


Installing MSVS + UF-Ex

Notes

You can either use [Visual Studio Code](#) or [Visual Studio](#) to use this engine, however this tutorial only covers MSVS, if you intend to use VSC, I might make a tutorial in the future, or you figure it out by yourself.

The PDF may get updated over time, for the most updated version, visit this [link](#).

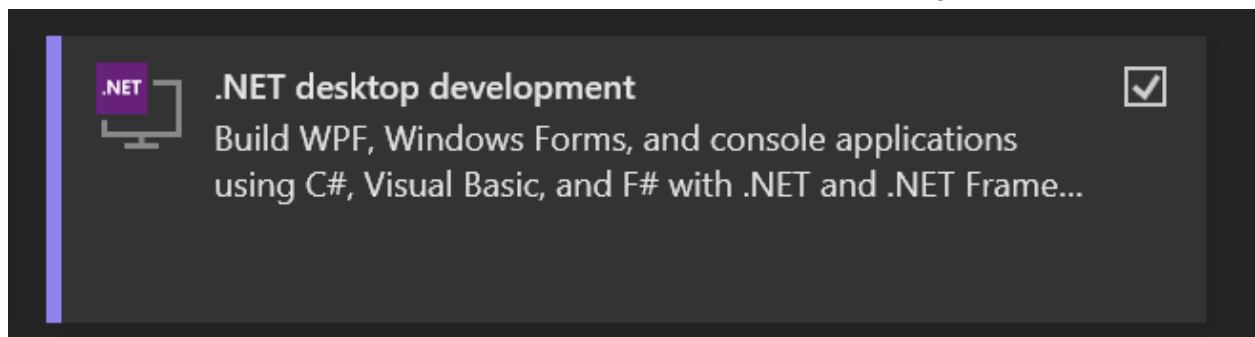
Requirements

- Microsoft Visual Studio 2022 (MSVS)
- [.NET](#) 9.0 (C# 13 or higher)
- UF-Ex engine pack ( **UndyneFight-Ex.dll**)
- UF-Ex .xml file (It is used for variable documentation)
- UF-Ex content pack (Will be distributed once 0.3.0 is released)
- 8 GB of disk space, 4 GB of RAM
- [The courage to ask for help](#).

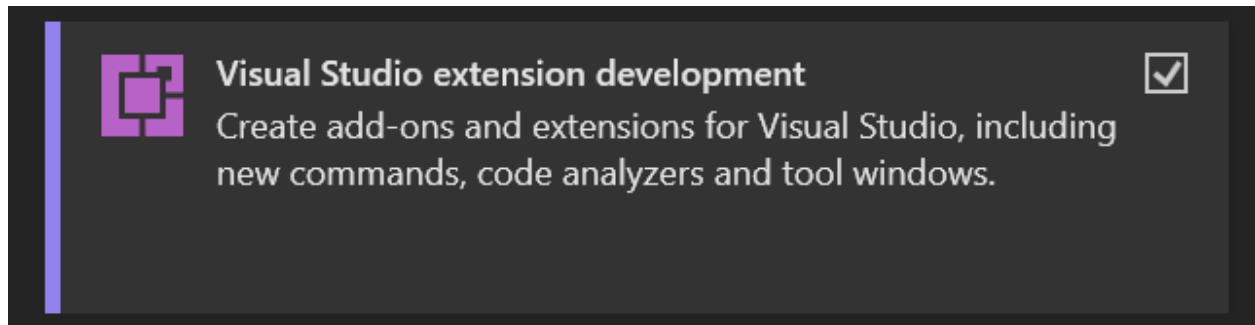
Installing Visual Studio

Visual Studio Community is sufficient for this engine, you don't have to install Professional or Enterprise.

You have to install .NET desktop development in order for the UF-Ex engine to function.

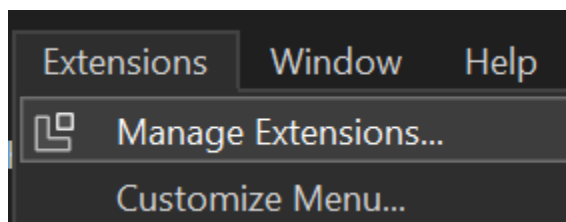


VS extension development is optional.

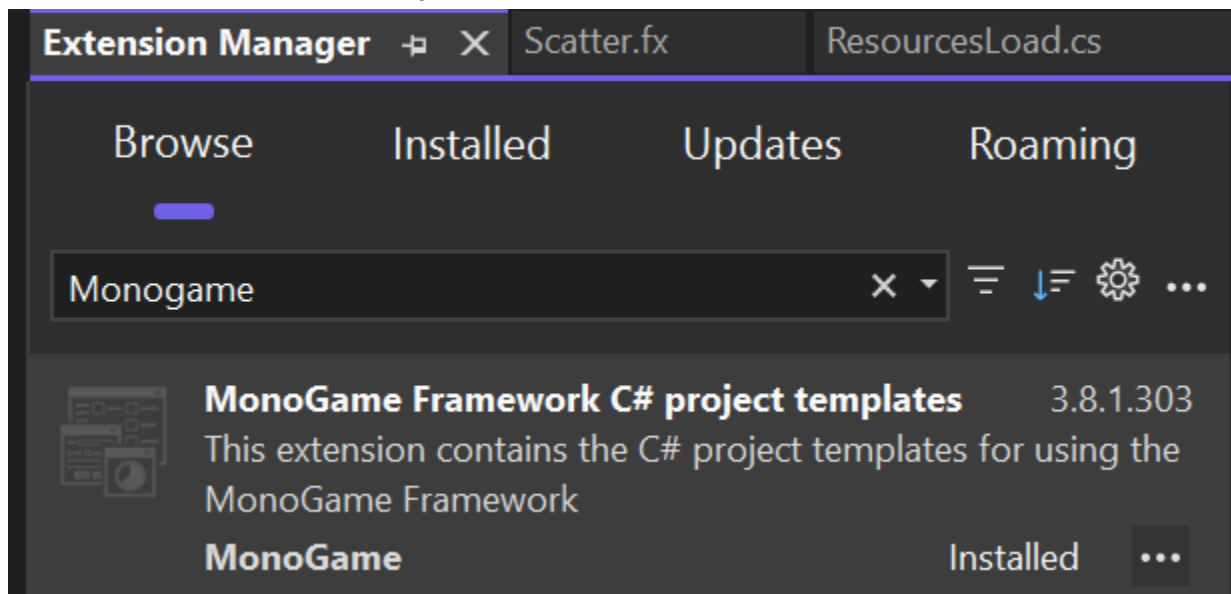


Setting up the solution

After installing MSVS, open the provided example solution, you can name it whatever you like. Open the solution (.sln), select “Extensions” at the top bar, and then choose “Manage Extensions”

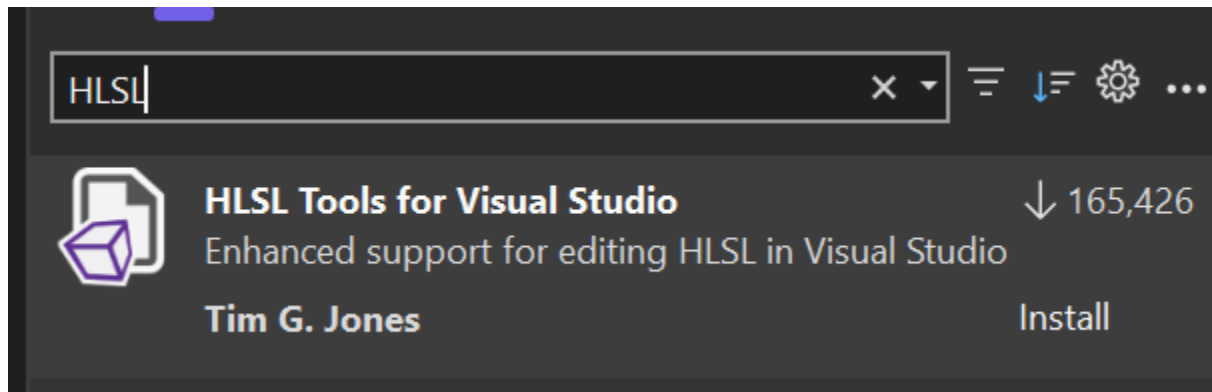


Choose it and search for “Monogame”



Install the extension shown in the image. (Note that the extension version in the image is outdated, ensure you have the latest version installed)

After installing Monogame Framework, search for “HLSL”

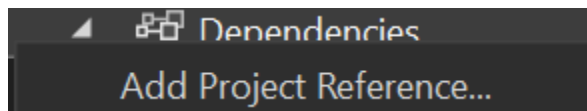


Install the extension shown in the image.

(You can choose to not install this if you can read HLSL shader codes in pure text.)

Note that you may need to restart your computer/MSVS after installing certain components.

After finishing the installation process, open MSVS and go to the solution explorer on the right hand side, Right click “Dependencies” and select “Add project reference”



Click “Browse” and select the engine .dll file and the data .dll file.

After importing the engine reference, by pressing (Ctrl +) F5, the game should run normally (And then you will see a hideous UI).

(Ctrl + F5 will not attach the debugger to the executable, it will run faster but will not show an error log when the game crashes.)

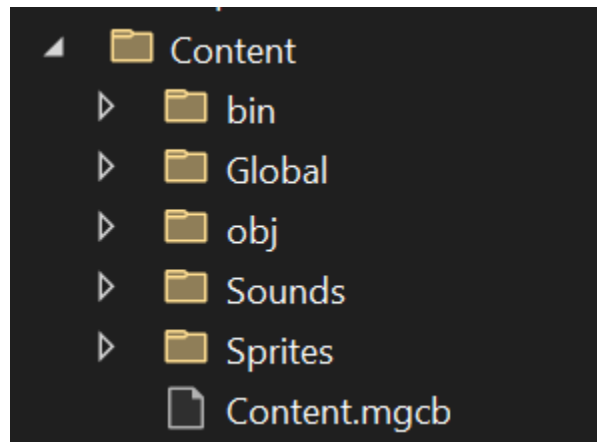
Importing music

If you try to select a song, you should see this.



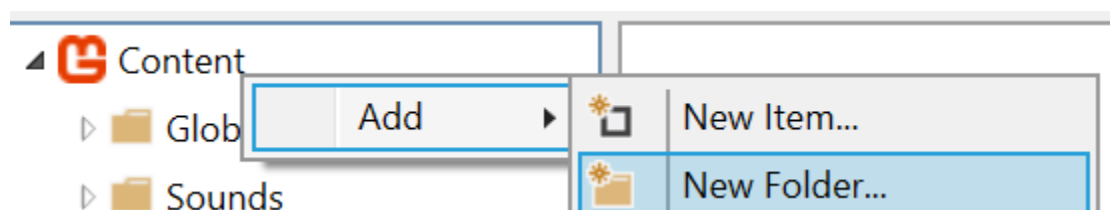
A bracket indicates that there is no associated music file for the chart, that means you need to import it.

To import assets, select "Content.mgcb" in the "Content" folder



You can just double click it or choose "Open With -> MGCB Editor".

If there is no such folder named "Musics", create one by right clicking "Content", and select "Add -> New Folder" and name it "Musics".



After having the “Musics” folder, create a folder that matches the name of the chart, in this case, it would be “My First Chart”.

Right click on the newly created folder and select “Add -> Existing Item”, then you can select the

music you want to import, make sure it says  **Copy the file to the directory** and

the imported music is set to a song  .

After importing the music, rename the music asset into “song.(file type)” (If the music you imported was an .ogg file, name it to song.ogg, and so on)

If you want to add a chart cover, simply import an image file and rename it into “paint”.

Note: The engine has auto scaling for chart covers, but it is best to set it to 640x480 before importing to avoid bugs.

After importing the chart assets, make sure to “Build” the MGCB by pressing F6, it is best to use “Rebuild” if you have removed or renamed assets.

Basic template information

Some comments are inside “Normal Model” and “Championship Model”, but here is the detailed explanation.

Method: IWaveSet

```
class MyFirstChart : WaveConstructor, IWaveSet
```

“WaveConstructor” contains the basic methods for charting such as beat calculation and the main chart making function.

“IWaveSet” is the [interface](#) that stores the basic information of the chart, such as the difficulties and music name.

```
public MyFirstChart() : base(62.5f / (100/*bpm*/ / 60f)) { }
```

The “100” shown in the image is the BPM of the chart, if you are unsure of the BPM, you can use an online BPM checker or use Malody (More accurate) to find the BPM of the music.

If there are multiple BPM in the chart, simply put them in arrays like this:

```
: base([10, 300], [10, 50], [999, 230])
```

This means that the first 10 beats have 300 bpm, the next 10 beats have 50 bpm, and the next 999 beats have 230 bpm.

```
public string Music => "My First Chart";
```

This is the file path to the music.

```
public string FightName => "My First Chart";
```

This is the name the chart will be shown in the chart select menu.

```
private class ThisInformation : SongInformation
{
    //Setup the difficulties of the chart
    0 references
    public override Dictionary<Difficulty, float> CompleteDifficulty => new<...>;
    0 references
    public override Dictionary<Difficulty, float> ComplexDifficulty => new<...>;
    0 references
    public override Dictionary<Difficulty, float> APDifficulty => new<...>;
    0 references
    public override string BarrageAuthor => "Name";
    0 references
    public override string AttributeAuthor => "Name";
    0 references
    public override string PaintAuthor => "Name";
    0 references
    public override string SongAuthor => "Name";
    //If MusicOptimized is set to true, the song speed will be adjusted when the game is slowed down/sped up
    1 reference
    public ThisInformation() { MusicOptimized = false; }
}
```

“Complete/Complex/AP Difficulty” represents the difficulty constant of the chart.

“BarrageAuthor” stands for the one who made the chart.

“AttributeAuthor” stands for the one who made the effects.

“PaintAuthor” stands for who made the chart cover.

“SongAuthor” stands for the song composer.

There are other variables in “SongInformation”, such as “Hidden” and “Extra”, they will be discussed in detail later.

```
public new void Start()
{
    ...
}
```

This function will run when the chart begins, you can initialize the chart here, such as box position and size, soul location, or shader variables.

```

public void Noob()...
0 references
public void Easy()...
0 references
public void Normal()...
0 references
public void Hard()...
0 references
public void Extreme()...
0 references
public void ExtremePlus()...

```

These are the functions that will be executed in each difficulty, Noob() will be executed in the noob difficulty, Easy() will be executed in easy difficulty, etc. You can not include the difficulties if it is not implemented.

Method: IWaveSetS

Sometimes, you may find that you need to copy and paste chunks of code from one difficulty to another, and when you need to change certain aspects of the code, you need to manually change them all. Not only is this process boring and tedious, it makes the code unreadable and too large. In short, this is poor coding practice and should be avoided.

To simplify the charting process, IWaveSetS is created, your code can now look as simple as this

```

public void Chart()
{
    Effect();
    switch (CurrentDifficulty)
    {
        case Difficulty.Noob:...
        case Difficulty.Normal:...
    }
}

```

Or even this

```
int spd = (int)CurrentDifficulty switch
{
    0 => 4,
    1 => 5,
    2 => 6
};
CreateChart(0, BeatTime(1), spd, ["R", "", "", "etc"]);
```

Not only is the code more efficient, it is more readable and easy to keep track of.