AshLib Usage Documentation

Documentation for AshLib v2.1.0, by Dumbelfo.
https://github.com/Dumbelfo08/AshLib

Index

Contents

This documentation is aimed at teaching how to use AshLib practically. It will contain examples, and explain the intended use of things.
Let's start!

Dates

This struct represents a date, a point in time. It goes down to second, so that is: years, months, days, hours, minutes and seconds. The years have a limitation, they can only be between 1488 and 2511.

It's important to note that this struct is immutable.

Their most useful characteristic is the CPTF, which stands for Compressed printable date format. This is a way to represent a date in just 6 printable characters! This uses base64 chars, so all of it is printable.

Creation

To create a new Date struct, you could use the constructor:

```
Date d = new Date(0, 0, 12, 12, 8, 1897);
```

The arguments are in the following order: seconds, minutes, hours, day, month, year.

So we created a date on the 12th of August of 1897, at 12:00.

You can also cast from a DateTime, so you can do things like this:

```
Date d = (Date) DateTime.Now;
```

CPTF

To convert a Date into the CPTF format, we use the ToCPTF method. Following the previous example:

```
Date d = new Date(0, 0, 12, 12, 8, 1897);
string cptf = d.ToCPTF();
Console.WriteLine(d);
Console.WriteLine(cptf);
```

This should output something like:

```
12/8/1897 12:0:0
AMAYZm
```

And if we do the inverse operation now, we should get the original data. To transform a CPTF to a Date again, we can use the static method, or alternatively the constructor directly.

```
Date d = new Date(0, 0, 12, 12, 8, 1897);
string cptf = d.ToCPTF();
Console.WriteLine(d);
Console.WriteLine(cptf);
Date r = Date.FromCPTF(cptf); //Also new Date(cptf); would have worked
Console.WriteLine(r);
```

We will get the output that we want to get:

```
12/8/1897 12:0:0
AMAYZm
12/8/1897 12:0:0
```

This is useful for places where you need to have Dates occupy very little space and needs to be printable.

DeltaHelper

DeltaHelper aims at providing an easy way for user to handle everything related to times in their applications. You will be able to calculate ***fps***, calculate ***deltaTime***, **target** a specific framerate, and calculate fps especially directed for showing it to the user without having them fluctuating.

To start, create a new instance and start it at the start of your code:

```
DeltaHelper dh = new DeltaHelper();
dh.Start();
```

Now, each frame, we need to call the ***Frame*** method. A graphic thread will be simulated.

```
while(true){
      Thread.Sleep(13); //Simulate calculations

      dh.Frame();
}
```

Its very important to **always call Frame at the very end** of the frame.

Using fps

We can access the fps value using the get-only properties ***fps***. For example, you can do:

```
while(true){
      Thread.Sleep(13); //Simulate calculations

      double fps = dh.fps;
      Console.WriteLine("FPS: " + fps);

      dh.Frame();
}
```

This fps value is updated each frame, so it will fluctuate rapidly:

```
FPS: 62.58448906023145
FPS: 67.74333405592868
FPS: 67.35548879878185
FPS: 66.27827596948552
FPS: 62.63544915880664

...
```

Using stableFps

As you can see, this number is not very good for displaying the value. We can instead use the ***stableFps*** property:

```
while(true){
      Thread.Sleep(13); //Simulate calculations

      double sfps = dh.stableFps;
      Console.WriteLine("FPS: " + sfps);

      dh.Frame();
}
```

This will output an fps value that changes only every **second** (one second is the **default**, it can be changed) and is an average of the fps that happened in that time (is more accurate at smoothing out values for having a correct bigger picture of everything).

This is how the output would look like:

```
FPS: 64.56245379439002
FPS: 64.56245379439002
FPS: 64.56245379439002
FPS: 64.56245379439002
FPS: 64.56245379439002
FPS: 64.56245379439002
FPS: 60.65326951946504
FPS: 60.65326951946504
FPS: 60.65326951946504
```

As it is noticeable, the value changes suddenly. For all the frames of a second, the same value will be outputted.

Setting the stable update time

The time of one second can be changed, using the ***SetStableUpdateTime*** method:

```
DeltaHelper dh = new DeltaHelper();
dh.SetStableUpdateTime(100d); //Set it to 100 miliseconds
dh.Start();

while(true){
      Thread.Sleep(13); //Simulate calculations

      double sfps = dh.stableFps;
      Console.WriteLine("FPS: " + sfps);

      dh.Frame();
}
```

As it can be seen, we set it to 100 **miliseconds**. This is not a good time for showing it to users, but its good for seeing how it works.

The output looks something like this:

```
FPS: 64.15538066595118
FPS: 64.15538066595118
FPS: 62.754998211482494
FPS: 62.754998211482494
FPS: 62.754998211482494
FPS: 62.754998211482494
FPS: 62.754998211482494
FPS: 62.754998211482494
FPS: 56.56705836211968
FPS: 56.56705836211968
```

It is visible that takes few less frames to update it. Nice!

DeltaTime

In many types of applications, including games, you need to know the time that a frame took to calculate to take it into account in calculations, so things will happen in the same time no matter the framerate.

This time between frame and frame is called ***deltaTime***. To get it here, it is as easy as to use the get-only property.

```
while(true){
      Thread.Sleep(13); //Simulate calculations

      player.Move(5d * (dh.deltaTime/1000d)); //Simulate some kind of movement

      dh.Frame();
}
```

In here, we want the player to move 5 units by second, so we multiply it by ***deltaTime***. Why do we divide it by 1000 first? Because ***deltaTime*** comes in **milliseconds**, and we we want to move 5 units by **second**.

Target framerate

In your applications, you sometimes want to get X number of frames per second. It is easy to do that here, just use the ***Target*** method before ***Frame***:

```
while(true){
      Thread.Sleep(13); //Simulate calculations

      double fps = dh.fps;
      Console.WriteLine("FPS: " + fps);

      dh.Target(30d); //Target 30 fps
      dh.Frame();
}
```

We call it with 30 as the argument. 30 is in this case **fps**.

As the output shows, this works pretty well:

```
FPS: 29.99967000362738
FPS: 29.999580005879615
FPS: 29.999310015869604
FPS: 29.99976000192168
FPS: 29.999940000123697
FPS: 29.999580005879615
```

Also, as another feature, we can get the total time the application has been running, using ***GetTime***:

```
Time: 2.9786007
Time: 3.0090410000000003
Time: 3.0560126
Time: 3.086485
Time: 3.1332364
```

This returns the time since the start in **seconds**. Great!

AshFile

An AshFile is a data structure composed of camps. Each **camp** has a **name** and a **value**. This value can be of different types(numbers, text...).

It also is a file format, with the extension **.ash**. It is very easy to save and load an AshFile from a file, making it a reliable and easy way to use it in programming projects and then store it.

Camp Manipulation

So we can start with the most important thing, the **camps**.

We can create a new AshFile like this:

```
AshFile af = new AshFile();
```

This AshFile is completely empty and ready to use!

So we can start creating a camp named "*Hello*" and having its value be a string of value "*World!*", using ***SetCamp***:

```
af.SetCamp("Hello", "World!");

Console.WriteLine(af.GetCamp("Hello"));
```

When we log into the console, we see it will output "*World!*".

But if we try to have a middle step, getting it into a string before logging, it won't compile:

```
af.SetCamp("Hello", "World!");

string l = af.GetCamp("Hello");

Console.WriteLine(l);
```

Why? Because ***GetCamp*** returns an **object**. Camps can have values of many types, and so an object can be all of them. So how do we get the string? We could cast it into a string, but there is a better way, using ***CanGetCampAsString***:

```
string l = "";
if(!af.CanGetCamp("Hello", out l)){
      Console.WriteLine("An error occured!");
}

Console.WriteLine(l);
```

if we can get the camp as a string, it will be outputted in the out argument. If we cannot, the if code will execute, and we can handle all errors there.

In the last example, we can actually get it as a string, and "*World!*" will be logged. But what if we couldn't?

```
af.SetCamp("Hello", 12);

string l = "";
if(!af.CanGetCamp("Hello", out l)){
      Console.WriteLine("An error occured!");
}

Console.WriteLine(l);
```

Here, it is an int. So, it will log that an error has occurred. That way, we can stop cast errors!

Here we have an example with more camps:

```
AshFile af = new AshFile();
af.SetCamp("Hello", "World!");
af.SetCamp("test1", 13211);
af.SetCamp("test2", true);
af.SetCamp("test3", 0.0003f);

Console.WriteLine(af.Visualize());
```

Visualize will display all of the camps in a nice format.

The output should be:

```
Hello: World!
test1: 13211
test2: True
test3: 0.0003
```

Nice way to display all!

What is we try to set a camp that already has a value?

```
af.SetCamp("Hello", "World!");
af.SetCamp("test1", 13211);
af.SetCamp("Hello", true);
```

If we log the whole contents, this happens:

```
Hello: True
test1: 13211
```

The last call replaces the old one without an issue.

There is much more things we can do to camps:

```
AshFile af = new AshFile();
af.SetCamp("1",103);
af.SetCamp("2",109);
af.SetCamp("3",123);

af.DeleteCamp("2");

af.RenameCamp("1", "s");
```

After these operations, we can see all the camps:

```
s: 103
3: 123
```

Also, there is the initialize method. If the camps doesn't exist, it will create with the given value. But if it exists, nothing will happen. This is an example:

```
af.SetCamp("a", 13.5f);
af.SetCamp("b", 90.46f);

af.InitializeCamp("a", 0f);
af.InitializeCamp("c", 10f);
```

And the output if we log it is:

```
a: 13.5
b: 90.46
c: 10
```

It is important to note is that the value of camps can be of **27** different types! You can check the Reference documentation for all of them.

Converting to File

Another of the important features of AshFiles is the ability to convert them into files in a new file format, ***.ash***. You can store and load them really easily.

In reality, what files are in computers is an array of bytes, so this conversion will be focused on that, and then putting that into a file.

We can save an AshFile to a file using the ***Save*** method:

```
AshFile af = new AshFile();
af.SetCamp("camp1", "You can actually");
af.SetCamp("camp2", "save these files!!!! :0");

af.Save("C:/ashFileTest/file1.ash");
```

For example, here we initialize an AshFile and then save it. As easy as that!

Now, how do we load files?

```
AshFile af = new AshFile();
af.Load("C:/ashFileTest/file1.ash");

Console.WriteLine(af.AsString());
```

If we do this after the previous operation (saving it), we should see the two camps. Instead of using the ***Load*** method, we can also just use a constructor:

```
AshFile af = new AshFile("C:/ashFileTest/file1.ash");
```

This will do the same.

When we load or save a file, we are actually saving the path internally (the ***path*** field). So we don't have to specify it always!

```
AshFile af = new AshFile("C:/ashFileTest/file1.ash");

af.SetCamp("never", "gonnaGiveYouUp");

af.Save();
```

Notice how we call ***Save*** with no arguments, because the path is stored from the constructor.

Error handling

Sometimes, when reading and writing to files, errors occur. It's important to at least know what happened. AshFiles have a nice way to see it. It consists of ***GetErrorCount*** and ***GetErrorLog***.

Here is an example of how they would be used:

```
//After an operation or a number of operations is when we check for errors

Console.WriteLine("Number of errors that occured: " + AshFile.GetErrorCount());
if(AshFile.GetErrorCount() > 0){
      Console.WriteLine("Error log: " + AshFile.GetErrorLog());
}

AshFile.EmptyErrors(); //We empty the errors so the current ones wont mix with the
ones of the next operation
```

This is a correct way to handle errors, but it can be implemented in many different ways.

Other

Only the most important parts were covered in this guide. Other stuff exists, and you can find information on it in the reference documentation. Also, there is example code on some of these things. Happy coding!