Enhanced Code Flow for Unreal Engine 4

This code plugin provides functions that drastically improve the quality of life during the implementation of game flow in C++.

It works very well with gameplay programming, UI programming with a lot of transitions or in any other situation.

Index:

Installation

This is a code only plugin. It means that to use it you have to:

1. Put the whole plugin into a Plugins directory of your project.
2. Add **"EnhancedCodeFlow"** entry to the **PublicDependencyModuleNames** in your project's **build.cs** file.
3. Add **#include "EnhancedCodeFlow.h"** to a file in which you want to use the plugin.

Usage

Run the following functions to use enhanced code flow!

> Note that every function must receive a pointer to an owner that runs this function in it's first argument.
>
> The owner must be able to return a World via **GetWorld()** function.

Delay

Execute specified action after some time. This can be useful in many various situations. Everytime when I was using a Delay node in blueprints I wish there was an equivalent of it in c++.

```
FFlow::Delay(this, 2.f, [this]()
{
  // Code to execute after 2 seconds.
});
```

Add Ticker

Creates a ticker. It can tick specified amount of time or until it won't be stopped or when owning object won't be destroyed.

Useful for actors and components that you don't want to be tickeable, but needs one tick to do something.

Run ticker for 10 seconds

```
FFlow::AddTicker(this, 10.f, [this](float DeltaTime)
{
  // Code to execute every tick
});
```

Run ticker for infinite time and stop it when you want to

```
FFlow::AddTicker(this, [this](float DeltaTime, FECFHandle TickerHandle)
{
  Code to execute in every tick.

  // Use this to stop the ticker
  FFlow::StopAction(this, TickerHandle);
});
```

Run ticker for infinite time and something else stops it

```
FECFHandle TickerHandle = FFlow::AddTicker(this, [this](float DeltaTime)
{
  // Code to execute in every tick.
});

// Use this to stop the ticker
FFlow::StopAction(this, TickerHandle);
```

Note 1: Tickers and every other plugin actions are impacted by global time dilation.

Note 2: You can check if the ticker (or any other action) is running using **FFlow::IsActionRunning(TickerHandle)**

Wait and execute

Waits until specific conditions are made and then executes code.

The conditions are defined in a form of predicate.

Perfect solution if code needs a reference to an object, which spawn moment is not clearly defined.

```
FFlow::WaitAndExecute(this, [this]()
{
  // Write your own predicate.
  // Return true when you want to execute the code below.
  return bIsReadyToUse;
},
[this]()
{
  // Implement code to execute when conditions are met.
});
```

While true execute

While the specified conditions are true tick the given code.

This one is useful when you want to write a loop that executes one run every tick until it finishes it's job.

```
FFlow::WhileTrueExecute(this, [this]()
{
  // Write your own predicate.
  //Return true when you want this action to continue.
  return bIsRunning;
},
[this](float DeltaTime)
{
  // Implement code to tick when conditions are true.
});
```

Add timeline

Easily launch the timeline and update your game based on them. Great solution for any kind of blends and transitions. The function requires the following parameters:

- StartValue - a value with which the timeline will begin;
- StopValue - a value with which the timeline will end. StopValue can be lesser than StartValue;
- Time - how long the timeline will work;
- TickFunc - a function that will tick with the timeline. It has the following arguments:
    - Value - a current value on this timeline;
    - Time - a time that passed on this timeline;
- CallbackFunc - a function that will run when the timeline comes to an end. Has the same arguments as TickFunc. This function is *optional*;
- BlendFunc - a function that describes a shape of the timeline:
    - Linear *(default)*
    - Cubic
    - EaseIn
    - EaseOut
    - EaseInOut
- BlendExp - an exponent defining a shape of EaseIn, EaseOut and EaseInOut function shapes. *(default value: 1.f)*;

```
FFlow::AddTimeline(this, 0.f, 1.f, 2.f, [this](float Value, float Time)
{
  // Code to run every time the timeline tick
},
[this](float Value, float Time)
{
  // Code to run when timeline stops
},
EECFBlendFunc::ECFBlend_Linear, 2.f);
```

Add custom timeline

Creates a discrete timeline which shape is based on a **UCurveFloat**. Works like the previously described timeline, but an asset with a curve must be given.

```
FFlow::AddCustomTimeline(this, Curve, [this](float Value, float Time)
{
  // Code to run every time the timeline tick
},
[this](float Value, float Time)
{
  // Code to run when timeline stops
});
```

Stopping actions

Every function described earlier returns a **FECFHandle** which can be used to check if the following action is running and to stop it.

```
FFlow::IsActionRunning(GetWorld(), Handle); // <- is this action
running?
FFlow::StopAction(GetWorld(), Handle); // <- stops this action!
```

Note that this function requires a pointer to the existing **World** in order to work properly.

You can also stop all of the actions from a specific owner or from everywhere:

```
FFlow::StopAllActions(GetWorld()); // <- stops all of the actions
FFlow::StopAllActions(GetWorld(), Owner); // <- stops all of the actions
started from this specific owner
```

You can also stop all of the **specific** actions. In this case you can also optionally specifiy an owner of this actions, or simply stop all of them.

```
FFlow::RemoveAllDelays(GetWorld());
FFlow::RemoveAllTickers(GetWorld());
FFlow::RemoveAllWaitAndExecutes(GetWorld());
FFlow::RemoveAllWhileTrueExecutes(GetWorld());
FFlow::RemoveAllTimelines(GetWorld());
FFlow::RemoveAllCustomTimelines(GetWorld());
```

Extending plugin

If you have a source code of this plugin you can easily extend it's functionalities!

Check how other actions are made to easier understand how to extend the plugin.

1. Create a class that inherits from **UECFActionBase**
2. Implement **Setup** function, which accepts all parameters you want to pass to this action. **Setup** function must return true if the given parameters are valid.

```
bool Setup(int32 Param1, int32 Param2, TUniqueFunction<void()>&&
Callback)
{
  CallbackFunc = MoveTemp(Callback);
  if (CallbackFunc) return true;
  return false;
}
```

   Any callback must be passed as an r-value reference and be moved to the action's variable.

3. Override **Init** and **Tick** functions if needed.
4. If you want this action to be stopped while ticking - use **MarkAsFinished()** function.
5. In the **FEnhancedCodeFlow** class implement static function that launches the action using **AddAction** function. The function must receive a pointer to the launching **UObject** pointer and every other argument that is used in the action's **Setup** function in the same order. It must return **FECFHandle**.

```
FECFHandle FEnhancedCodeFlow::NewAction(UObject* InOwner, int32 Param1,
int32 Param2, TUniqueFunction<void()>&& Call)
{
  if (UECFSubsystem* ECF = UECFSubsystem::Get(InOwner))
    return ECF->AddAction<UECFNewAction>(InOwner, Param1, Param2,
MoveTemp(Call));
  else
    return FECFHandle();
}
```

6. You can optionally add static function which will stop this action

```
void FFlow::RemoveNewActions(const UObject* WorldContextObject, UObject*
InOwner)
{
  if (UECFSubsystem* ECF = UECFSubsystem::Get(InOwner))
  {
    ECF->RemoveActionsOfClass<UECFNewAction>(InOwner);
  }
}
```

It is done! Now you can run your own action:

```
FFlow::NewAction(this, 1, 2, [this]()
{
  // Callback code.
});
```

Contact

If you have any question ask it on forum thread: https://forums.unrealengine.com/unreal-engine/marketplace/1868196-enhanced-code-flow I will try my best to answer it quickly :)

Thanks

I want to send special thanks to Monica, because she always supports me and believes in me, and to Pawel, for allowing me to test this plugin on his project.

Also, I want to thank You for using this plugin! It is very important for me that my work is useful for someone!

Happy coding!