Custom User Modules

This is a tutorial that aims to teach you, the prospective HPL3 modder, on how to use a user module. It assumes that you have a general grasp of programming in AngelScript and are more or less comfortable with HPL3's scripting process. Basically, by the time you start this tutorial, you should already have some map scripting experience under your belt.

What is a User Module?

A user module is basically a self-contained bundle of code that is active at all times. This is contrary to a map script that is only active when that particular map is loaded.

Many of the systems you see in SOMA are handled with the use of user modules (which I am going to hence forth just shorten to “modules”). Default player behavior, the game over screen, the main game menu, and even the credits are all controlled with modules. You can check out the User Module Page for a more complete list on the existing modules that come default with the game.

In addition to containing behavior and storing data that persists across maps, you can also use a module to draw graphics and widgets using ImGui directly onto the player's view, much like you can onto a terminal entity. You can even have the cursor appear for an interactive experience right on the screen.

As you can guess, modules can be used for a variety of things that cannot be done with a map script alone. These can be such cases like:

- A global inventory system
- An oxygen monitoring system
- A secret message viewer
- Contextual visuals and sounds
- An AI-like monitoring system (think Left 4 Dead's AI Director)

Getting Started

The first thing you need when creating a module is the script file itself. The script file is named using the convention “——Handler.hps”. Module scripts consist of a primary class, with the naming scheme of “cScr——Module” that inherits from `iScrUserModule` and `iScrUserModule_Interface`. In addition, you must include both “`interfaces/UserModule_Interface.hps`” and “`modules/ModuleInterfaces.hps`”. There are a number of built-in functions, and while not all of them are required to exist in your script, it's generally a good idea to have them in there just in case.

You can save the file more or less wherever you want (as long as the place is added to your resources.cfg), but the convention used by SOMA is to store module code files in a “modules” folder that is inside the “script” folder.

As a baseline, you can copy the following code:

```
#include "interfaces/UserModule_Interface.hps"
#include "modules/ModuleInterfaces.hps"

class cScrBaseHandler : iScrUserModule, iScrUserModule_Interface
{
//////////////////////////////////////////////////////////////////////////
//////////////
    // ==============
    // MAIN CALLBACKS
    // ==============
//{//////////////////////////////////////////////////////////////////////////
//////////////

    //-------------------------------------------------------

    void Init() {}
    void Update(float afTimeStep) {}
    void PostUpdate(float afTimeStep) {}
    void VariableUpdate(float afDeltaTime) {}
    void Reset() {}
    void OnDraw(float afFrameTime) {}
    void OnPostRender(float afFrameTime) {}
    void LoadUserConfig() {}
    void SaveUserConfig() {}
    void OnMapEnter(cLuxMap @apMap) {}
    void OnMapLeave(cLuxMap @apMap) {}
    void CreateWorldEntities(cLuxMap @apMap) {}
    void DestroyWorldEntities(cLuxMap @apMap) {}
    void PreloadData(cLuxMap @apMap) {}
    void OnEnterContainer(const tString&in asOldContainer) {}
    void OnLeaveContainer(const tString&in asNewContainer) {}
    void OnExitPressed() {}
    void OnAction(int alAction,  bool abPressed) {}
    void OnAnalogInput(int alAnalogId,  const cVector3f &in avAmount) {}
    void AppGotInputFocus() {}
    void AppLostInputFocus() {}

    //-------------------------------------------------------

    //} END MAIN CALLBACKS

//////////////////////////////////////////////////////////////////////////
//////////////
    // ================
    // GLOBAL FUNCTIONS
    // ================
//{//////////////////////////////////////////////////////////////////////////
//////////////
```

```
    //------------------------------------------------------

    //------------------------------------------------------

    //} END GLOBAL FUNCTIONS

//////////////////////////////////////////////////////////////////////////
//////////////
    // ==============
    // GUI CALLBACKS
    // ==============
//{//////////////////////////////////////////////////////////////////////////
//////////////

    //------------------------------------------------------

    void OnGui(float afTimeStep) {}

    //------------------------------------------------------

    //} END GUI CALLBACKS
}
```

Next, you need to add your module to your mod's “Modules.cfg” file. That file is an XML file that lists every module loaded when the game starts. (If your mod doesn't have a “Modules.cfg” file, you can copy the one from SOMA's config folder to use as a starting point.)

There are several attributes that need to be set for your module's entry. An entry looks like this:

```
<Module
    Name = "MenuHandler"
    ScriptFile = "modules/MenuHandler.hps"
    ScriptClass = "cScrMenuHandler"
    ID = "13"
    IsGlobal = "false"
    Container = "Default"
    UseInputCallbacks = "true"
/>
```

- **Name:** The name of the module. This name will be used in various places of a script to call functions within your module.
- **ScriptFile:** The path/name of the script file for your module.
- **ScriptClass:** The name of the class for your module script.
- **ID**: The ID of the module for the in-engine module list. This ID must be unique and should be sequential among every declared module. (0-19 is used by SOMA's built-in modules, so I recommend that any custom modules start at 20.)
- **IsGlobal:** *<Unknown effect>**
- **Container:** *<Unknown effect>**
- **UseInputCallbacks:** Whether the game's input actions should be rerouted to the module's `OnAction` function. (If false, the `OnAction` function will never get called.)

** The IsGlobal and Container attributes are unknown in what effect they have on the module, but every default SOMA module uses the same value for both of them - "false" and "Default", respectively.*

Scripting

There are four general categories of of modules that you can make: Always-On, Toggled, Heads Up Display, and Input-Based. Many modules will fall into one or more of these categories, but I'll boil the basics of each one, and how they are generally handled.

Always-On

As you may have guessed, these modules are always running. Whether that means they are always monitoring something, or always playing a sound, or always draws something to the screen. The `PlayerToolHandler` module is a good example of this, as it is constantly checking if the player is looking at an entity that can be interacted with.

The Always-On modules make heavy use of the `Update` function. This function runs on every frame, so anything you do that needs to be updated constantly can be placed in this function. It looks like this:

```
void Update(float afTimeStep)
{

}
```

- **afTimeStep:** The amount of time in seconds that have passed since the last time Update was called.

Toggled

The Toggled modules are modules that run continuously like Always-On modules, but they are turned on and off at different times. They can be toggled manually via a script, or it can be a condition that is met or unmet that the module checks through use of the `Update` function. The `PlayerHandsHandler` does this a lot, as it is only activated when something happens that causes the player's hand models to become visible.

Like the Always-On modules, the Toggled modules generally make use of the `Update` function, but they are also heavily affected by the use of global functions (discussed below) that other scripts can call in order to toggle the module on or off, or to hand it important information.

Heads Up Display

Heads Up Displays differ from other modules in that they draw information onto the screen, and optionally allows for interaction. As the name suggests, this screen layer is called the heads-up display, or the HUD. `MenuHandler` is a great example of this; as the module responsible for

displaying the title and pause menus, it makes ample use of graphics and interactivity.

The bulk of the work in Heads Up Display modules take place in the `OnGui` function. This function will look familiar, as it's identical to the `OnGui` functions that are used for terminals:

```
void OnGui(float afTimeStep)
{

}
```

- **afTimeStep:** The amount of time in seconds that have passed since the last time OnGui was called.

Input-Based

Input-Based modules are modules that derive much of their function as a direct response to player actions. Actions are just the mapped keys that the player could press, such as jump, walk, or interact. `InventoryHandler`, though a small module, makes use of player input in order to display the player's inventory.

Input-Based modules make use of the `OnAction` function to capture and respond to player input. (Since it uses that function, if you make an Input-Based module, you will need to set the "UseInputCallbacks" attribute in the "Modules.cfg" to *true*.)

```
void OnAction(int alAction,  bool abPressed)
{

}
```

- **alAction:** An integer that represents what action was fired. Check against various `eAction` values (such as `eAction_Interact`).
- **abPressed:** Whether the action is the result of a button/key getting pressed (*true*) or released (*false*).

(When making an Input-Based module, you can also use the OnAnalogInput function to receive player actions from controllers.)

Global Functions

Many times when you create a module, you want to be able to interact with or control the module from elsewhere in your mod. In cases like this, you are going to want to create a few functions that are designed to interoperate with external scripts. These functions are the global functions.

You can make the global functions do whatever you want, but their use has to follow a few rules. The first is strict - the function has to have no parameters, and it has to have a return value of void. The second is less important, but still recommended - the function name should have the naming convention of "_Global_———". A global function can look like this:

```
void _Global_DoSomeWork()
```

```
{

}
```

Now you might notice that, since the function must return nothing and have no parameters, there is no obvious way to pass information in or get information out of these functions. For this purpose, HPL3 has a series of `cScript_X` functions. These functions fall into a number of classes:

- **GetGlobalVarX/SetGlobalVarX:** Gets or sets a value for a global variable.
- **GetGlobalArgX/SetGlobalArgX:** Gets or sets an argument (parameter) for a global function call.
- **GetGlobalReturnX/SetGlobalReturnX:** Gets or sets a return value for a global function call.

Since the “GlobalArg” and “GlobalReturn” families of functions deal primarily with global function use, they will be the ones that you use the most. The “GlobalVar” family of functions are better suited for manipulating global variables, you won't necessarily be using them much with global functions.

In addition to knowing the families, you should also know about the types you are able to work with. These types will replace the “X” in the above family names:

- String
- Bool
- Int
- ID
- Float
- Vector2f
- Vector3f
- Vector4f
- Matrix
- Color

The last piece of these functions are the parameters. The parameters are as follows:

- **GetGlobalArgX:** (int alIdx)
  - **alIdx:** The index of the global argument to retrieve.
- **SetGlobalArgX:** (int alIdx, X aX)
  - **alIdx:** The index of the global argument to set.
- **aX:** The value to store as an argument. (The type will depend on which type of function you are using.)
- **GetGlobalReturnX:** ()
  - *No parameters*
- **SetGlobalReturnX:** (X aX)
  - **aX:** The value to store as a return value. (The type will depend on which type of function you are using.)

The “GlobalArg” functions deal with an index. This index mirrors which position you are trying to reference (i.e. first argument, second argument, third argument, etc.). The “GlobalReturn” functions don't require an index, as they only need to deal with the one return value.

Now that you know all the pieces, you know how these functions are structured. The complete

signature of a script function is `cScript_FunctionNameFunctionType(Parameters)`. For example, the signature for setting a string argument is `cScript_SetGlobalArgInt(0, "string")`.

Now, we've reached this point where we can pass information to and from the global functions, but we still need a way to actually call the function. This is done with one additional "cScript" function:

```
cScript_RunGlobalFunc(tString asObjName, tString asClassName, tString asFuncName)
```

- **asObjName:** The name of the module for which to execute the function. (The name that you gave in the "Modules.cfg" file.)
- **asClassName:** The name of the class object to call the function on. (Pass "" to use the default module loaded by the game.)
- **asFuncName:** The name of the function to call.

Now that we've got all these parts, I can show you an example of how to use them in practice:

(In your map script)

```
void DoSomeWork(int alSomeInt, tString asSomeString)
{
    cScript_SetGlobalArgInt(, alSomeInt);
    cScript_SetGlobalArgString(1, asSomeString);
    cScript_RunGlobalFunc("SomeHandler", "", "_Global_DoSomeWork");

    bool abResult = cScript_GetGlobalResultBool();

    // Do some nonsense with the result
}
```

(In your module script)

```
void _Global_DoSomeWork()
{
    int lSomeInt = cScript_GetGlobalArgInt();
    string sSomeString = cScript_GetGlobalArgString(1);

    // Do whatever module wizardry here...

    cScript_SetGlobalReturnBool(true);
}
```

So the process of this script is as follows. In the map script, it sets the global arguments at indices 0 and 1 to an integer and string value, respectively. It then launches the function `_Global_DoSomeWork` from the module `SomeHandler`. In the function, the module retrieves the arguments that the map script set and does... whatever... with them. Once the work is done, the module assigns the return value to a boolean value and exits. Back in the map script, the return value is retrieved, and the map script continues.

Helper Script

In the previous section, I covered how you can use global functions in your map script to call functions set in your modules. In practice, you generally don't want to call the “cScript” functions directly from within your map scripts - the script is bulky and verbose, not to mention difficult to easily read and understand. Also, if it's a module function you will need to call multiple times, it's cumbersome to maintain that bulky code in more than one place. Instead, what you'll want to do is create another script file that handles all the heavy lifting and provides an easy-to-understand interface between the map script and the module.

While a helper script necessitates creating a new script file, the good news is the only measure you need to take is to make sure the script loaded via the “resources.cfg” file. There are no classes or includes that you need to add to the script file (other than the ones you need depending on what you're trying to do). In fact, the only things you need to do is add the functions themselves to the script file. A couple convention things to keep in mind is that the files typically go in a “script/helpers” folder and take the name “helper_——.hps”, and the functions themselves generally take the name of `ModuleName_FunctionName`.

To take the example from the previous section, you might set up the helper script like so:

(In your helper script)

```
bool SomeModule_DoSomeWork(int alSomeString, tString asSomeString)
{
    cScript_SetGlobalArgInt(, alSomeInt);
    cScript_SetGlobalArgString(1, asSomeString);
    cScript_RunGlobalFunc("SomeHandler", "", "_Global_DoSomeWork");

    return cScript_GetGlobalResultBool();
}
```

As you can see, the logic is generally the same from the map script from the previous example - it takes the parameters and passes them as arguments to the global function, then takes what the global function returns and passes it along to whatever script called the helper. Speaking of which, your map script should now look like this:

(In your map script)

```
// Make sure to include the helper script file
#include "helpers/helper_somehandler"

void DoSomeWork(int alSomeInt, tString asSomeString)
{
    bool abResult = SomeModule_DoSomeWork(alSomeInt, asSomeString);

    // Do some nonsense with the result
}
```

Conclusion

That's pretty much it for the general explanation of modules. The rest is entirely dependent on what kind of module you want to write.