Script

Code style

The following is an overview of how the code should look like. For anybody working for Frictional Games, this is a must. For all else, see it as guidelines to have consistent code which is especially good if you want share your stuff.

Remember also to look at existing code and try and copy the looks.

Prefixes

System Hungarian notation is to be used for all variable names. These are the notations used:

Prefix	Description	Example
c	a class	class cMyClass {}
i	an interface	interface iMyInterface {}
f	a float or double value.	float fX;
l	any integer (char, int, uint, etc)	int lX;
p	an angel script handle or C++ pointer.	iPhysicsBody@ pBody; iPhysicsBody *pBody;
s	a string	cString sWord;
v	a vector or array	int[] vNumbers; cVector3f vPoint
mtx	a matrix	cMatrixf mtxTransform;
q	a quaternion	cQuaternion qRotation;
a	an argument	void Func(int alX) {

m	a member variable	class cMyClass {
		int mlMember;
		}

g	a global variable, defined outside of a class or function.	int glMyGlobal; class cMyClass {}
id	this is for tID type.	tID m_idEntity.

a variable name always starts with a lower case letter, so anything of type not specified must start with a lower case word. Example:

```
cMyOwnClass myClass;
```

If combining two prefixes, and one consist of more than one letter, then have an underscore between them. Example:

```
cMatrix m_mtxMemberMatrix;
```

Comments

Always comment the code you write; when scripting, it is always certain somebody else will work on it. Comment to make other people understand what it means and to make the code look nicer.

Function/Class comment
Use comment blocks for the main comment for each class/function, this so that the parser can show the correct info for code compleition/hints. Only for helpers and similar type of base functions where the information is needed.

```
/**
 * Change the light radius to the specified value.
 *
 * @param tString asLight, name of the light to change radius of.
 * @param float afRadius, the radius in meters to set for the light.
 **/
void LightChangeRadius(const tString &in asLight, float afRadius)
{

}
```

Section comment
When starting a new section of some kind do a comment that explains what is going on.

```
//////////////////////////////////////////
// Update the monster damage
cMonster@ pMonster = CurrentHit();
float fDamage = RandomDamage();
fDamage -= pMonster.Defense();
if(fDamage<) fDamage =;
fDamage *= mfHitMultiplier;
pMonster.DecHealth(fDamage);
```

Single line comment
When commenting a single line

```
//The square of the distance will be enough
float fDistSqr = vDiff.x*vDiff.x +vDiff.y*vDiff.y;
```

Only do this in rare cases when it is really hard to figure what the code means. Normally, simply using section comments is enough. If you split the code into nice sections and name them well, that should be more than enough for other people to figure out what it is about.

Function/Class separation
Use this to make it easier to see functions and classes. Can also be used to encapsulate other data.

```
void Foo()
{

}

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

```
void Bar()
{

}
```

Containers

There are some containers that are not part of the angelscript language, but added as extension, these will be described here.

Array

Arrays are simple linear storage containers.

You can access elements using *varname[index]*.

Syntax:

```
array<T> or T[]
```

Methods

```
void insertAt(uint, const T&in)
void removeAt(uint)
void insertLast(const T&in)
void insertBack(const T&in)
void removeFirst()
void removeLast()
uint length() const
void resize(uint)

void sortAsc()
void sortAsc(uint, uint)
void sortDesc()
void sortDesc(uint, uint)
void reverse()
int find(const T&in) const
int find(uint, const T&in) const

void push_back(const T&in)
void push_front(const T&in)
void pop_back()
void pop_front()
uint size() const
```

Included files

File can be included using the *#include* key word. This will look in all folders that have been declared in resources.cfg.

An important thing to think about is that any enums, classes, interfaces, etc that are meant to be shared, needs to be declared with the *shared* keyword. Example:

```
 shared enum eMyEnum {
shard interface iMyInterface {
```

```
If this is not done, then the data types will be declared as different types in all files that include the files, which will lead to exceptions, or worse crashes.
```

User Classes

Script user classes are C++ classes that implement the iScriptUpdateableUserClassInterface interface.

The script for these classes must contain the code:

```
cUserClass@ mBaseObj;
void SetupBaseInterface(cUserClass@ aObj){@mBaseObj = aObj;}
```

```
Where "cUserClass" is the name of the implemented user class.
```

These classes can choose to implement the method:

```
void Init()
```

```
This method is called upon creation of the class object, but __not__ on reload.
```

The constructor of the class is called on creation and on reload. However, note that the handle *mBaseObj* is not initialized when the constructor is called and hence may only be used in *Init()*.

Script Classes

Script classes are whatever class that is created in script.

Handles

For any script that is to be saved (which means all scripts pretty much), handles to script classes can

NOT be member variables. So a script file like:

```
class cMyClass {
 ...
}
class cMyOtherClass{
 ...
 cMyClass@ mpMyClass;
}
```

is NOT supported. The reason for this is that when a script is destroyed any handle to these classes are destroyed too and cannot be retrieved again!

It is however okay to to pass handles as arguments:

```
void MyFunc(cMyClass@ apArg) {
...
}
```

or use them as local or global variables.

```
cMyClass@ gpGlobalVar
void MyFunc() {
```

```
void MyFunc() {
 cMyClass@ pLocalVar
 ...
}
```

Its only when they are to be saved that problems arise.

ID Handles

When working with engine types it is recommended to use ID handles instead of class pointers.

```
class cMyClass {
 iPhyiscsBody@ mpBody
}
```

Using class handler directly like this works but it is unsafe and does not support saving. Instead the handle can be saved like this.

```
class cMyClass {
 tID mBody;
}
```

This saves a unqiue identifier to the body. The ID of a object can be retrived by calling GetID() function. There are script functions for converting the ID to a class handle.

```
void MyFunc() {

 iPhysicsBody@ pBody = cLux_ID_PhysicsBody(mBody);

 if(pBody !is null) {

  ...
 }

}
```

This retrives the class handle from the engine so that it can be used. Retriving the class handle from the ID is very fast.

Using a ID instead of a class handle has two big advantages. Unlike class handles the ID can be saved. When loading the save file the ID will still work and retrive the same object.

If the object that the ID handle points to gets deleted anywhere else in the code the ID will still be safe. When trying to retrive a deleted object the function will return null. Accessing null pointers will never cause the game to crash. When accessing a class handle that has been deleted that leads to accessing deallocated memory and the game will crash.

Script Callbacks

Calling script functions from scripts

If you want to run a script function/method from script using the Prepare, Execute, SetArg, etc you need to make sure that string arguments are NOT refences from C++ code. For example this is NOT allowed:

```
if(mObj.ScriptPrepare("void DoSomething(string &in asStr)") ){
SetArgString(, mObj.GetName() );
ScriptExecture();
}
```

Where mObj is of class cMyObj implemented in C++:

```
cMyObj {
 ...
 tString& GetName();
}
```

What is wrong here is that the argument is “string &in”, an in reference. The reason why this is bad is because when the string sent to the callback function is a reference and the string returned from C++ is also this. What happens when a reference is returned from script is that the Script makes a copy of the class and then provides a reference to this. In the example above, the variable returned from GetName() only has a scope lasting the function call. So when we arrive at ScriptExecute, the reference is gone.

To fix this you can do one of two things. One is to make sure that the argument for the called function is not a reference, like:

```
if(mObj.ScriptPrepare("void DoSomething(string in asStr)") ){
```

Or you can simply save the variable as a script one and then set that one, like:

```
tString sName = mObj.GetName()
SetArgString(, sName );
ScriptExecture();
```

In the last example, the variable (sName) is in scope until after ScriptExecute is called, and hence for the as long as needed.

Global Functions

Global functions make it easier to communicate between separate script modules. They are the slowest form of function calling but can also be very nice to use, just do not use it in performance sensitive functions (eg functions that get called many times in an update loop).

1) Before running the Global function, all the arguments need to be set up. Example:

```
cScript_SetGlobalArgBool(, true);
cScript_SetGlobalArgVector3f(1, cVector3f(1,2,3));
cScript_RunGlobalFunc("MyClassObj", "cMyClass", "MyGlobalFunc");
```

2) The return value must be gotten before any other global function is called, else it goes away. This is okay:

```
...
cScript_RunGlobalFunc("MyClassObj", "cMyClass", "MyGlobalFunc");
bool bReturnFromMyGlobalFunc = cScript_GetGlobalReturnBool()
cScript_RunGlobalFunc("MyClassObj", "cMyClass", "SomeOtherFunc");
```

```
This is NOT okay:
```

```
cScript_RunGlobalFunc("MyClassObj", "cMyClass", "MyGlobalFunc");
cScript_RunGlobalFunc("MyClassObj", "cMyClass", "SomeOtherFunc");
bool bReturnFromMyGlobalFunc = cScript_GetGlobalReturnBool()
```

(it would be getting the return value from *SomeOtherFunc*.)

3) When creating a function that is to be global, it must have void as return value and no arguments. Example:

```
void MyGlobalFunc()
```

4) The implemented global function must get all arguments using

```
void MyGlobalFunc() {
bool bArg0 = cScript_GetGlobalArgBool(0);
cVector3f vArg1 = cScript_GetGlobalArgVector3f(1);
```

5) When calling a global function, it is okay to have one or more asterix in the object name and then several object will be called. For example *"My*test"* would call: *"MyNiceTest"* and *"MyBadTest"*. However, this will be a bit slower, so be careful when using it (eg not during things called every update).

Asterix is also supported for the class name (eg "cMyClass*"). If you do not care about the class of the objects the class param can simply be empty (_ckgedit_QUOTckgedit>). The following will call MyGlobalFunc in all script modules named "MyClassObj" no matter the class name: cScript_RunGlobalFunc(GESHI_QUOTMyClassObjGESHI_QUOT, GESHI_QUOTGESHI_QUOT, GESHI_QUOTMyGlobalFuncGESHI_QUOT); ===== Saving ===== To skip saving a variable, you must use metadata keyword [nosave] in front of the the variable name. Examples: [nosave] float mfT; [nosave] cMyClass@ mHandle; If you only want to save the actual pointer for a handle, but not any data, then use the prefix [nodatasave] Example: [nodatasave] cMyClass@ mNoSaveHandle; However it is almost always better to use the tID type for these situations! Also, there is a the prefix [volatile], this will do the same as a [nosave] but will also make sure that variable is not even saved during script reload. This should be used on handles where you are never sure if saved variable can invalid, for instance cSoundEntity and cParticleSystem (that can be deleted by the engine). [volatile] cSoundEnity@ mpCurrentSoundEntity; Instead of using class handle here it is better to store the ID to the object. The ID can always be saved and will work correctly on load even if the object no longer exists. Worst thing that can happen is that when getting the actual class NULL is returned. This leads to an excpection, stopping further execution of the current script file, but does not lead to a crash or similar major failure. tID mCurrentSoundEntity; In case you have an array of handles where none of the properties should be saved, this is NOT possible. Instead save names of IDs of the objects in the array. ===== Optimizing ===== All of the optimizations tips are meant to be used when speed is of essence. It is good to always use them when possible, but it is even more important to have the code readable, so take that into account too! **Declare Local Variables Outside Loops**
Try and keep the local variables outside a loop. So instead of doing: for(int i=0; iGESHI_OPEN1000; ++i) { float fX = GetSomeThing(i); … } Do it like this instead: float fX; for(int i=0; iGESHI_OPEN__1000; ++i) { fX = GetSomeThing(i); … } That way they script does not need to create the local var for every loop and this can boost performance quite a bit!

Include only what is needed
Make sure that you only include hps files that are really needed. Having includes that are not used can eat up a lot of extra loading time very easily!

Important notes

C++ scriptable classes with classes in can not be abstract ones. It must always be the top class in the hierarchy that is used and saved.

Specific Guidelines

Helper Functions

- Helper functions should use degrees, not radians