Lecture7: Python Scripting

References:

https://docs.python.org/3.8/c-api/index.html

https://docs.python.org/3/extending/index.html

Quick Tangent: Pre-Compiled Headers

- Why they're (sometimes) necessary
  - If your project has many static headers (Ogre.h, Python.h, etc.)
  - Build times are SLOW
- How they are set up:
  - Create a common header (stdafx.h, pch.h, often)
  - Put all common includes in there
  - Include stadafx.h as the first include in every .h and .cpp file you create
    - Make a stdafx.cpp that *just* includes stdafx.h
  - VC++ settings:
    - Properties=>C/C++=>Pre-compiled headers (change to "Using" – verify it's using your stdafx.h)
    - Right-click on just stdafx.cpp and set the same FILE property to "Create"
- The first build will be slow, subsequent builds should be modestly to dramatically faster
  - Rebuild's will remake the pch file and should be avoided if possible

python GameObject: Overview

- Create a python GameObject class
  - Implemented in C++
  - Wraps select methods of C++ GameObject class
  - Will look like a Python class to the Python User
- The python user…
  - calls a create_game_object function
    - can be used to create a “vanilla” Python GO
      - Has only methods exposed by the Python GO class
    - can be used to create an “enhanced” Python GO
      - Is created from a user class that’s derived from Python GO
      - The user class can define “special” methods that the C++ game engine will call
        - create
        - update
  - Python and C++ GameObject <u>instances</u> know about each other
    - C++ GameObject has a pointer to a PyObject (it’s “twin” in python)
      - NULL if this object is not “script aware”
    - Python object has a pointer to the C++ GameObject
      - The actual GameObject is owned by the GOM
      - Python has a PyCapsule object we’ll use to store the pointer

Creating a Python class in C

- (in script_game_object.h)
- Step1: Make a C structure to hold any attributes our python-GameObject needs
  - When one of the methods is called, you will be passed a PyObject* for self.
  - This can be safely cast as one of our ssuge::script::GameObject*'s
  - Which can be used to get the C++ "twin" of this python game object

```
#include <game_object.h>      // Contains ssuge::GameObject - the normal c++ one
namespace ssuge
{
   namespace script
   {
       typedef struct
       {
            PyObject_HEAD                        // C-style "inheritance"
            ssuge::GameObject* mTwin;            // The c++ back-end for this game object - does all the real work
       } GameObject;          // <= see the need for the nested namespace?
   }
}
```

Creating a Python class in C, cont.

- (in script_game_object.cpp)
- Step2: The Type structure (aka “the big giant struct”)
  - Anywhere a PyTypeObject* is required, pass the address of this thing
  - PyTypeObject*’s can safely be cast as a PyObject*
    - b/c the beginning part of the struct contains a PyObject – this is hacky C-style “inheritance”

[Look at the file from blackboard]

Creating a Python class in C, cont.

- (in script_manager.cpp, the PyInit function)
- Step3: Add our GameObject type to the ssuge module
  - In your PyInit_whatever function, after the module's been created.
  - Call PyTypeReady on the GameObject type structure
    - Inherits "special" fields that we didn't define from python's object class
  - Call PyModule_AddObject to add "GameObject" to the python ssuge module
    - Call INCREF on the Type before this (b/c AddObject steals the only reference)

Creating a Python class in C, cont.

- Step4: Add “non-special” methods to GameObject
  - See the note in ssuge_game_object_methods.h/cpp

Creating a Python class in C, cont.

- Step5: create_game_object(group_name, object_name, tag, class="")
  - a new top-level function (script_functions.h/cpp)
  - responsible for creating the python AND c++ game objects
    - and connecting them to each other
    - and calling a "create" method (later)
  - Use PyObject_Call on…
    - GameObjectType (the big-giant-structure), cast to PyObject*
    - A GameObject-derived class (later)
  - Return the result, which should be an instance of a GameObject(-derived) class

Creating a Python class in C, cont.

- Step6: Allow custom types
  - This is the icing on the delicious python/ssuge wedding cake!
  - We want the (python) user to be able to be able to create a class like this:

```
class MyAwesomeClass(ssuge.GameObject):
    def create(self, *args):
        # do stuff to set up this game object
        # args is a tuple passed from create_game_object
    def update(self, dt):
        # Called every frame by C++’s GameObject::update function
```

x

Creating a Python class in C, cont.

- Step6, cont.
  - There are two places the user can create a class like this:
    - In a separate file.
      - While running ScriptManager::run_script, you can…
      - …iterate through the module's directory (see PyModule_GetDict and PyDict_Next)…
      - …if one of the objects passes a PyType_Check, it's a class – I cached these away in c++ map
    - In the same file they want to use it
      - You can use PyEval_GetGlobals and PyDict_GetItem to see if we have that class
  - [See the python test script(s) for an example of both]