1. Introduction

This is the last exercise which is based on the *LuaCmdGame* framework. The goal is to bind two simple C++ classes to Lua, so that they can be accessed and used within the script code.

First, open the solution *lua.sln* with Visual Studio 2012 and make sure that *LuaCmdGame* is set as startup project. Also make sure that *OOTest.lua* is set in the field *Command Arguments* in the project's properties. This will cause the program to load and run the script *OOTest.lua* when you execute it in Visual Studio.

In the source file *LuaCmdGame.cpp*, you can find a function named `Exercise_03_Lua_OO_UnitTest(...)`, which is called before the actual game starts up. It runs a couple of unit tests which help you to make sure that the bindings you implement work correctly. The game only starts if all tests passed successfully.

2. Binding the Value Type Class 'Vec2'

The source filed *Vec2.h* and *Vec2.cpp* define a very simple `struct` which represents a two-dimensional vector. It has two `public` members `m_x` and `m_y` of the type `float`, can calculate its length, and has two overloaded operators for addition and substraction. These functions are fully implemented and functional.

There are a couple of `TODOs` in the file *Vec2.cpp*, which indicate where you will have to place your own code. The `TODOs` also explain what exactly your code is supposed to do. However, if you want to add additional member functions to the `struct`, feel free to edit the respective header file as well.

a) The first function you should implement is `lua_bind(...)`. It is called once at the beginning of the unit test and its purpose is to create and register the needed Lua meta table.

b) Now finish the code of the function `lua_CFunction_Vec2(...)`. It serves as a constructor for the `Vec2 struct` inside the Lua script code.

c) Implement the remaining `lua_CFunctions`. These are the member functions that are stored in the meta table. When you are done, all the unit tests for `Vec2` should pass without any error.

Hint 1: Make sure to run the unit tests frequently to test your code. They will give you respective messages with detailed error descriptions.

Hint 2: You can take a look into the file *OOTest.lua* in order to see how the bindings are supposed to work. The respective test functions are called `test_Vec2()` and `test_Player()`.

Hint 3: You should consider to write two member functions which can push and pop a `Vec2` instance to the Lua stack, respectively. Otherwise, you might end up with a considerable amount of redundant source code.

3. Binding the Reference Type Class 'Player'

The code of the `class Player` can be found in the files *Player.h* and *Player.cpp*. The `class` is a simple *Singleton*, which means that it has only one global instance. The respective pointer can be obtained by calling `Player::getInstance()`. The `class` has get/set functions for a name of the type `std::string` and a position of the type `Vec2`.

The file *Player.cpp* also contains a bunch of `TODOs`, which indicate where you will have to place your own code and what it is supposed to do. Again, if you want to add additional member functions to the `class`, feel free to edit the respective header file as well.

a) Implement the function `lua_bind(...)`. It is called once at the beginning of the unit test and its purpose is to create and register the needed Lua meta table. In addition, it should register a global `Player` instance to Lua.

b) Write the missing code of the set/get name functions. Their implementation is straight forward since you can simply utilize the Lua type `string`. However, make sure that you properly reconstruct the `Player` pointer from the implicitly passed table `self`. After all, we don't want the binding code to fail if we later introduce additional players...

c) Implement the functions set/get position. You will have to pop and push `Vec2` instances using your own binding code from exercise 2 for this.

Hint 1: You will have to store a hidden field of the type `lightuserdata` in the `Player` Lua table, which stores the respective pointer address. You could call it `__ptr`, for example.

When you are done, all the unit tests should pass and you will be able to play the extraordinarily amazing “Move the P around” game. ;-)

4. Bonus Exercise

Create an object oriented version of your own *LuaCmdGame* script, which uses several Lua-bound C++ classes to represent its structures and objects.