acrionlua

Presentation on October 10, 2022
who we are

Stefan Zipproth
Managing director
✉️ s.zipproth-at-acrion.ch

Alexander Braun
Managing director
✉️ a.braun-at-acrion.ch
personal background

• studied computer science at Technical University of Munich
• freelancer since 1999, worked for 17 large companies in Germany and Switzerland, mainly embedded systems in C++
• own products:
  - Straton
    image segmentation used by astro photographers
  - jetpix
    image compression technology
  - Aristarch
    chess engine
previous product

GUI (Windows, C#, Window Forms)

.NET Assembly (C++)
new product

GUI (multi platform, C++)

shared library 1

shared library 2

old module

new commercial module
better use generic plugin interface?

GUI (multi platform, C++)

plugin manager

plugins (multi platform, C++)

refactoring: move image I/O of GUI into 3rd plugin

shared libraries implementing plugin interface
better use generic plugin interface?

GUI (multi platform, C++)

plugin manager

plugins (multi platform, C++)

What's the benefit?
What's the requirement?

refactoring: move image I/O of GUI into 3rd plugin

shared libraries implementing plugin interface
make users want batch processing

GUI (multi platform, C++)

command line interface (multi platform, C++)

plugin manager

plugins (multi platform, C++)

cool refactoring suddenly becomes essential
first task of the plugin manager

- GUI (multi platform, C++)
- command line interface (multi platform, C++)
- plugins (multi platform, C++)

- fetch list of available plugins from github repo

centralized
we also want scripting

GUI (multi platform, C++)

command line interface (multi platform, C++)

plugin manager

scripting

plugins (multi platform, C++)

the problem: how to use functions from shared libraries?

- fetch list of available plugins from github repo
- provide scripting functions to load arbitrary shared libraries
let’s use Lua!

GUI (multi platform, C++)

Lua command line interface

acrionlua static library

Lua plugin

shared libraries (multi platform, C++)

- manage plugins
- fetch list of available plugins from github repo
- provide Lua functions to load arbitrary shared libraries

new function
import()
make pure Lua plugins possible

GUI (multi platform, C++)

Lua command line interface

acrionlua static library

Lua plugin

optional shared libraries (multi platform, C++)

- manage plugins
- fetch list of available plugins from github repo
- provide Lua functions to load arbitrary shared libraries
- provide Lua functions to read and write from light user data

Stefan Zipproth
take care of a common usability problem

GUI (multi platform, C++)

Lua command line interface

acrionlua static library

Lua plugin

optional shared libraries (multi platform, C++)

- manage plugins
- fetch list of available plugins from github repo
- provide Lua functions to load arbitrary shared libraries
- provide Lua functions to read and write from light user data
- send asynchronous messages

do Lua co-routines help?
take care of a common usability problem

GUI (multi platform, C++)

Lua command line interface

acriOnlua static library

Lua plugin

optional
shared libraries (multi platform, C++)

- manage plugins
- fetch list of available plugins from github repo
- provide Lua functions to load arbitrary shared libraries
- provide Lua functions to read and write from light user data
- send asynchronous messages

Stefan Zipproth
Jenkins

3 jobs to build ImageMagick

3 jobs to build plugin 1 (acrion image tools)

3 jobs to publish acrion image tools

3 jobs to publish meta data of acrion image tools

3 jobs to build plugin 2 (acrion imago)

3 jobs to publish acrion imago

3 jobs to publish meta data of acrion imago

3 jobs to build acrionlua lib + exe

3 jobs to build acrionphoto + installer

3 jobs to publish acrionphoto installer
Lua plugin interface

```lua
local correctionTool = {}
correctionTool.name = "Correction"
correctionTool.description = "Reverse the distortion based on the results of the
 correctionTool.icon = "Correction.svg"
correctionTool.parameters = {
    workingImage = { type = "void" },
    referenceImage = { type = "void" },
    width = { type = "long long" },
    height = { type = "long long" },
    channels = { type = "long long" },
    Intensity = { type = "double", default = "0.2", minimum = "0.1", max = "1" },
    InterpolationModel = { type = "enum", default = "Center And " },
    SectionSize = { type = "long long", default = "480", min = "1" },
    DetectionsPerSectionForContinuousInterpolation = { type = "long long", def = "480" },
    SmoothingForContinuousInterpolation = { type = "long long", default = "3" },
    SectionRangeForContinuousInterpolation = { type = "long long", def = "480" },
    DetectionRadius = { type = "long long", default = "48", min = "1" },
    BrightnessDiscontinuityThreshold = { type = "long long", default = "57600", min = "1" },
    BrightnessPercentageLimit = { type = "double", default = "0.7", min = "0.3", max = "1" },
    BrightnessRequired = { type = "double", default = "4.0", min = "0.0", max = "10.0" },
    LocalInterpolation = { type = "double", default = "2.0", min = "0.0", max = "100.0" },
}
ConsiderEdges = { type = "enum", default = "yes", values = {
    ["0"] = "do not preserve textures",
    ["1"] = "simple",
    ["2"] = "use average, no diagonals",
    ["3"] = "use average, use diagonals",
    ["4"] = "use minimum, no diagonals",
    ["5"] = "use minimum, use diagonals", internal = "yes"},
ConsiderCenter = { type = "enum", default = "no", values = {
    ["0"] = "do not preserve textures",
    ["1"] = "simple",
    ["2"] = "use average, no diagonals",
    ["3"] = "use average, use diagonals",
    ["4"] = "use minimum, no diagonals",
    ["5"] = "use minimum, use diagonals", internal = "yes"},
ConsiderTopLeftCorner = { type = "enum", default = "yes", values = {
    ["0"] = "do not preserve textures",
    ["1"] = "simple",
    ["2"] = "use average, no diagonals",
    ["3"] = "use average, use diagonals",
    ["4"] = "use minimum, no diagonals",
    ["5"] = "use minimum, use diagonals", internal = "yes"},
ConsiderTopRightCorner = { type = "enum", default = "yes", values = {
    ["0"] = "do not preserve textures",
    ["1"] = "simple",
    ["2"] = "use average, no diagonals",
    ["3"] = "use average, use diagonals",
    ["4"] = "use minimum, no diagonals",
    ["5"] = "use minimum, use diagonals", internal = "yes"},
ConsiderBottomLeftCorner = { type = "enum", default = "yes", values = {
    ["0"] = "do not preserve textures",
    ["1"] = "simple",
    ["2"] = "use average, no diagonals",
    ["3"] = "use average, use diagonals",
    ["4"] = "use minimum, no diagonals",
    ["5"] = "use minimum, use diagonals", internal = "yes"},
ConsiderBottomRightCorner = { type = "enum", default = "yes", values = {
    ["0"] = "do not preserve textures",
    ["1"] = "simple",
    ["2"] = "use average, no diagonals",
    ["3"] = "use average, use diagonals",
    ["4"] = "use minimum, no diagonals",
    ["5"] = "use minimum, use diagonals", internal = "yes"},
Smoothing = { type = "double", default = "0.2", min = "0.0", max = "10.0" },
PreserveTexturesDeprecated = { type = "enum", default = "yes", values = {
    ["0"] = "do not preserve textures",
    ["1"] = "simple",
    ["2"] = "use average, no diagonals",
    ["3"] = "use average, use diagonals",
    ["4"] = "use minimum, no diagonals",
    ["5"] = "use minimum, use diagonals", internal = "yes"},
EnhanceDetails = { type = "enum", default = "no", values = {
    ["0"] = "do not preserve textures",
    ["1"] = "simple",
    ["2"] = "use average, no diagonals",
    ["3"] = "use average, use diagonals",
    ["4"] = "use minimum, no diagonals",
    ["5"] = "use minimum, use diagonals", internal = "yes"},
}```
showing plugin meta data in a GUI

generic widgets based on Lua code (plugin description)
new Lua function import

```lua
function CallSubtractLeftRightWrap(parameters)
  import("acrion_image_tools", "SubtractWorkingImageFromReference", "long long(void"

  local result = SubtractWorkingImageFromReference(touserdata(parameters.workingImag

  if result==0 then
    return "", 0
  else
    return "SubtractWorkingImageFromReference: error '" .. result .. "'", result
  end

end
```

Lua plugin calls import

C++ function Import
- loads the shared library via `boost::dll_shared_library`
new Lua function import

```lua
function CallSubtractLeftRightWrap(parameters)
  import("acrion_image_tools", "SubtractWorkingImageFromReference", "long long(void"
  local result = SubtractWorkingImageFromReference(touserdata(parameters.workingImage

  if result==0 then
    return "", 0
  else
    return "SubtractWorkingImageFromReference: error '" .. result .. "'", result
  end
end
```

Lua plugin calls `import`

C++ function Import
- loads the shared library via `boost::dll_shared_library`
- stores the signature of the new Lua function + reference to the shared library in a C++ map using the name of the new Lua function as key

StoreImportedFunction(s);
new Lua function import

```lua
function CallSubtractLeftRightWrap(parameters)
    import("acrion_image_tools", "SubtractWorkingImageFromReference", "long long(void"

    local result = SubtractWorkingImageFromReference(touserdata(parameters.workingImage

    if result == 0 then
        return "", 0
    else
        return "SubtractWorkingImageFromReference: error '\".. result .. ''": result
    end
end
```

**Lua plugin calls import**

**C++ function Import**
- loads the shared library via `boost::dll_shared_library`
- stores the signature of the new Lua function + reference to the shared library in a C++ map using the name of the new Lua function as key
- registers a new Lua function that calls the C++ function `CallDllFunction`

**all imported functions call the same C++ function**

```lua
StoreImportedFunction(s);
lua_pushcfunction(L, CallDllFunction);
lua_setglobal(L, s.functionName.c_str());
```
Calling the imported function

- CallDllFunction gets the name of the calling Lua function via lua_getinfo.
- In the map it finds the required data
- It searches the signature... through a code-generated chain of if clauses, e.g.

```cpp
1 if (signature=="void(long long,long long,long long,bool,std::string)")
2 {
3 boost::dll::import<void(long long,long long,long long,bool,std::string)>({dll
4 (lua_tointeger(l,1),
5 lua_tointeger(l,2),
6 lua_tointeger(l,3),
7 lua_tobool(l,4),
8 lua_tostring(l,5));
9 return;
10 }
```

lua_find_signature_if_chain.cpp (3497 lines, 2 MB)
Calling the imported function

- CallDllFunction gets the name of the calling Lua function via lua_getinfo.
- In the map it finds the required data
- It searches the signature...

... in a C++ std::map of std::function instances that are pre-initialized like e.g....

... through a code-generated chain of if clauses, e.g.

... and called as follows ...

lua_find_signature_if_chain.cpp (3497 lines, 2 MB)
what’s possible?

```lua
local cpp_argument_types = {
  {max_sequence=2, types="void"},
  {max_sequence=6, types="long long"}, -- needs to match lua_Integer, see lua.h
  {max_sequence=3, types="double"},
  {max_sequence=3, types="bool"},
  {max_sequence=1, types="std::string"}
}
local cpp_argument_type_list = concat_tables(cpp_argument_types[1].types, concat_table(cpp_argument_types))
local cpp_return_types = table.move(cpp_argument_type_list, 1, #cpp_argument_type_list)
local n_use_map = 4
local max_arguments = 15
```

Plugin developers may use wither

1. only Lua
2. Lua with own C++ libraries

Accessing 3rd party C++ libraries

1. is possible in many cases
2. can be made possible easily

Stefan Zipproth
acrion.ch
github.com/acrion
multithreading

- used in commercial product
- prevents common multithreading issues

The reason for using single threading lies solely in the design of the programming languages
what's required to make clime available in Lua?

care about Lua state
what's required to make clime available in Lua?

```
namespace ACRIONLUA
{
    struct Table;

typedef std::map<std::string, std::string> StringTable;

typedef std::map<std::string, Table>    SubTables;

struct ACRIONLUA_LIBRARY_EXPORT Table  // We call a "Table" the composition of a table
{
    StringTable data;
    SubTables  subTables;
};

class ACRIONLUA_LIBRARY_EXPORT MessageFromLua
{
    public:
    std::string name;
    Table    parameters;
};

using MessageManagerType = clime::message_manager<MessageToLua, MessageFromLua>;

extern ACRIONLUA_LIBRARY_EXPORT std::string shortenClassName(const std::string& className);

extern ACRIONLUA_LIBRARY_EXPORT int    demangling_status;

#define CLASS_NAME::ACRIONLUA::shortenClassName(__DEMANGLED_CLASS_NAME(ACRIONLUA::demangling_namespace, className))

// convenience macros to make sending messages more readable
#define SEND_TO_LUA(name, parameters) ::ACRIONLUA::LuaThreadPool::Get().SendMessage(std::string(name), parameters)
```
What's required to make clime available in Lua?

Register function to handle a message.

```cpp
class LuaThread
{
public:
  LuaThread(std::filesystem::path luaFilePath, MessageManagerType& messageManager)
  : lua luaFilePath
  {
    messageManager.add_handler<MessageToLua>(
      [this](std::shared_ptr<MessageToLua> messageToLua)
      {
        handleMessageToLua(messageToLua);
      },
      [this](const std::exception& ex)
      {
        handleExceptionInLua(ex);
      },
      [this]()
      {
        handleLuaIsIdle();
      });
  }
};
```
suggestions on how to design the new messaging functions in acrionlua?
new Lua functions

library acrionlua

executable acrionlua

plugin manager with meta data on github

clime C++ Light Message passing library

import sendmessage

Stefan Zipproth