Engine Scripting with Lua

Lion Kortlepel `@lionkor`

July 12, 2020

Contents

Abstract

This engine lets you script behavior in Lua. In order for this to be useful, the engine provides a number of functions and globals, which are documented in this PDF. Further, it will show how to use Lua to script some basic behavior by providing some examples.

1 Setup

For scripts to be read by the engine, an `Entity` needs to have a `ScriptableComponent`. The constructor of the latter accepts a scriptfile name, like `example.lua`. For the file to be loaded it needs to be in the `Data/` directory and listed in the `Data/res.list` of your project.

An example program follows that will be used for the rest of this pdf.

```cpp
#include "Engine.h"
int main() {
    // Create an application
    Application app("Scripting101 Program", { 800, 600 });
    // Create an entity
    WeakPtr<Entity> entity_weak = app.add_entity();
    // Lock the Ptr for temporary access
    auto entity = entity_weak.lock();
    // Add ScriptableComponent
    entity->add_component<ScriptableComponent>("example.lua");
    // Run the application
    return app.run();
}
```

example.cpp

Additionally, the files `Data/example.lua` and `Data/res.list` exist.

```
Engine.log_info("Hello, World!")
```

Data/example.lua

```
example.lua
```

Data/res.list

With these files in place and the `example.cpp` compiled, we can now write code in `Data/example.lua`.

2 Exposed Functions

2.1 Engine API

The `Engine` namespace provides general engine functionality, mostly used for debugging and core engine functionalities.

2.1.1 Engine.log_info(message)

Description
Prints an info message to the debug output.

Arguments

1. `message - string`
   The message to be printed.

Returns
Nothing

2.1.2 Engine.log_warning(message)

Description
Prints a yellow warning message to the debug output.

Arguments

1. `message - string`
   The message to be printed.

Returns
Nothing

2.1.3 Engine.log_error(message)

Description
Prints a red error message to the debug output.

Arguments

1. `message - string`
   The message to be printed.

Returns
Nothing

3 Constants

The engine exposes several constant values and tables to all scripts. These are read-only.

3.1 Tables

3.1.1 MouseButton

Description
A table which holds the integer values used in identifying mouse buttons in mouse-event related callbacks. The actual values are implementation defined and may change.

Entries

- `MouseButton.LMB` - Left mouse button integer equivalent
- `MouseButton.RMB` - Right mouse button integer equivalent
- `MouseButton.MMB` - Middle mouse button integer equivalent
- `MouseButton.XB1` - Extra mouse button 1 integer equivalent
- `MouseButton.XB2` - Extra mouse button 2 integer equivalent

3.2 Values

Please note that values with "unfriendly" names such as `g_parent` are only to be used internally, but are documented here to provide a single and complete reference.

3.2.1 g_scriptfile_name

Description
The full name of the current script file. Used automatically by the engine in calls to `Engine.log_*` and similar function families.

3.2.2 g_parent

Description
The address of the parent `Entity`, as `std::uintptr_t`. Used internally in the implementation of parent-modifying functions. May be used to compare entities in a really crude way.

4 Special Lua Functions

The engine calls specific lua functions (if they exist), at specific points in time or when events occur. The following is a listing of all of those special functions. If they do not exist, a warning is printed into the debug output once and any further attempts at calling the function will not occur.

4.1 Periodically Called Functions

These functions are called periodically. It is advised not to put any heavy calculations in any of these, unless absolutely unavoidable. Any "power-hungry" code in these functions will cause a slowdown.

4.1.1 update()

Description
Called every frame from the moment the `ScriptableComponent` is created until it or the `Entity` is destroyed.

Arguments
None

Returns
Nothing

Example

```
function update()
   -- do things here
end
```

4.2 Event Callbacks

These functions are called when a specific event occurs, for example a mouse click.

4.2.1 on_mouse_down(mouse_button, x, y)

Description
Called when the user presses any mouse button.

Arguments

1. `mouse_button - integer`
   The mouse button that has been pressed. The buttons are represented as integers, but the engine provides global `MouseButton` table to compare against:
   - `MouseButton.LMB` - The left mouse button
   - `MouseButton.RMB` - The right mouse button
   - `MouseButton.MMB` - The middle mouse button
   - `MouseButton.XB1` - The extra button 1 (only exists on some mice)
   - `MouseButton.XB2` - The extra button 1 (only exists on some mice)
2. `x - number`
   The x-position of the mouse in the world.
3. `y - number`
   The y-position of the mouse in the world.

Returns
Nothing

Example
This example prints "`lmb left mouse pressed!`" in the debug output whenever the user presses the left mouse button.

```
function on_mouse_down(mb, x, y)
    if mb == MouseButton.LMB then
        Engine.log_info("left mouse pressed!")
    end
end
```