LuaController: Integrando interpretação de Lua ao Godot para permitir criação de jogos de programação

Nome e Matrícula: **Rodrigo Leite da Silva 2021593**
Professor Orientador: **Edward Hermann Haeusler**

Resumo

Para permitir que eu use a escrita de programas em um jogo educativo de programação no meu projeto do mestrado preciso que um *game-engine* forneça uma classe/gui de *input* de texto e interpretação do mesmo como alguma linguagem de programação. Proponho, então, este projeto para desenvolver um módulo para o *game-engine* de código aberto **Godot** (escrito em c++) que oferece: uma interface de programação em **Lua** para as jogadoras escreverem programas; e, à desenvolvedora, controle sobre a execução desse código, cadastro de funções para permitir controle de elementos do jogo.

Vocabulário

Descrever este trabalho feito para o *game-engine* pode ser confuso porque há várias camadas de usuário, desenvolvedor e código que influenciam umas às outras. Para deixar claro a quem me refiro a todo momento, este é o vocabulário usado:

Módulo: O módulo para Godot desenvolvido neste trabalho, chamado **LuaController**. Oferece uma ferramenta nova para o **editor**.

Editor: A janela principal de edição do motor de jogos (*game*-engine) Godot. A janela que tem ferramentas para **Usuária** fazer um **jogo**.

Jogo: O produto final em execução, compilado pelo **editor** e abstraído dos códigos que o **desenvolvedor** e a **usuária** enxergam e alteram.

Desenvolvedor: do próprio *game-engine* Godot e seus módulos, aquele que escreve código em C++ para prover funcionalidades para **usuária** usar dentro do **editor**. É meu papel como desenvolvedor do **módulo**.

Usuária: Aquela que desenvolve o **jogo**. Que usufrui das funcionalidades do **editor** para criar o **jogo**.

Jogadora: Aquela que joga o **jogo** final.

Documentação

Especificação do Programa

Escopo

O objetivo é que uma cadeia de caracteres escrita pela **jogadora** dentro do **jogo** possa fazer alterações na cena do **jogo**. Para isso meu programa precisa: interpretar a cadeia como um *script* em uma linguagem de programação; precisa oferecer à **usuária** uma forma de escolher e definir as

alterações que o *script* faz no **jogo**; precisa ser integrado ao **editor** de jogos com interface gráfica e programática.

Por que Godot como motor de jogos?

A escolha de Godot (https://godotengine.org/) como o motor de jogos para qual desenvolveria meu programa se deu por: ser um programa de código aberto; estar sob a licença do MIT; e ser escrito em C++, algo que facilita a integração com Lua como a linguagem usada para os *scripts* da **jogadora**.

Godot é projetado para ser facilmente extensível. Ele integra com facilidade módulos externos às funcionalidades de seu **editor** e ao motor de jogos.

Por que Lua como a linguagem interpretada?

Precisava de uma linguagem de fácil integração em um programa maior, de sintaxe relativamente simples, extensível e portátil. Lua atende a todos esses requisitos, em parte graças a sua API C. Por essa API que escolhi um motor de jogos escrito em C++.

Por que LuaCpp como ponte entre Lua e o código fonte de Godot?

Decidi usar LuaCpp de jordanvrtanoski (https://github.com/jordanvrtanoski/luacpp), uma biblioteca de amarração entre Lua e C++, para poder focar no projeto do **módulo** em si. Essa biblioteca traz facilitadores para as tarefas mais repetitivas na API C: compilação e execução de código; transferência de valores através da Pilha em *lua_State*; e facilidade de criação de *userdata*.

Projeto do programa

O programa resultante é um módulo, no padrão Godot, chamado **LuaController**. É homônimo da classe principal do módulo porque esta é a única classe visível para a usuária.

Arquitetura

O módulo consiste em três classes principais (LuaController, LuaControllerContext e LuaCallable) e uma que auxilia na execução de testes unitários (LuaControllerUnitTester). Vide Figura 1 “LuaController_ClassDiagram.png”.

As classes englobadas por Godot são derivadas da classe *Node*, que traz muitas facilidades de integração ao **editor**, dependem fortemente das classes e tipos do *game-engine* e fazem cadastros de seus métodos e propriedades ao banco de dados de classes de Godot (ClassDB), para que sejam integrados e usados com facilidade pela **usuária**.

As classes englobadas em LuaCpp são encarregadas de fazer a abstração entre LuaController e a execução do *script* Lua. Suas implementações dependem fortemente dos tipos da biblioteca LuaCpp.

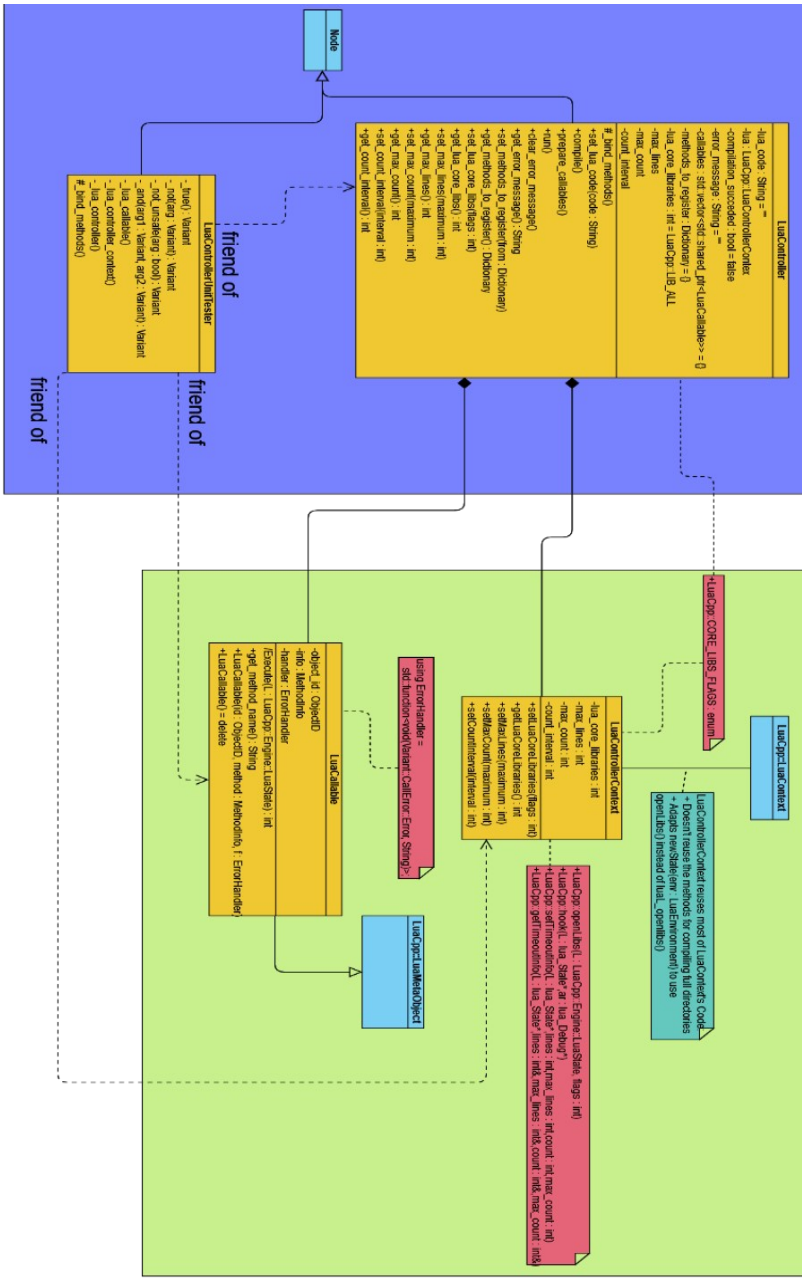

Figura 1 “LuaController_ClassDiagram.png”

Especificação das Classes do Módulo

LuaController

A única classe disponibilizada à **usuária** no **editor**. É derivada de *Node* para aproveitar a arquitetura de Godot (que favorece composição) o que, combinado com uma flexibilidade para definir métodos invocáveis de dentro de uma execução de Lua, permite que esta classe controle uma *Scene*.

Através desta classe a **usuária** pode: interpretar uma cadeia de caracteres como código Lua; definir métodos que podem ser chamados durante a execução do código, e o nome com que serão inseridos no ambiente global; receber códigos de erro referentes a erros de compilação e execução do código; definir quais bibliotecas padrão de Lua são abertas no ambiente onde o *script* da **jogadora** executa; e definir o método virtual "*lua_error_handler*".

Se a **usuária** define o método virtual "*lua_error_handler(call_error_code : int, message : String)*", este é chamado por LuaCallable se ocorre algum erro em seu método Execute(), durante a invocação do método cadastrado na LuaCallable em questão.

A classe oferece dois tipos de *timeout*: por quantidade de linhas executadas; e por quantidade de comandos contados. Como isso é implementado usando o mecanismo de Ganchos de Lua, o manual de lua5.3 pode esclarecer as especifidades (como o fato que a contagem para linha é incrementada **antes** de executá-la). Para que a **usuária** tenha controle disso, as propriedades *max_lines*, *max_count* e *count_interval* são oferecidas:

- *max_lines* é a quantidade máxima de linhas que a execução pode alcançar (a última não é executada). Deve ser pelo menos 0. Se definida como 0, não é feita a contagem nem pode ocorrer um *timeout* por contagem de linhas. *max_count* e *max_lines* não podem ser, simultaneamente, 0.
- *count_interval* define de quantos-em-quantos comandos executados incrementa o contador correspondente a *max_count*. Deve ser pelo menos 1. Se for mais que 1, um *timeout* por contagem de comandos quer dizer que a quantidade real de comandos executados foi: ***count_interval * max_count***.
- *max_count* é a quantidade máxima que o contador descrito em *count_interval* é incrementado. Deve ser pelo menos 0. Se definida como 0, não é feita a contagem nem pode ocorrer um *timeout* por contagem de comandos. *max_count* e *max_lines* não podem ser, simultaneamente, 0.

LuaContextController

Esta classe é, em sua maior parte, um clone de LuaContext, a classe de LuaCpp que gerencia a execução de código Lua. Precisei cloná-la pois precisava incrementá-la com as seguintes funcionalidades:

- Poder escolher quais bibliotecas padrão de Lua eu abro no contexto de execução. Para a **usuária** poder evitar que a **jogadora** tenha acesso à biblioteca do sistema operacional.
- Poder, eventualmente, definir um *timeout* para o código executado.

LuaCallable

Derivada de LuaCpp::LuaMetaObject, é a classe que permite que um método definido pela **usuária** no **editor** possa ser invocado pelo *script* Lua.

Na sua construção recebe as informações do método que irá invocar e o identificador do *Object* sobre o qual o método será invocado. Recebe, também, uma *callback* que corresponde ao método virtual “LuaController::lua_error_handler()”. Esta *callback* é invocada se ocorre erro durante a chamada do método.

Repassa sem alteração para o método Godot apenas argumentos do tipo: nil, string, number, boolean. Em teoria daria para converter uma *table* para *Dictionary*, mas esse e outros tipos são repassados como *nil*. Da mesma forma que Lua trata argumentos a mais ou a menos, LuaCallable ignora algum argumento se foi excedente, e argumentos que faltam são passados como *nil*.

Para valores retornados do método para Lua, o comportamento é o mesmo. Com um detalhe: métodos em Godot não podem retornar mais de um valor.

Importante notar que LuaCallable não verifica os tipos esperados pelo método que irá chamar, portanto o método passado deve ser seguros de chamar com argumentos de tipo variante. Isto é um erro de chamada do método portanto, se ocorrer, LuaCallable invocará a *callback* recebida em sua construção.

LuaControllerUnitTester

Derivada de *Node*, tanto para poder ser usada nos testes unitários no **editor** com GdUnit3, quanto para ser usada como o objeto invocado em testes de LuaCallable. Esta classe é amiga de todas do módulo para poder executar testes unitários invasivos.

Provê, na sua interface com Godot, métodos homônimos às classes (e.g. “lua_callable” e “lua_controller_context”) que retornam um *Array* de *String*, com cada cadeia sendo uma descrição detalhada do teste que falhou.

Projeto

LuaController é o ponto de entrada:

LuaController é implementado de forma que contenha todas as informações necessárias para tentar executar um *script* Lua e é o único responsável para avisar à **usuária** de qualquer erro que ocorra. Como os métodos para definir, compilar e executar código são feitas individualmente, pode ser complicado entender o estado que o módulo se encontra a qualquer momento. A Figura 2 “LuaController_States.png” evidencia tais estados e suas transições.

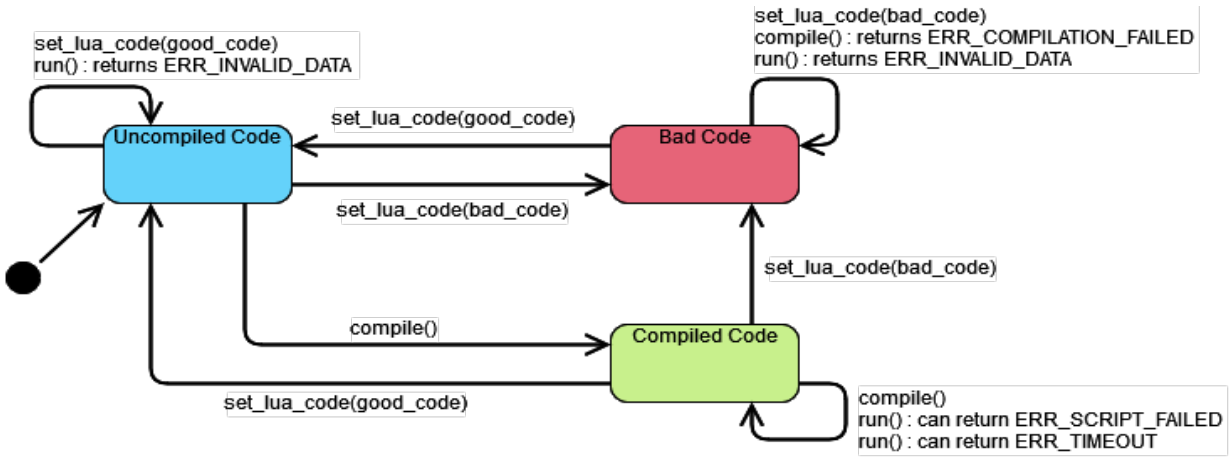

Figura 2 “LuaController_States.png”

Como a usuária dá controle do jogo para o script Lua:

O fluxo para ter código Lua executado está evidenciado na Figura 2. Entretanto, para que a execução gere alterações na cena, código de Godot, implementado pela **usuária**, precisa ser invocado de alguma forma no ambiente.

Chamadas aos métodos do objeto LuaController são usados para essa finalidade. Como a **usuária**, dentro do **editor,** tem total liberdade de definir e implementar novos métodos para o *Node* LuaController, o objeto cadastra seus métodos como funções no ambiente global do *script*. Dessa forma, qualquer alteração da cena já possível usando métodos, é possível com uma chamada a função no *script*.

Para a **usuária** poder escolher os métodos que quer cadastrar (um método que provavelmente não quer cadastrar é *queue_free*) e com quais nomes serão cadastrados, é definida a *Property* ***methods_to_register*** do objeto. É um *Dictionary*, cujas chaves e valores devem ser todas do tipo *String*. Para cada par *key*/*value* deste dicionário, *key* é o nome de um método de LuaController que será cadastrado e *value* é o nome da função que corresponderá à sua chamada.

Por fim, a **usuária** pode querer limitar quais bibliotecas padrão de Lua (https://www.lua.org/manual/5.3/manual.html#6) são abertas no ambiente (pode não querer dar à **jogadora** acesso ao sistema operacional). Para isso a *Property* ***lua_core_libraries*** é usada para indicar quais as bibliotecas devem ser abertas. Estão disponíveis para abrir: base, coroutine, table, io, os, string, utf8, math, debug e package.

Como o módulo comunica à usuária erros ocorridos

O **módulo** foi desenvolvido com o objetivo de ser usado em contexto educacional, onde erros de compilação e de execução da cadeia passada são **esperados e considerados comportamento normal do módulo**. Portanto:

- Assim como é a filosofia de Godot e Lua, erros identificados nunca resultam na interrupção do programa. LuaController sempre retorna um código de erro e guarda uma mensagem indicando qual o erro e se mantém em um estado útil, de forma que Godot possa continuar a execução normalmente.

- Fica a encargo da **usuária** reconhecer os erros que o modulo indica e tomar providências para que o **jogo** não quebre. Por exemplo: avisando a **jogadora** que algo deve ser corrigido; ou compilando a última versão do *script* feito pela **jogadora** que não teve erros.

O único erro que o módulo não se proteje e não trata é quando LuaCallable tenta acessar um ponteiro inválido para um objeto da classe LuaController. Isso porque, se for inválido, algo deu muito errado e a execução do **jogo** não deveria continuar, pois o LuaCallable que tentou acessar o ponteiro deveria ter sido destruído **com** o objeto que o ponteiro se referia.

Para comunicar erros durante a execução do **jogo** mas fora da execução do *script*, os métodos de LuaController *compile()* e *run()* retornam *Error* diferente de *OK*. Se isso ocorre, uma mensagem detalhada do erro que ocorreu é recuperável através do método *get_error_message*, e permanece válida até que outra chamada retorne um *Error* ou o método *clear_error_message* seja invocado.

Para capturar erros de chamadas aos métodos de LuaController que LuaCallable faz durante a execução do *script*, a **usuária** pode definir o método virtual de LuaController *lua_error_handler()*, que recebe um *Variant::CallError::Error* e uma *String* que é a mensagem montada através de *Variant::get_call_error_text()*.

Testes

Para fazer testes automatizados sobre o módulo optei por uma boa integração com o **editor**, para poder aproveitar projetos concretos em Godot e dar ênfase em testes de integração do módulo. **GdUnit3** (https://godotengine.org/asset-library/asset/867), uma ferramenta da *Asset Library* de Godot desenvolvida pela comunidade, foi a escolhida para a tarefa.

Com essa ferramenta, criei o projeto Godot “tester-project” para executar os testes. O projeto é distribuído junto com o **módulo** LuaController e, para executar os testes automatizados a partir da linha de comando, basta seguir as instruções no README.md do **módulo**. Um log dos testes formatado em html é salvo em “tester-project/reports”. Todos *scripts* de teste estão na pasta “tester-project/test”.

Testes Unitários

Os testes unitários são feitos por LuaControllerUnitTester. A classe foi implementada como *Node* por dois motivos:

- Para ser o ponto de contato entre o *framework* de GdUnit3 e as classes que Godot não pode alcançar.
- Para poder ser, durante os testes, o objeto cujos métodos são cadastrados em um ambiente Lua.

Cada método de LuaControllerUnitTester corresponde a uma classe testada. E cada mensagem de erro retornada é montada no formato: “[FAILURE] <nome_do_método> @ TEST <número_do_teste> ‘<nome_do_teste>’ : <descrição_da_assertiva_que_falhou>”. A chamada aos métodos de LuaControllerUnitTester estão no *script* “UnitTest.gd”.

Testes de Integração

Os testes de integração estão nos *scripts*: RealSceneTest.gd, StateBadCodeTest.gd, StateCompiledCodeTest.gd, StateTransitionsTest.gd, StateUncompiledCodeTest.gd.

Os testes em RealSceneTest.gd instanciam uma cena real (“tester-project/RealScene/ScriptRunner.tscn”), simulam interações que a **jogadora** faria e testa se o **módulo** se comportou como esperado.

Os *scripts* de nome “State*” focam em testar a validez dos estados do **módulo**, como estão descritos na Figura 2. Alguns testes aparecem em mais de um *script* porque os escopos dos testes de cada *script* tendem a ter interseção.