Godot script language

Gdscript

Hands-out

GdScript é a linguagem de programação usada na engine Godot!

GODOT

Visite o site oficial

Repositório Flappy

Hands out

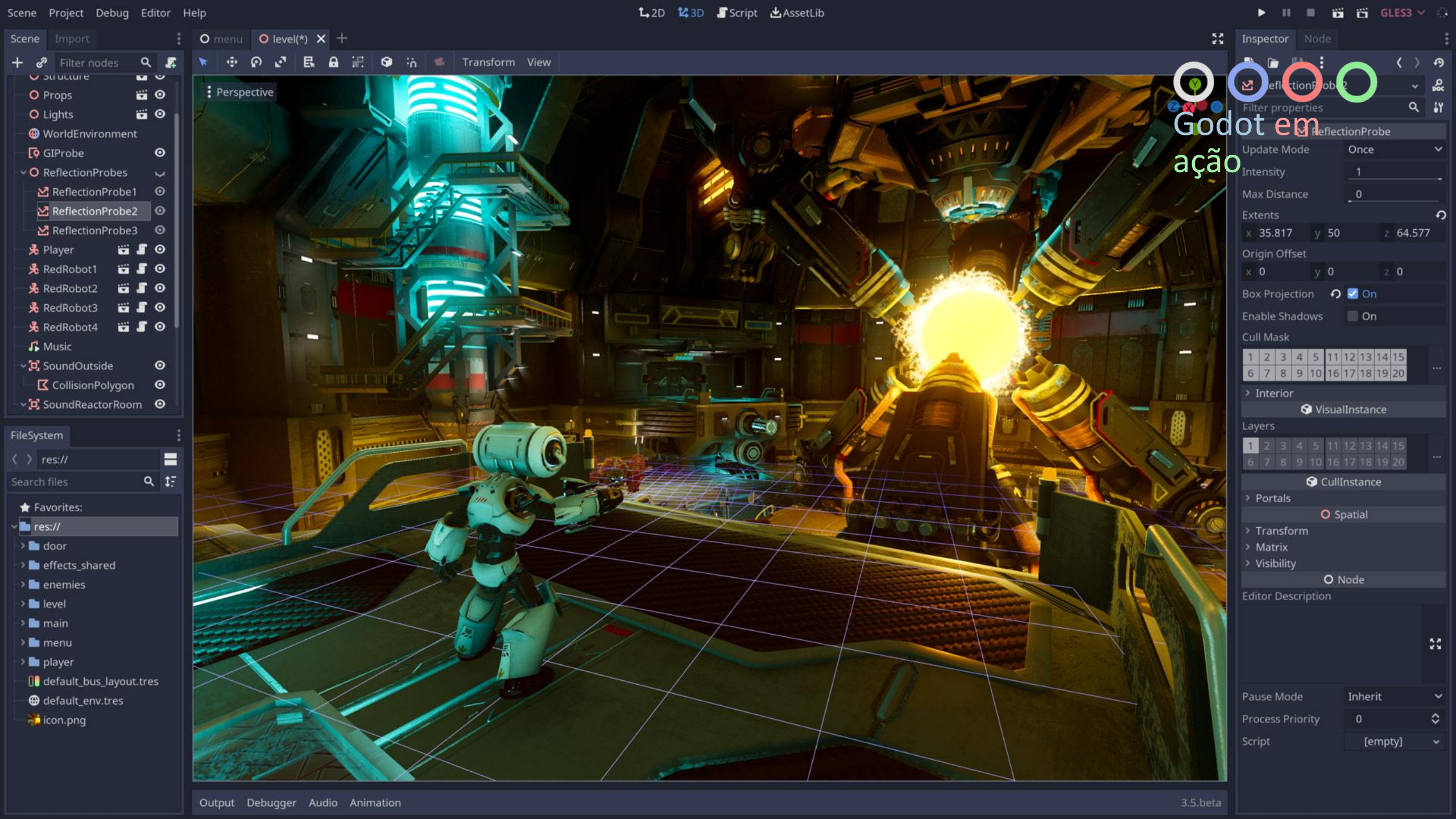
Motor de jogo 2D/ 3D escrito em C e C++ criado por Juan Linietsky e Ariel Manzur em conjunto com a OKAM Studio no começo de *2001*, sua primeira versão de **código aberto** lançada no dia 15 de dezembro de *2014*.

Multiplataforma:

- **PC:** Windows, Linux, OSX, Web (com WASM), e
- **Smartphones**: Android e IOS, por padrão;
- **Plataformas de realidade virtual (VR) / estendida (AR):** HTC Vive, Valve Index, Oculus Rift, Oculus Go, Óculos Quest, todos os headsets Microsoft MR, ARKit da Apple e outros;
- **Consoles** *(via licenciamento)* e muitos outros **portes não-oficiais**, como o Raps Berry Pi;
- O motor roda no Windows, Linux e Mac, nos web browsers (com diversas limitações), mas teoricamente pode ser configurado para rodar em qualquer outra plataforma citada acima.

Design simples e moderno. Pensado em jogos e no desenvolvedor de jogos. Atualmente, amplamente adotado por desenvolvedores independentes.


SHOWREEL
2022 Godot
Desktop Games


Scene Project Debug Editor Help
2D 3D Script AssetLib
GLES3
Scene Import
Filter nodes
Structure
Props
Lights
WorldEnvironment
GIProbe
ReflectionProbes
ReflectionProbe1
ReflectionProbe2
ReflectionProbe3
Player
RedRobot1
RedRobot2
RedRobot3
RedRobot4
Music
SoundOutside
CollisionPolygon
SoundReactorRoom
FileSystem
res://
Search files
Favorites:
res://
door
effects_shared
enemies
level
main
menu
player
default_bus_layout.tres
default_env.tres
icon.png
menu
level(*)
Transform View
Perspective
Godot em ação
Inspector Node
ReflectionProbe2
Filter properties
ReflectionProbe
Update Mode Once
Intensity 1
Max Distance 0
Extents
x 35.817 y 50 z 64.577
Origin Offset
x 0 y 0 z 0
Box Projection On
Enable Shadows On
Cull Mask
Interior
VisualInstance
Layers
CullInstance
Portals
Spatial
Transform
Matrix
Visibility
Node
Editor Description
Pause Mode Inherit
Process Priority 0
Script [empty]
Output Debugger Audio Animation
3.5.beta

Possui uma das licenças mais permissivas:

The MIT License (MIT)

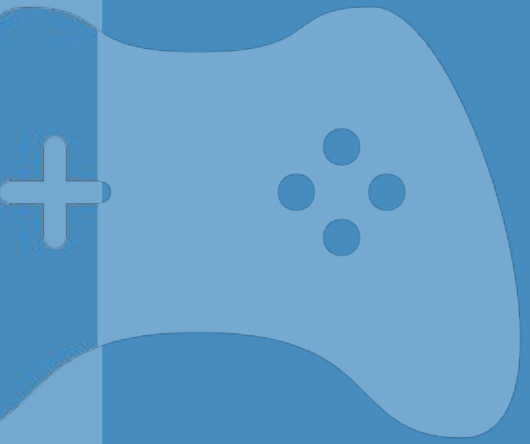
Hands out

Hands out

GDScript

Projeto FLOSS de linguagem de programação para jogos, projetada especificamente para a Godot.

O lead dev do motor, e um dos fundadores do projeto, é o argentino *Juan Linietsky,* e a cabeça no desenvolvimento da linguagem GDScript é o brasileiro *George Marques*.



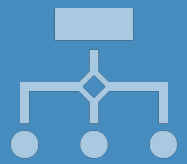
It just works!

Principal filosofia

Necessidades de um projeto de jogo comercial

Eficiência: jogabilidade suave (e em múltiplas plataformas — não só PC e Mobile, como hardware especializado: Consoles, Handhelds, VR/AR, etc...)

Praticidade: rápido de desenvolver, iterar sobre (refatorar) e experimentar. – Jogos, em especial protótipos, precisam de um fluxo veloz na dinâmica de criação.



Flexibilidade: fácil de modificar, modularidade. – Jogos requerem uma variabilidade em conteúdo e mecânicas relacionados de diversas formas.

Em primeiro lugar, o domínio de uma linguagem é, em geral, o fator determinante para sua adesão por um programador. Então escolher usar uma DSL é questão de escolher trabalhar sobre esse dado domínio.

Um bom programador busca usar a ferramenta mais amigável, mais prática e mais flexível para o tipo de problemas que se busca resolver.

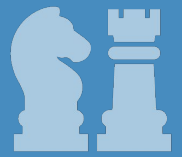
— E o design de GDScript é construído pensando sobre a criação de projetos, em geral, jogos na Godot!

GDScript foi pensado para lidar com os três problemas citados anteriormente, em especial praticidade e flexibilidade.

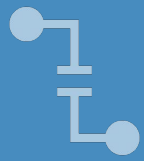
Vantagens de uma DSL (linguagem de domínio específico)

Características da linguagem *GDScript*

Eficiência muitas vezes vem como uma terceira prioridade porque...



Os sistemas de jogos modernos tendem a ser bem poderosos por padrão.



Godot suporta um poderoso sistema de *bindings* e múltiplas linguagens (C-based) trabalhando em conjunto, o que permite otimizar os gargalos (partes específicas de código crítico).

O foco de uma linguagem de script está, ou deveria estar, sempre nos humanos (desenvolvedores);

É importante ter uma ideia sobre o que é possível fazer com linguagens especializadas caso você tenha de lidar com uma delas algum dia.

Conhecendo conceitos de linguagens de programação torna muito mais rápido de aderir à tais linguagens que costumam ser muito simples (em sintaxe, semântica, até estrutura) e objetivas nos seus propósitos. A produtividade obtida ao usar uma dessas linguagens, em geral, costuma valer o esforço de aprender sua determinada sintaxe e aplicação.

DSLs algumas vezes podem até ser movidas de seus contextos originais para outras aplicações, dependendo mais ou menos do quão acoplada ela é com o seu sistema de origem. Como exemplo, temos a ECMAScript, mais especificamente o Javascript, que era uma DSL destinada à navegadores e que atualmente foi aplicada em outros contextos.

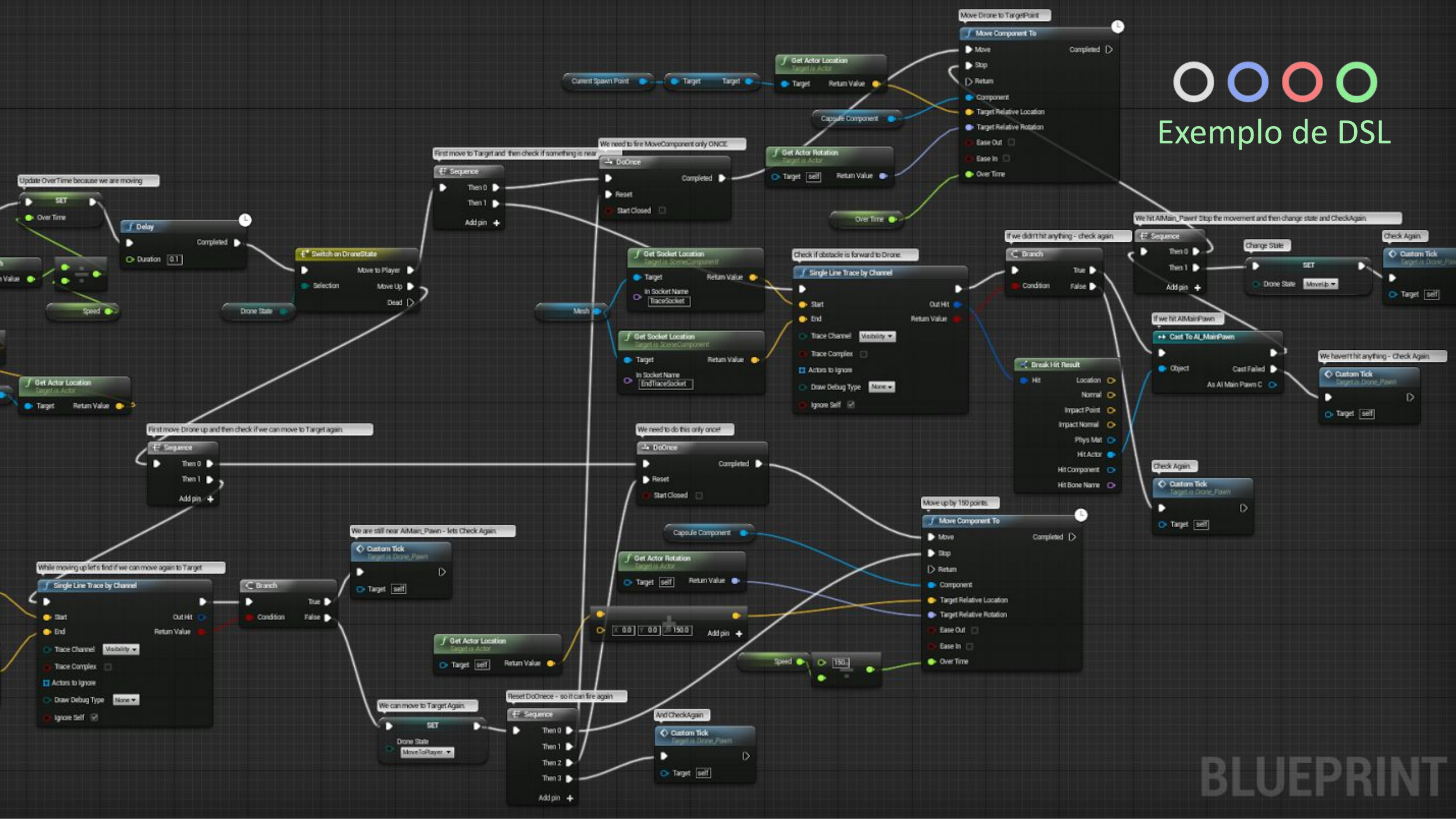
Vantagens de uma DSL (linguagem de domínio específico)

Exemplos de DSL:

- Game Description Language;
- GLSL (OpenGL Shading Language);
- Gradle (build management and automation);
- ActionScript from Flash;
- ELisp (Emacs Lisp);
- Visual Basic (domínio de programas visuais do Microsoft Windows).

E a grande maioria, se não todas, as linguagens visuais (como as linguagens de blocos em aplicações educacionais, ex.: Scratch), motores de jogos (ex.s: Unreal Blueprints, Gdvelop Event System, ...), ferramentas de automação em algumas aplicações de modelagem de áudio, softwares de produção gráfica e multimídia (ex.s: Blender Nodes, DaVinci Resolve Fusion, Natron editor, ...), etc...

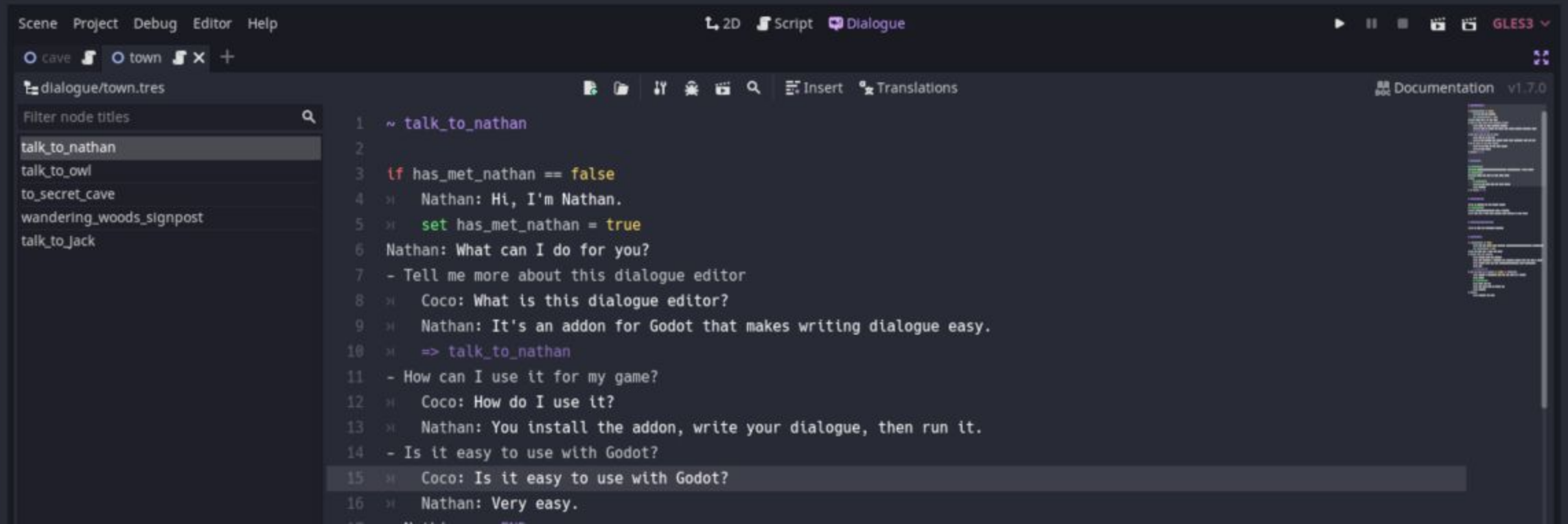
Vantagens de uma DSL (linguagem de domínio específico)


Exemplo de DSL
BLUEPRINT

Exemplo de DSL

Dialogue Manager

A Godot addon for simple branching dialogue



“Por que não ‘*legacy*’, por que não C#/C++?”

GDScript é uma linguagem de programação imperativa, contextual (interpretada, compilada JIT e AOT), orientada à objetos, construída como parte da Godot Game Engine. Foi criada com intuito de economizar o tempo dos desenvolvedores na codificação de seus jogos.*

Sintaxe simples, moderna, produtiva, sem muito overhead, permissiva à otimizações;
Muito **amigável** para iniciantes, e muito prática para profissionais;
Flexibilidade: GDScript é popularmente usada para criação de protótipos*, first slices* e ferramentas de propósito específico, devido a sua natureza simples e prática;
Altamente **vinculada com o motor**: code completion rico e contextualizado com a cena sendo editada, além de permitir criar interfaces de forma rápida já que sua API conversa muito bem com o editor;
Possui diversas **syntactic sugars** pensados em *game dev:*

- Por exemplo, uso dos atalhos para caminho relativo do projeto (res://), caminho do diretório do usuário na pasta (user://) - para arquivos *save*;
- Tipos de dados especiais, como Vectors e Transforms para manipulações lineares;
- Implementa acesso à rede na sintaxe: palavras reservadas remote, master e puppet (como annotations em GDScript 2.0);
- Constantes comuns, de uso frequente, disponíveis globalmente: PI, TAU, INF e NAN;
- Vem carregada com muitos métodos e atributos usados em jogos: lerp, clerp, clamp (max e min), smoothstep, rand, draw_, e muito mais...

Características da linguagem GDScript

```gdscript
extends Node2D

export var path: NodePath
onready var node = get_node_or_null(path)


func _ready() -> void:
    if not node:
        return
    print(node)
```

Desde a primeira versão do motor, a linguagem GDScript vem evoluindo rapidamente.
A versão 4.0 de Godot (atualmente, em fase alfa) trará grandes mudanças no ecossistema, sendo a mais significativa dela a implementação de uma nova versão da linguagem GDScript.

Visando isso, doravante, abordaremos as duas sintaxes nessa apresentação, sendo GDScript 1 apresentado à esquerda no tom escuro, e a correspondente 2.0 à direita em tema claro.

```gdscript
extends Node2D

@export var path: NodePath
@onready var node = get_node_or_null(path)


func _ready() -> void:
    if not node:
        return
    print(node)
```

Características da linguagem *GDScript*

Sintaxe: Inspirado em Python, Lua, Nim, Squirrel e outras linguagens de script;

```gdscript
# A file is a class!

# Inheritance

extends BaseClass

# (optional) class definition with a custom icon

class_name MyClass, "res://path/to/optional/icon.svg"


# Member variables

var a = 5
var s = "Hello"
var arr = [1, 2, 3]
var dict = {"key": "value", 2: 3}
var typed_var: int
var inferred_type := "String"

# Constants

const ANSWER = 42
const THE_NAME = "Charly"
```

```gdscript
# Enums.
enum {UNIT_NEUTRAL, UNIT_ENEMY, UNIT_ALLY}
enum Named {THING_1, THING_2, ANOTHER_THING = -1}

# Built-in vector types.
var v2 = Vector2(1, 2)
var v3 = Vector3(1, 2, 3)

```

```gdscript
# Enums

enum {UNIT_NEUTRAL, UNIT_ENEMY, UNIT_ALLY}
enum Named {THING_1, THING_2, ANOTHER_THING = -1}

# Built-in vector types

var v2 = Vector2(1, 2)
var v3 = Vector3(1, 2, 3)

```

```gdscript
# Everything after "#" is a comment.
# A file is a class!

# (optional) class definition:
class_name MyClass

# Inheritance:
extends BaseClass

# (optional) icon to show in the editor dialogs:
@icon("res://path/to/optional/icon.svg")


# Member variables.
var a = 5
var s = "Hello"
var arr = [1, 2, 3]
var dict = {"key": "value", 2: 3}
var other_dict = {key = "value", other_key = 2}
var typed_var: int
var inferred_type := "String"

# Constants.
const ANSWER = 42
const THE_NAME = "Charly"

```

Características da linguagem *GDScript*

Sintaxe: Inspirado em Python, Lua, Nim, Squirrel e outras linguagens de script;

```
# Function

func some_function(param1, param2, param3):
    var local_var = 5

    if param1 < local_var:
        print(param1)
    elif param2 > 5:
        print(param2)
    else:
        print("Fail!")

    for i in range(20):
        print(i)

    while param2 != 0:
        param2 -= 1

    match param3:
        3:
            print("param is 3")
        _:
            print("param is not 3")

    var local_var2 = param1 + 3
    return local_var2
```

```
# Functions.
func some_function(param1, param2, param3):
    const local_const = 5

    if param1 < local_const:
        print(param1)
    elif param2 > 5:
        print(param2)
    else:
        print("Fail!")

    for i in range(20):
        print(i)

    while param2 ≠ 0:
        param2 -= 1

    match param3:
        3:
            print("param3 is 3!")
        _:
            print("param3 is not 3!")

    var local_var = param1 + 3
    return local_var
```

Características da linguagem *GDScript*

Sintaxe: Inspirado em Python, Lua, Nim, Squirrel e outras linguagens de script;

```
# Functions override functions with the same name on the base/parent class.
# If you still want to call them, use '.' (like 'super' in other languages).

func something(p1, p2):
    .something(p1, p2)


# It's also possible to call another function in the super class:
func other_something(p1, p2):
    .something(p1, p2)

# Inner class

class Something:
    var a = 10


# Constructor

func _init():
    print("Constructed!")
    var lv = Something.new()
    print(lv.a)

```

```
# Functions override functions with the same name on the base/super class.
# If you still want to call them, use "super":
func something(p1, p2):
    super(p1, p2)


# It's also possible to call another function in the super class:
func other_something(p1, p2):
    super.something(p1, p2)


# Inner class
class Something:
    var a = 10


# Constructor
func _init():
    print("Constructed!")
    var lv = Something.new()
    print(lv.a)

```

Características da linguagem *GDScript*

Sintaxe: Inspirado em Python, Lua, Nim, Squirrel e outras linguagens de script;

Sintaxe

palavras reservadas

GDScript 1.0 only:

tool, onready, export, setget, remote, master, puppet, remotesync, mastersync, puppetsync - transformadas em *annotations* na 2.0;

yield - convertida para await;

GDScript 1.0 & 2.0:

if, elif, else, for, while, match, break, continue, pass, return, class, class_name, extends, is, as, self, signal, func, static, const, enum, var, breakpoint, preload, assert, PI, TAU, INF, NAN

GDScript 2.0 only:

await

Sintaxe

palavras reservadas

A seguir estão listadas as palavras reservadas, ***tokens***, da linguagem GDScript. Operadores (como, in, not, and ou or) e nomes de tipos built-in, que serão apresentadas posteriormente, também são reservadas.

Keyword	Description
if	See if/else/elif.
elif	See if/else/elif.
else	See if/else/elif.
for	See for.
while	See while.
match	See match.
break	Exits the execution of the current `for` or `while` loop.
continue	Immediately skips to the next iteration of the `for` or `while` loop.
pass	Used where a statement is required syntactically but execution of code is undesired, e.g. in empty functions.
return	Returns a value from a function.
class	Defines an inner class.
class_name	Defines a class name and optional icon for your script.
extends	Defines what class to extend with the current class.

Sintaxe

palavras reservadas

A seguir estão listadas as palavras reservadas, ***tokens***, da linguagem GDScript. Operadores (como, in, not, and ou or) e nomes de tipos built-in, que serão apresentadas posteriormente, também são reservadas.

Keyword	Description
is	Tests whether a variable extends a given class, or is of a given built-in type.
as	Cast the value to a given type if possible.
self	Refers to current class instance.
tool [1]	Executes the script in the editor.
signal	Defines a signal.
func	Defines a function.
static	Defines a static function. Static member variables are not allowed.
const	Defines a constant.
enum	Defines an enum.
var	Defines a variable.
onready [1]	Initializes a variable once the Node the script is attached to and its children are part of the scene tree.
export [1]	Saves a variable along with the resource it's attached to and makes it visible and modifiable in the editor.
setget [1]	Defines setter and getter functions for a variable.

[1] GDScript 1

Sintaxe

palavras reservadas

Keyword	Description
breakpoint	Editor helper for debugger breakpoints.
preload	Preloads a class or variable. See Classes as resources.
yield [1]	Coroutine support. See Coroutines with yield.
assert	Asserts a condition, logs error on failure. Ignored in non-debug builds. See Assert keyword.
remote	Networking RPC annotation. See high-level multiplayer docs.
master [1]	Networking RPC annotation. See high-level multiplayer docs.
puppet [1]	Networking RPC annotation. See high-level multiplayer docs.
remotesync [1]	Networking RPC annotation. See high-level multiplayer docs.
mastersync [1]	Networking RPC annotation. See high-level multiplayer docs.
puppetsync [1]	Networking RPC annotation. See high-level multiplayer docs.
PI	PI constant.
TAU	TAU constant.
INF	Infinity constant. Used for comparisons.
NAN	NAN (not a number) constant. Used for comparisons.
await [2]	Waits for a signal or a coroutine to finish. See Awaiting for signals.

[1] GDScript 1 [2] GDScript 2.0

Sintaxe

operadores

A seguir estão listados os operadores suportados em sua ordem de precedência.
(! && e || foram removidos em GDScript 2.0).

Operator	Description
`x[index]`	Subscription (highest priority)
`x.attribute`	Attribute reference
`foo()`	Function call
`is`	Instance type checker
`**` (Gd 2.0)	Power operator Multiplies value by itself `x` times, similar to calling `pow` built-in function
`~`	Bitwise NOT
`-x`	Negative / Unary negation
`*` `/` `%`	Multiplication / Division / Remainder These operators have the same behavior as C++. Integer division is truncated rather than returning a fractional number, and the % operator is only available for ints (`fmod` for floats), and is additionally used for Format Strings
`+`	Addition / Concatenation of arrays
`-`	Subtraction
`<<` `>>`	Bit shifting
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`<` `>` `==` `!=` `>=` `<=`	Comparisons
`in`	Content test
`!` `not`	Boolean NOT
`and` `&&`	Boolean AND
`or` `||`	Boolean OR
`if x else`	Ternary if/else
`as`	Type casting
`=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `<<=` `>>=`	Assignment (lowest priority)

Sintaxe

literais

Literal	Type
`45`	Base 10 integer
`0x8f51`	Base 16 (hexadecimal) integer
`0b101010`	Base 2 (binary) integer
`3.14`, `58.1e-10`	Floating-point number (real)
`"Hello"`, `"Hi"`	Strings
`"""Hello"""`	Multiline string
`&"name"`	ref StringName
`^"Node/Label"`	ref NodePath
`$NodePath`	Shorthand for `get_node("NodePath")`

```
# Integers and floats can have their numbers separated with _ to make them
# more readable. The following ways to write numbers are all valid:

12_345_678  # Equal to 12345678.
3.141_592_7  # Equal to 3.1415927.
0x8080_0000_ffff  # Equal to 0x80800000ffff.
0b11_00_11_00  # Equal to 0b11001100.
```

Sintaxe

annotations

Em GDScript 2.0 há *tokens* especiais (palavras-chave) que agem como as palavras reservadas de GDScript 1 mas não são, ao invés disso são tratadas como *annotations*. Cada *annotation* começa com o caractere @ seguida de seu nome.

Elas afetam como o *script* é tratado por ferramentas externas e, normalmente, não alteram o comportamento.

Por exemplo, as *annotations* @export_* permitem que um atributo seja disponibilizado no editor da Godot.

Scene Project Debug Editor Help 2D 3D Script AssetLib

Scene Import

Filter no

Control
LineEdit
LineEdit2

FileSystem

res://src/Node.gd

Search files

Favorites:
res://
src
icon.png

Exemplo

Control

File Search Edit Go To Debug Online Docs Search Help

```
4   @export var statement := ""
5
6   @export_range(1, 100, 1, "or_greater")
7   var ranged_var: int = 42
8
9   # Podem estar na mesma linha, ou não
10  @onready
11  @export_node_path(TextEdit, LineEdit)
12  var input_field
13  @onready @export_node_path(TextEdit, LineEdit) var input_field2
14
15
16  func _ready() -> void:
17      prints(statement, "more than", ranged_var, get_node(input_field).text)
18
```

18 : 1

```
--- Debugging process started ---
Godot Engine v4.0.alpha8.official.cc3ed63af - https://godotengine.org
Vulkan API 1.2.131 - Using Vulkan Device #0: AMD - AMD Radeon R7 M260

There are more than 8000 reasons to use Godot
--- Debugging process stopped ---
```

4 0 0 2

Filter messages

Output Debugger (1) Search Results Audio Animation Replication 4.0.alpha8

Inspector Node

Control

Filter properties

Control.gd
Statement There are
Ranged Var 8000
Input Field LineEdit
Input Field 2 Assign...

Control
Layout
Auto Translate
Hint
Focus
Mouse
Theme

CanvasItem
Visibility
Texture
Material

Node
Process
Editor Description
Script Control.g

Add Metadata

Sintaxe

annotations

Aqui estão listadas todas as *annotations* disponíveis em GDScript 2.0:

Annotation	Description
`@tool`	Enable the Tool mode.
`@onready`	Defer initialization of variable until the node is in the tree. See @onready annotation.
`@icon(path)`	Set the class icon to show in editor. To be used together with the `class_name` keyword.
`@rpc`	RPC modifiers. See high-level multiplayer docs.

`@export`
`@export enum`
`@export file`
`@export dir`
`@export global file`
`@export global dir`
`@export multiline`
`@export placeholder`
`@export range`
`@export exp easing`
`@export color no alpha`
`@export node path`
`@export flags`
`@export flags 2d render`
`@export flags 2d physics`
`@export flags 3d render`
`@export_flags_3d_physics`
	Export hints for the editor. See GDScript exports.

Sintaxe

comentários e continuação de linhas

Comentários em GDScript são feitos com o caractere #

```
# This is a comment.
```

GDScript 2.0 introduziu uma sintaxe de docstring para auto geração de documentação

```
extends Node2D

## A brief description of your script.
##
## A more detailed description of the script.
##
## @tutorial:            http://the/tutorial1/url.com
```

Assim como em C e Python, uma linha em GDScript pode ser continuada na seguinte, ao usar o caractere \ no final de uma linha, como uma forma de “escapar” a quebra.

```
var a = 1 + \
2
```

Isso pode ser feito várias vezes para a mesma linha:

```
var a = 1 + \
4 + \
10 + \
4
```

Filter scripts

- Control.gd
- Node.gd
- CanvasItem
- Control
- Node

Node

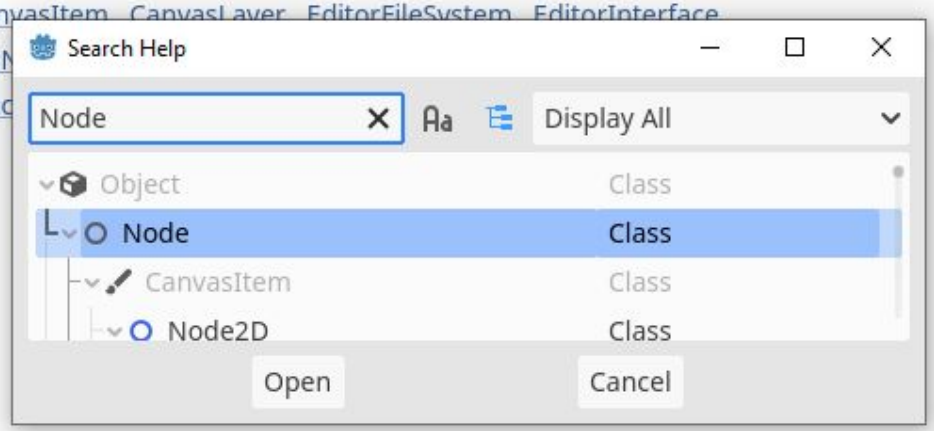
Filter methods

- Top
- Description
- Properties
- Methods
- Signals
- Enumerations
- Constants
- Property Descriptions
- Method Descriptions

Class: Node

Inherits: Object

Inherited by: AnimationPlayer , AnimationTree , AudioStreamPlayer , BaseClass , CanvasItem , CanvasLayer , EditorFileSystem , EditorInterface , EditorPlugin , EditorResourcePreview , HTTPRequest , InstancePlaceholder , MissingN , NavigationAgent2D , NavigationAgent3D , NavigationObstacle2D , NavigationObstac , SkeletonIK3D , Timer , Viewport , WorldEnvironment



Base class for all *scene* objects.

Description

Nodes are Godot's building blocks. They can be assigned as the child of another node, resulting in a tree arrangement. A given node can contain any number of nodes as children with the requirement that all siblings (direct children of a node) should have unique names.

A tree of nodes is called a *scene*. Scenes can be saved to the disk and then instantiated into other scenes. This allows for very high flexibility in the architecture and data model of Godot projects.

Scene tree: The `SceneTree` contains the active tree of nodes. When a node is added to the scene tree, it receives the `NOTIFICATION_ENTER_TREE` notification and its `_enter_tree()` callback is triggered. Child nodes are always added *after* their parent node, i.e. the `_enter_tree()` callback of a parent node will be triggered before its child's.

Once all nodes have been added in the scene tree, they receive the `NOTIFICATION_READY` notification and their respective `_ready()` callbacks are

Tipos built-in

Os tipos built-in de Gdscript são:

- Tipos primitivos:
  - null - vazio
  - bool - true | false
  - int - inteiro .ex.: -2, 0, 3, INF
  - float - real .ex.: -1., .0, .5, PI
  - Variant - “qualquer”
- Tipos compostos:
  - Array e Pool - listas
  - Dicionary - dicionário { “chave” : valor }
  - String, StringName e NodePath - “texto”
  - Vector2, Vector3, Rect2, Plane, Transform2D, Quat, AABB, Color, Basis e Transform - espaço .ex.: P=(x, y, z)
- Tipos especiais:
  - Signal, Callable e Object

Tipos built-in

```gdscript
# Passagem por referência
var arr = Array([1, 2, 3, 4])
var arr2 = arr
print(arr, arr2)
arr2.invert()
print(arr, arr2)
```

```gdscript
# Passagem por valor
var pool = PoolIntArray([1, 2, 3, 4])
var pool2 = pool
print(pool, pool2)
pool2.invert()
print(pool, pool2)
```

```gdscript
# Passagem por referência
var dict = Dictionary()
dict.a = 1
dict.b = 2
dict.c = 3
var dict2 = dict
print(dict, dict2)
dict2.erase("a")
print(dict, dict2)
```

Tipos built-in

String

String - Uma sequência de caracteres em formato Unicode. Usam contagem de referência, e implementam uma estratégia de copiar-ao-escrever, diminuindo seu custo em recursos.
Elas podem conter as seguintes sequências de escape:

Escape sequence	Expands to
`\n`	Newline (line feed)
`\t`	Horizontal tab character
`\r`	Carriage return
`\a`	Alert (beep/bell)
`\b`	Backspace
`\f`	Formfeed page break
`\v`	Vertical tab character
`\"`	Double quote
`\'`	Single quote
`\\`	Backslash
`\uXXXX`	Unicode codepoint `XXXX` (hexadecimal, case-insensitive)

Tipos built-in

String

GDScript também suporta formatação de Strings

Operador %

```
# Define a format string with placeholder '%s'
var format_string = "We're waiting for %s."

# Using the '%' operator, the placeholder is replaced with the desired value
var actual_string = format_string % "Godot"

print(actual_string)
# Output: "We're waiting for Godot."
```

String.format()

```
# Define a format string
var format_string = "We're waiting for {str}"

# Using the 'format' method, replace the 'str' placeholder
var actual_string = format_string.format({"str": "Godot"})

print(actual_string)
# Output: "We're waiting for Godot"
```

Múltiplos *placeholders*

```
var format_string = "%s was reluctant to learn %s, but now he enjoys it."
var actual_string = format_string % ["Estragon", "GDScript"]

print(actual_string)
# Output: "Estragon was reluctant to learn GDScript, but now he enjoys it."
```

Tipos built-in

Container types: Arrays e Pools

Em *GDScript 2.0* Arrays podem ser tipados estaticamente usando a sintaxe à seguir:

```gdscript
var array: Array[int] = [1, 2, 3, 4]
# Atribuição resulta em erro, apresentado em tempo real no editor.
# array[0] = "a"

var array2: Array[String] = ["a", "b", "c"]
array2[0] = "a"

var arrayObj: Array[Object] = [null, self, Node.new()]
# A atribuição abaixo resulta em erro, pois, diferente de Python,
# Uma String não é um objeto, mas um tipo Variant.
#arrayObj[0] = String()

# Porém, Object também é uma Variant na Godot
# É equivalente a um Array genérico
var arrayVar: Array[Variant] = [null, self, Node3D.new(), Color.WHITE, 0]
arrayVar[0] = String()
```

Tipos built-in

Container types: Dictionaries

Dictionary - Associative container which contains values referenced by unique keys.

```
var d = {4: 5, "A key": "A value", 28: [1, 2, 3]}
d["Hi!"] = 0
d = {
    22: "value",
    "some_key": 2,
    "other_key": [2, 3, 4],
    "more_key": "Hello"
}
```

Lua-style table syntax is also supported.
Lua-style uses = instead of : and doesn't use quotes to mark string keys (making for slightly less to write). However, keys written in this form can't start with a digit (like any GDScript identifier).

```
var d = {
    test22 = "value",
    some_key = 2,
    other_key = [2, 3, 4],
    more_key = "Hello"
}
```

Tipos built-in

Container types: Dictionaries

To add a key to an existing dictionary, access it like an existing key and assign to it:

```
var d = {} # Create an empty Dictionary.
d.waiting = 14 # Add String "waiting" as a key and assign the value 14 to it.
d[4] = "hello" # Add integer 4 as a key and assign the String "hello" as its value.
d["Godot"] = 3.01 # Add String "Godot" as a key and assign the value 3.01 to it.

var test = 4
# Prints "hello" by indexing the dictionary with a dynamic key.
# This is not the same as `d.test`. The bracket syntax equivalent to
# `d.test` is `d["test"]`.
print(d[test])
```

Note

The bracket syntax can be used to access properties of any Object, not just Dictionaries. Keep in mind it will cause a script error when attempting to index a non-existing property. To avoid this, use the Object.get() and Object.set() methods instead.

Dados

variáveis

Variáveis podem existir como membros de classes ou locais em funções.

Elas podem ser criadas com a palavra reservada **var** e podem, opcionalmente, ser assinadas com um valor na inicialização.

```
var a # Data type is 'null' by default.
var b = 5
var c = 3.8
var d = b + c # Variables are always initialized in order.
```

Variáveis podem, opcionalmente, ter **especificação de tipo – vinculação explícita** . Quando um tipo é especificado, a variável será forçada a sempre ter esse mesmo tipo, e tentar assinar um tipo incompatível resultará em erro.

```
var my_vector2: Vector2
var my_node: Node = Sprite.new()
```

Tipos são especificados na declaração da variável usando o símbolo : após o nome da variável, seguido do tipo.

Se a variável for inicializada na declaração, o tipo pode ser **inferido**, então é possível omitir o nome do tipo, **– vinculação implícita**:

```
var my_vector2 := Vector2() # 'my_vector2' is of type 'Vector2'.
var my_node := Sprite.new() # 'my_node' is of type 'Sprite'.
```

Inferência de tipos somente é possível se o valor do tipo assinado for de um tipo definido, caso contrário será gerado um erro.

Dados

variáveis

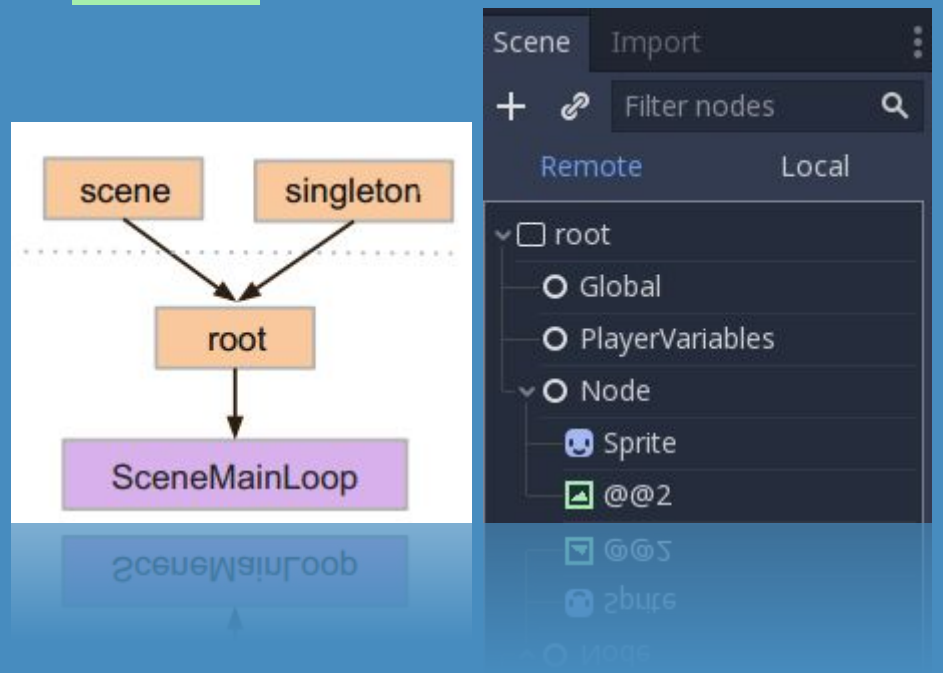
GDScript usa **vinculação dinâmica fortemente tipada**: todas as variáveis são **implicitamente** do tipo Variant que é uma espécie de “descritor” das variáveis que são gerenciadas pela ClassDB.

```
var foo = 2
match typeof(foo):
	TYPE_NIL:
		print("foo is null")
	TYPE_INT:
		print("foo is an integer")
	TYPE_OBJECT:
		# Note that Objects are their own special category.
		# To get the name of the underlying Object type,
		# you need the `get_class()` method.
		print("foo is a(n) %s" % foo.get_class())
		# inject the class name into a formatted string.
		# Note also that there is not yet any way to get a script's
		# `class_name` string easily.
		# To fetch that value, you need to dig deeply into a hidden
		# ProjectSettings setting: an Array of Dictionaries called
		# "_global_script_classes".
		# Open your project.godot file to see it up close.
```

A função global typeof retorna o valor enumerado, de um tipo Variant, armazenado na variável.

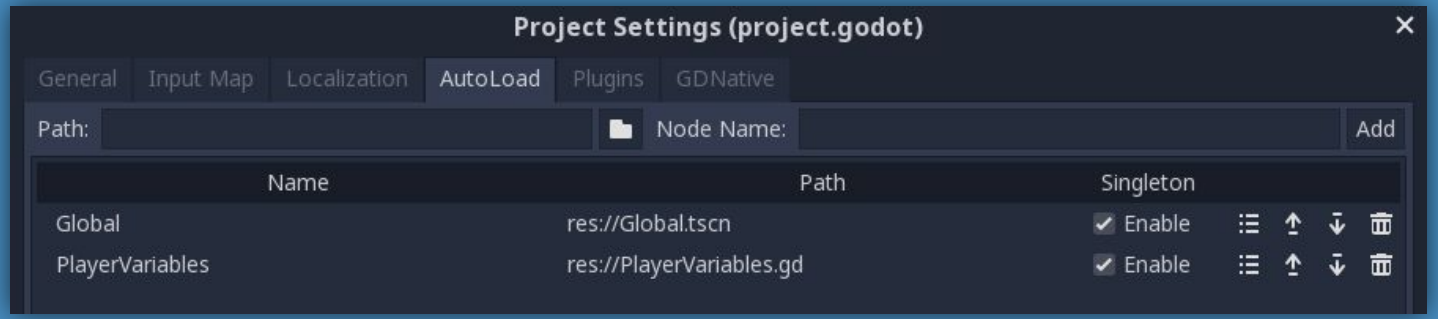
Dados

variáveis



Godot não permite a criação de variáveis globais (disponíveis entre *scripts*), pois já que todos os *scripts* são definições de classes, não existe um “escopo global” acessível ao usuário da linguagem.

Mas, caso seja necessário manter dados consistentes entre cenas, é possível implementar o padrão singleton, definindo um conjunto de classes (nós) globais que serão carregados na cena junto a inicialização do game. Assim, atributos e métodos poderão ser acessados à partir do nome dessa classe.




```
PlayerVariables.health -= 10
```

Dados

casting

Valores atribuídos à variáveis tipadas devem ser de um tipo compatível. Se for necessário ‘coerir’ um valor para ser de um certo tipo, em particular para subtipos de objetos, você pode fazer **casting** com o operador **as**.

```
var my_node2D: Node2D
my_node2D = $Sprite as Node2D # Works since Sprite is a subtype of Node2D.
```

Casting entre tipos de objetos resulta no mesmo objeto se o valor é do mesmo tipo ou subtipo do tipo coerido.

```
var my_node2D: Node2D
my_node2D = $Button as Node2D # Results in 'null' since a Button is not a subtype of Node2D.
```

Se o valor não for um subtipo, a operação de casting resultará no valor null.

Dados

casting

Valores atribuídos à variáveis tipadas devem ser de um tipo compatível. Se for necessário ‘coerir’ um valor para ser de um certo tipo, em particular para subtipos de objetos, você pode fazer **casting** com o operador **as**.

```
var my_int: int
my_int = "123" as int # The string can be converted to int.
my_int = Vector2() as int # A Vector2 can't be converted to int, this will cause an error.
```

Para tipos built-in, eles serão, forçadamente, convertidos se possível, caso contrário, o motor irá gerar um erro.

```
# Will infer the variable to be of type Sprite.
var my_sprite := $Character as Sprite

# Will fail if $AnimPlayer is not an AnimationPlayer, even if it has the method 'play()'.
($AnimPlayer as AnimationPlayer).play("walk")
```

Casting também é útil para obter variáveis mais type-safe quando interagindo com a árvore da cena.

Dados

constantes

Constants are values you cannot change when the game is running.
Their value must be known at compile-time.
Using the const keyword allows you to give a constant value a name.
Trying to assign a value to a constant after it's declared will give you an error.

```
const A = 5
const B = Vector2(20, 20)
const C = 10 + 20 # Constant expression.
const D = Vector2(20, 30).x # Constant expression: 20.
const E = [1, 2, 3, 4][0] # Constant expression: 1.
const F = sin(20) # 'sin()' can be used in constant expressions.
const G = x + 20 # Invalid; this is not a constant expression!
const H = A + 20 # Constant expression: 25 (`A` is a constant).
```

Although the type of constants is inferred from the assigned value, it's also possible to add explicit type specification:

```
const A: int = 5
const B: Vector2 = Vector2()
```

Assigning a value of an incompatible type will raise an error.

Dados

enums

Enums are basically a shorthand for constants, and are pretty useful if you want to assign consecutive integers to some constant.

```
enum {TILE_BRICK, TILE_FLOOR, TILE_SPIKE, TILE_TELEPORT}
# Is the same as:
const TILE_BRICK = 0
const TILE_FLOOR = 1
const TILE_SPIKE = 2
const TILE_TELEPORT = 3

enum State {STATE_IDLE, STATE_JUMP = 5, STATE_SHOOT}
# Is the same as:
const State = {STATE_IDLE = 0, STATE_JUMP = 5, STATE_SHOOT = 6}
# Access values with State.STATE_IDLE, etc.
```

If you pass a name to the enum, it will put all the keys inside a constant dictionary of that name.

Important

In Godot 3.1 and later, keys in a named enum are not registered as global constants. They should be accessed prefixed by the enum's name (`Name.KEY`); see an example above.

Dados

funções

Functions always belong to a class. The scope priority for variable look-up is: local → class member → global.

The self variable is always available and is provided as an option for accessing class members, but is not always required (and should not be sent as the function's first argument, unlike Python).

```
func my_function(a, b):
    prints(self, a, b)
    return a + b  # Return is optional; without it 'null' is returned.
```

Functions can also have type specification for the arguments and for the return value. Types for arguments can be added in a similar way to variables:

```
func my_function(a: int, b: String) -> void:
    pass
```

If a function argument has a default value, it's possible **to infer** the type:

```
func my_function(int_arg := 42, String_arg := "string"):
    pass
```

Dados

funções: escopo

Como não há funções internas, nem escopo global (já que *scripts* são classes), os únicos escopos existentes são:

- Escopo "global" (local em relação à classe): apenas definição de propriedades e métodos;
- Escopo local (em relação à função ou bloco de estrutura – como if, else, match, etc.): declarações, exceto definições de métodos e classes – e sinais, em GDScript 1.

Dados

funções: escopo

```
extends Node
var x: int = 2

func _ready() -> void:
    # x é vinculado ao escopo local, pois foi declarado localmente.
    print(x) # Imprime null na tela, pois x existe porém não foi inicializado.
    #print(z) # Causa erro, pois z não existe nesse momento
    # (não foi inicializado, nem redeclarado).

    var z: int = 4
    var x = func hello() -> void:
        #var z = "hello" # Resulta em erro: z já está declarado no escopo local
        while true:
            var k = "hello"; # k é declarado no escopo local do bloco while
            break;
        # print(k) # Resulta em erro:
        # k não está declarado no escopo local de hello

    x.call() # Chama o método vinculado à variável local x
    do() # Imprime o valor da propriedade x da classe
    # x volta a ser vinculado ao escopo global

func do():
    print(x) # x está vinculado ao escopo global, pois não foi definido aqui.
```

Note que em GDScript 2.0, embora funções sejam cidadãos de primeira ordem, não é possível defini-las e acessá-las no escopo da função, exceto em atribuições.

Dados

[referenciando funções: GDScript 1]

In GDScript 1: contrary to Python, functions are not first-class objects. This means they cannot be stored in variables, passed as an argument to another function or be returned from other functions. This ~~is~~ was for performance reasons.

```
# Call a function by name in one step.
my_node.call("my_function", args)

# Store a function reference.
var my_func = funcref(my_node, "my_function")
# Call stored function reference.
my_func.call_func(args)
```

To reference a function by name at run-time, (e.g. to store it in a variable, or pass it to another function as an argument) one must use the call or funcref helpers:

Dados

referenciando funções: GDScript 2.0

In GDScript 2.0: Functions are first-class items in terms of the Callable object. Referencing a function by name without calling it will automatically generate the proper callable. This can be used to pass functions as arguments.

```gdscript
func map(arr: Array, function: Callable) -> Array:
    var result = []
    for item in arr:
        result.push_back(function.call(item))
    return result

func add1(value: int) -> int:
    return value + 1;

func _ready() -> void:
    var my_array = [1, 2, 3]
    var plus_one = map(my_array, add1)
    print(plus_one) # Prints [2, 3, 4].
```

Note

Callables **must** be called with the `call` method. You cannot use the `()` operator directly. This behavior is implemented to avoid performance issues on direct function calls.

Statements & Control Flow

Declarações são *standard* (seguem o padrão) e podem ser atribuições/ assinaturas, chamadas de funções, estruturas de fluxos de controle, etc.

; como um separador de declarações é inteiramente opcional.

Statements & Control Flow

Expressões em GDScript

```
2 + 2 # Binary operation.
-5 # Unary operation.
"okay" if x > 4 else "not okay" # Ternary operation.
x # Identifier representing variable or constant.
x.a # Attribute access.
x[4] # Subscript access.
x > 2 or x < 5 # Comparisons and logic operators.
x == y + 2 # Equality test.
do_something() # Function call.
[1, 2, 3] # Array definition.
{A = 1, B = 2} # Dictionary definition.
preload("res://icon.png") # Preload builtin function.
self # Reference to current instance.
```

Expressões retornam valores que podem ser atribuídos à destinos válidos. Operandos de algumas operações podem ser outras expressões. Uma atribuição/ assinatura não é uma expressão, portanto, não retorna valor.

Identificadores, atributos, e subscritos são destinos válidos. Outras expressões não podem ser posicionadas à esquerda de uma atribuição.

Statements & Control Flow

[if-else-elif]

Simple conditions are created by using the if/else/elif syntax. Parenthesis around conditions are allowed, but not required. Given the nature of the tab-based indentation, elif can be used instead of else/if to maintain a level of indentation.

```
if (expression):
    statement(s)
elif (expression):
    statement(s)
else:
    statement(s)
```

Short statements can be written on the same line as the condition:

```
if 1 + 1 == 2: return 2 + 2
else:
    var x = 3 + 3
    return x
```

Sometimes, you might want to assign a different initial value based on a boolean expression. In this case, ternary-if expressions come in handy:

```
var x = [value] if [expression] else [value]
y += 3 if y < 10 else -1
```

Statements & Control Flow

if-else-elif

Ternary-if expressions can be nested to handle more than 2 cases. When nesting ternary-if expressions, it is recommended to wrap the complete expression over multiple lines to preserve readability:

```
var count = 0

var fruit = (
        "apple" if count == 2
        else "pear" if count == 1
        else "banana" if count == 0
        else "orange"
)
print(fruit)  # banana

# Alternative syntax with backslashes instead of parentheses (for multi-line expressions).
# Less lines required, but harder to refactor.
var fruit_alt = \
        "apple" if count == 2 \
        else "pear" if count == 1 \
        else "banana" if count == 0 \
        else "orange"
print(fruit_alt)  # banana
```

Statements & Control Flow

loops: while e for

while - Simple loops are created by using while syntax. Loops can be broken using break or continued using continue:

```
while (expression):
    statement(s)
    if expression: continue
    statement(s)
    if expression: break
    statement(s)
```

for - To iterate through a range, such as an array or table, a for loop is used. When iterating over an array, the current array element is stored in the loop variable. When iterating over a dictionary, the key is stored in the loop variable.

```
for x in [5, 7, 11]:
    statement # Loop iterates 3 times with 'x' as 5, then 7 and finally 11.

var dict = {"a": 0, "b": 1, "c": 2}
for i in dict:
    print(dict[i]) # Prints 0, then 1, then 2.
```

Statements & Control Flow

loops: while e for

for - To iterate through a range, such as an array or table, a for loop is used. When iterating over an array, the current array element is stored in the loop variable. When iterating over a dictionary, the key is stored in the loop variable.

```
for i in range(3):
    statement # Similar to [0, 1, 2] but does not allocate an array.

for i in range(1, 3):
    statement # Similar to [1, 2] but does not allocate an array.

for i in range(2, 8, 2):
    statement # Similar to [2, 4, 6] but does not allocate an array.

for c in "Hello":
    print(c) # Iterate through all characters in a String, print every letter on new line.

for i in 3:
    statement # Similar to range(3)

for i in 2.2:
    statement # Similar to range(ceil(2.2))
```

Statements & Control Flow

match

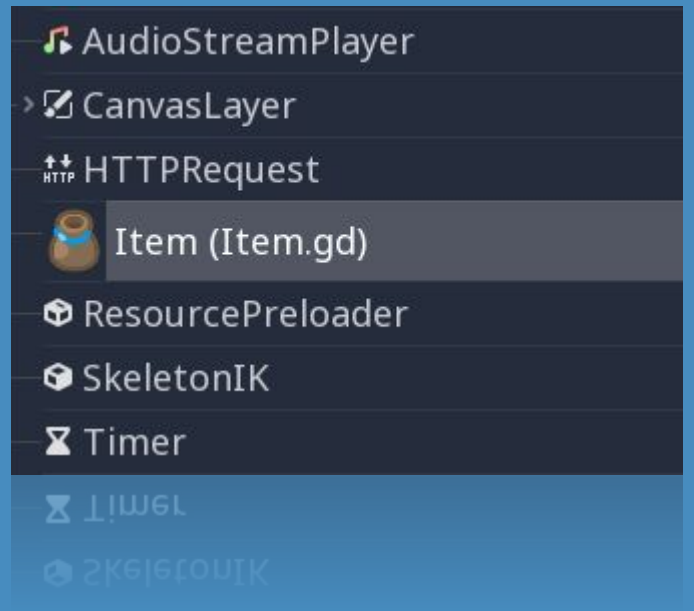
match - statement used to branch execution of a program. It's the equivalent of the switch statement found in many other languages, but offers some additional features. Basic syntax:

```
match (expression):
    [pattern](s):
        [block]
    [pattern](s):
        [block]
    [pattern](s):
        [block]
```

From switch to match:

1. Replace switch with match;
2. Remove case;
3. Remove any breaks;
4. Change default to a single underscore.

Classes



By default, all script files are **unnamed classes**. In this case, you can only reference them using the file's path, using either a relative or an absolute path. For example, if you name a script file character.gd:

```gdscript
# Inherit from 'Character.gd'.

extends "res://path/to/character.gd"

# Load character.gd and create a new node instance from it.

var Character = load("res://path/to/character.gd")
var character_node = Character.new()
```

You can give your class a name to register it as a new type in Godot's editor. For that, you use the class_name keyword. You can optionally: add a comma followed by a path to an image/ use the @icon annotation with the path to show it as an icon. Your class will then appear with its new icon in the editor:

```gdscript
# Item.gd

extends Node
class_name Item
@icon("res://interface/icons/item.png")
```

```gdscript
# Item.gd

extends Node
class_name Item, "res://interface/icons/item.png"
```

* GDScript 1 * GDScript 2.0

Classes

exemplo

```gdscript
# Saved as a file named 'character.gd'.

class_name Character


var health = 5


func print_health():
    print(health)


func print_this_script_three_times():
    print(get_script())
    print(ResourceLoader.load("res://character.gd"))
    print(Character)
```

Note

Godot's class syntax is compact: it can only contain member variables or functions. You can use static functions, but not static member variables. In the same way, the engine initializes variables every time you create an instance, and this includes arrays and dictionaries. This is in the spirit of thread safety, since scripts can be initialized in separate threads without the user knowing.

Classes

herança

A class (stored as a file) can inherit from:

- A global class.
- Another class file.
- An inner class inside another class file.

Multiple inheritance is **not** allowed.

Inheritance uses the extends keyword:

```
# Inherit/extend a globally available class.
extends SomeClass

# Inherit/extend a named class file.
extends "somefile.gd"

# Inherit/extend an inner class in another file.
extends "somefile.gd".SomeInnerClass
```

Classes

classes como recursos

Classes stored as files are treated as resources. They must be loaded from disk to access them in other classes.
This is done using either the load or preload functions (see below). Instancing of a loaded class resource is done by calling the new function on the class object:

```gdscript
# Load the class resource when calling load().
var MyClass = load("myclass.gd")

# Preload the class only once at compile time.
const MyClass = preload("myclass.gd")


func _init():
    var a = MyClass.new()
    a.some_function()
```

Classes

setters & getters

Em GDScript 1, a palavra chave setget pode ser usada para definir funções assessoras para determinada propriedade de uma classe.

```
var variable = value setget setterfunc, getterfunc
```

Sempre que o valor da variável for alterada por uma fonte externa, a função *set* será chamada. O *setter* deve definir o que será feito com a variável a qual está vinculado. A mesma ideia se aplica a quando uma variável é acessada (o método *getter* é ativado, e deve retornar um valor associado).

```
# Only a setter.
var my_var = 5 setget my_var_set
# Only a getter (note the comma).
var my_var = 5 setget ,my_var_get
```

O método setter (ou getter) pode ser omitido, caso não se deseje declará-lo.

Classes

setters & getters

Em GDScript 2.0, métodos assessores são definidos diretamente com a definição do atributo:

```
var milliseconds: int = 0
var seconds: int:
    get:
        return milliseconds / 1000
    set(value):
        milliseconds = value * 1000
```

No caso em que se deseja repetir o bloco do getter ou setter para ser usada de forma compartilhada por outras propriedades, ou repetir a lógica em fragmentos do código, você pode usar a notação abaixo para vincular a propriedade à funções existentes:

```
var my_prop:
    get = get_my_prop, set = set_my_prop
```

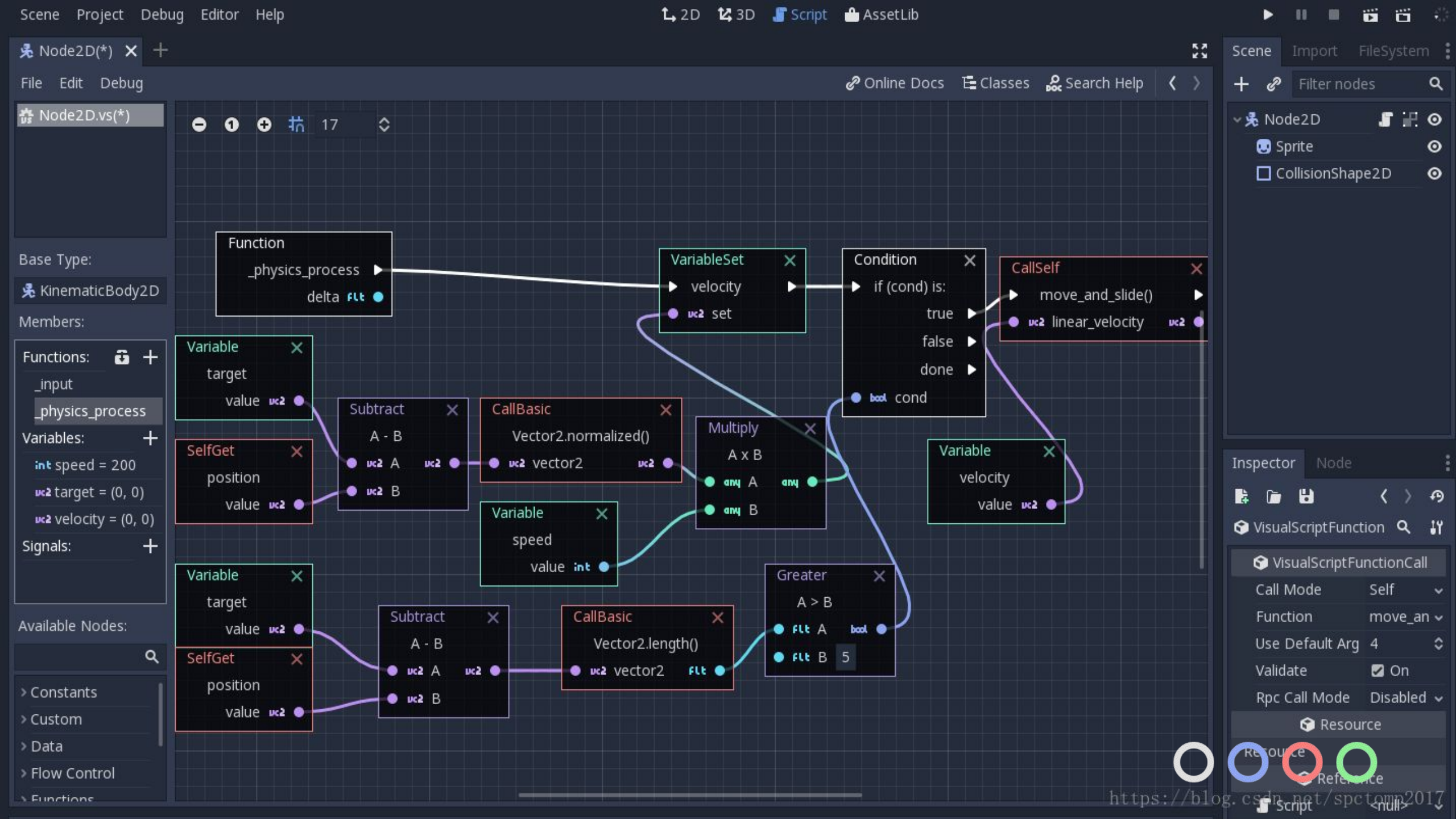
Bibliotecas built-in

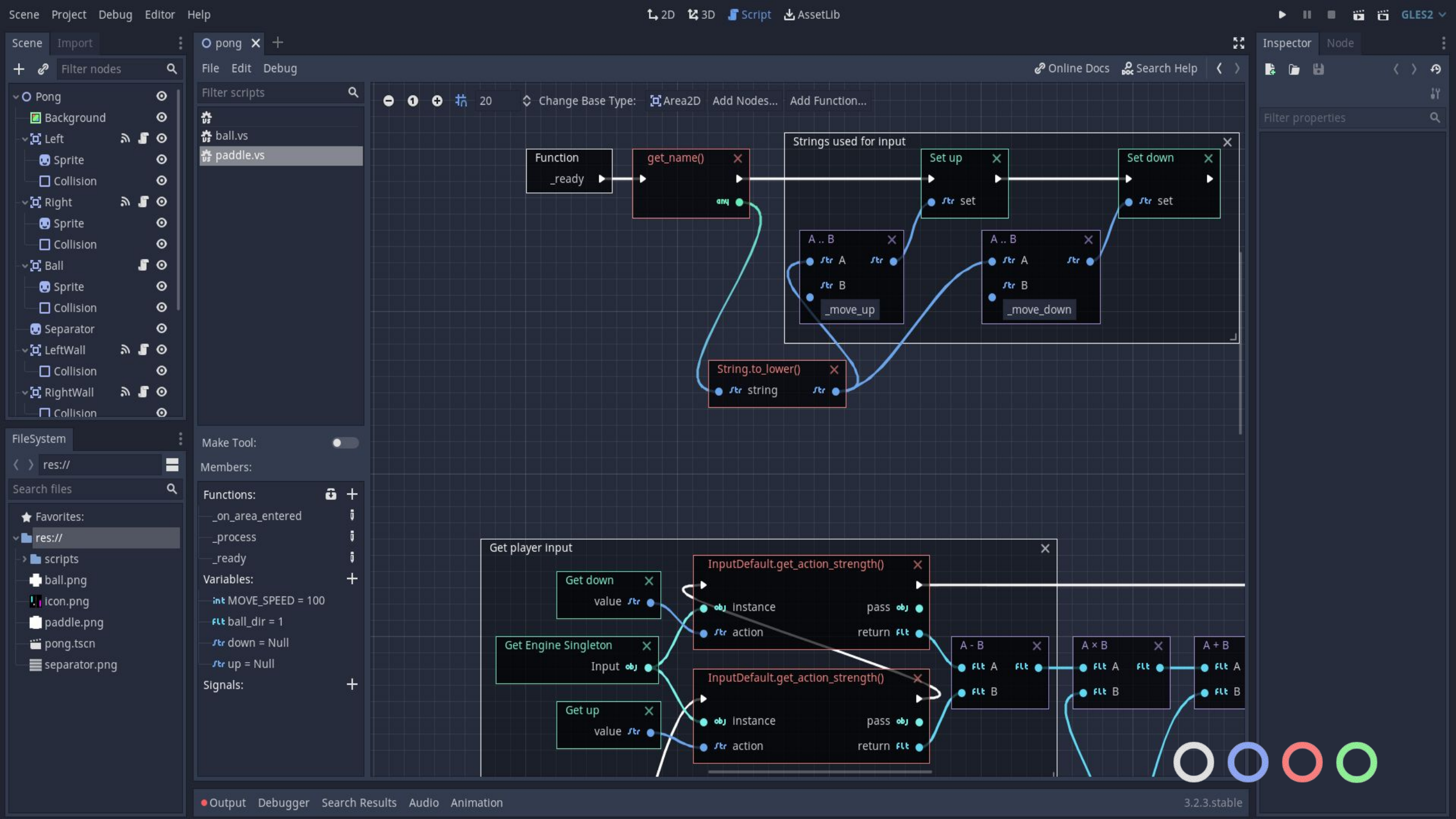
GDScript é carregado com uma série de funcionalidades contidas em “bibliotecas” (classes) incluídas no motor.

- Matemática: Geometry, RandomNumberGenerator, NoiseShaders...
- I/O: Input, InputEvent, InputMap...
- Arquivos e assets: File, JSON, ConfigFile, Directory, Themes, AtlasTexture, TileSets, TileMaps e GridMaps...
- (IA para jogos): Astar (pathfinding) + Astar2D (Godot 4.+), NavigationServer...
- Cinemática: nós Tween, Timer, AnimationPlayer , AnimationTreePlayer e outros nós e recursos.
- Runtime: AudioStreamPlayer +2D e 3D, AudioServer, LanguageServer, PhysicsServer, VisualServer (Godot 4.+), OS...
- Plugin: dentre os principais GodotXR (para AR e VR).

GDScript to Visual Language

Godot também possui uma DSL alternativa à GDScript, semelhante a linguagem em termos de funcionalidade e estrutura, traduzida em uma interface de blocos e fios:
A linguagem VisualScript.


Scene Project Debug Editor Help
2D 3D Script AssetLib
Node2D(*)
File Edit Debug
Online Docs Classes Search Help
Node2D.vs(*)
Base Type:
KinematicBody2D
Members:
Functions:
_input
_physics_process
Variables:
int speed = 200
vec2 target = (0, 0)
vec2 velocity = (0, 0)
Signals:
Available Nodes:
Constants
Custom
Data
Flow Control
Functions
Function
_physics_process
delta flt
VariableSet
velocity
vec2 set
Condition
if (cond) is:
true
false
done
bool cond
CallSelf
move_and_slide()
vec2 linear_velocity vec2
Variable
target
value vec2
Subtract
A - B
vec2 A vec2
vec2 B
CallBasic
Vector2.normalized()
vec2 vector2 vec2
Multiply
A x B
any A any
any B
SelfGet
position
value vec2
Variable
velocity
value vec2
Variable
speed
value int
Variable
target
value vec2
Subtract
A - B
vec2 A vec2
vec2 B
CallBasic
Vector2.length()
vec2 vector2 flt
Greater
A > B
flt A bool
flt B 5
SelfGet
position
value vec2
Scene Import FileSystem
Filter nodes
Node2D
Sprite
CollisionShape2D
Inspector Node
VisualScriptFunction
VisualScriptFunctionCall
Call Mode Self
Function move_an
Use Default Arg 4
Validate On
Rpc Call Mode Disabled
Resource
Script <null>
https://blog.csdn.net/spctomp2017


Scene Project Debug Editor Help
2D 3D Script AssetLib
GLES2
Scene Import
Filter nodes
Pong
Background
Left
Sprite
Collision
Right
Sprite
Collision
Ball
Sprite
Collision
Separator
LeftWall
Collision
RightWall
Collision
FileSystem
res://
Search files
Favorites:
res://
scripts
ball.png
icon.png
paddle.png
pong.tscn
separator.png
pong
File Edit Debug
Online Docs Search Help
Filter scripts
ball.vs
paddle.vs
Make Tool:
Members:
Functions:
_on_area_entered
_process
_ready
Variables:
int MOVE_SPEED = 100
flt ball_dir = 1
str down = Null
str up = Null
Signals:
20
Change Base Type: Area2D
Add Nodes...
Add Function...
Function
_ready
get_name()
any
Strings used for input
Set up
str set
Set down
str set
A .. B
str A
str B
_move_up
A .. B
str A
str B
_move_down
String.to_lower()
str string
Get player input
Get down
value str
Get Engine Singleton
Input obj
Get up
value str
InputDefault.get_action_strength()
obj instance
str action
pass obj
return flt
InputDefault.get_action_strength()
obj instance
str action
pass obj
return flt
A - B
flt A
flt B
A × B
flt A
flt B
A + B
flt A
flt B
Inspector Node
Filter properties
Output Debugger Search Results Audio Animation
3.2.3.stable

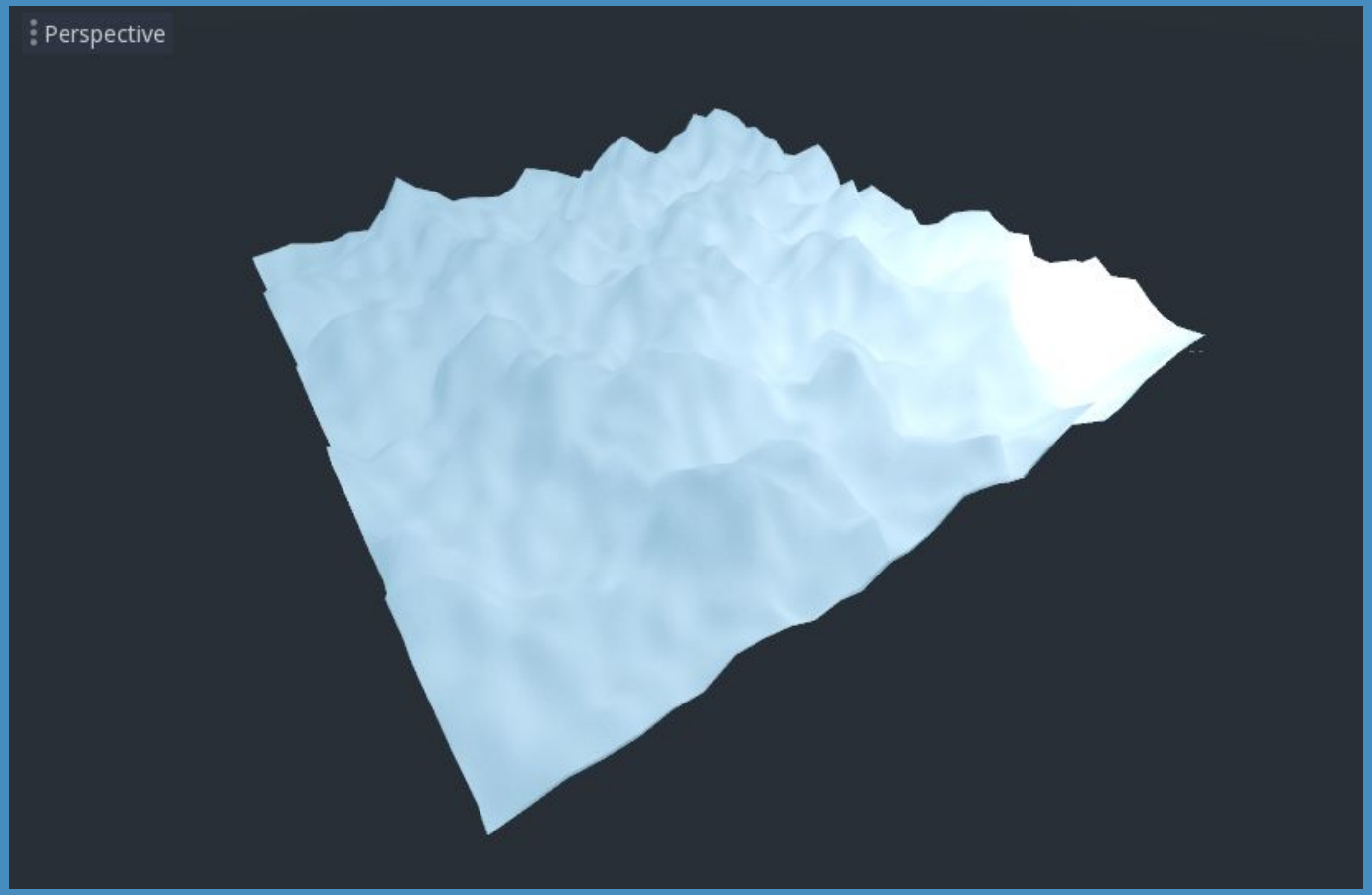
Godot shaders

Além disso, Godot também suporta uma DSL especializada na criação de *shaders* (fragmentos de códigos que rodam exclusivamente na GPU).

Sua sintaxe é idêntica à C, porém sem ponteiros.
Executado por meio de funções especiais, e possui diversos parâmetros relativos ao tipo de gráfico que está processando.
Por exemplo: fragment que é chamada para cada pixel da textura que está associada.

```
shader_type spatial;

uniform float height_scale = 0.5;
uniform sampler2D noise;
uniform sampler2D normalmap;

varying vec2 tex_position;

void vertex() {
  tex_position = VERTEX.xz / 2.0 + 0.5;
  float height = texture(noise, tex_position).x;
  VERTEX.y += height * height_scale;
}

void fragment() {
  NORMAL_MAP = texture(normalmap, tex_position).xyz;
}
```

Godot shaders



Godot bindings

É possível vincular Godot e GDScript com linguagens baseadas em C utilizando o sistema de extensão do motor:
GDNative (GDS 1) | GDExtension (GDS 2.0).

Linguagens atualmente disponíveis:
C, C++ (oficiais);
D, Kotlin, Nim, Python, Rust (não-oficiais).

C# usa uma versão especial da engine carregada com o motor (compilador) Mono Develop.

Recomendações

GODOT

Showcase

Godot Brasil - Discord

Godot Engine – Discord

Imagem: Sonic Colors Ultimate (jogo portado em Godot em 2021)

Apresentação disponível em: github.com/GersonFeDutra/gdscript-language

Referências

Godot Docs – 3.4 branch. Godot Docs, 2014-2022. Disponível em https://docs.godotengine.org/en/stable/. Acesso em 05 de maio - 25 de Junho de 2022.

Godot Docs – master branch — Godot Docs (latest), 2014-2022. Disponível em https://docs.godotengine.org/en/latest/. Acesso em 05 de maio - 22 de Junho de 2022.

Script Languages. Godot Docs, 2014. Disponível em https://docs.godotengine.org/en/stable/getting_started/step_by_step/scripting_languages.html#gdscript. Acesso em 05 de Maio - 22 de Junho de 2022.

GDScript Basics. Godot Docs, 2014. Disponível em https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html#doc-gdscript. Acesso em 05 de Maio - 22 de Junho de 2022.

A documentação, em especial conteúdo do master branch, é atualizada constantemente, portanto podem não corresponder ao conteúdo usado nessa apresentação.