FICHA TÉCNICA

AUTOR

Equipe de Desenvolvimento Gillis Interactive

EDIÇÃO TÉCNICA

Coordenação de Desenvolvimento Gillis Interactive

REVISÃO & CONTROLE DE QUALIDADE

Equipe de Desenvolvimento

DESENVOLVEDOR DE GAMES COM GODOT 3.0 MD. I

Des. de Games com Godot 3.0 – Módulo I / Gillis Interactive, 2021

Edição: 1ª

Idioma: português

Em conformidade com o Acordo Ortográfico da Língua Portuguesa.

Sumário

AULA 01 5

INTRODUÇÃO 5

Instalação 6

Instalando os templates de exportação 7

Introdução ao Editor da Godot 8

Gerenciador de Projetos 8

Criar ou importar um projeto 9

Seu primeiro olhar no editor do Godot 11

Os Espaços de Trabalho 13

Modificar a interface 16

Mover e redimensionar painéis 16

AULA 02 18

CENAS E NÓS 18

Nós 18

Cenas 19

AULA 03 21

INSTÂNCIAS 21

AULA 04 24

LINGUAGEM DE CONCEPÇÃO 24

Sobrecarga de informações 26

AULA 05 28

SCRIPTING 28

GDScript 28

VisualScript 29

.NET / C# 30

GDNative / C++ 30

O papel do roteiro 31

Manipulando um sinal 31

AULA 06 32

PROCESSAMENTOS 32

Grupos 34

Notificações 34

Criando (Nós) 35

Criando instâncias de cenas 36

Registrar scripts como classes 37

AULA 07 40

SINAIS 40
Conectando sinais por código *40*
Sinais personalizados *40*
AULA 08 43
SEU PRIMEIRO JOGO 43
Configuração do projeto *43*
Organizando o projeto *44*
Cena do Jogador *45*
Estrutura de nós *45*
Animação por Sprite *46*
Movendo Jogador *49*
Selecionado as Animações *53*
Preparando para colisões *54*
AULA 09 57
CENA DO INIMIGO 57
Configuração de nós *57*
Salvar Cena: Script do inimigo *59*
AULA 10 61
CENA PRINCIPAL 61
Gerando monstros *62*
Script principal *64*
Testando a cena *67*
AULA 11 68
HUD 68
Conectando HUD a Principal *73*
AULA 12 76
REMOVENDO ANTIGAS CRIATURAS 76
Terminando *77*
Plano de fundo *77*
Efeitos sonoros *78*
Atalho de teclado *78*
AULA 13 80
EXPORTANDO 80
Preparando o projeto *80*
Definindo uma cena principal *84*
Exportar modelos *84*
Exportando por plataforma *86*

CONTEÚDO EXTRA 88
Filosofia de Design do Godot 88
Design e composição orientada a objetos 88
Engine 2D e 3D separados 91
Design de interfaces com os nós de Controle 91
TextureRect 93
TextureButton 94
TextureProgress 95
Rótulo 95
NinePatchRect 96
Como alterar a âncora 97
Âncoras são relativas ao contêiner pai 98
As margens mudam com a âncora 99
Organize nós de controle automaticamente com contêineres 101
Os 5 contêineres mais úteis 101
O MarginContainer adiciona uma margem de 40px ao redor da interface do usuário do jogo 102
O HBoxContainer alinha horizontalmente os elementos da interface do usuário 103
Nós e recursos 103
Externo vs embutido 104
Carregando recursos a partir do código 106
Carregando cenas 107
Liberando recursos 107
Sistema de arquivos 108
Implementação 108
Delimitador de caminho 109
Caminho de recursos 109
Sistema de arquivos da máquina 109
Desvantagens 110
Árvore de cena 111
Viewport raiz 111
Alterando a cena atual 113
Singletons (Carregamento Automático) 114

AULA 01

Introdução

Neste curso vamos aprender a criar jogos e compartilhar com as principais plataformas com a ferramenta 100% gratuita: Godot Engine. Vamos aprender desde os primeiros passos na aventura da programação e construção de um jogo totalmente funcional para as principais plataformas da atualidade.

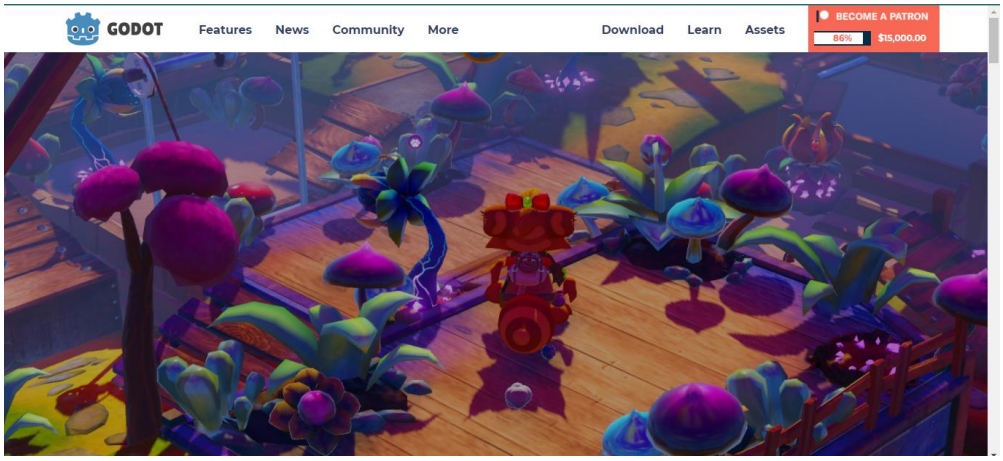
O curso será dividido em 3 módulos, criaremos diversos jogos, desde os mais simples até os jogos mais complexos, progredindo e absorvendo os conhecimentos à medida que vemos os resultados reais do nosso trabalho.

Utilizando a Godot Engine, uma máquina de jogos de código aberto e muito poderosa, o aluno vai aprender a configurar e desenvolver jogos desde os primeiros passos, permitindo a alunos sem experiência dar início ao mundo do desenvolvimento de jogos, tirando proveito do poder da engine utilizando o mínimo de programação possível.

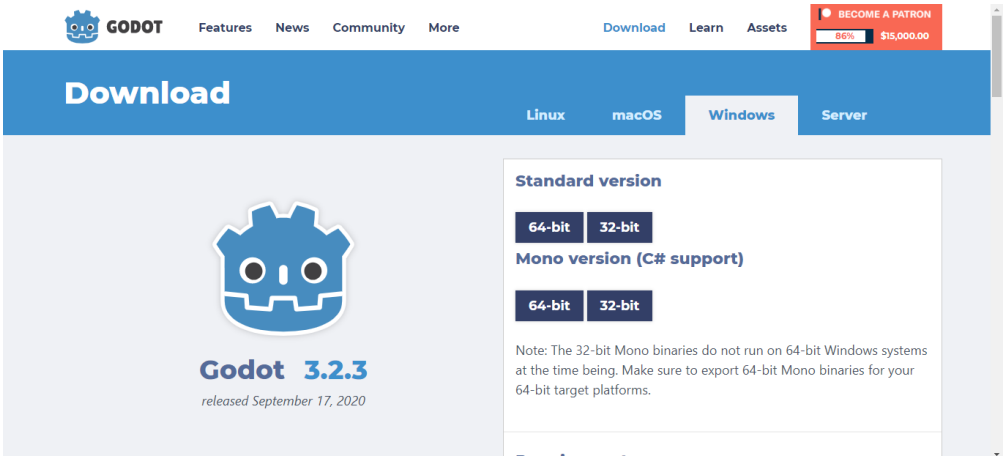
Para utilizar a Godot você vai precisar de conhecimentos básicos de informática, um PC com placa de vídeo que suporte a Godot e vontade de aprender programação. Com o Godot você consegue desenvolver nos sistemas operacionais Windows, Linux e Mac.

Instalação

Para instalar é simples, acessamos o site da Godot https://godotengine.org/ e clicamos em Download.



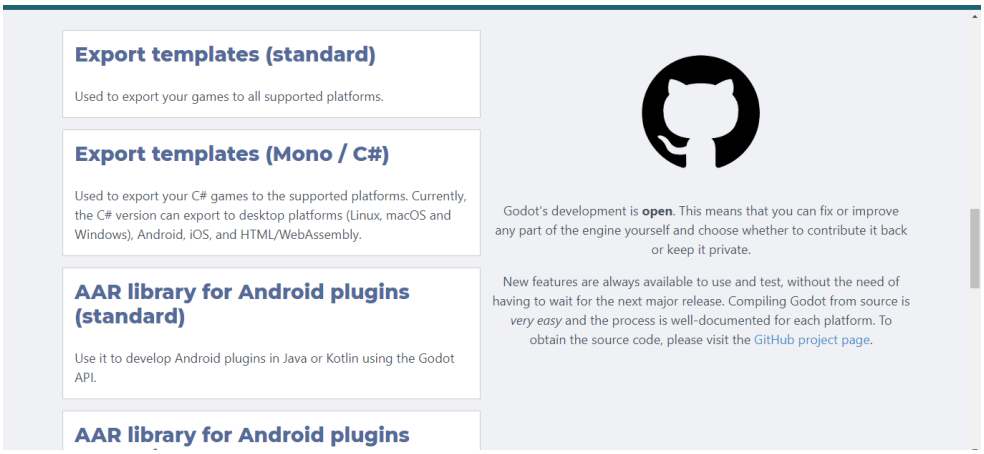
Na aba de download vamos ter diversas opções para os sistemas operacionais disponíveis, você seleciona o sistema operacional e a sua versão.



Você efetua o download e armazena o arquivo no local desejado, após isso iremos obter um arquivo no formato ZIP, descompactando esse arquivo vamos obter o executável, bastando dar um duplo clique nele para abrir a Godot. É simples e muito rápido.

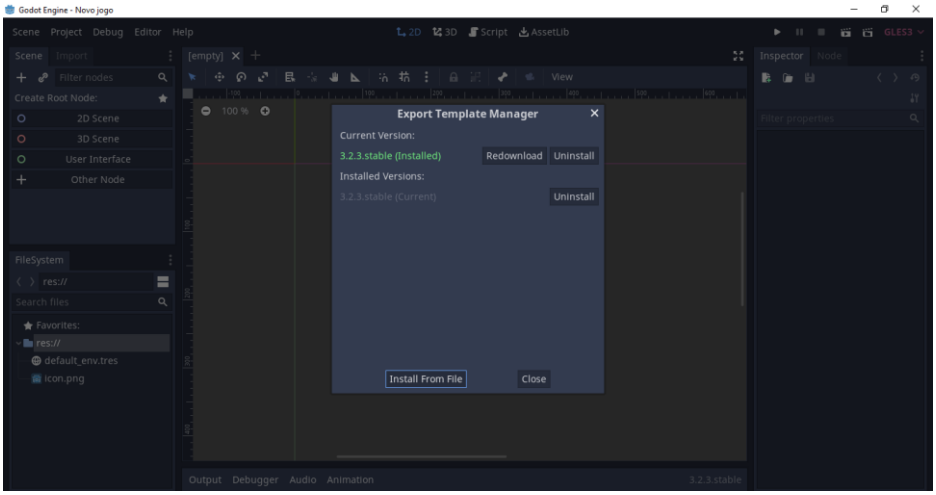
Instalando os *templates* de exportação

Para que futuramente possamos exportar nossos projetos para outros projetos, é necessário instalar na Godot os *templates* de exportação. Na página de download encontramos essa opção, a export templates.



Basta efetuar o download dos templates e uma vez feito o download é só abrir a Godot e qualquer projeto aberto.

Para instalar vá em Editor > Manage Export Templates > Install From File > Selecione o Template que foi realizado Download > Open.

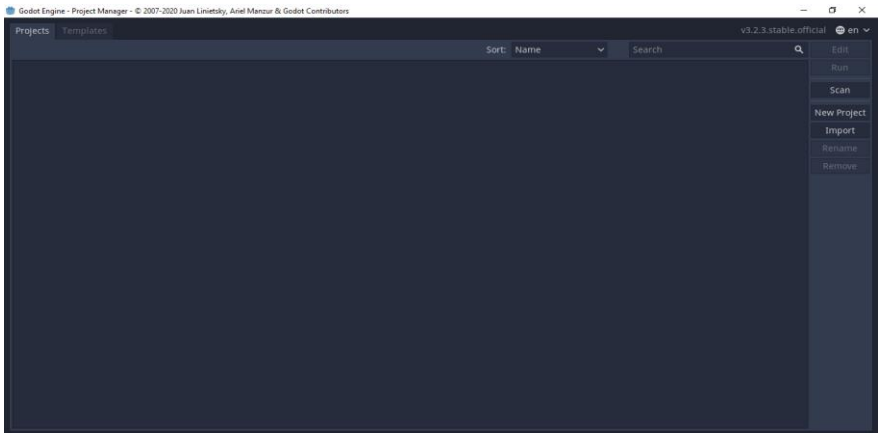


Introdução ao Editor da Godot

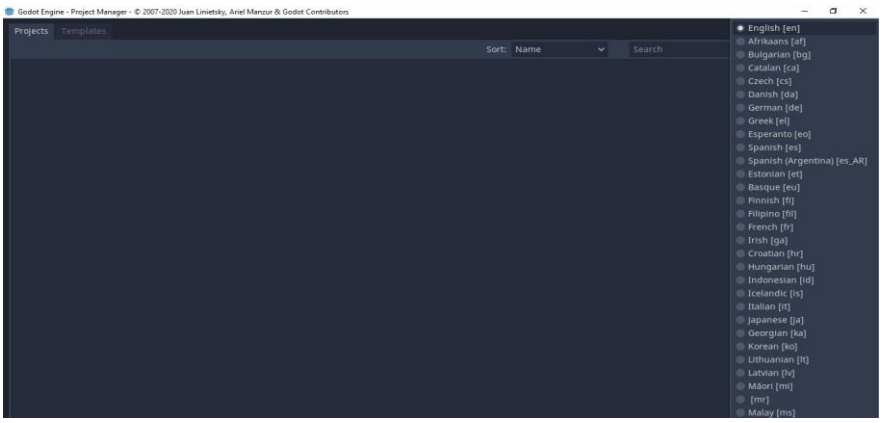
Agora vamos apresentar a interface da Godot, vamos dar uma olhada no gerenciador de projetos, painéis, área de trabalho e tudo que você precisa saber para utilizar a Engine.

Gerenciador de Projetos

O gerenciador de projetos, permite você criar, remover, importar ou jogar projetos de jogos.

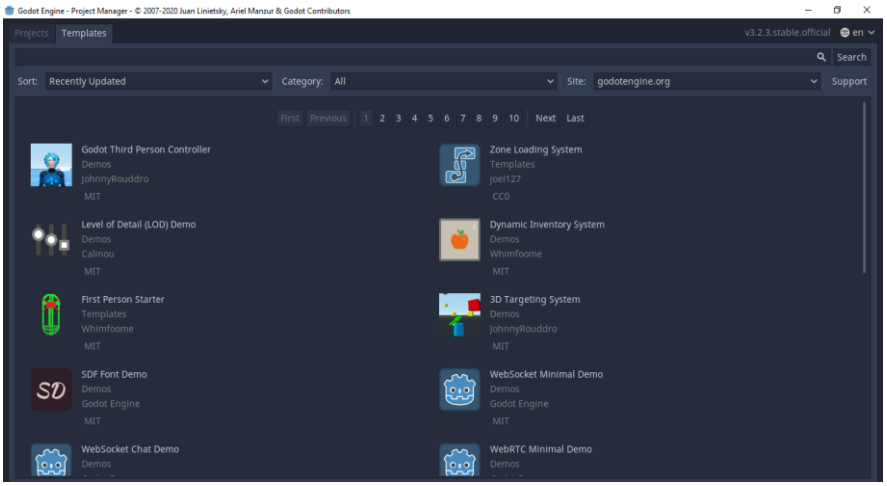


No canto superior direito, você encontra um menu suspenso para alterar o idioma do editor.



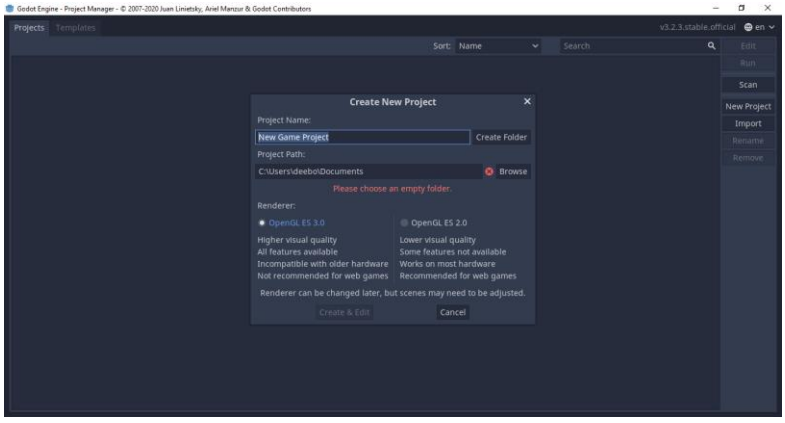
A partir da guia Templates, você pode baixar modelos e demonstrações de projeto de código aberto para lhe ajudar a aprender mais rápido. Apenas selecione o modelo ou demo desejado, clique em baixar, uma vez que tenha

terminado de baixar, clique em instalar e escolha onde você quer que o projeto fique. Você pode aprender mais sobre isso em Sobre a Biblioteca de Assets.

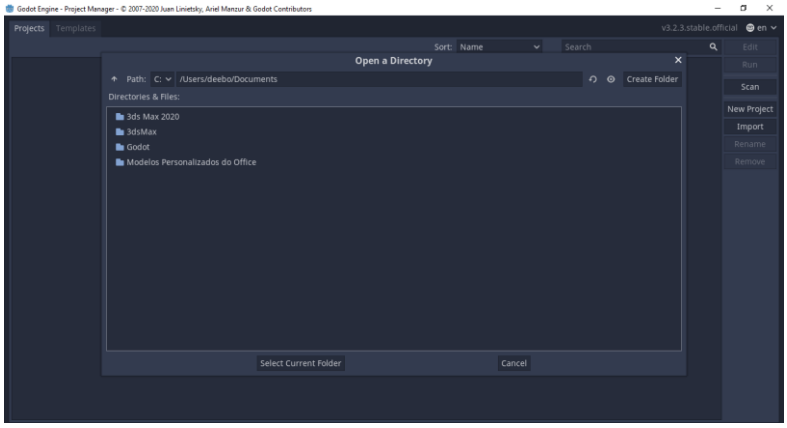


Criar ou importar um projeto

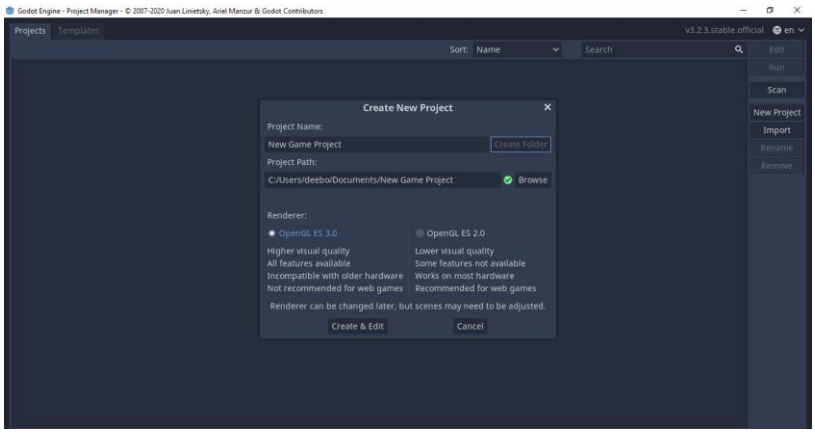
Para criar um projeto, clique no botão New Project à direita. Dê um nome, escolha uma pasta vazia do seu computador para salvá-lo e escolha um renderizador.



Clique no botão Browse para abrir o navegador de arquivos e escolher um local, ou digite o caminho da pasta no campo Caminho do Projeto.

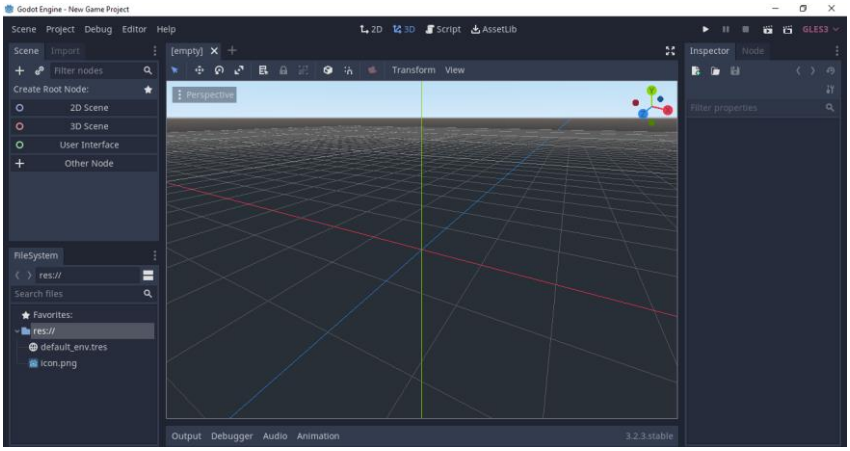


Quando você vir uma marca verde na direita, significa que a engine detectou uma pasta vazia. Você também pode clicar no botão Criar Pasta perto do nome do seu projeto e uma pasta vazia vai ser criada com aquele nome para o projeto.

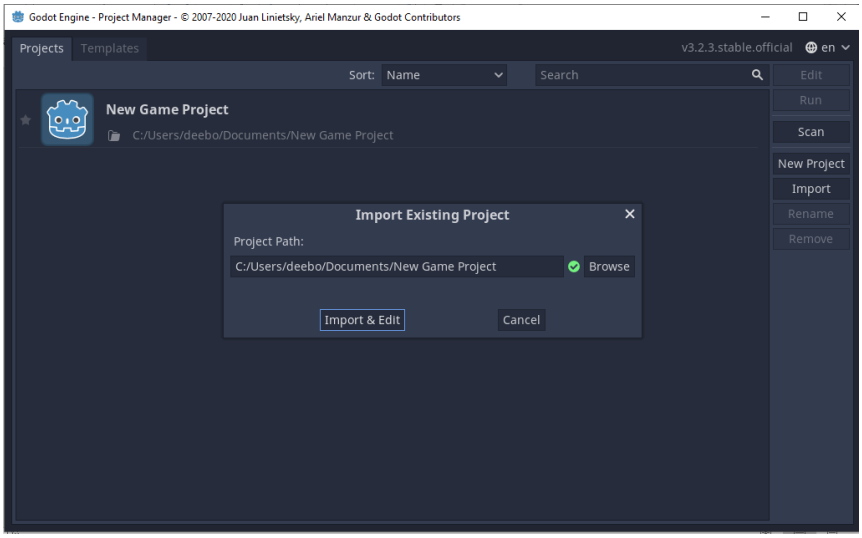


Por último, você precisa escolher qual renderizador usar (OpenGL ES 3.0 ou OpenGL ES 2.0). As vantagens e desvantagens de cada um estão listadas para ajudá-lo a escolher.

Quando estiver tudo pronto, clique em Create & Edit. Godot vai criar o projeto para você e abrir ele no editor.



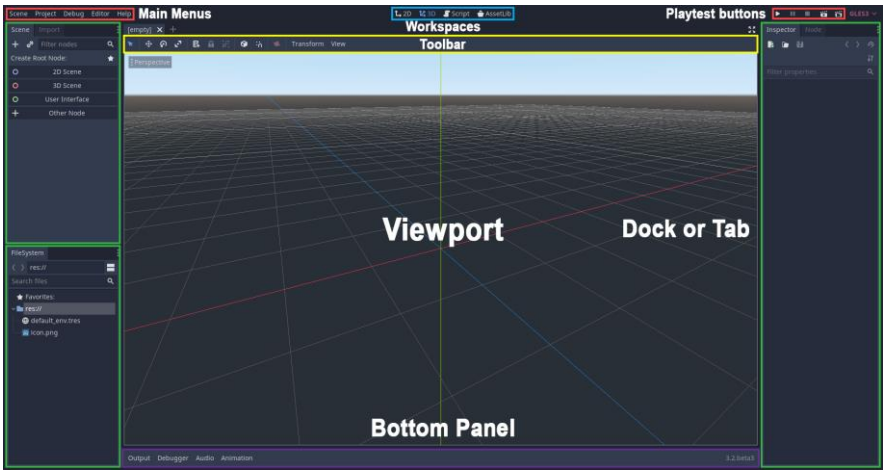
Você pode importar projetos existentes de maneira similar, usando o botão Importar. Localize a pasta que contém o projeto ou o arquivo project.godot para importá-lo e editá-lo.



Quando o caminho da pasta está correto, você vê uma marca verde de confirmação.

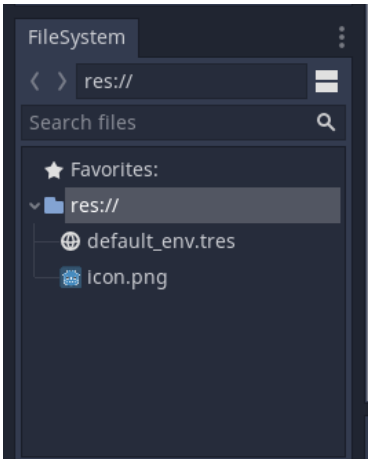
Seu primeiro olhar no editor do Godot

Com seu projeto aberto, você verá a interface do editor com menus ao longo do topo da interface e painéis ao longo de ambas as extremidades laterais da janela de exibição.

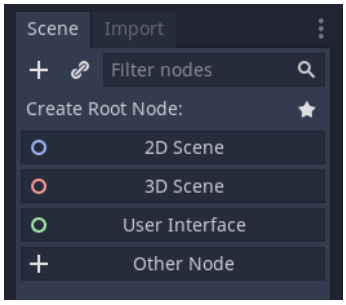


No topo, da esquerda para a direita, você pode ver os menus principais, as workspaces e os botões de teste.

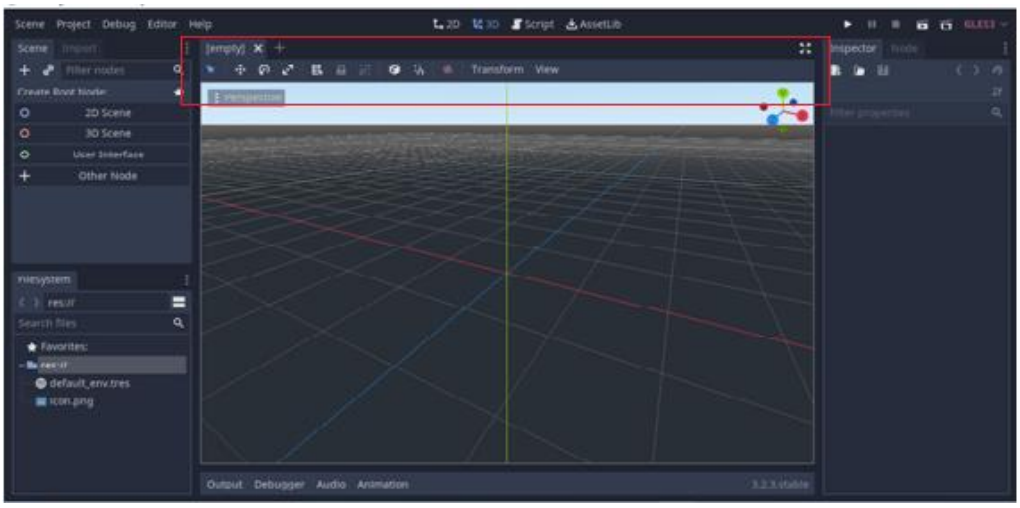
O Painel Arquivos (File System dock) é onde você vai gerenciar seus recursos e arquivos do projeto.



O Painel de Cenas (Scene dock) lista o conteúdo da cena ativa e o Inspetor(Inspector) permite o gerenciamento das propriedades do conteúdo da Cena.



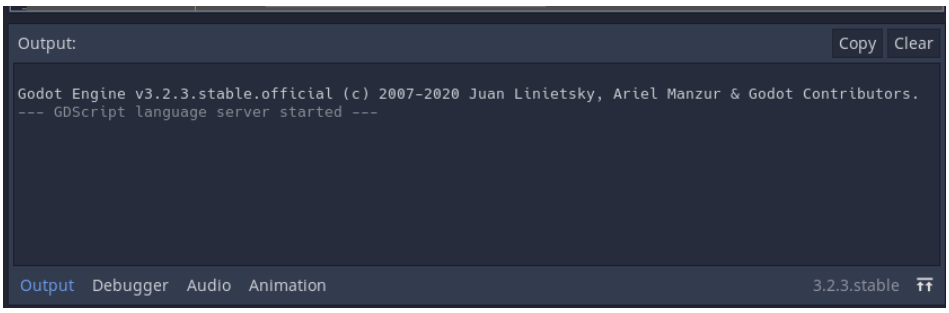
No centro, você tem a Barra de Ferramentas no topo, onde você encontra as ferramentas para mover, redimensionar ou travar objetos da sua cena. A barra se altera conforme você alterna entre diferentes espaços de trabalho.



Barra de Ferramentas 3D

Barra de Ferramentas 2D

O Painel Inferior é onde moram o console de depuração, o editor de animação, o mixer de áudio. Eles são largos e podem ocupar um espaço precioso. Por esse motivo eles ficam recolhidos por padrão.



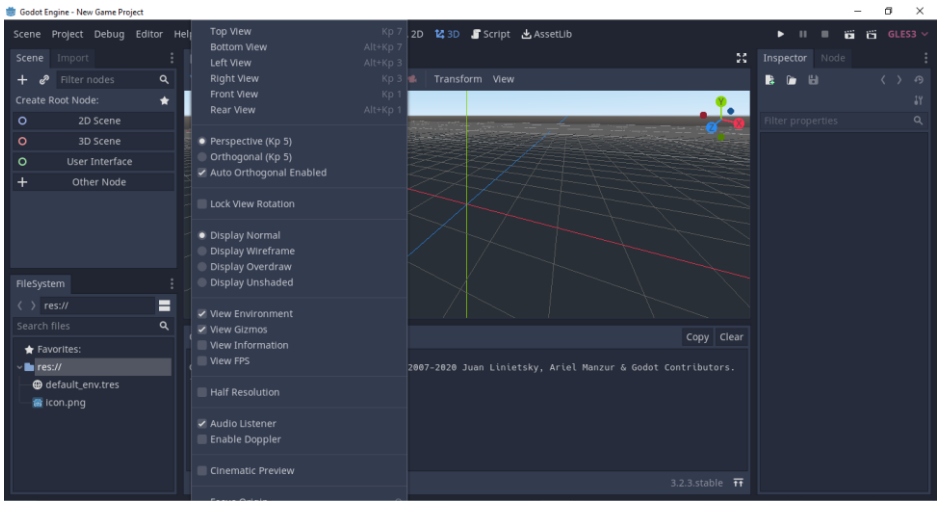
Os Espaços de Trabalho

Você pode ver quatro botões de espaços de trabalho no topo: 2D, 3D, Script e AssetLib.

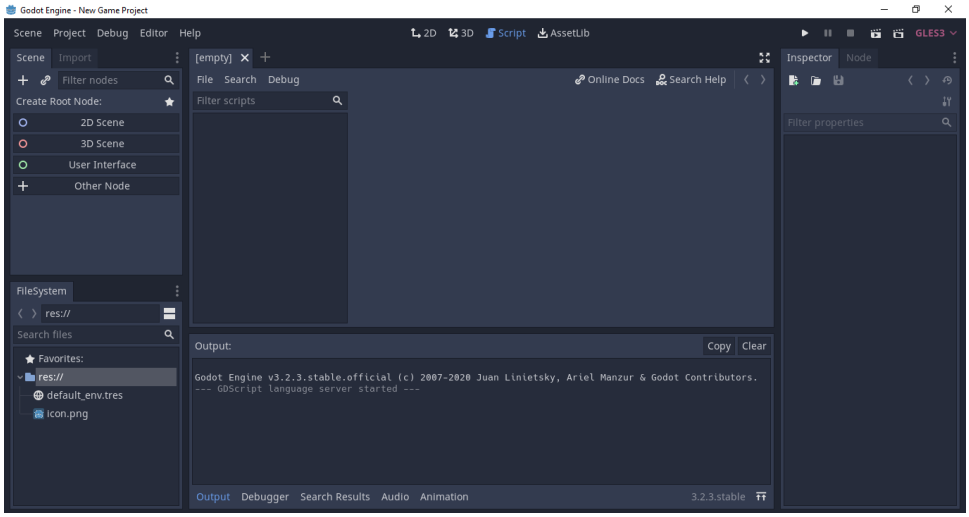
Você usará o Espaço de trabalho 2D para todos os tipos de jogos. Além de ser onde os jogos 2D são desenvolvidos, este espaço de trabalho é onde você pode criar as suas interfaces de usuário (UI). Pressione F1 (ou Alt + 1 no macOS) para acessá-lo.

No Espaço de Trabalho 3D, você pode trabalhar com malhas (meshes), luzes e projetar níveis para jogos 3D. Pressione F2 (ou Alt + 2 no macOS) para acessá-lo.

Repare no botão Perspectiva abaixo da barra de ferramentas, ele abre uma lista de opções relacionadas ao ponto de vista 3D.

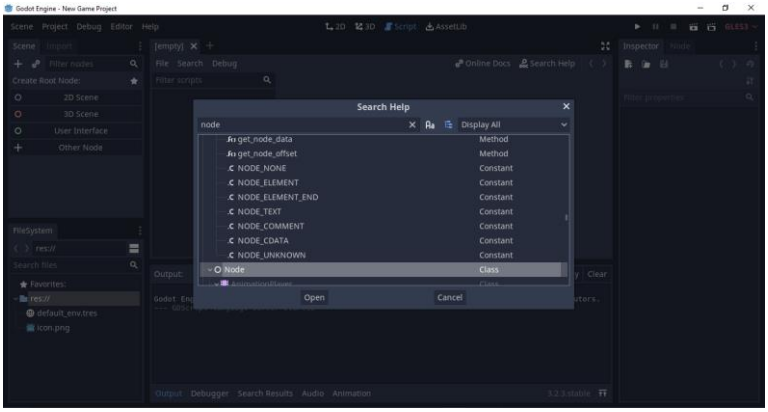


O espaço de trabalho Script é um editor de código completo, com um depurador, um rico autocompletar e referência de código embutida.

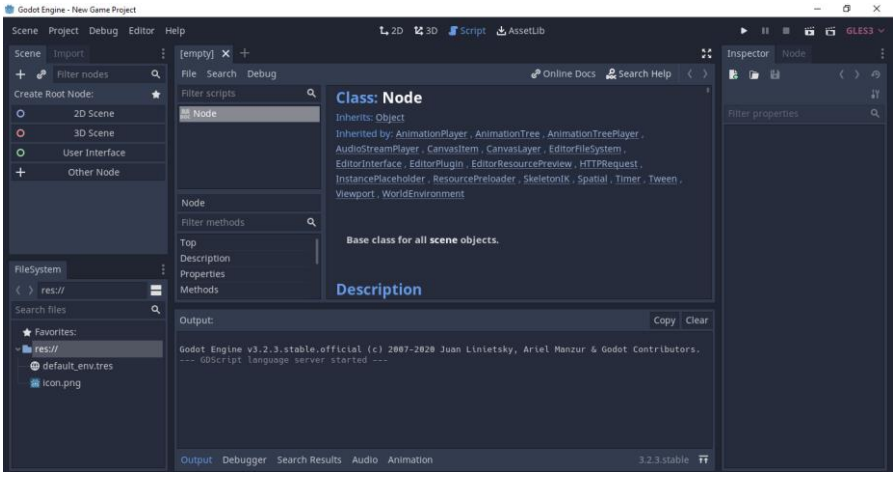


Para procurar informações sobre uma classe, método, propriedade, constante ou sinal na engine enquanto você escreve um script, pressione o botão "Search Help" no canto superior direito do espaço de trabalho de Script.

Uma nova janela surgirá. Procure pelo item que você quer encontrar informações a respeito.



Clique no item que você está procurando e pressione abrir. A documentação para o item será apresentada no espaço de trabalho de script.



Finalmente, o AssetLib é uma biblioteca de complementos, scripts e assets gratuitos para usar nos seus projetos

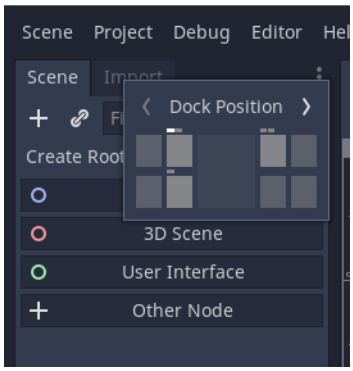
Modificar a interface

A interface do Godot fica em uma única janela. Você não pode dividi-la para várias telas, embora possa trabalhar com um editor de código externo como Atom ou Visual Studio, por exemplo.

Mover e redimensionar painéis

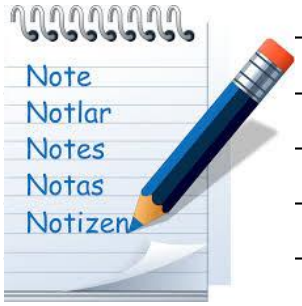
Clique e arraste na borda de qualquer painel para redimensioná-lo horizontal ou verticalmente.

Clique no ícone de três-pontos no topo de qualquer painel para alterar sua localização.



Vá para o menu Editor > Editor Settings para um ajuste na aparência e comportamento.

ANOTAÇÕES



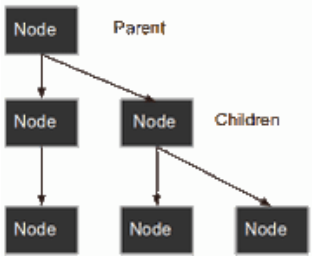
AULA 02

Cenas e Nós

Nós

Os nós (node) são peças fundamentais na criação de um jogo. Como já mencionado, um nó pode desempenhar uma variedade de funções especializadas. Entretanto, todo nó sempre terá os seguintes atributos:

- Tem um nome.
- Tem propriedade editáveis.
- Pode receber uma chamada para processar cada quadro.
- Pode ser estendido (ter mais funções).
- Pode ser adicionado a outros nós como filho.

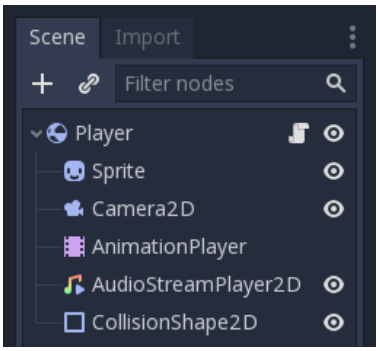


Esse último é importante: **NÓS** podem ter outros **NÓS** como filhos. Quando arranjados dessa maneira, os nós compõem uma árvore.

No Godot, a habilidade de arranjar os nós desta forma, cria uma ferramenta poderosa para organizar projetos. Já que diferentes **NÓS** têm diferentes funções, combiná-los permite a criação de funções mais complexas.

Não se preocupe se ainda não entendeu. Continuaremos a explorar esses conceitos nas próximas seções. O fato mais importante para lembrar por ora é que **NÓS** existem e eles podem ser organizados desta forma.

Cenas



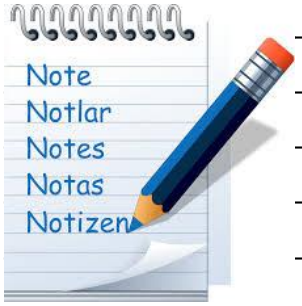
Agora que o conceito de nós foi definido, o próximo passo lógico é explicar o que é uma Cena. Uma cena é composta por um grupo de nós organizados hierarquicamente (no estilo árvore). Além disso, uma cena:

- sempre tem um único nó raiz;
- pode ser salva no disco e carregada posteriormente;
- pode ser instanciada (mais sobre isso depois).

Executar um jogo significa executar uma cena. Um projeto pode conter muitas cenas, mas, para o jogo começar, uma delas deve ser selecionada como cena principal.

Basicamente, o editor do Godot é um editor de cena. Ele tem várias ferramentas para edição de cenas 2D e 3D, assim como interfaces com usuário, mas o editor é baseado no conceito de editar uma cena e os nós que a compõem.

ANOTAÇÕES

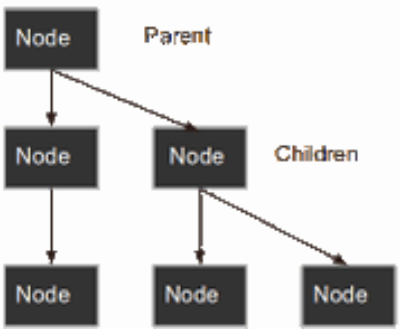


AULA 03

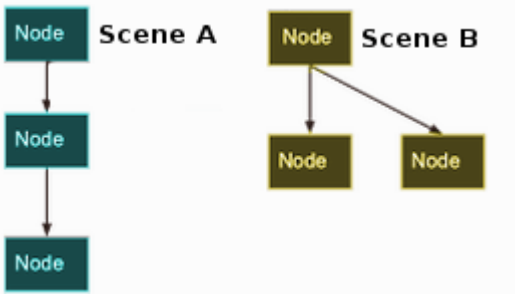
Instâncias

Criar uma única cena e adicionar NÓS a ela, pode funcionar para projetos pequenos, mas, à medida que o projeto cresce em tamanho e complexidade, a quantidade de NÓS pode se tornar impraticável rapidamente. Para resolver isso, o Godot permite que um projeto seja separado em qualquer quantidade de cenas. Isso é uma ferramenta poderosa que lhe ajuda a organizar os diferentes componentes do seu jogo.

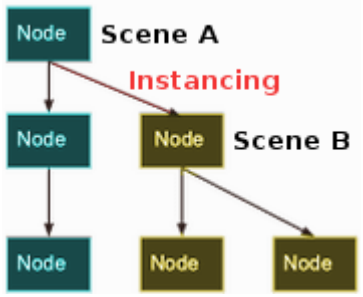
Em Cenas e NÓS, você aprendeu que uma cena é uma coleção de NÓS organizados em uma estrutura de árvore, com um único nó como nó raiz.



Você pode criar quantas cenas quiser e salvá-las no disco. Cenas salvas assim são chamadas de "Cenas Empacotadas" e têm a extensão de arquivo. tscn.



Uma vez salva, a cena pode ser instanciada dentro de outra como se fosse qualquer outro nó.

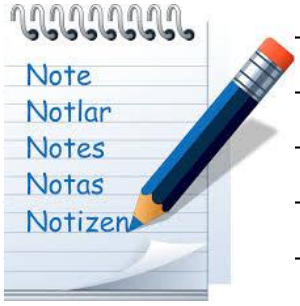


Na figura acima, a Cena B foi adicionada à Cena A como uma instância.

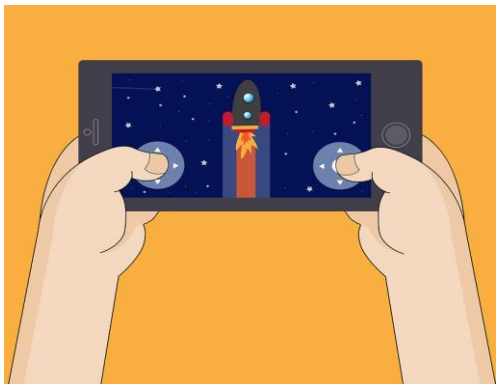
Você pode adicionar quantas instâncias quiser à cena, seja usando o botão "Instância" novamente, ou clicando na instância da bola e pressionando "Duplicar".

Instâncias podem ser úteis quando você quer criar muitas cópias do mesmo objeto. Também é possível criar instâncias em código usando GDScript.

ANOTAÇÕES



AULA 04

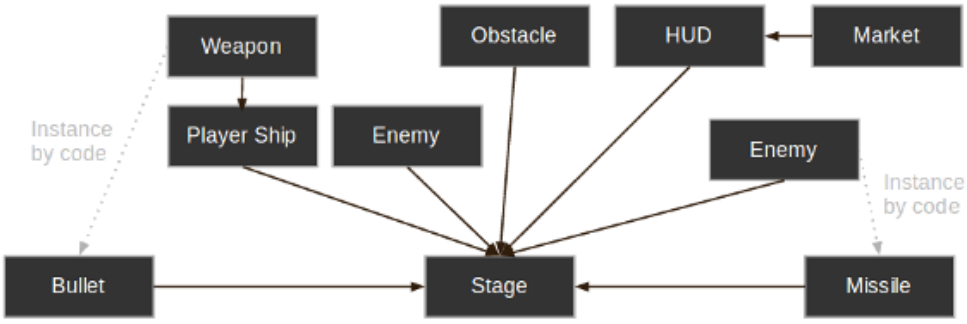


Linguagem de concepção

O maior ponto forte que vem das instâncias de cenas é que isso funciona como uma excelente linguagem para design. Isso é o que mais distingue Godot de todas as outras engines por aí. O Godot foi projetado pensando neste conceito.

Ao fazer jogos com o Godot, a abordagem recomendada é dispensar os padrões de concepção mais comuns, como MVC e diagramas de Entidade-Relacionamento, e em vez disso, pensar nas suas cenas de uma forma mais natural. Comece imaginando os elementos visíveis do seu jogo, aqueles que conseguem ser citados não só pelo programador, mas por qualquer um.

Por exemplo, aqui está como um jogo simples de tiro poderia ser imaginado:

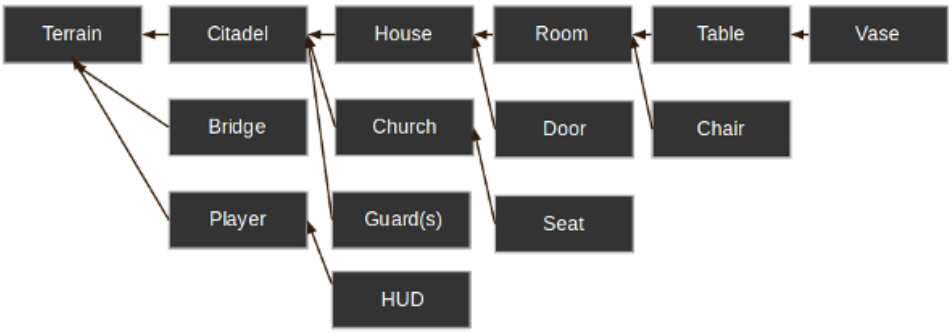


É possível chegar a um diagrama desse tipo para quase qualquer tipo de jogo. Anote as partes do jogo que consegue visualizar e daí adicione setas que representam a posse de um componente por outro.

Com um diagrama desse tipo, o processo recomendado para fazer um jogo é criar uma cena para cada elemento listado no diagrama. Você usará instâncias (seja por código ou diretamente pelo editor) para as relações de posse.

Muito do tempo gasto em programação de jogos (ou de softwares em geral) é projetando a arquitetura e fazendo os componentes do jogo se encaixar nela. O projeto baseado em cenas substitui essa abordagem e torna o desenvolvimento muito mais rápido e mais direto, lhe permitindo se concentrar na lógica do jogo em si. Pelo fato de a maioria dos componentes do jogo serem mapeados diretamente a uma cena, usar projeto baseado em instâncias de cena significa que se fazem necessários poucos códigos extras para arquitetura.

Vamos dar uma olhada em mais um exemplo, um tanto mais complexo, de um jogo do tipo mundo aberto, com muitos ativos e elementos aninhados:



Dê uma olhada no elemento "Room". Digamos que começamos por lá. Nós poderíamos fazer algumas outras cenas de aposentos diferentes, com diferentes disposições dos móveis (que também são cenas) neles. Depois, poderíamos fazer uma cena de casa, conectando aposentos para compor seu interior.

Então, poderíamos fazer a cena de uma cidadela, que é composta por muitas instâncias de casas. Assim, poderíamos começar a trabalhar no terreno do mapa do mundo, adicionando a cidadela nele.

Posteriormente, poderíamos criar cenas que representem guardas (e outros personagens não jogáveis) e adicioná-los na cidadela também. Como resultado, eles seriam indiretamente adicionados no mundo do jogo.

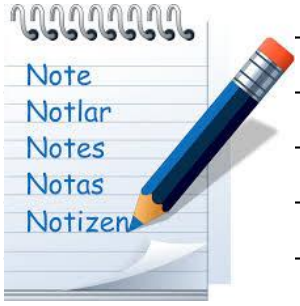
Como Godot, é fácil fazer esse tipo de iteração no seu jogo, já que tudo que você precisa fazer é criar mais cenas e suas instâncias. Além disso, a interface do editor é projetada para ser amigável para programadores e não-programadores. Um processo típico de desenvolvimento em equipe pode envolver artistas 2D e 3D, projetistas de níveis, projetistas de jogo e animadores, todos trabalhando com a interface do editor.

Sobrecarga de informações

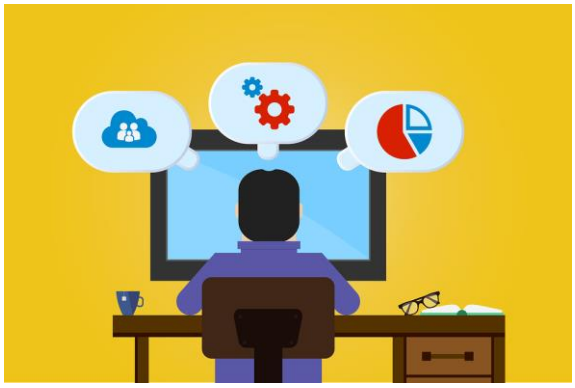
Esta página fez chover muitas informações de alto nível de uma vez na sua cabeça. Entretanto, a parte importante deste tutorial é conscientizar como as cenas e suas instâncias são usadas em projetos reais.

Tudo discutido aqui se tornará natural assim que você começar a fazer jogos e colocar esses conceitos em prática.

ANOTAÇÕES



AULA 05



Scripting

Antes do Godot 3.0, a única maneira de roteirizar um jogo era usar doc_gdscript. Agora, Godot tem quatro linguagens oficiais e a habilidade de adicionar linguagens de roteirização extras dinamicamente.

Isso é ótimo, principalmente por causa da enorme flexibilidade oferecida, mas também dificulta nosso trabalho de dar suporte às linguagens.

As linguagens "principais" no Godot, entretanto, são a GDScript e a VisualScript. O principal motivo para escolhê-las é seu nível de integração com o Godot, já que isso torna a experiência mais suave: ambas têm uma integração boa com o editor, enquanto, C# e C++ precisam ser editadas em uma IDE separada. Se você é fã de linguagens estaticamente tipadas, vá de C# ou C++.

GDScript

GDScript é, como já mencionado, a principal linguagem usada no Godot. Usá-la tem alguns pontos positivos quando comparada às demais dada sua alta integração com o Godot:

- É simples, elegante e projetada para ser familiar aos usuários de outras linguagens como Lua, Python, Squirrel, etc.
- Carrega e compila rapidamente.

- A integração com o editor é agradável de trabalhar, com completação de código para NÓS, sinais e muitos outros itens pertinentes à cena em edição.
- Tem tipos de vetores embutidos (como Vectors, transformações etc.), tornando-a eficiente para uso pesado de álgebra linear.
- Suporte a múltiplas threads tão eficientemente quanto como em linguagens tipadas estaticamente – uma das limitações que nos fez evitar máquinas virtuais como Lua, Squirrel etc.
- Não usa coletor de lixo, então ele troca um pouquinho de automação (a maioria dos objetos usa referências contadas, de qualquer forma) por determinismo.
- Sua natureza dinâmica torna fácil otimizar seções de código em C++ (via GDNative) se mais desempenho for necessário, sem ter que recompilar o motor de jogo.

Se está indeciso e tiver experiência com programação, especialmente com linguagens tipada dinamicamente, opte por GDScript.

VisualScript

A partir da versão 3.0, o Godot oferece Visual Scripting. Esta é uma implementação típica de uma linguagem de "blocos e conexões", mas adaptada para o jeito que o Godot funciona.

Roteirização visual é uma ótima ferramenta para não programadores, ou mesmo para desenvolvedores experientes que queiram tornar partes do código mais acessíveis para outros, como projetistas de jogos e artistas.

Também pode ser usado por programadores para construir máquinas de estado ou fluxos de trabalhos de NÓS visuais personalizados - por exemplo, um sistema de diálogo.

.NET / C#

C# é uma linguagem madura, com toneladas de código escritos para ela e cujo suporte foi adicionado graças a uma generosa doação da Microsoft.

Ela tem uma excelente relação de compromisso entre desempenho e facilidade de uso, embora tenha que se atentar para seu coletor de lixo.

Já que o Godot usa plataforma .NET Mono, em teoria qualquer biblioteca ou infraestrutura .NET terceirizada pode ser usada para roteirizar no Godot, assim como qualquer linguagem de programação compatível com a Infraestrutura de Linguagem Comum (CLI), tais como F#, Boo ou ClojureCLR. Na prática, entretanto, C# é a única opção .NET com suporte oficial.

GDNative / C++

Finalmente, uma das nossas adições mais brilhantes à versão 3.0: GDNative permite roteirizar em C++ sem precisar recompilar (ou mesmo reiniciar) o Godot.

Qualquer versão da C++ pode ser usada, e misturar marcas e versões de compiladores para as bibliotecas compartilhadas geradas funciona perfeitamente; tudo graças ao uso de uma ponte de API C interna.

Esta linguagem é a melhor escolha para performance e não precisa ser usada no jogo todo, já que outras partes podem ser escritas em GDScript ou

Visual Script. Entretanto, a API é clara e fácil de usar, porque se assemelha, em sua maior parte, à API C++ do Godot.

Mais linguagens podem ser disponibilizadas através da interface GDNative, mas tenha em mente que elas não têm suporte oficial.

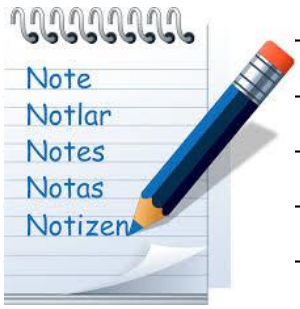
O papel do roteiro

Um roteiro adiciona comportamento a um nó. É usado para controlar como o nó funciona e, também como ele interage com outros NÓS: filhos, pais, irmãos e assim por diante. O escopo local do roteiro é o nó. Em outras palavras, o roteiro herda as funções fornecidas por tal nó.

Manipulando um sinal

Sinais são "emitidos" quando algum tipo específico de ação acontece e eles podem ser conectados a qualquer função de qualquer instância de script. Sinais são usados majoritariamente em NÓS de interface gráfica – embora outros NÓS também os tenham – e você pode até mesmo definir sinais personalizados em seus próprios roteiros.

ANOTAÇÕES



AULA 06

Processamentos

Muitas ações no Godot são disparadas por funções virtuais ou de retorno, então não há necessidade de escrever um código que execute o tempo todo.

Entretanto, ainda é comum precisar que um roteiro seja processado a cada quadro. Existem dois tipos de processamento: o ocioso e o físico.

Processamento ocioso é ativado quando o método Node.process() é encontrado em um roteiro. Ele pode ser desativado e reativado com a função Node.set_process().

Esse método será chamado toda vez que um quadro é desenhado:

```
func _process(delta):
    # Faça alguma coisa...
    pass
```

É importante ter em mente que a frequência em que a função _process() será chamada depende de quantos quadros por segundo (FPS) sua aplicação está rodando. Essa taxa pode variar com o tempo e dispositivos.

Para ajudar a gerenciar essa variabilidade, o parâmetro delta contém o tempo decorrido em segundos, sendo um número decimal, desde a chamada anterior de _process().

Este parâmetro pode ser usado para garantir que as coisas sempre levem a mesma quantidade de tempo, independentemente da taxa de quadros (FPS) do jogo.

Por exemplo, o movimento é frequentemente multiplicado pelo delta para tornar a velocidade de movimento constante independente da taxa de quadros.

Processamento físico com _physics_process() funciona similar, mas deveria ser usado para processo que tenham que acontecer antes de cada passo da física, tais como controlar um personagem. Ele sempre é executado antes de um passo da física e é chamado a intervalos fixos de tempo: 60 vezes por segundo por padrão. Você pode alterar o intervalo nas Configurações do Projeto, em Physics -> Common -> Physics Fps (Física -> Comum -> Fps Física).

A função _process(), contudo, não é sincronizado com a física. Sua taxa de quadros não é constante e depende do hardware e da otimização do jogo. Sua execução é feita depois do passo da física em jogos de thread única.

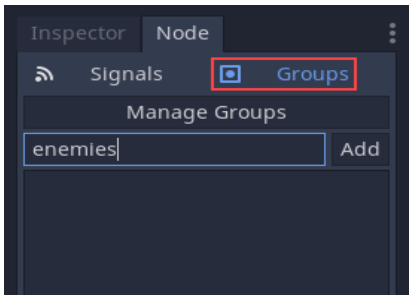
Uma maneira simples de ver a função _process() funcionando é criar uma cena com um único nó Label, com o seguinte roteiro:

```
var accum = 0

func _process(delta):

 accum += delta

 text = str(accum) # 'text' é uma propriedade de rótulo embutida.
```

Que vai mostrar um contador aumentando a cada quadro.

Grupos

Grupos no Godot funcionam como tags que você pode encontrar em outros softwares. 'NÓS' podem ser adicionados a grupos, a quantidade que desejar por nó e isso é uma funcionalidade útil para organizar cenas grandes. Há duas formas de fazer isso. A primeira é pela interface gráfica, a partir do botão Grupos dentro do painel Nó:



Notificações

O Godot tem um sistema de notificações. Elas não costumam ser necessárias para roteirizar, já que são programação de baixo nível e se provê funções virtuais para a maioria delas. Mas é bom saber que elas existem.

Funções sobrescrevíveis

Tais funções sobrescrevíveis, descritas em seguida, podem ser aplicadas a NÓS:

```
func _enter_tree():
    # Quando o nó entra na Scene Tree, ele se torna ativo
    # e esta função é chamada. NÓS filhos não entraram
    # a cena ativa ainda. Em geral, é melhor use_ready ()
    # para a maioria dos casos.
    pass


func _ready():
    # Esta função é chamada após _enter_tree, mas garante
```

```
        # que todos os NÓS filhos também entraram na Scene Tree,
        # e tornou-se ativo.
        pass

    func _exit_tree():
        # Quando o nó sai da Scene Tree, esta função é chamada.
        # NÓS filhos todos saíram da Scene Tree neste ponto
        # e todos ficaram inativos.

        pass
    func _process(delta):
        # Esta função é chamada em cada quadro.
        pass

    func _physics_process(delta):
        # Está função é chamada em todos os quadros físicos.
        Pass
```

Como já mencionado, é melhor usar essas funções ao invés do sistema de notificações.

Criando (NÓS)

Para criar um nó a partir do código, chame o método .new(), assim como faria para qualquer tipo de dado baseado em classe. Por exemplo:

```
var s
func _ready():
s = Sprite.new() # Cria um  sprite!
add_child(s) # Adiciona como filho nó deste
```

Para excluir um nó, esteja ele dentro ou fora da cena, deve-se usar free():

```
func _someaction():

s.free() # Remove imediatamente o nó da cena e o libera.
```

Quando um nó é liberado da memória, ele também libera todos os seus NÓS filhos. Por isso, excluir NÓS manualmente é mais simples do que parece. Libere o nó base e todo o resto vai embora junto com ele.

Uma situação pode acontecer quando queremos excluir um nó que está "bloqueado" atualmente, por estar emitindo um sinal ou chamando uma função. Isso quebra o jogo. Executar o Godot com o depurador irá pegar esse caso frequentemente e lhe alertar sobre isso.

A forma mais segura de excluir um nó é usando Node.queue_free(). Isso apaga o nó com segurança durante o tempo ocioso.

```
func _someaction():

s.queue_free() # Remove o nó da cena e o libera quando for seguro fazê-lo.
```

Criando instâncias de cenas

Criar uma instância de cena a partir do código é feito em duas etapas. A primeira é carregar a cena do seu disco rígido:

```
var scene = load("res://myscene.tscn") # Será carregado quando o script for instanciado.
```

Pré-carregamento pode ser mais conveniente, já que acontece durante o tempo de análise (apenas para GDScript):

```
var scene = preload("res://myscene.tscn") # Irá carregar ao analisar o script.
```

Mas cena ainda não é um nó. Ela está empacotada em um recurso especial chamado PackedScene. Para criar um nó de fato, é preciso chamar a função PackedScene.instance(). Ela vai retornar a árvore de NÓS que pode ser adicionada à cena ativa:

```
var node = scene.instance()

add_child(node)
```

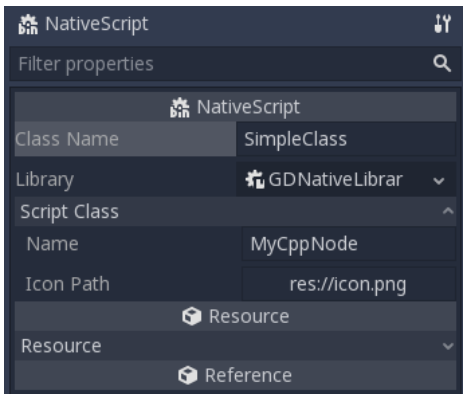
A vantagem deste processo de duas etapas é que uma cena empacotada pode ser mantida carregada e pronta para uso, de forma que você possa gerar quantas instâncias desejar. É útil especialmente para, rapidamente, criar instâncias de vários inimigos, balas e outras entidades na cena ativa.

Registrar scripts como classes

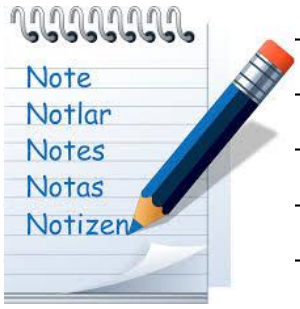
Godot tem uma função "Classe Script" para registrar scripts individuais com o editor. Por padrão, você só pode acessar scripts sem nome ao carregar o arquivo diretamente.

Você pode nomear um script e o registrar como um tipo no editor com a palavra-chave class_name seguido pelo nome da classe. Você pode adicionar uma vírgula e um caminho opcional a uma imagem para usar como ícone. Então, você encontrará seu novo tipo na janela de criação de nodes e recursos.

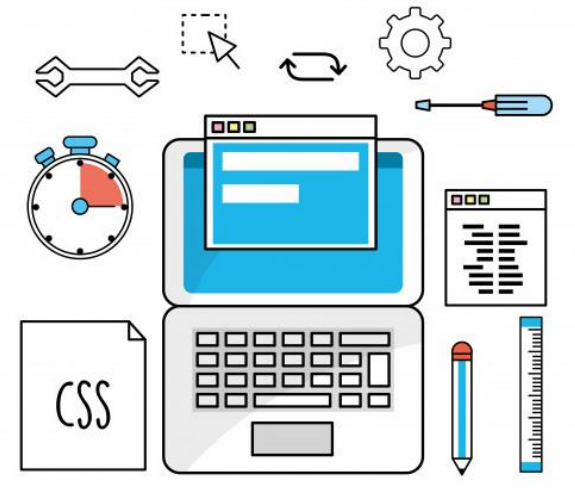
```
extends Node


# Declare o nome da classe aqui

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


func _ready ():

    var this = ScriptName          # referência ao script.

    var cppNode = MyCppNode.new()    # nova instância de uma classe chamada MyCppNode


    cppNode.queue_free()
```



ANOTAÇÕES



AULA 07



Sinais

Sinais são a versão do Godot do padrão observador. Eles permitem que um nó envie uma mensagem que outros NÓS podem captar e responder. Por exemplo, ao invés de continuamente verificar um botão para saber se ele está sendo pressionado, o botão pode emitir um sinal quando for pressionado.

Sinais são um meio de desacoplar seus objetos de jogo, o que leva a um código mais organizado e gerenciável. Ao invés de forçar objetos de jogo a esperar que outros objetos estejam sempre presentes, eles podem emitir sinais para os quais qualquer objeto interessado pode inscrever-se e responder.

Conectando sinais por código

Você também pode fazer a conexão do sinal em código ao invés do editor. Isso é geralmente necessário quando você está instanciando (NÓS) via código e por isso não pode usar o editor para fazer a conexão.

Sinais personalizados

Você também pode declarar seus próprios sinais no Godot:

```
extends Node2D

signal my_signal
```

Quando declarado, seus sinais personalizados aparecerão no Inspetor e podem ser conectados em um nó da mesma forma que os sinais predefinidos.

Para emitir um sinal por código, use a função emit_signal:

```
extends Node2D

signal my_signal

func _ready():

emit_signal("my_signal")
```

Um sinal também pode opcionalmente declarar um ou mais argumentos. Especifique os nomes dos argumentos entre parênteses:

```
extends Node

signal my_signal(value, other_value)
```

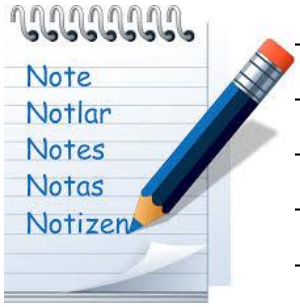
Estes argumentos aparecem na aba Nó do editor e a Godot pode usá-los para criar funções para você. Entretanto, você ainda pode emitir qualquer número de argumento quando você emite sinais; é você quem tem que emitir os valores corretos.

Para emitir valores, adicione-os como o segundo argumento à função emit_signal:

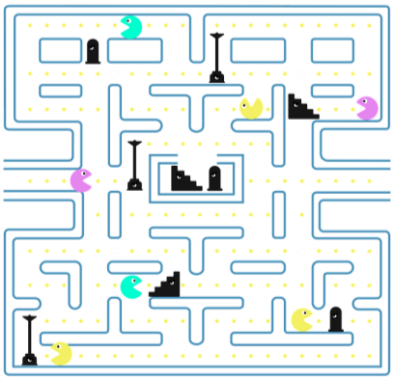
```
extends Node

signal my_signal(value, other_value)

func _ready():

    emit_signal("my_signal", true, 42)
```

No Godot, muitos dos NÓS embutidos proporcionam sinais que você pode usar para detectar eventos. Por exemplo, um Area2D <class_Area2D>`representando uma moeda emite um sinal "body_entered` sempre quando o corpo físico do jogador entra na forma de colisão, permitindo você saber quando o jogador a coletou.

ANOTAÇÕES



AULA 08



Seu primeiro jogo

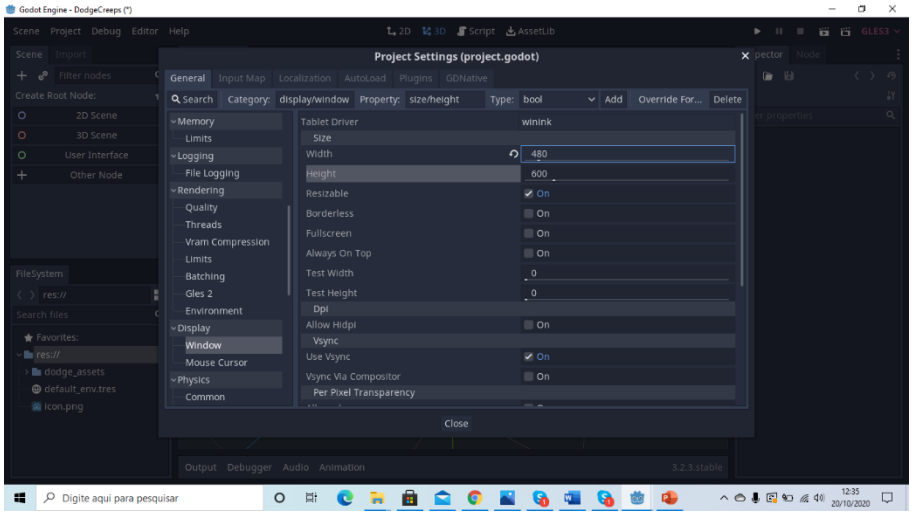
Você aprenderá como o editor Godot funciona, como estruturar um projeto e como construir um jogo 2D.

O jogo se chama "Dodge the Creeps!" (em tradução livre, "Desvie das Criaturas!"). Seu personagem deve se mover e evitar os inimigos por quanto tempo puder. Aqui está uma prévia do resultado:

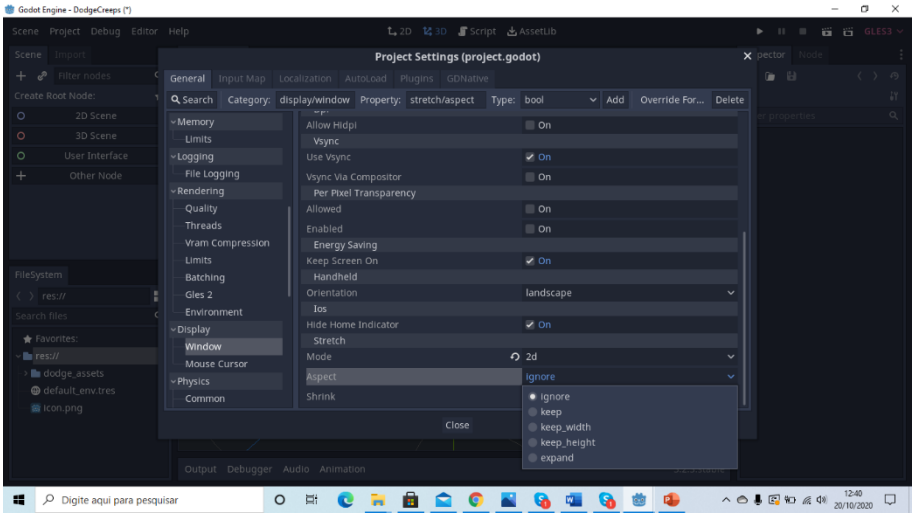
Por que 2D? Jogos 3D são muito mais complexos do que os 2D. Você deve se concentrar em 2D até que tenha uma boa compreensão do processo de desenvolvimento de jogos e de como deve-se usar o Godot.

Configuração do projeto

Vamos executar o Godot e iniciar um novo projeto. Este jogo usará o modo retrato, então precisamos ajustar as dimensões da janela de jogo. Clique em Projeto -> Configurações do Projeto -> Exibição -> Janela e configure "Largura" para 480 e "Altura" para 720.

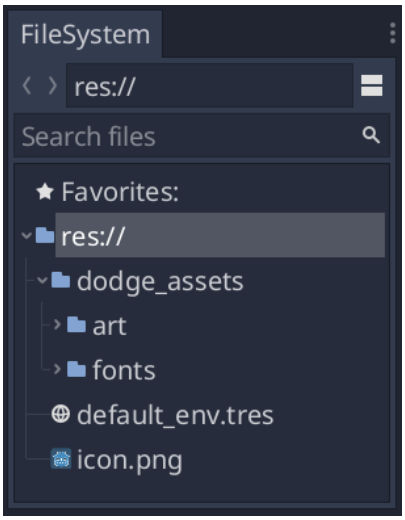


Também nesta seção, nas opções "Alongamento (Stretch)", vamos definir "Modo (Mode)" para "2d" e "Aspecto (Aspect)" para "manter(keep)". Isso garante que o jogo seja escalado consistentemente em telas de tamanhos diferentes.



Organizando o projeto

Neste projeto, vamos criar três cenas independentes: Jogador, Inimigo e HUD, que combinaremos na cena Principal do jogo. Em um projeto maior, pode ser útil criar pastas para armazenar as várias cenas e seus scripts, mas para este jogo relativamente pequeno, você pode salvar tudo na pasta raiz, referenciada como res://. Você pode ver as pastas do seu projeto no painel Arquivos no canto inferior esquerdo:

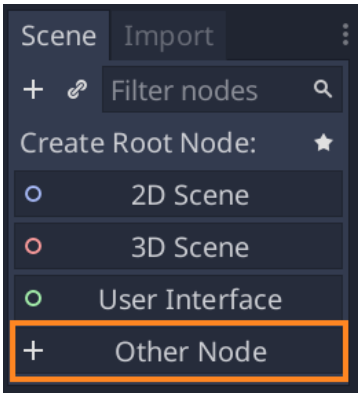


Cena do Jogador

A primeira cena definirá o objeto Player. Um dos benefícios de criar uma cena separada do jogador é que podemos testá-la separadamente, mesmo antes de criarmos outras partes do jogo.

Estrutura de NÓS

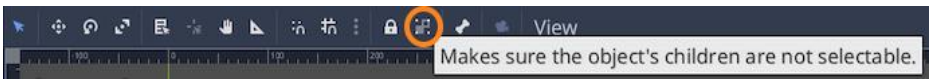
Para iniciarmos, precisamos escolher o nó raiz para o objeto jogador. Como regra geral, o nó raiz da cena deve refletir à funcionalidade desejado do objeto - o que o objeto *é*. Clique no botão "Outros NÓS" e adicionamos o nó Area2D à cena.



Godot vai mostrar um ícone de aviso próximo do nó na árvore de cenas. Você pode ignorar isso por enquanto. Nós vamos falar disso mais tarde.

Com Area2D, nós podemos detectar objetos que se sobreponham ou vão de encontro ao jogador. Mudamos seu nome para Jogador com um clique duplo no nome do nó. Já que nós configuramos o nó raiz, nós agora podemos inserir NÓS adicionais para adicionar mais funcionalidades.

Antes de adicionar filhos ao nó Jogador, queremos ter certeza de que não os moveremos nem os redimensionaremos acidentalmente ao clicar neles. Selecione o nó e clique no ícone à direita do cadeado; seu texto de dica diz "Garante que os filhos do objeto não sejam selecionáveis."



Vamos salvar a cena. Clique em Cena -> Salvar ou pressione Ctrl + S no Windows/Linux ou Cmd + S no macOS.

Para esse projeto, vamos seguir as convenções de nomeação Godot.

- **GDScript**: classes (NÓS) usam o estilo PascalCase (IniciaisMaiúsculas), variáveis e funções usam snake_case (minúsculas_separadas_por_sublinhadas), e constantes usam ALL_CAPS (TODAS_MAIÚSCULAS).

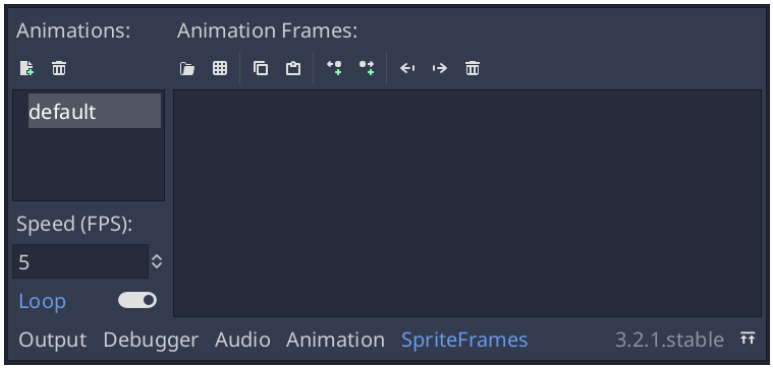
- **** C # **:** classes, variáveis de exportação e métodos usam PascalCase, campos privados usam _camelCase, variáveis locais e parâmetros usam camelCase.

Tenha o cuidado de digitar os nomes dos métodos com precisão ao conectar os sinais.

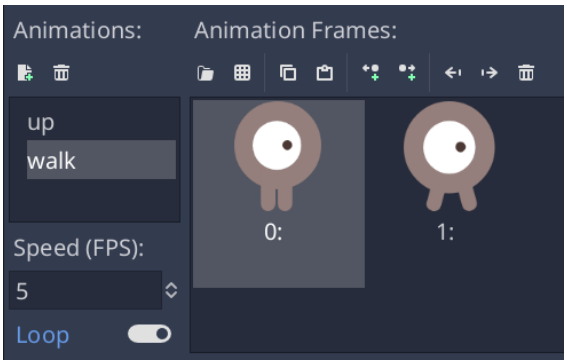
Animação por Sprite

Vamos clicar no nó Jogador e adicionar um nó AnimatedSprite (sprite animado) como filho. O AnimatedSprite irá lidar com a aparência e as animações do nosso jogador. Note que existe um símbolo de alerta ao lado do nó. Um AnimatedSprite exige um recurso do tipo SpriteFrames (quadros de sprite),

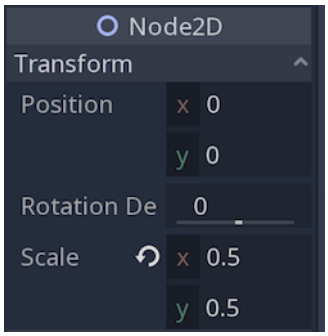
que é uma lista das animações que ele pode mostrar. Para criar um, encontramos a propriedade Frames no Inspetor e clicamos em "[empty]" -> "Novo SpriteFrames". Clique novamente para abrir o painel de "SpriteFrames**":**



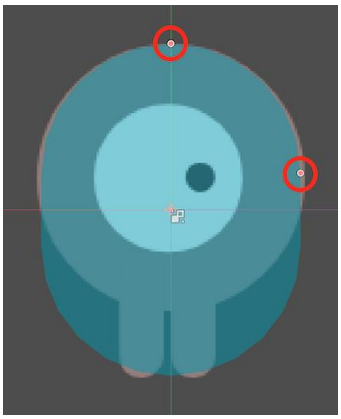
À esquerda está uma lista de animações. Clicamos em "default" e a renomeamos para "walk". Então, clicamos no botão "Nova Animação" para criar uma segunda animação chamada "up". Buscamos as imagens do jogador na aba "Sistema de Arquivos" - eles estão na pasta art que temos as artes do jogo. Arrastamos as duas imagens de cada animação, chamadas playerGrey_up[1/2] e playerGrey_walk[1/2] para dentro do "Animation Frames" ao lado do painel correspondente de cada animação:



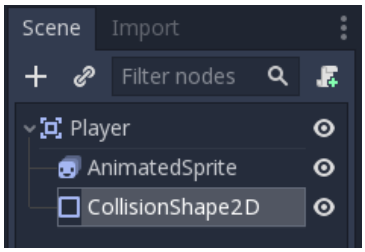
As imagens do jogador são um tanto grandes demais para a janela de jogo, então precisamos reduzir sua escala. Clicamos no **NÓ** AnimatedSprite e configuramos a propriedade Scale para (0.5, 0.5). Você pode encontrá-la no Inspetor na seção Node2D.



Finalmente, adicionamos um CollisionShape2D (forma de colisão 2D) como filho de Jogador. Isso determina a "caixa de acerto" do jogador, ou seja, os limites da sua área de colisão. Para este personagem, um nó CapsuleShape2D (forma cápsula 2D) é o que melhor se encaixa. Então, ao lado de "Shape" (forma) no Inspetor, clique em "<null>" -> "Novo CapsuleShape2D". Utilizando os dois manipuladores, redimensione a forma para cobrir o sprite:



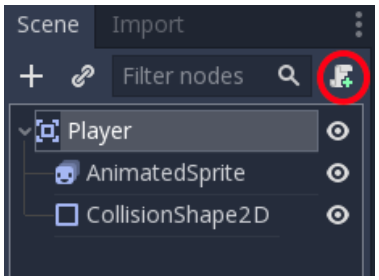
Quando tiver finalizado, a cena Jogador deveria se parecer assim:



Vamos salvar a cena novamente após as alterações.

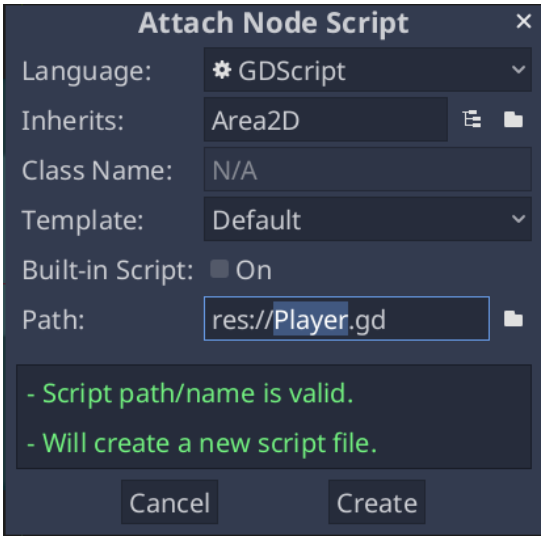
Movendo Jogador

Agora precisamos adicionar algumas funcionalidades que não conseguimos em um nó embutido, então adicionaremos um script. Clicamos no nó Jogador e depois no botão "Adicionar Script":



Na janela de configurações de roteiro, você pode deixar as configurações padrões do jeito que estão. Apenas clique em "Criar":

Observações: Se você estiver criando um script em C# ou em outras linguagens, selecione a linguagem em no menu suspenso *Idioma* antes de clicar em Criar.

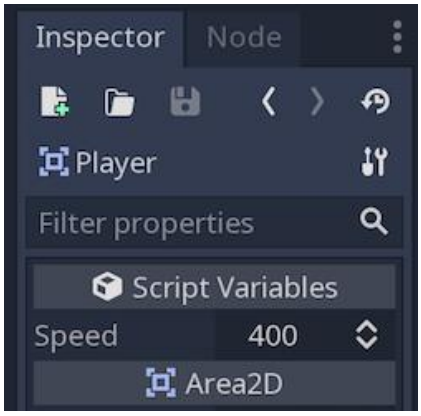


Comece declarando as variáveis membros que este objeto irá precisar:

GDScript

```
export var speed = 400  # Quão rápido o jogador se moverá (pixels / seg).

    var screen_size  # Tamanho da janela do jogo.
```

Usar a palavra-chave export na primeira variável, speed, nos permite definir seu valor pelo Inspetor. Isto pode ser útil para os valores que você deseja ser capaz de ajustar do mesmo jeito que se faz com as propriedades internas de um nó. Clicando no nó Jogador você verá que a propriedade agora aparece na seção "Variáveis de Script" do Inspetor. Lembre-se, se você modificar o valor aqui, ele irá sobrepor o valor que está definido no script.



A função _ready() é chamada quando um nó entra na árvore de cena, que é uma boa hora para descobrir o tamanho da janela do jogo:

```
    func _ready():screen_size = get_viewport_rect().size
```

Agora podemos usar a função _process() para definir o que o jogador fará. _process() é chamada a cada quadro, então usaremos ela para atualizar os elementos de nosso jogo que esperamos que mudem frequentemente. Para o jogador, precisamos fazer o seguinte:

- Verificação de entradas.
- Movimentação em uma certa direção.

- Reprodução da animação apropriada.

Primeiro, precisamos verificar as entradas – o jogador está pressionando uma tecla **?** Para este jogo, temos 4 entradas de direção para verificar. Ações de entrada são definidas nas Configurações do Projeto na aba "Mapa de entrada". Você pode definir eventos personalizados e atribuir diferentes teclas, eventos de mouse ou outras entradas para eles. Para esta demonstração, vamos usar os eventos padrões que são atribuídos para as teclas de seta no teclado.

Você pode detectar se uma tecla é pressionada usando Input.is_action_pressed(), que retorna true (verdadeiro) se é pressionada ou false (falso) em caso contrário.

```
func _process(delta):
    var velocity = Vector2()  # The player's movement vector.
    if Input.is_action_pressed("ui_right"):
        velocity.x += 1
    if Input.is_action_pressed("ui_left"):
        velocity.x -= 1
    if Input.is_action_pressed("ui_down"):
        velocity.y += 1
    if Input.is_action_pressed("ui_up"):
        velocity.y -= 1
    if velocity.length() > 0:
        velocity = velocity.normalized() * speed
        $AnimatedSprite.play()
    else:
        $AnimatedSprite.stop()
```

Iniciamos definindo a velocidade como sendo (0, 0) - por padrão o jogador não deve estar se movendo. Então nós verificamos cada entrada e adicionamos/subtraímos da velocidade para obter a direção resultante. Por exemplo, se você segurar direita e baixo ao mesmo tempo, o vetor resultante velocidade será (1, 1). Neste caso, já que estamos adicionando um

movimento vertical e um horizontal, o jogador iria se mover *mais rápido* do que se apenas se movesse horizontalmente.

Podemos evitar isso se *normalizarmos* a velocidade, o que significa que podemos definir seu *comprimento* (módulo) para 1 e multiplicar pela velocidade desejada. Isso resolve o problema de movimentação mais rápida nas diagonais.

Nós também verificamos se o jogador está se movendo, para que possamos iniciar ou parar a animação do AnimatedSprite.

$ é um atalho para get_node(). Então, no código acima, $AnimatedSprite.play() é o mesmo que get_node("AnimatedSprite").play().

Dica: Em GDScript, $ retorna o nó no caminho relativo a partir deste ou retorna null (nulo) se o nó não for encontrado. Já que AnimatedSprite é um filho do nó atual, podemos usar $AnimatedSprite.

Agora que temos uma direção de movimento, podemos atualizar a posição do jogador. Podemos também usar clamp() para impedir que ele saia da tela.

Clamp (fixar) um valor significa restringi-lo a um determinado intervalo. Adicionamos o seguinte código embaixo da função "_process"(Tenha certeza de que não está identado ao *else*):

```
    position += velocity * delta
    position.x = clamp(position.x, 0, screen_size.x)
    position.y = clamp(position.y, 0, screen_size.y)
```

Dica: O parâmetro *delta* na função *_process()* se refere ao *comprimento do frame* - a quantidade de tempo que o frame anterior levou para ser completado. Usando este valor você se certifica que seu movimento será constante, mesmo quando o frame rate sofrer alterações.

Clicamos em "Rodar Cena" (F6) e conferimos se consegue mover o jogador pela tela em todas as direções.

Aviso: Se aparecer um erro no painel "Depurador" que diz A tentativa de chamar a função 'play' na base 'instance null' em uma instância n ula. Se você vir no painel "Depurador" um erro referente a "null instance" (instância nula), provavelmente significa que você digitou o nome do nó errado. Letras maiúsculas e minúsculas fazem diferença nos nomes dos NÓS, e $NomeDoNo ou get_node("NomeDoNo") tem que coincidir com o nome que você vê na árvore da cena.

Selecionado as Animações

Agora que o jogador consegue se mover, precisamos alterar qual animação do AnimatedSprite é reproduzida conforme a direção. Temos uma animação "andar", que mostra o jogador andando para a direita. Essa animação deve ser espelhada horizontalmente usando a propriedade flip_h para o jogador se movimentar à esquerda. Temos também a animação "up", que deve ser espelhada verticalmente com flip_v para o movimento para baixo. Vamos colocar este código no fim da nossa função _process():

```
    if velocity.x != 0:
        $AnimatedSprite.animation = "walk"
        $AnimatedSprite.flip_v = false
        # See the note below about boolean assignment
        $AnimatedSprite.flip_h = velocity.x < 0
    elif velocity.y != 0:
        $AnimatedSprite.animation = "up"
        $AnimatedSprite.flip_v = velocity.y > 0
```

Nota: As atribuições booleanas no código acima são um encurtamento comum para programadores. Já que estamos fazendo um teste de comparação(booleano) e *atribuindo* um valor booleano, é possível fazer os dois ao mesmo tempo. Compare esse código a atribuição booleana encurtada, acima:

```
    if velocity.x < 0:
        $AnimatedSprite.flip_h = true
```

```
    else:
        $AnimatedSprite.flip_h = false
```

Executamos a cena novamente e verificamos se as animações estão corretas em cada uma das direções.

Dica: Um erro comum aqui é digitar os nomes das animações de forma errada. O nome das animações no painel SpriteFrames deve ser igual ao que foi digitado no código. Se você nomeou a animação "Walk" você também deve usar a letra maiúscula "W" no código.

Quando tiver certeza que a movimentação está funcionando corretamente, adicione esta linha à _ready() para que o jogador fique oculto no início do jogo:

```
hide()
```

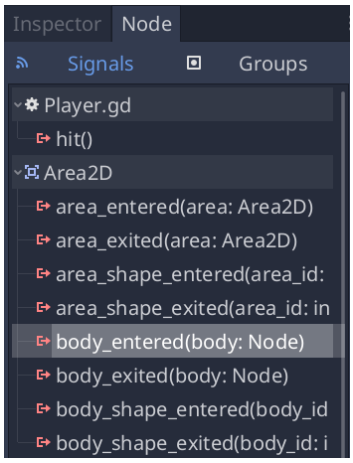
Preparando para colisões

Nós queremos que o Jogador detecte quando é atingido por um inimigo, mas nós não fizemos nenhum inimigo ainda! Não tem problema, porque nós iremos utilizar a funcionalidade de *sinal* do Godot para fazer com que funcione.

Adicione o trecho a seguir no início do script, depois de extends Area2D:

```
signal hit
```

Isto define um sinal personalizado chamado "hit" (atingir) que faremos com que nosso jogador emita (envie) quando colidir com um inimigo. Iremos utilizar Area2D para detectar a colisão. Selecione o nó Jogador e clique na aba "Nó" ao lado da aba "Inspetor" para ver a lista de sinais que o jogador pode emitir:



Observe que o nosso sinal personalizado "hit" também está lá. Já que nossos inimigos serão NÓS do tipo RigidBody2D, queremos o sinal body_entered(Object body); ele será emitido quando um corpo entrar em contato com o jogador. Clique em "Conectar." e a janela "Conectando Sinal" aparece. Nós não precisamos alterar nenhuma destas configurações – o Godot criará automaticamente uma função chamada _on_Jogador_body_entered no script do jogador. Esta função será chamada sempre que o sinal for emitido - ela *trata* do sinal.

```
42 func _on_Player_body_entered(body):
43     pass # Replace with function body.
```

Note que o icone verde indica que um sinal está conectado a esta função. Adicionamos esse código à função:

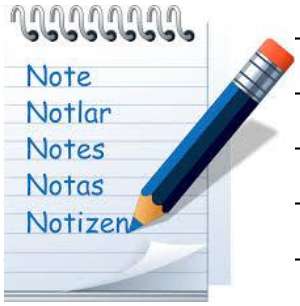
```
func _on_Player_body_entered(body):
    hide()  # O jogador desaparece após ser atingido.
    emit_signal("hit")
    $CollisionShape2D.set_deferred("disabled", true)
```

Cada vez que um inimigo atinge o jogador, o sinal será emitido. Precisamos desativar a colisão do jogador para que não acione o sinal hit mais de uma vez.

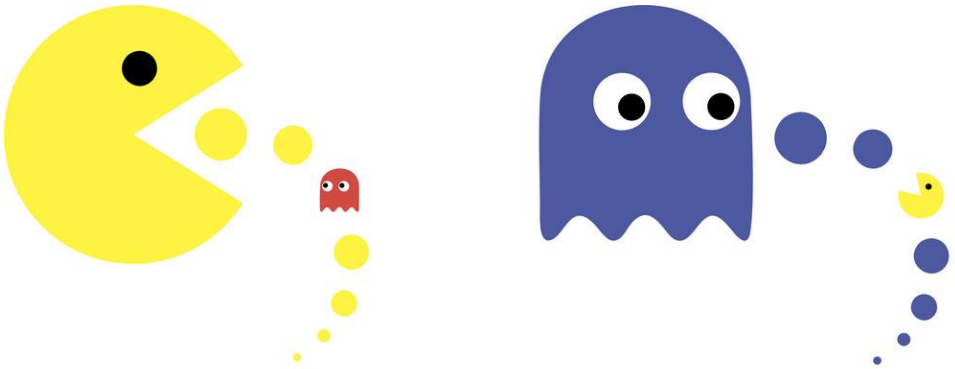
A última peça para nosso jogador é adicionar uma função que possamos chamar para reiniciar o jogador para começarmos um novo jogo.

```
func start(pos):
    position = pos
    show()
    $CollisionShape2D.disabled = false
```

ANOTAÇÕES



AULA 09



Cena do Inimigo

Agora é hora de criarmos os inimigos que nosso jogador terá que se desviar. Seu comportamento não será tão complexo: inimigos irão nascer aleatoriamente nos cantos da tela e irão se mover em uma direção também aleatória, só que em linha reta, para, então desaparecerem ao sair da tela.

Nós iremos construir isso em uma cena Inimigo, que poderemos então *instanciar* para criar uma quantidade de inimigos independentes no jogo.

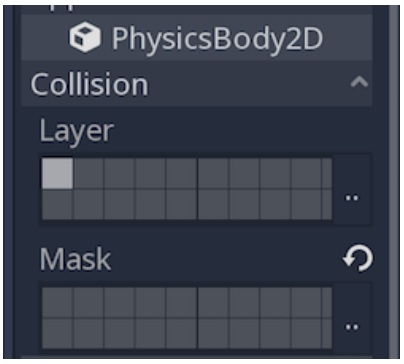
Configuração de NÓS

Clique Cena -> Nova Cena e adicione os seguintes NÓS:

- RigidBody2D (chamado Turba)
- AnimatedSprite
- CollisionShape2D
- VisibilityNotifier2D (chamado Visibilidade)

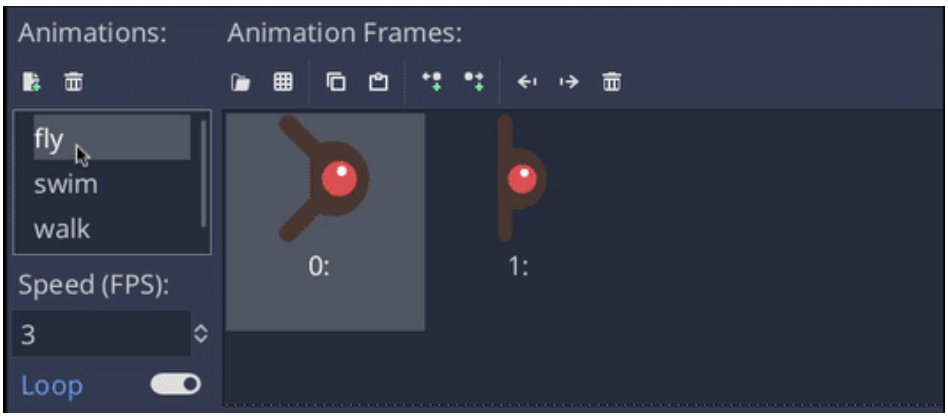
Nas propriedades de RigidBody2D, configuramos Gravity Scale (escala da gravidade) para 0, para que os inimigos não caiam para a parte de baixo da tela. Além disso, dentro da seção PhysicsBody2D, clicamos na

propriedade Mask (máscara) e desmarcamos a primeira caixa. Isso vai garantir que os inimigos não colidam uns com os outros.



Configuramos o AnimatedSprite assim como você fizemos para o jogador. Desta vez, temos 3 animações: voar (*fly*), nadar (*swim*) e andar (*walk*). Definimos a propriedade Playing reproduzindo no Inspetor para "Ativo" e ajuste a configuração "Velocidade (FPS)" como mostrado abaixo. Vamos selecionar uma dessas animações aleatoriamente para que a turba tenha alguma variedade.

Ajustamos o "Speed (FPS)" para 3 em todas as animações.



Definimos a propriedade Playing no Inspector para "On".

Vamos selecionar uma das animações aleatoriamente para que os inimigos tenham alguma variedade.

Como as imagens do jogador, essas imagens da turba precisam ser reduzidas. Defina a propriedade Scale (escala) do AnimatedSprite para (0.75, 0.75).

Como na cena do Jogador, adicionamos um CapsuleShape2D para a colisão. Para alinhar a forma com a imagem, você precisará definir a propriedade Rotation Degrees (graus de rotação) como 90 na seção Node2D.

Salvar Cena: Script do inimigo

Adicione um roteiro ao nó Turba e adicione as seguintes variáveis membros:

```
extends RigidBody2D

export var min_speed = 150  # Faixa de velocidade miníma.

export var max_speed = 250  # Faixa de velocidade máxima.
```

Quando gerarmos um inimigo, vamos escolher um valor aleatório entre min_speed (velocidade mínima) e max_speed (velocidade máxima) para definir a velocidade com que cada inimigo irá se mover (seria bem entediante se todos eles se movessem na mesma velocidade).

Agora, vamos ver o resto do roteiro. Em "_ready ()", escolhemos aleatoriamente um dos três tipos de animação:

```
var mob_types = $AnimatedSprite.frames.get_animation_names()

$AnimatedSprite.animation = mob_types[randi() % mob_types.size()]
```

Primeiro nós pegamos a lista dos nomes das animações da propriedade frames do AnimatedSprite. Que retorna um vetor contendo os nomes das três animações: ["walk", "swim", "fly"].

Nós precisamos, então, de um número randômico entre 0 e 2 para selecionar um dos nomes da lista (os índices de um vetor começam do 0). randi() % n"seleciona randomicamente um inteiro entre "0 e n-1.

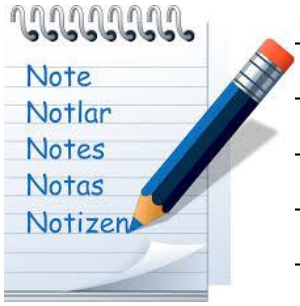
Nota: Você deve usar randomize() se quiser que sua sequência de números "aleatórios" seja diferente toda vez que você executar a cena. Nós vamos usar randomize() em nossa cena Principal, então não precisaremos dessa função aqui. randi () % n é a maneira padrão de obter um inteiro aleatório entre 0 e n-1.

A última parte é fazer os inimigos se autodestruírem ao sair da tela. Conecte o sinal screen_exited() (saiu da tela) do nó VisibilityNotifier2D e adicione este código:

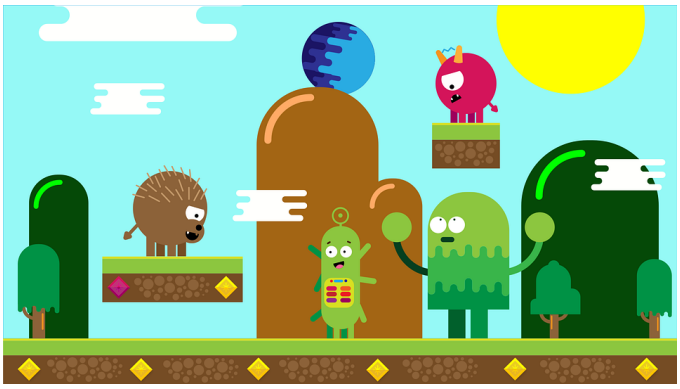
```
func _on_VisibilityNotifier2D_screen_exited():

queue_free()
```

Isto completa a cena dos Inimigos.

ANOTAÇÕES

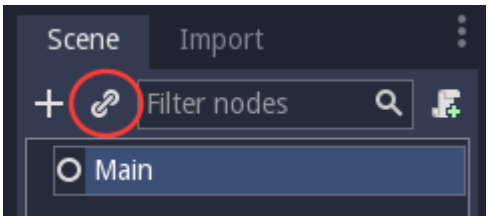


AULA 10



Cena Principal

Agora é hora de juntar tudo. Criamos uma cena e adicionamos um nó chamado Main. Clicamos no botão "Instância" e selecione o Player.tscn salvo.



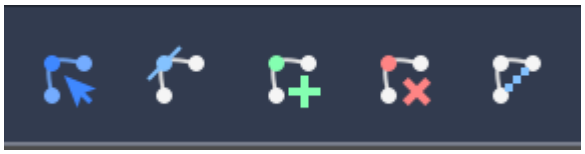
Agora adicionamos os seguintes NÓS como filhos de Principal, e os nomeie como mostrado (os valores estão em segundos):

- Timer (nomeado MobTimer) - para controlar a frequência com que a turba é gerada
- Timer (nomeado ScoreTimer) - para incrementar a pontuação a cada segundo
- Timer (nomeado StartTimer) - para dar um atraso antes de começar
- Position2D (nomeado StartPosition) - para indicar a posição inicial do jogador

Além disso, configuramos a propriedade One Shot ("uma só vez") de StartTimer para "Ativo" e Position do nó StartPosition para (240, 450).

Gerando monstros

O nó Principal ficará gerando novos inimigos, e nós queremos que eles apareçam em lugares aleatórios nos cantos da tela. Adicionamos um nó Path2D chamado CaminhoTurba como filho de Principal. Quando você selecionar Path2D, aparecerão alguns botões novos na parte superior do editor:



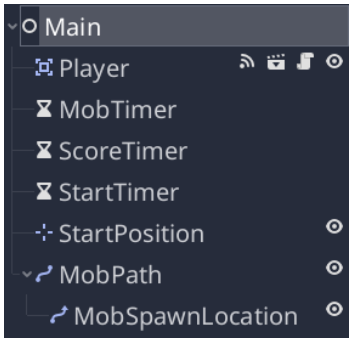
Selecione o do meio ("Adicionar Ponto") e desenhe o caminho clicando para adicionar os pontos nos cantos mostrados. Para que os pontos se encaixem na grade, certifique-se de que "Usar snap à grade" e "Usar snap" estejam selecionados. Essas opções podem ser encontradas à esquerda do botão "Bloquear", aparecendo como um ímã próximo a alguns pontos e linhas que se cruzam, respectivamente.

Importante: Desenhamos o caminho em sentido *horário*, ou sua turba vai surgir apontando para *fora* em vez de para *dentro*!

Depois de colocar o ponto 4 na imagem, clicamos no botão "Fechar Curva", e sua curva estará completa.

Agora que o caminho está definido, adicionamos um nó PathFollow2D como filho de CaminhoTurba e dê o nome de LocalGeraçãoTurba. Esse nó vai rotacionar automaticamente e seguir o caminho conforme ele se move, para que possamos usá-lo para selecionar uma posição e uma direção aleatória ao longo do caminho.

A cena deve se parecer com isso:



Script principal

Adicione um roteiro à Principal. No começo do script, nós usamos export (PackedScene) para permitir-nos escolher a cena Inimigo que queremos instanciar.

```
extends Node

export (PackedScene) var Mob
var score

func _ready():
    randomize()
```

Clicamos no nó Principal e podemos ver verá a propriedade "Mob" no inspetor, abaixo das Variáveis do Script.

Podemos atribuir o valor dessa propriedade de duas formas:

- Arraste Mob.tscn do painel do "Sistema de Arquivos" e solte-o na propriedade Mob.
- Clique na seta para baixo ao lado de "[vazio]" e escolha "Carregar". Selecione Mob.tscn.

Em seguida, selecione o nó do Player no painel Cena e acesse o nó na barra lateral. Certificamos de selecionar a aba Sinais no Painel de **NÓS**.

Devemos ver uma lista de sinais para o nó Player. Em seguida, encontre e dê dois cliques no sinal hit da lista (ou clique com o botão direito e selecione "Conectar..."). Isso abrirá a caixa de diálogo de conexão de sinal. Queremos criar uma função chamada game_over, que lidará com tudo o que precisa acontecer quando um jogo acabar. Digitamos "game_over" na caixa "Método no Nó" na parte inferior da janela de conexão de sinal e clique em "Conectar". Adicionamos

o código a seguir à nova função, assim como uma função new_game (novo jogo) para definir tudo para um novo jogo:

```
func game_over():
    $ScoreTimer.stop()
    $MobTimer.stop()

func new_game():
    score = 0
    $Player.start($StartPosition.position)
    $StartTimer.start()
```

Agora, conectamos o sinal timeout() de cada um dos NÓS de Timer (StartTimer, ScoreTimer , e"MobTimer") para o script principal. StartTimer irá iniciar os outros dois temporizadores. ScoreTimer irá incrementar a pontuação em 1.

```
func _on_StartTimer_timeout():
    $MobTimer.start()
    $ScoreTimer.start()

func _on_ScoreTimer_timeout():
    score += 1
```

Em _on_MobTimer_timeout(), vamos criar uma instância de inimigo, pegar um local de início aleatório ao longo do Path2D, e pôr o mob em movimento. O nó PathFollow2D irá rotacionar automaticamente à medida em que ele segue o caminho, então usaremos isso para escolher a direção do inimigo, bem como sua posição.

Note que uma nova instância deve ser adicionada à cena usando add_child() (adicionar filho).

```
    func _on_MobTimer_timeout():
        #Escolha uma localização aleatória em Path2D
        $MobPath/MobSpawnLocation.offset = randi()
        # Crie uma instância de Mob e adicione-a à cena.    var mob = Mob.instance()
        add_child(mob)
        # Set the mob's direction perpendicular to the path direction.
        var direction = $MobPath/MobSpawnLocation.rotation + PI / 2
        # Defina a direção do mob perpendicular à direção do caminho...
        mob.position = $MobPath/MobSpawnLocation.position
        # Adicione alguma aleatoriedade à direção.
        direction += rand_range(-PI / 4, PI / 4)
        mob.rotation = direction
        # Defina a velocidade (velocidade & direção).
        mob.linear_velocity = Vector2(rand_range(mob.min_speed, mob.max_speed), 0)
        mob.linear_velocity = mob.linear_velocity.rotated(direction)
```

Importante: Por que PI? Em funções que demandem ângulos, GDScript usa *radianos*, e não graus. Se você se sente mais confortável trabalhando com graus, precisará usar as funções deg2rad() (graus para radianos) e rad2deg() (radianos para graus) para fazer as conversões.

Testando a cena

Vamos testar a cena para garantir que tudo está funcionando. Adicionamos isso ao _ready():

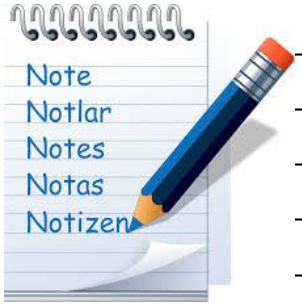
```
func _ready():
    randomize()
    new_game()
```

Vamos também atribuir Main como nossa "Cena Principal" - aquela que é executada automaticamente quando o jogo é iniciado. Pressione o botão "Reproduzir" e selecione " Main.tscn" quando solicitado.

Devemos ser capazes de mover o jogador, ver os inimigos nascendo, e ver o jogador desaparecer quando atingido por um inimigo.

Quando tiver certeza de que tudo está funcionando, remova a chamada de new_game() em _ready().

ANOTAÇÕES



AULA 11

HUD

A peça final que nosso jogo precisa é uma interface com usuário: uma interface que mostra coisas como pontuação, uma mensagem de "fim de jogo" e um botão de reinício. Criamos uma cena e adicionamos um nó CanvasLayer chamado HUD. "HUD" significa "heads-up display", um mostrador informativo que aparece como uma camada de sobreposição à visão do jogo.

O nó CanvasLayer nos permite desenhar nossos elementos de interface em uma camada acima do resto do jogo, de forma que as informações que ela mostrar não fiquem cobertas por quaisquer elementos do jogo, como o jogador ou os inimigos.

O HUD exibirá as seguintes informações:

- Pontuação, alterado por ScoreTimer.
- Uma mensagem, como "Game Over" ou "Prepare-se!"
- Um botão "Iniciar" para começar o jogo.

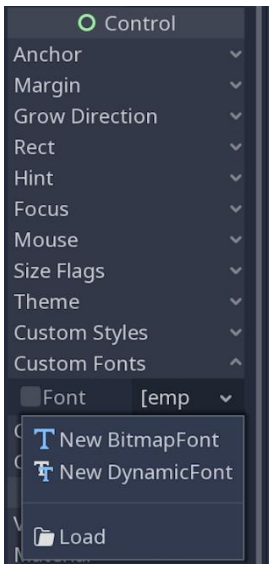
O nó básico para elementos de interface é Control. Para criar nossa interface, usaremos dois tipos de NÓS Control: Label e Button.

Crie os seguintes itens como filhos do nó HUD:

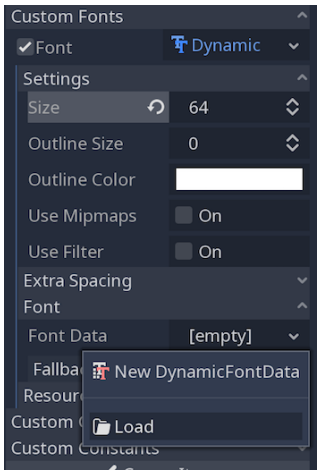
- Label nomeado ScoreLabel.
- Label nomeado MessageLabel.
- Button nomeado StartButton.
- Timer nomeado MessageTimer.

Clique no ScoreLabel e digite um número no campo Text "no Inspetor". A fonte padrão para os **NÓS** "Control" é pequena e não escala bem. Há um arquivo de fonte incluído nos assets do jogo chamado "Xolonium-Regular.ttf". Para usar esta fonte, faça o seguinte para cada um dos três **NÓS** Control:

1. Na propriedade "Custom Fonts" (Fontes Personalizadas), escolha "Novo DynamicFont"



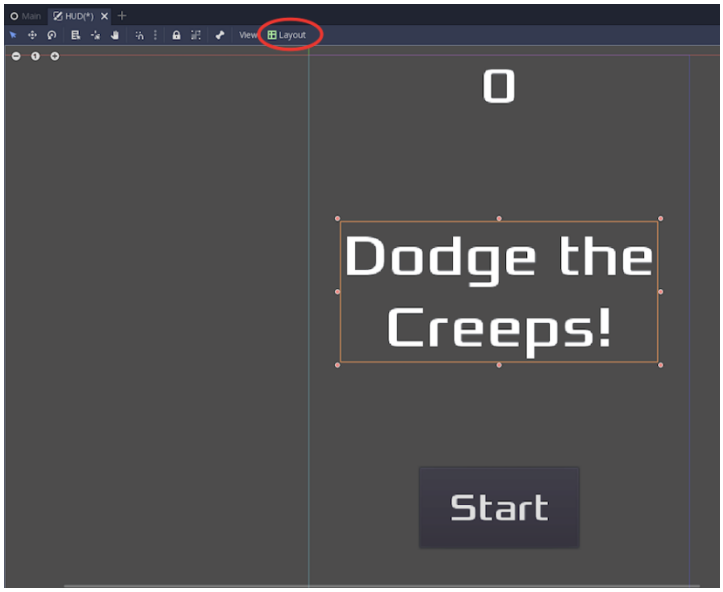
2. Clicamos na "DynamicFont" que adicionamos e em "Font/Font Data" (dados da fonte), escolha "Carregar" e selecione o arquivo "Xolonium-Regular.ttf". Você tem que definir o tamanho (Size) da fonte. Uma configuração de 64 funciona bem.



Depois de fazer isso no "ScoreLabel", você pode clicar na seta para baixo ao lado da propriedade DynamicFont e escolher "Copiar", depois "Colar" no mesmo local que os outros dois NÓS de Controle.

Nota: **Âncoras e Margens: NÓS** Control têm uma posição e um tamanho, mas eles também têm âncoras e margens. Âncoras definem a origem – o ponto de referência para as bordas do nó. As margens atualizam automaticamente quando você move ou redimensiona um nó de controle. Elas representam a distância das bordas do nó Control até sua âncora.

Organizamos os **NÓS** conforme indicado abaixo. Clicamos no botão "Disposição (Layout)" para definir a disposição de um nó de controle:



Você pode arrastar os **NÓS** para colocá-los manualmente ou, para um posicionamento mais preciso, usar as seguintes configurações:

ScoreLabel

- *Layout* : "Top Wide"
- *Texto*: 0
- *Alinhamento* : "Centralizado"

MessageLabel

- *Layout* : "Center" (Centro)
- *Text*: Desvie dos Bichos!
- *Alinhamento* : "Centralizado"
- *Autowrap* : "Ativo"

StartButton

- Text: Iniciar
- Layout: "Center Bottom"
- Margem (Margin) :
- Top: -200
- Inferior: -100

No MessageTimer, definimos o Wait Time (Tempo de Espera) para 2 e configuramos a propriedade One Shot (Apenas uma Vez) como "Ativo".

Agora, adicionamos este script ao HUD:

```
extends CanvasLayer
signal start_game
```

O sinal start_game diz ao **NÓ** Main (principal) que o botão foi pressionado.

```
func show_message(text):
    $Message.text = text
    $Message.show()
    $MessageTimer.start()
```

Esta função é chamada quando queremos mostrar uma mensagem temporariamente, como a "Prepare-se".

```
func show_game_over():
show_message("Game Over")
    # # Espere até que o MessageTimer tenha feito a contagem regressiva.
    yield($MessageTimer, "timeout")
    $Message.text = "Dodge the\nCreeps!"
    $Message.show()
    # Faça um cronômetro de disparo único e espere terminar.
    yield(get_tree().create_timer(1), "timeout")
    $StartButton.show()
```

Esta função é chamada quando o jogador perde. Ela mostrará "Game Over" por 2 segundos e depois retornará à tela de título após uma breve pausa e mostrará o botão "Iniciar".

Nota: Quando você precisa pausar por um breve tempo, uma alternativa ao do nó Timer é usar a função create_timer() do SceneTree. Isso pode ser muito útil para atrasar, como no código acima, onde queremos esperar um pouco de tempo antes de mostrar o botão "Iniciar".

```
func update_score(score):

    $ScoreLabel.text = str(score)
```

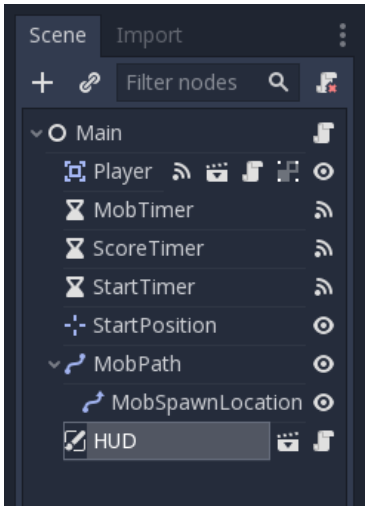
Esta função é chamada por "Main"sempre que a pontuação for alterada.

Conectamos o sinal "timeout()" do "MessageTimer" e o sinal pressed() do StartButton e adicione o seguinte código às novas funções:

```
func _on_StartButton_pressed():
    $StartButton.hide()
    emit_signal("start_game")

func _on_MessageTimer_timeout():
    $Message.hide()
```

Conectando HUD a Principal

Agora que terminamos de criar a cena HUD, salve-a e volte para a Principal. Crie uma instância da cena HUD como fez com a cena Jogador, e coloque-a no final da árvore. A árvore completa deveria se parecer assim, então confira se não falta alguma coisa:



Agora precisamos conectar a funcionalidade de HUD ao roteiro de Principal. Isso exige algumas adições à cena Principal:

Na guia Nó, conectamos o sinal " start_game" do HUD à função " new_game ()" do nó Principal, digitando "new_game" no "Receiver Method" na janela "Connect a Signal". Verificamos se o ícone de conexão verde agora aparece ao lado de "func new_game ()" no script.

Em new_game(), atualizamos o mostrador de pontuação e mostre a mensagem "Prepare-se":

```
$HUD.update_score(score)

$HUD.show_message("Get Ready")
```

Em game_over(), precisamos chamar a correspondente função de HUD:

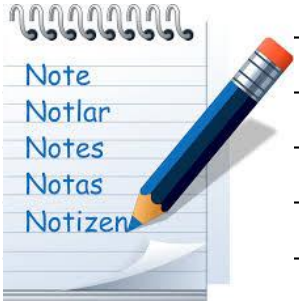
```
$HUD.show_game_over()
```

Finalmente, adicione isto a _on_ScoreTimer_timeout() para manter o mostrador em sincronia com as mudanças de pontuação:

```
$HUD.update_score(score)
```

Agora está tudo pronto para jogar! Clicamos no botão "Rodar o Projeto". Será solicitada a seleção de uma cena principal, então escolhemos, Principal.tscn.

ANOTAÇÕES

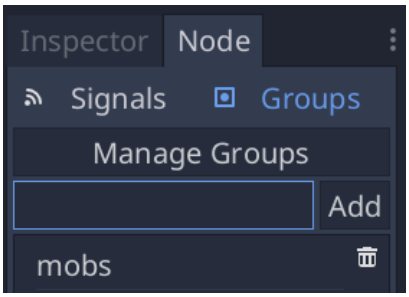


AULA 12

Removendo antigas criaturas

Se você jogar até o "Game Over" e iniciar um novo jogo, as criaturas do jogo anterior ainda poderão estar na tela. Seria melhor se todas elas desaparecessem no início de cada partida. Nós só precisamos de um jeito de *falar* para todos os inimigos se autodestruírem. Nós podemos fazer isso com a funcionalidade "group"(grupo).

Na cena do Inimigo, selecionamos o nó raiz e clique na aba "Nó" próxima ao Inspetor (No mesmo lugar onde você encontra os sinais do nó). Próximo a "Sinais", clique "Grupos", você pode digitar o nome do novo grupo e clicamos em "Adicionar".



Agora todos os inimigos estarão no grupo "inimigos". Podemos então adicionar a seguinte linha à função game_over() em Main:

```
get_tree().call_group("mobs", "queue_free")
```

A função "call_group()"chama a função passada como parâmetro em cada nó do grupo - neste caso nós estamos falando para cada inimigo se autodestruir.

Terminando

Agora completamos toda a funcionalidade do nosso jogo. Abaixo estão alguns passos restantes para adicionar um pouco mais de "sabor" para melhorar a experiência do jogo. Sinta-se livre para expandir a jogabilidade com suas próprias ideias.

Plano de fundo

O plano de fundo padrão cor cinza não é muito apelativo, então vamos mudar sua cor. Uma maneira de fazer isso é usar um Nó ColorRect ("retângulo colorido"). Vamos fazer ele ser o primeiro Nó de Principal para que ele seja desenhado por trás dos outros NÓS. ColorRect tem apenas uma propriedade: Color ("cor").

Escolhemos uma cor que gostamos e selecionamos, em "Layout", "Full Rect" para cobrir toda a tela.

Você também pode adicionar uma imagem de plano de fundo, se tiver uma, ao usar um nó TextureRect.

Efeitos sonoros

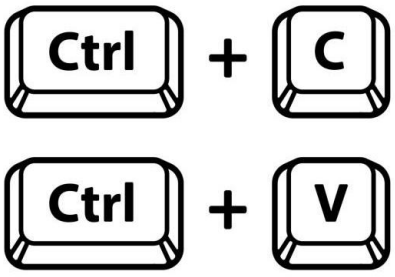
Som e música podem ser a forma mais efetiva de adicionar um atrativo à experiência de jogo. Na pasta de ativos do seu jogo, temos dois arquivos de áudio: "House In a Forest Loop.ogg" para música de fundo e "gameover.wav" para quando o jogador perde.

Adicionamos dois NÓS AudioStreamPlayer como filhos de Principal. Nomeie um deles como Música e o outro como SomDeMorte. Em cada um, clique na propriedade Stream ("fluxo"), selecione "Carregar", e escolha o arquivo sonoro correspondente.

Para reproduzir a música, adicionamos $Musica.play() "na função" new_game() e $Musica.stop() na função game_over().

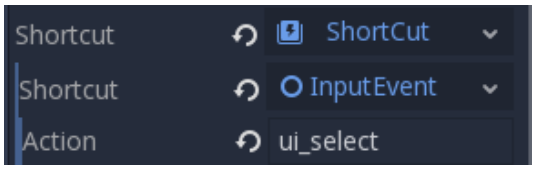
Por fim, adicionamos $SomDeMorte.play() na função game_over().

Atalho de teclado



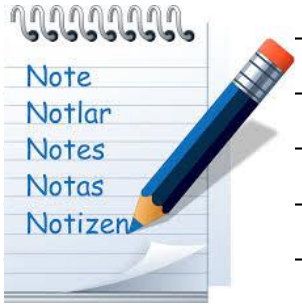
Como o jogo é jogado com controles de teclado, seria conveniente se pudéssemos iniciar o jogo pressionando uma tecla do teclado. Uma maneira de fazer isso é usando a propriedade "Atalho" do nó Button.

Na cena HUD, selecionamos o StartButton e encontrar, a propriedade *Atalho* no Inspetor. Selecionamos "Novo atalho" e clicamos no item "Atalho". Uma segunda propriedade *Atalho* aparecerá. Selecionamos "New InputEventAction" e clicamos no novo "InputEventAction". Finalmente, na propriedade *Action*, digitamos o nome "ui_select". Este é o evento de entrada padrão associado à barra de espaço.



Agora, quando o botão iniciar aparecer, você pode clicar nele ou pressionar :kbd: *Barra de Espaço* para iniciar o jogo.

ANOTAÇÕES



AULA 13

Exportando

Quando você tem um jogo em funcionamento, provavelmente quer compartilhar seu sucesso com os outros. No entanto, não é prático pedir a seus amigos que façam o download do Godot apenas para que possam abrir seu projeto. Em vez disso, você pode exportar seu projeto, convertendo-o em um "pacote" que pode ser executado por qualquer pessoa.

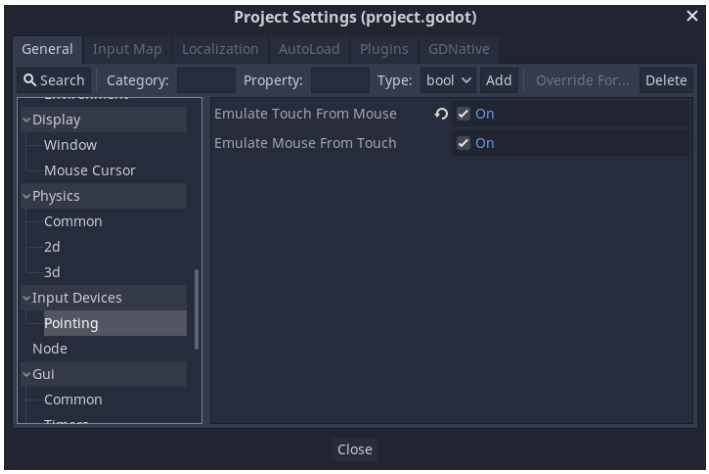
A maneira como você exporta seu jogo depende em qual plataforma você está objetivando. Neste tutorial, você aprenderá como exportar o jogo "Dodge the Creeps" para Android.

Preparando o projeto

Em "Dodge the Creeps", usamos controles de teclado para mover o personagem do jogador. Isso é bom se o seu jogo está sendo jogado em uma plataforma de PC, mas em um telefone ou tablet, você precisa suportar a entrada de tela sensível ao toque. Como um evento de clique pode ser tratado da mesma forma que um evento de toque, convertemos o jogo em um estilo de entrada de clicar-e-mover.

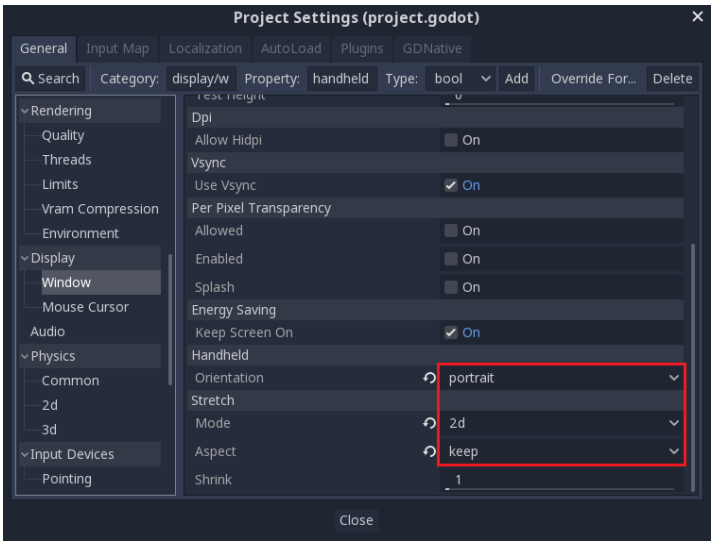
Por padrão o Godot emula o mouse quando há entrada de toque. Quer dizer que se tudo é programado para acontecer em um evento de mouse, o toque

vai ativar da mesma forma. O Godot também pode emular o toque a partir de cliques do mouse, o que precisaremos para poder jogar nosso jogo no computador depois que trocarmos para entrada de toque. Nas "Configurações do Projeto" em *Input Devices* e *Pointing*, ligue a opção *Emulate Touch From Mouse*.



Também queremos ter certeza de que o jogo mudará de tamanho consistentemente em telas de diferentes tamanhos, então nas configurações do projeto vá para *Display* e clique em *Window*. Nas opções de *Stretch*, mude *Mode* para "2d" e *Aspect* para "keep".

Desde que já estamos nas configurações da *Janela*, também devemos definir a *Orientação* para "portrait" em *Handheld*.



Em seguida, precisamos modificar o script Player.gd para alterar o método de entrada. Removeremos as entradas principais e faremos com que o jogador se mova em direção a um "alvo" definido pelo evento de toque (ou clique).

Aqui está o script completo para o jogador, com comentários observando o que mudamos:

```
extends Area2D

signal hit

export var speed = 400
var screen_size
# Adicione esta variável para manter a posição clicada.
var target = Vector2()

func _ready():
    hide()
    screen_size = get_viewport_rect().size

func start(pos):
    position = pos
    # O alvo inicial é a posição inicial.
    target = pos
    show()
    $CollisionShape2D.disabled = false

# Altere o alvo sempre que ocorrer um evento de toque.
func _input(event):
    if event is InputEventScreenTouch and event.pressed:
        target = event.position
```

```
func _process(delta):
    var velocity = Vector2()
    # Mova-se em direção ao alvo e pare quando estiver perto.
    if position.distance_to(target) > 10:
        velocity = target - position

# Removendo controles do teclado
#    if Input.is_action_pressed("ui_right"):
#        velocity.x += 1
#    if Input.is_action_pressed("ui_left"):
#        velocity.x -= 1
#    if Input.is_action_pressed("ui_down"):
#        velocity.y += 1
#    if Input.is_action_pressed("ui_up"):
#        velocity.y -= 1

    if velocity.length() > 0:
        velocity = velocity.normalized() * speed
        $AnimatedSprite.play()
    else:
        $AnimatedSprite.stop()

    position += velocity * delta
    # Ainda precisamos fixar a posição do jogador aqui porque em dispositivos que não correspondem à proporção do seu jogo, Godot tentará mantê-la o máximo possível criando bordas pretas, se necessário.
    # Sem clamp (), o jogador seria capaz de se mover sob essas bordas.
    position.x = clamp(position.x, 0, screen_size.x)
    position.y = clamp(position.y, 0, screen_size.y)

    if velocity.x != 0:
        $AnimatedSprite.animation = "walk"
```

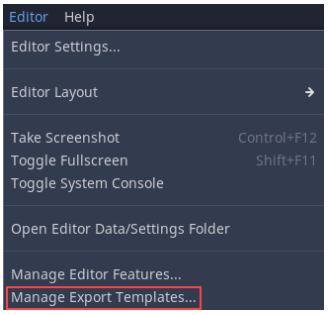
```
            $AnimatedSprite.flip_v = false
            $AnimatedSprite.flip_h = velocity.x < 0
        elif velocity.y != 0:
            $AnimatedSprite.animation = "up"
            $AnimatedSprite.flip_v = velocity.y > 0

    func _on_Player_body_entered( body ):
        hide()
        emit_signal("hit")
        $CollisionShape2D.set_deferred("disabled", true)
```

Definindo uma cena principal

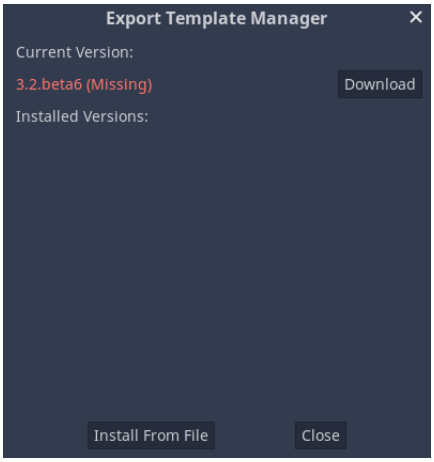
A cena principal é na qual o seu jogo iniciará. Em *Projeto -> Opções do Projeto -> Aplicação -> Rodar*, defina *Cena Principal* para "Main.tscn" clicando no ícone da pasta e selecionando-a.

Exportar modelos

Para exportar, você precisa baixar os *export templates* de http://godotengine.org/download. Esses modelos são versões otimizadas do mecanismo sem o editor previamente compilado para cada plataforma. Você também pode baixá-los em Godot clicando em *Editor -> Gerenciar Modelos de Exportação*:



Na janela que aparece, você pode clicar em "Download" para obter a versão do modelo que corresponde à sua versão do Godot.

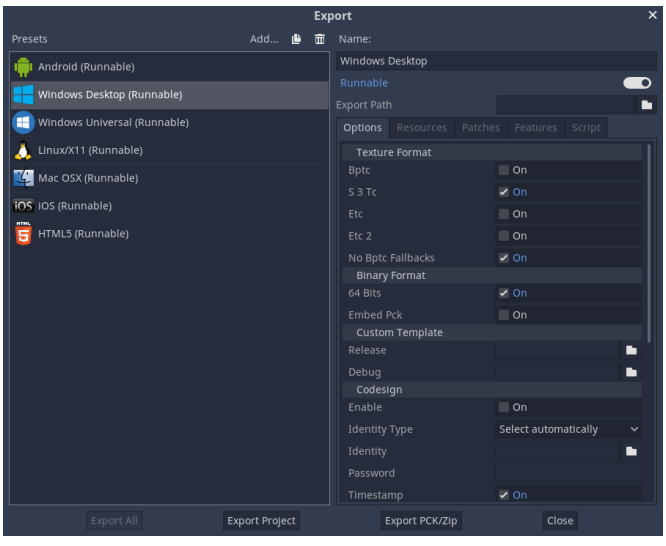


Nota: Se você atualizar o Godot, deverá baixar modelos que correspondam à nova versão ou seus projetos exportados podem não funcionar corretamente.

Predefinições de Exportação

Em seguida, você pode definir as configurações de exportação clicando em *Projeto -> Exportar*.

Crie uma predefinição de exportação clicando em "Adicionar ..." e selecionando uma plataforma. Você pode fazer quantas predefinições desejar com configurações diferentes.



Na parte inferior da janela há dois botões. "Exportar PCK / ZIP" cria apenas uma versão compactada dos dados do seu projeto. Isso não inclui um executável, portanto, o projeto não pode ser executado por conta própria.

O segundo botão, "Exportar Projeto", cria uma versão executável completa do seu jogo, como um *.apk* para o Android ou um *.exe* para o Windows.

Nas abas "Recursos" e "Funcionalidades" você pode personalizar como o jogo é exportado para cada plataforma. Podemos deixar essas configurações como estão por enquanto.

Exportando por plataforma

Nesta seção, vamos percorrer o processo da plataforma Android, incluindo qualquer software ou requisitos adicionais que você precise.

Antes de exportar seu projeto para Android, você deve fazer o download do seguinte software:

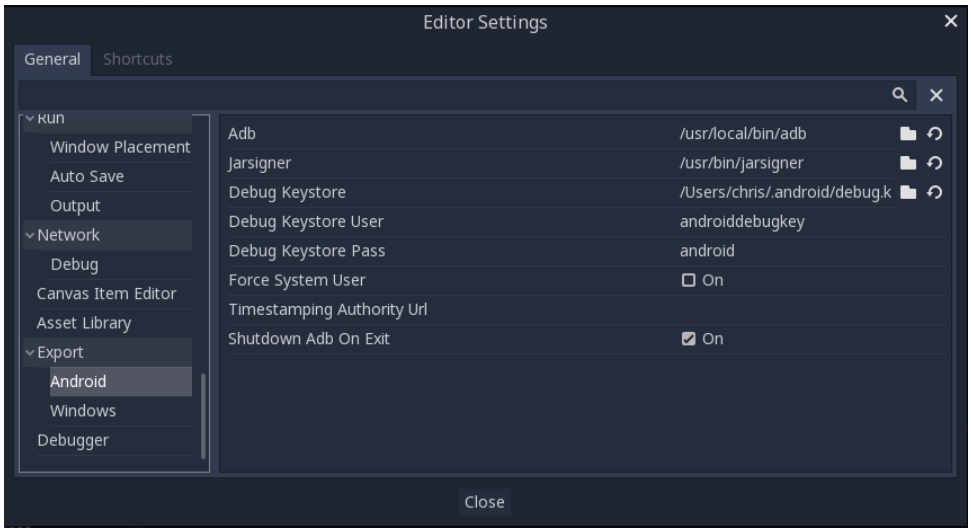
- Android SDK: https://developer.Android.com/studio/
- Open JDK(A versão 8 é necessária, versões mais recentes não funcionarão): https://adoptopenjdk.net/index.html

Quando você executar o Android Studio pela primeira vez, clique em *Configure -> SDK Manager* e instale "Android SDK Platform Tools". Isso instala a ferramenta de linha de comando *adb* que o Godot usa para se comunicar com o seu dispositivo.

Em seguida, crie um keystore de depuração executando o seguinte comando na linha de comandos do seu sistema:

```
keytool -keyalg RSA -genkeypair -alias Androiddebugkey -keypass Android -keystore debug.keystore -storepass Android -dname "CN=Android Debug,O=Android,C=US" -validity 9999
```

Clicamos em *Editor -> Configurações do Editor* no Godot e selecione a seção *Export / Android*. Aqui, você precisa definir os caminhos para os aplicativos do Android SDK em seu sistema e o local do keystore que você acabou de criar.



Agora você está pronto para exportar. Clicamos em *Projeto -> Exportar* e adicionamos uma predefinição para Android (veja acima). Selecionamos as predefinições para Android e em *Opções* vá para *Screen* e mude *Orientation* para "Portrait".

Clique no botão "Exportar Projeto" e o Godot criará um APK que você pode baixar no seu dispositivo. Para fazer isso na linha de comando, use o seguinte:

```
adb install dodge.apk
```

Se o seu sistema for compatível, a conexão de um dispositivo Android compatível fará com que um botão "Implantação com um clique" apareça na área de botões de teste de jogo (playtest) do Godot:

Clique neste botão cria o APK e o copie para o seu dispositivo em uma única etapa.

Conteúdo extra

Filosofia de Design do Godot

Cada motor de jogo é diferente e atende a diferentes necessidades. Não apenas oferecem uma gama de recursos, mas o design de cada mecanismo é único. Isso leva a diferentes fluxos de trabalho e diferentes maneiras de formar as estruturas dos seus jogos. Tudo isso decorre de suas respectivas filosofias de design.

Design e composição orientada a objetos

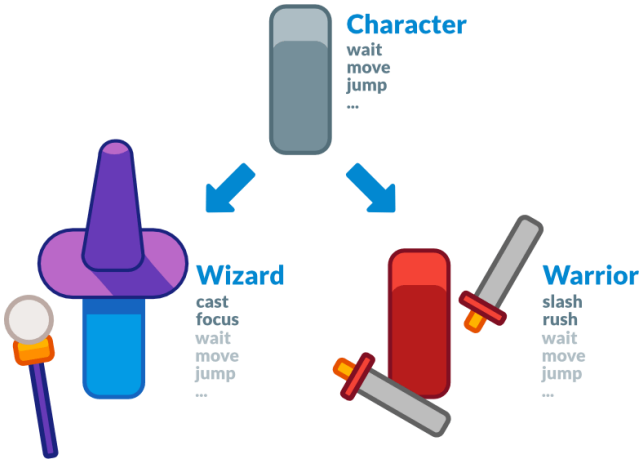
O Godot adota o design orientado a objetos em seu núcleo, com seu sistema de cena flexível e hierarquia de (**NÓS**). Ele tenta ficar longe de padrões estritos de programação para oferecer uma maneira intuitiva de estruturar seu jogo.

Por um lado, o Godot permite compor ou agregar cenas. É como prefabs aninhados: você pode criar uma cena BlinkingLight e uma cena BrokenLantern que usa o BlinkingLight. Em seguida, crie uma cidade repleta de BrokenLanterns. Mude a cor do BlinkingLight, salve e todos os BrokenLanterns na cidade serão atualizados instantaneamente.

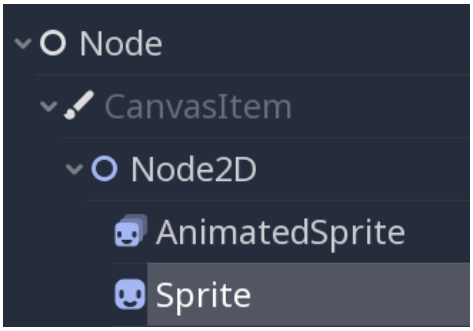
Além disso, você pode herdar de qualquer cena. Uma cena Godot pode ser uma arma, um personagem, um item, uma porta, um nível, parte de um nível, qualquer coisa que você quiser. Funciona como uma classe em código puro, com a exceção de que você é livre para projetá-la usando o editor, usando apenas o código ou misturando e combinando os dois.

É diferente das prefabs que você encontra em vários motores 3D, pois você pode herdar e estender essas cenas. Você pode criar um Mago que

estenda seu Personagem. Modifique o Personagem no editor e o Mago também atualizará. Isso te ajuda a criar seus projetos de modo que sua estrutura corresponda ao design do jogo.



Observe também que o Godot oferece muitos tipos diferentes de objetos chamados **NÓS**, cada um com um propósito específico. Os **NÓS** são parte de uma árvore e sempre herdam de seus pais até a classe Node. Embora o motor tenha componentes como formas de colisão, eles são a exceção, não a regra.



Sprite é um Node2D, um CanvasItem e um Node. Ele tem todas as propriedades e recursos de suas três classes principais, como transformações ou a capacidade de desenhar formas personalizadas e renderizar com um sombreador personalizado.

Pacote com tudo incluído

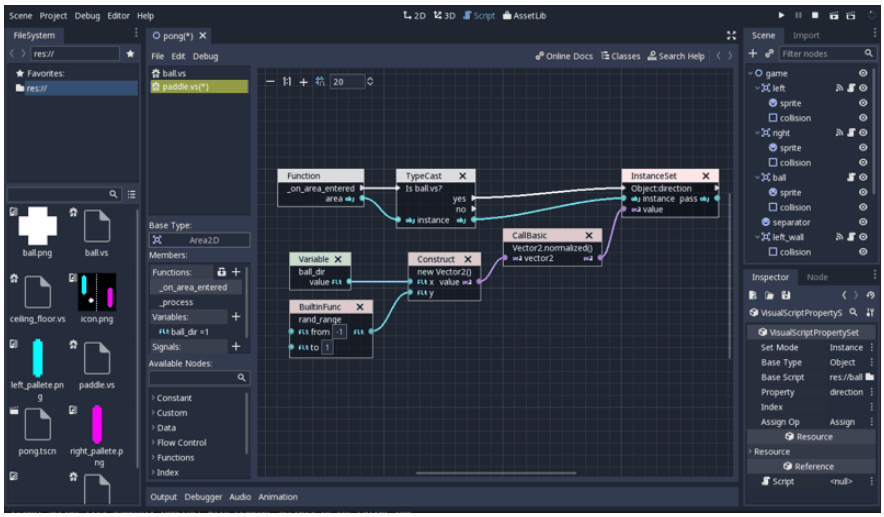
O Godot tenta fornecer suas próprias ferramentas para responder às necessidades mais comuns. Tem um editor de script dedicado, um editor de animação, um editor de tilemaps, um editor de shaders, um depurador, um profiler, a capacidade de hot-reload local e em dispositivos remotos etc.

O objetivo é oferecer um pacote completo para criar jogos e uma experiência de usuário contínua. Você ainda pode trabalhar com programas externos desde que exista um plugin de importação para ele. Ou você pode criar um, como o Tiled Map Importer.

Isso também é, em parte, o porquê de o Godot oferecer suas próprias linguagens de programação GDscript e VisualScript, junto com C#. Elas são projetadas para as necessidades de desenvolvedores de jogos e designers de jogos, e estão totalmente integradas ao motor e ao editor.

O GDscript permite que você escreva um código simples usando uma sintaxe parecida com o Python, porém ele detecta tipos e oferece a característica de preenchimento automático de uma linguagem estática. Também é otimizado para código de jogo com tipos embutidos como Vetores e Cores.

Observe que com o GDNative, você pode escrever código de alto desempenho usando linguagens compiladas como C, C++, Rust ou Python (usando o compilador Cython) sem recompilar a engine.



VisualScript é uma linguagem de programação baseada em **NÓS** que se integra bem ao editor. Você pode arrastar e soltar **NÓS** ou recursos no gráfico para criar blocos de código.

Observe que o espaço de trabalho 3D não possui tantas ferramentas quanto o espaço de trabalho 2D. Você precisará de programas externos ou complementos para editar terrenos, animar caracteres complexos e assim por diante. O Godot fornece uma API completa para estender a funcionalidade do editor usando o código do jogo.

Engine 2D e 3D separados

O Godot oferece mecanismos dedicados de renderização 2D e 3D. Como resultado, a unidade base para cenas 2D é pixels. Mesmo que os mecanismos sejam separados, você pode renderizar 2D em 3D, 3D em 2D e sobrepor sprites 2D e fazer a interface em seu mundo 3D.

Design de interfaces com os NÓS de Controle

Telas de computadores, celulares e TVs existem em todas as formas e tamanhos. Para enviar um jogo, será necessário suportar diferentes proporções e resoluções de tela. Pode ser difícil criar interfaces responsivas que se adaptem a todas as plataformas. Felizmente, o Godot vem com ferramentas robustas para projetar e gerenciar de forma responsiva a interface de usuário.

Esta guia irá ajudá-lo com o design da interface do usuário. Você vai aprender:

- Os cinco **NÓS** de controle mais úteis para criar sua interface de jogos

- Como trabalhar com a âncora de elementos da interface do usuário
- Como colocar e organizar eficientemente sua interface de usuário usando contêineres
- Os cinco contêineres mais comuns.

Para desenhar sua UI, você irá usar os **NÓS** de Controle. Eles são os **NÓS** com ícones verdes no editor. Existem dezenas deles, para criar qualquer coisa desde barras de vida até aplicações complexas. O próprio editor do Godot é construído usando (**NÓS**) Controle.

Os **NÓS** de Controle possuem propriedades exclusivas que permitem que eles funcionem bem uns com os outros. Outros **NÓS** visuais, como Node2D e Sprite não possuem estas capacidades. Então para facilitar sua vida use os **NÓS** de Controle sempre que possível quando for construir suas UIs.

Todos os **NÓS** de controle compartilham as mesmas propriedades principais:

- Âncora (anchor)
- Retângulo delimitador (Bounding rectangle)
- Foco e foco vizinho (Focus e focus neighbor)
- Sinalizadores de tamanho (Size flags)
- Margem (Margin)
- O tema opcional da interface do usuário (The optional UI theme)

Depois de entender os conceitos básicos do nó de Controle, você levará menos tempo para aprender todos os **NÓS** que derivam dele.

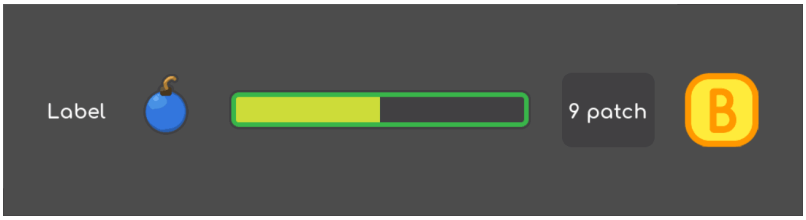
Os 5 elementos de interface do usuário mais comuns:

Godot vem com dezenas de **NÓS** de controle. Muitos deles estão aqui para ajudá-lo a criar plug-ins de editor e aplicações.

Para a maioria dos jogos, você só precisará de cinco tipos de elementos de interface do usuário e alguns contêineres. Esses cinco **NÓS** de controle são:

1. Label: para exibir texto

2. TextureRect: usado principalmente para planos de fundos, ou tudo o que deveria ser uma imagem estática
3. TextureProgress: para barras de vida, barras de carregamento, horizontais, verticais ou radiais
4. NinePatchRect: para painéis escalonáveis
5. TextureButton: para criar botões



TextureRect

TextureRect exibe uma textura ou imagem dentro de uma interface do usuário. Parece semelhante ao **NÓ** Sprite, mas oferece vários modos de escala. Defina a propriedade Stretch Mode para alterar seu comportamento:

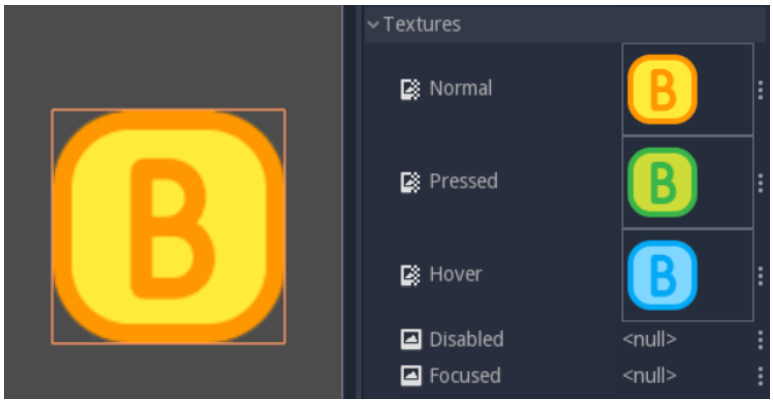
- Scale On Expand (compat) redimensiona a textura para se ajustar ao retângulo delimitador do nó, somente se a propriedade expand for true; caso contrário, ele se comporta como o modo Keep. Modo padrão para compatibilidade com versões anteriores.
- Scale redimensiona a textura para caber no retângulo delimitador do **NÓ**.
- Tile repete a textura, mas não faz com que ela escale.
- Keep e Keep Centered forçam a textura a permanecer no seu tamanho original, no canto superior esquerdo ou no centro do quadro, respectivamente.
- Keep Aspect e Keep Aspect Centered dimensionam a textura, mas a forçam-a a manter sua proporção original, no canto superior esquerdo ou no centro do quadro, respectivamente.
- Keep Aspect Covered funciona da mesma forma que Keep Aspect Centered, mas o lado mais curto se ajusta ao retângulo delimitador e o outro se encaixa aos limites do nó.

Assim como os NÓS Sprite, você pode modular a cor do TextureRect. Clique na propriedade Modulate e use o seletor de cores.

TextureButton

TextureButton é como TextureRect, exceto que elé tem 6 slots de textura: um para cada estado do botão. Na maioria das vezes, você vai utilizar as texturas Normal, Pressed e Hover. Focused é útil se sua interface escuta a entrada do teclado. O sexto slot de imagem, a Click Mask, permite definir a área clicável usando uma imagem em preto e branco puro de 1 bit.

Na seção Base Button, você encontrará algumas caixas de seleção que mudam como o botão se comporta. Quando o Toggle Mode está ativado, o botão alternará entre os estados ativo e normalmente quando você o pressionar. O Disabled faz com que seja desabilitado por padrão, caso em que usará a textura Disabled. O TextureButton compartilha algumas propriedades com o quadro de textura: ele tem uma propriedade modulate, para alterar sua cor e os modos Resize e Stretch para alterar seu comportamento de escala.

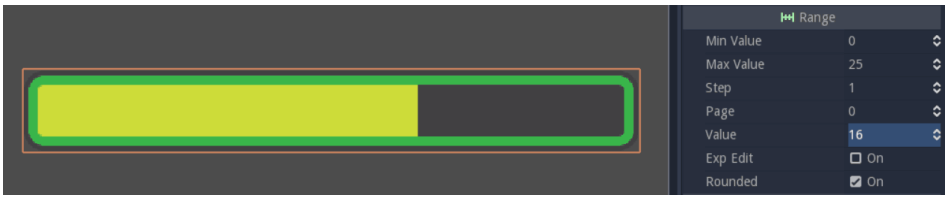


TextureProgress

Camadas TextureProgress usam até 3 sprites para criar uma barra de progresso. As texturas Under e Over se encaixam entre a de Progress, que exibe o valor da barra.

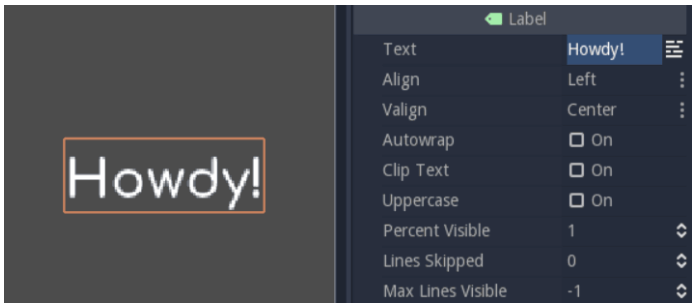
A propriedade Mode controla a direção na qual a barra cresce: horizontalmente, verticalmente ou radialmente. Se você o definir como radial, as propriedades Initial Angle e Fill Degrees permitem limitar o intervalo do medidor.

Para animar a barra, você vai querer olhar para a seção range. Defina as propriedades Min e Max para definir o intervalo do medidor. Por exemplo, para representar a vida de um personagem, você vai querer definir Min para 0, " e "Max para a vida máxima do personagem. Altere a propriedade Value para atualizar a barra. Se você deixar os valores Min e Max para o padrão de 0 e 100, " e definir a propriedade "Value para 40, 40% da textura Progress aparecerá e 60% dela permanecerá oculta.



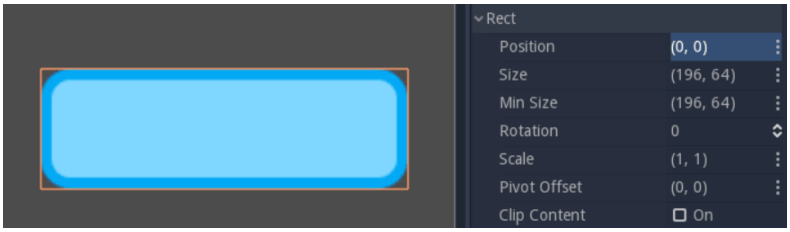
Rótulo

O Label exibe texto na tela. Você encontrará todas as suas propriedades na seção Label, no Inspetor. Escreva o texto na propriedade Text e verifique o Autowrap se você quiser que ele respeite o tamanho da caixa de texto. Se o Autowrap estiver desativado, você não poderá dimensionar o nó. Você pode alinhar o texto horizontalmente e verticalmente com Align e Valign, respectivamente.



NinePatchRect

NinePatchRect divide uma textura em 3 linhas e 3 colunas. O centro e os lados são ladrilhados quando você redimensiona a textura, mas nunca redimensiona os cantos. É útil criar painéis, caixas de diálogo e planos de fundo escaláveis para sua interface do usuário.



Existem dois fluxos de trabalho para criar interfaces de usuário responsivas e existem dois fluxos de trabalho para criar interfaces escalonáveis e flexíveis no Godot:

Você tem muitos **NÓS** de contêiner à sua disposição que dimensionam e posicionam elementos de interface do usuário para você. Eles assumem o controle de seus filhos.

Do outro lado, você tem o menu de layout. Ele ajuda você a ancorar, posicionar e redimensionar um elemento de interface do usuário dentro de seu pai.

As duas abordagens nem sempre são compatíveis. Como um contêiner controla seus filhos, você não pode usar o menu de layout neles. Cada contêiner tem um efeito específico, então você pode precisar aninhar vários deles para obter uma interface funcional. Com a abordagem de layout você trabalha de

baixo para cima nos filhos. Como você não insere contêineres extras na cena, pode criar hierarquias mais limpas, mas é mais difícil organizar itens em uma linha, coluna, grade etc.

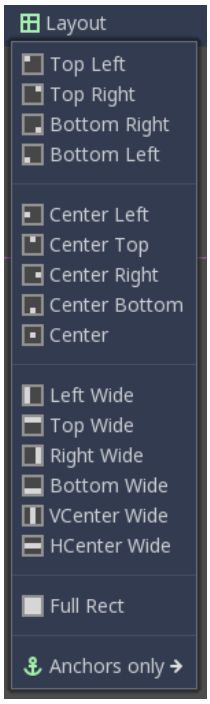
À medida que você cria interfaces de usuário para seus jogos e ferramentas, você desenvolverá uma noção do que se encaixa melhor em cada situação.

Coloque os elementos da interface do usuário precisamente com âncoras

NÓS, de controle têm uma posição e tamanho, mas eles também têm âncoras e margens. As âncoras definem a origem ou o ponto de referência para as arestas esquerda, superior, direita e inferior do nó. Altere qualquer uma das quatro âncoras para alterar o ponto de referência das margens.

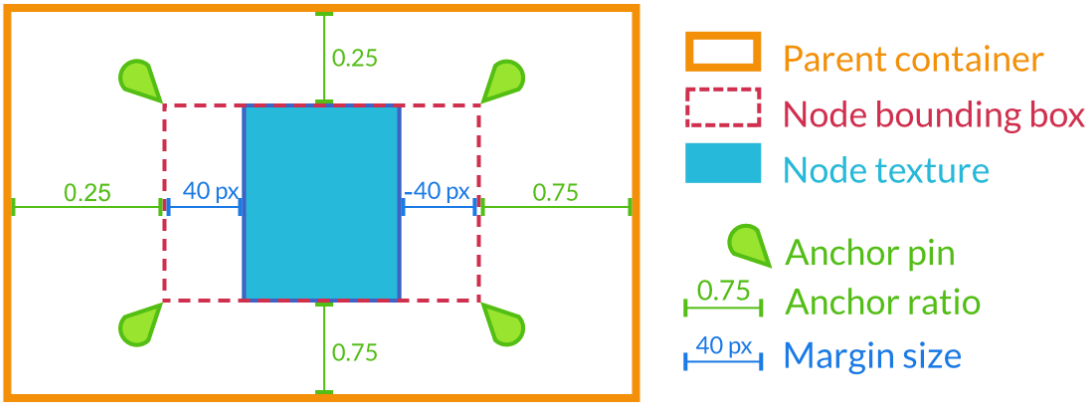
Como alterar a âncora

Como qualquer propriedade, você pode editar os 4 pontos de ancoragem no Inspetor, mas essa não é a maneira mais conveniente. Quando você seleciona um **NÓ** de controle, o menu de layout aparece acima da janela de exibição, na barra de ferramentas. Ele fornece uma lista de ícones para definir todas as 4 âncoras com um único clique, em vez de usar as 4 propriedades do inspetor. O menu de layout só será exibido quando você selecionar um nó de controle.



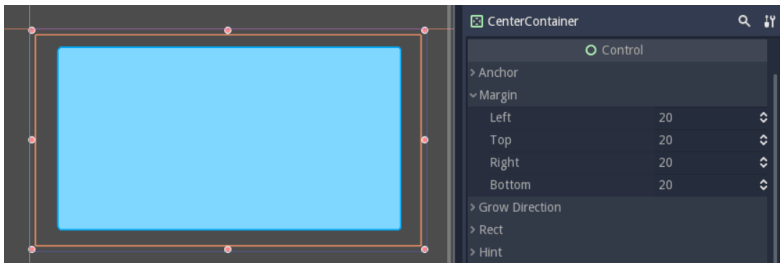
Âncoras são relativas ao contêiner pai

Cada âncora é um valor entre 0 e 1. Para as âncoras esquerda e superior, um valor de 0 significa que sem qualquer margem, as arestas do nó serão alinhadas às arestas esquerda e superior de seu pai. Para as bordas direita e inferior, um valor de 1 significa que eles se alinharão às bordas direita e inferior do contêiner pai. Por outro lado, as margens representam uma distância para a posição da âncora em pixels, enquanto as âncoras são relativas ao tamanho do contêiner pai.



As margens mudam com a âncora

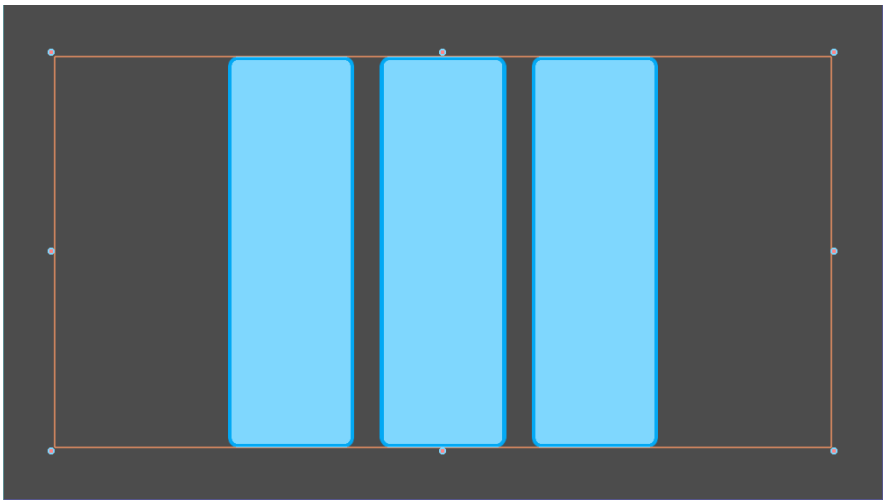
As margens são atualizadas automaticamente quando você move ou redimensiona um nó de controle. Elas representam a distância entre as bordas do nó de controle e sua âncora, que é relativa ao nó ou contêiner de controle pai. É por isso que os **NÓS** de controle devem estar sempre dentro de um contêiner, como veremos daqui a pouco. Se não houver pai, as margens serão relativas ao Retângulo delimitador do próprio nó, definido na seção Rect, no inspetor.



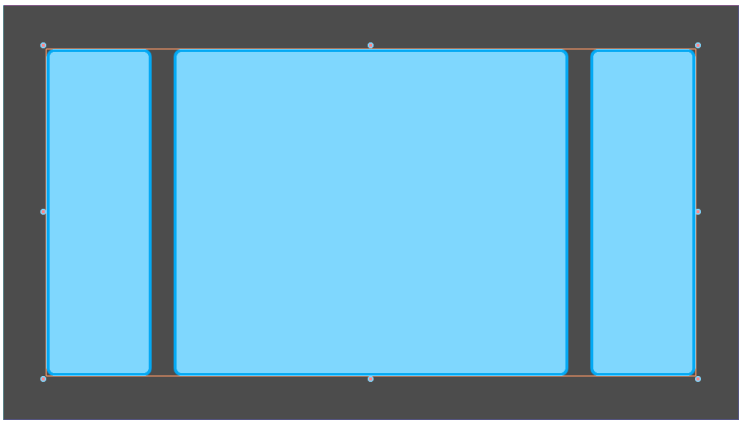
Tente alterar as âncoras ou aninhe seus **NÓS** de controle dentro de Contêineres: as margens serão atualizadas. Você raramente precisará editar as margens manualmente. Sempre tente encontrar um contêiner para ajudá-lo primeiro; o Godot vem com (**NÓS**) para resolver todos os casos comuns para você. Precisa adicionar espaço entre uma barra de vida e a borda da tela? Use o MarginContainer. Quer construir um menu vertical? Use o VBoxContainer.

Use tags de tamanho para alterar como os elementos da interface do usuário preenchem o espaço disponível:

Todo nó de controle possui sinalizadores de tamanho. Eles dizem aos contêineres como os elementos da interface do usuário devem ser dimensionados. Se você adicionar o sinalizador "Fill" à propriedade Horizontal ou Vertical, a caixa delimitadora do nó ocupará todo o espaço possível, mas respeitará seus irmãos e manterá seu tamanho. Se houver 3 **NÓS** TextureRect em um HBoxContainer, com os sinalizadores "Fill" nos dois eixos, cada um deles ocupará um terço do espaço disponível, mas não mais. O contêiner assumirá o nó e o redimensionará automaticamente.



O sinalizador "Expand" permite que o elemento da interface do usuário ocupe todo o espaço possível e empurre seus irmãos. Seu retângulo de delimitação crescerá nas bordas de seu pai ou até que seja bloqueado por outro nó de interface do usuário.



Você precisará de alguma prática para entender as tags de tamanho, já que o efeito delas pode mudar um pouco dependendo de como você configura sua interface.

Organize NÓS de controle automaticamente com contêineres

Os contêineres organizam automaticamente todos os **NÓS** filhos de controle, incluindo outros contêineres em linhas, colunas e mais. Use-os para adicionar preenchimento ao redor da interface ou dos **NÓS** centrais em seus retângulos delimitadores. Todos os contêineres internos são atualizados no editor, assim você vê o efeito instantaneamente.

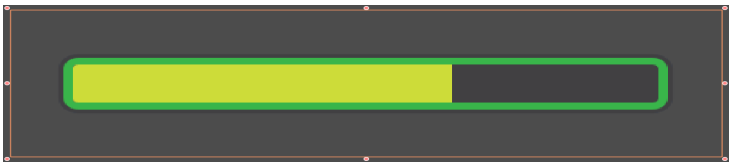
Os contêineres têm algumas propriedades especiais para controlar como eles organizam os elementos da interface do usuário. Para alterá-los, navegue até a seção Constantes personalizadas no Inspetor.

Os 5 contêineres mais úteis

Se você criar ferramentas, poderá precisar de todos os contêineres. Mas para a maioria dos jogos, um punhado será suficiente:

- MarginContainer: para adicionar margens em torno da parte da interface do usuário
- CenterContainer: para centralizar seus filhos em sua caixa delimitadora
- VboxContainer e HboxContainer: para organizar elementos da interface do usuário em linhas ou colunas.
- GridContainer: para organizar **NÓS** de controles em um padrão semelhante a uma grade
- CenterContainer: centraliza todos os seus filhos dentro de seu retângulo delimitador. Costuma ser usado para telas de título, se você quiser que as opções permaneçam no centro da viewport. Como ele centraliza tudo,

podemos usar quando desejarmos um único contêiner aninhado dentro dele. Se você usar texturas e botões, eles serão empilhados.



CenterContainer em ação. A barra de vida é centralizada dentro de seu contêiner pai.

O MarginContainer adiciona uma margem em qualquer lado dos **NÓS** filhos. Adicione um MarginContainer que englobe toda a viewport para adicionar uma separação entre a borda da janela e a interface do usuário. Você pode definir uma margem no lado superior, esquerdo, direito ou inferior do contêiner. Não é necessário marcar a caixa de seleção: clique na caixa de valor correspondente e digite qualquer número. Ele será ativado automaticamente.

O MarginContainer adiciona uma margem de 40px ao redor da interface do usuário do jogo

Existem dois BoxContainers: VBoxContainer e HBoxContainer. Você não pode adicionar o próprio nó BoxContainer, pois ele é uma classe auxiliar, mas você pode usar contêineres verticais e horizontais. Eles organizam (**NÓS**) em linhas ou colunas. Use-os para alinhar itens em uma loja ou para construir grades complexas com linhas e colunas de tamanhos diferentes, pois você pode aninhá-los da forma que desejar.

O HBoxContainer alinha horizontalmente os elementos da interface do usuário

O VBoxContainer organiza automaticamente seus filhos em uma coluna. Isso os coloca um após o outro. Se você usar o parâmetro de separação, ele deixará um espaço entre seus filhos. HBoxContainer organiza elementos da interface do usuário em uma linha. É semelhante ao VBoxContainer, com um método extra add_spacer para adicionar um nó de controle espaçador antes de seu primeiro filho ou depois de seu último filho, a partir de um script.

O GridContainer permite organizar elementos da interface do usuário em um padrão semelhante a uma grade. Você só pode controlar o número de colunas que possui e definir o número de linhas sozinho, com base na contagem de seus filhos. Se você tem nove filhos e três colunas, você terá 9 ÷ 3 = 3 linhas. Adicione mais três filhos e você terá quatro linhas. Em outras palavras, ele criará linhas à medida que você adicionar mais texturas e botões. Como os contêineres de caixa, ele possui duas propriedades para definir a separação vertical e horizontal entre as linhas e colunas, respectivamente.

Um GridContainer com 2 colunas. Ele dimensiona cada coluna automaticamente.

NÓS e recursos

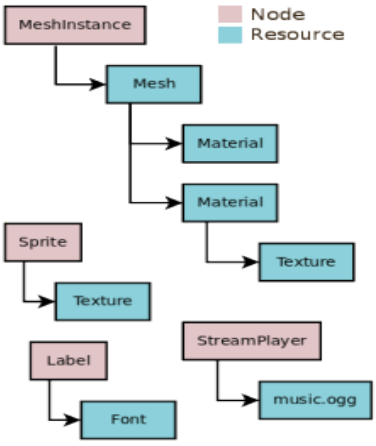
Até agora, focamos muito na classe de (**NÓS**) no Godot, já que a maioria dos comportamentos e funcionalidades do motor são implementados através deles. Existe outro tipo de dados que é igualmente importante: Recurso.

NÓS, dão funcionalidade: eles desenham sprites, modelos 3D, simulam física, organizam a interface do usuário etc. Recursos são containers de dados. Eles não fazem nada sozinhos: ao invés disso, os **NÓS** usam os dados contidos em recursos.

Qualquer coisa que o Godot salva ou carrega do disco é um recurso, seja ele uma cena (um arquivo .tscn or .scn), uma imagem, um script... Exemplos de Recursos são: Textura, Script, Malha, Animação, Fluxo de áudio, Fonte, Tradução.

Quando um recurso é carregado do disco, é carregado só uma vez. Se houver uma cópia desse recurso já carregado na memória, tentar carregar o recurso novamente retornará a mesma cópia várias vezes. Como recursos são apenas contêineres de dados, não é necessário duplicá-los.

Todos os objetos em Godot (**NÓS**, Recursos ou qualquer outra coisa) podem exportar propriedades. Elas podem ser de vários tipos (como texto, número inteiro, Vector2 etc.), e qualquer desses tipos pode virar um recurso. Isso significa que os **NÓS** e os recursos podem conter recursos como propriedades:

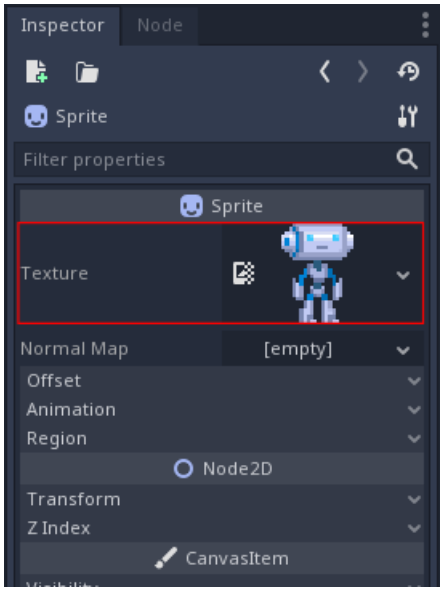


Externo vs embutido

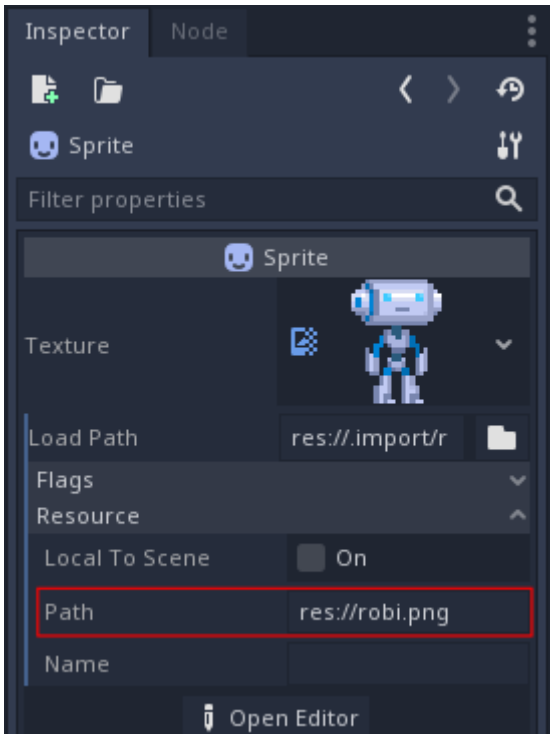
Existem duas maneiras de salvar recursos. Pode ser:

1. Externo a uma cena, salvo no disco como arquivos individuais.
2. Embutido, salvo dentro do arquivo .tscn ou .scn ao qual estão anexados.

Para ser mais específico, aqui está uma Textura em um nó Sprite:



Clicar no recurso permite que você veja e edite suas propriedades.



A propriedade caminho nos diz de onde o recurso vem. Nesse caso ele vem de uma imagem PNG chamada robi.png. Quando o recurso vem de um arquivo como esse, ele é um recurso externo. Se você apagar o caminho ou esse caminho estiver vazio, ele se transforma em um recurso embutido.

A mudança entre recursos embutidos e externos acontece quando você salva a cena. No exemplo acima, se você apaga o caminho "res://robi.png" e salva, o Godot vai salvar a imagem dentro do arquivo de cena .tscn.

Carregando recursos a partir do código

Existem duas maneiras de carregar recursos por código. Primeiro, você pode usar a função load() a qualquer momento:

```
var res = load("res://robi.png") #Godot carrega o recurso quando lê essa linha.

get_node("sprite").texture = res
```

Você pode também pré-carregar recursos. Diferente de load, essa função vai ler o arquivo do disco e carregá-lo ao compilar. Como resultado, você não pode chamar preload com um caminho variável: você preciso usar uma string constante.

```
var res = preload("res://robi.png") # Godot carrega o recurso e compila ao mesmo

                                                    tempo.

get_node("sprite").texture = res
```

Carregando cenas

Cenas também são recursos, mas há um detalhe. Cenas salvas no disco são recursos do tipo PackedScene. Isso significa que a cena é compactada dentro de um recurso.

Para obter uma instância da cena, o método PackedScene.instance() deve ser usado.

```
func _on_shoot():

var bullet = preload("res://bullet.tscn").instance()

add_child(bullet)
```

Esse método cria os NÓS na hierarquia da cena, configura-os e retorna o nó raiz da cena, que pode ser adicionado como filho de qualquer outro nó.

Essa abordagem tem várias vantagens. Como a função PackedScene.instance() é bastante rápida, novos inimigos, tiros, efeitos etc. podem ser adicionados ou removidos rapidamente, sem ter que carregá-los novamente do disco toda vez. É importante lembrar que, como sempre, imagens, malhas etc. são todos compartilhados entre as instâncias da cena.

Liberando recursos

Quando um "Recurso "não estiver mais em uso, ele será automaticamente liberado. Como, na maioria dos casos, os recursos estão contidos em **NÓS**, scripts ou outros recursos, quando um nó é removido ou liberado, todos os recursos filhos são liberados também se nenhum outro nó os usa.

Sistema de arquivos

O sistema de arquivos gerencia como os assets são armazenados e como são acessados. Um sistema de arquivos bem projetado também permite que vários desenvolvedores editem os mesmos arquivos fonte e assets enquanto colaboram. O Godot guarda todos os assets como arquivos em seu sistema de arquivos.

Implementação

O sistema de arquivos armazena recursos no disco. Qualquer coisa, desde um roteiro até uma cena ou uma imagem PNG, é um recurso para o motor. Se um recurso contiver propriedades que fazem referência a outros recursos no disco, os caminhos para esses recursos também serão incluídos. Se um recurso tiver sub-recursos embutidos, ele será salvo em um único arquivo junto com todos os sub-recursos inclusos. Por exemplo, um recurso de fonte geralmente é empacotado junto com as texturas de fonte.

O sistema de arquivos do Godot evita usar arquivos de metadados. Gerenciadores de assets e sistemas de controle de versão existentes são muito melhores do que qualquer coisa que poderíamos implementar, então o Godot tenta de tudo para funcionar bem com SVN, Git, Mercurial, Perforce etc.

Exemplo de conteúdo do sistema de arquivos:

- /project.godot
- /enemy/enemy.tscn
- /enemy/enemy.gd
- /enemy/enemysprite.png
- /player/player.gd
- project.godot

O arquivo project.godot é o arquivo de descrição do projeto e é sempre encontrado na raiz do projeto. Na verdade, sua localização define onde está a raiz. Este é o primeiro arquivo que o Godot procura ao abrir um projeto.

Delimitador de caminho

O godot suporta apenas / como delimitador de caminho, isso acontece por diversos motivos de portabilidade. Todas os sistemas operacionais suportam isso, até Windows, então a pasta C:\project\project.godot precisa ser digitada como C:/project/project.godot.

Caminho de recursos

Ao acessar recursos, usar o layout do sistema de arquivos do sistema operacional da máquina pode ser incômodo e não portátil. Para resolver este problema, o caminho especial res:// foi criado.

Este sistema de arquivos é para leitura e escrita somente quando se executa o projeto localmente a partir do editor. Quando exportado ou quando executado em dispositivos diferentes (como em telefones ou consoles ou executando a partir de DVD), o sistema de arquivos se tornará somente leitura e a gravação não será mais permitida.

Sistema de arquivos da máquina

Alternativamente, os caminhos do sistema de arquivos da máquina também podem ser usados, mas isso não é recomendado para um produto a ser lançado, já que esses caminhos não têm garantia de funcionar em todas as plataformas. No entanto, o uso de caminhos do sistema de arquivos da máquina pode ser útil ao escrever ferramentas de desenvolvimento no Godot.

Desvantagens

Existem algumas desvantagens nesse desenho simples do sistema de arquivos. A primeira questão é que a movimentação de ativos (renomeá-los ou movê-los de um caminho para outro dentro do projeto) quebrará as referências existentes a esses ativos. Essas referências terão que ser redefinidas para apontar para o novo local do ativo.

Para evitar isso, faça todas as suas movimentações, exclusões e renomeações de dentro do Godot, no painel Arquivos. Nunca mova ativos de fora do Godot, ou as dependências terão que ser consertadas manualmente.

Cenas

Nos tutoriais anteriores, tudo girava em torno do conceito de **NÓS**. Cenas são simplesmente uma coleção de **NÓS**. Elas se tornam ativos quando entram na árvore de cena.

Loop principal

A maneira como Godot trabalha internamente é a seguinte: *ref: OS <class_OS>*, que é a única classe de instância que é executada no começo. Depois disso, todos os drivers, servidores, linguagens de script, sistema de cena etc. são carregados.

Quando a inicialização estiver completa: ref: OS <class_OS> precisa ser fornecido a: ref: MainLoop <class_MainLoop> para ser executado. Até este ponto, tudo isso é funcionamento interno (você pode verificar o arquivo main / main.cpp no código-fonte, se você estiver interessado em ver como isso funciona internamente).

O programa do usuário, ou jogo, inicia no MainLoop. Essa classe tem alguns métodos, para inicialização, ocioso (retorno de chamada com sincronia de quadros), fixo (retorno de chamada com sincronização física) e entrada.

Novamente, isso é de baixo nível e ao fazer jogos no Godot, escrever seu próprio MainLoop raramente faz sentido.

Árvore de cena

Uma das maneiras de explicar como Godot funciona é que é um mecanismo de jogo de alto nível sobre um middleware de baixo nível.

O sistema de cena é o mecanismo do jogo, enquanto o: ref: OS <class_OS> e os servidores são a API de baixo nível.

O sistema de cena fornece seu próprio loop principal para o SO, SceneTree. Isso é automaticamente instanciado e definido ao executar uma cena, sem necessidade de qualquer trabalho extra.

É importante saber que essa classe existe porque tem alguns usos importantes:

Ele contém o Viewport raiz, para o qual uma cena é adicionada como filha quando é aberta pela primeira vez, para se tornar parte da Árvore da Cena (mais sobre isso a seguir).

Ele contém informações sobre os grupos e tem meios para chamar todos os NÓS de um grupo ou obter uma lista deles.

Ela contém alguma funcionalidade de estado global, como configurar o modo de pausa ou encerrar o processo.

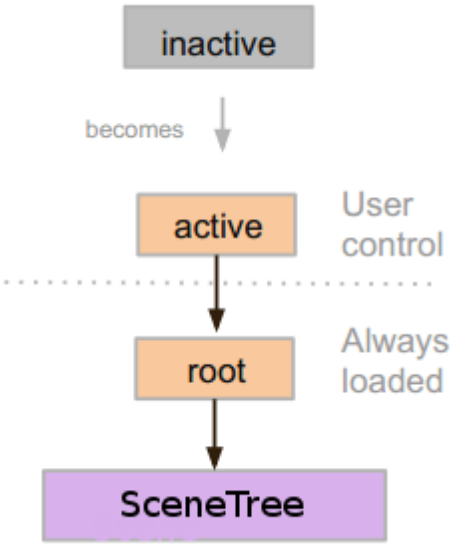
Quando um nó é parte da Scene Tree, o singleton SceneTree pode ser obtido chamando Node.get_tree().

Viewport raiz

A raiz: ref: `Viewport <class_Viewport> está sempre no topo da cena. De um nó, ele pode ser obtido de duas maneiras diferentes:

```
get_tree().get_root() # Acesso via loop principal da cena

get_node("/root") # Acesso via caminho absoluto da pasta
```

Este nó contém o viewport principal. Qualquer coisa que é filho de um Viewport é desenhada dentro dele por padrão, então faz sentido que o topo de todos os NÓS seja sempre um nó deste tipo, caso contrário nada seria visto.



Enquanto outras viewports podem ser criadas na cena (para efeitos de tela dividida e tal), esta é a única que nunca é criada pelo usuário. É criado automaticamente dentro do SceneTree.

Árvore de cena

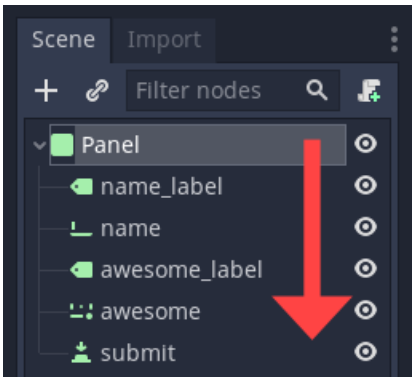
Quando um nó é conectado, direta ou indiretamente, à viewport raiz, ele se torna parte da * árvore de cena *.

Isso significa que, como explicado nos tutoriais anteriores, ele obterá os retornos de chamada _enter_tree () e _ready () (assim como _exit_tree ()).

Quando os **NÓS** entram na Scene Tree, eles se tornam ativos. Eles têm acesso a tudo o que precisam para processar, obter entrada, exibir visuais em 2D e 3D, enviar e receber notificações, reproduzir sons etc. Quando são removidos da árvore de cena, eles perdem essas habilidades.

Ordem da árvore

A maioria das operações de nó no Godot, como desenhar 2D, processar, ou obter notificações, é feita em ordem de árvore. Isso significa que pais e irmãos com uma classificação menor na ordem de árvore serão notificados antes do **NÓ** atual.



"Tornando-se ativo" entrando na * Scene Tree *

1. Uma cena é carregada do disco ou criada pelo script.
2. O nó raiz dessa cena (apenas uma raiz, lembra?) É adicionado como um filho da viewport "root" (de SceneTree) ou para qualquer filho ou neto dela.
3. Cada nó da cena recém-adicionada receberá a notificação "enter_tree" (callback _enter_tree () no GDScript) na ordem de cima para baixo.
4. Uma notificação extra, "ready" (retorno de chamada _ready () no GDScript) é fornecida por conveniência, quando um nó e todos os seus filhos estão dentro da cena ativa.
5. Quando uma cena (ou parte dela) é removida, eles recebem a notificação "exit scene" (retorno de chamada _exit_tree() na GDScript) na ordem de baixo para cima.

Alterando a cena atual

Depois que uma cena é carregada, muitas vezes é desejado mudar essa cena para outra. A maneira simples de fazer isso é usar a função: ref: SceneTree.change_scene() <class_SceneTree_method_change_scene>:

```
func _my_level_was_completed():

    get_tree().change_scene("res://levels/level2.tscn")
```

Em vez de usar caminhos de arquivo, também é possível usar recursos prontos: ref: PackedScene <class_PackedScene> usando a função equivalente: ref:

SceneTree.change_scene_to(PackedScene cena) <class_SceneTree_method_change_scene_to>:

```
var next_scene = preload("res://levels/level2.tscn")

func _my_level_was_completed():

    get_tree().change_scene_to(next_scene)
```

Estas são maneiras rápidas e úteis de trocar de cena, mas têm a desvantagem de o jogo travar até que a nova cena seja carregada e executada. Em algum momento do seu jogo, pode ser necessário criar telas de carregamento adequadas com barra de progresso, indicadores animados ou carregamento de thread (no fundo). Isso deve ser feito manualmente usando autoloads e Background loading.

Singletons (Carregamento Automático)

O sistema de cenas de Godot, embora poderoso e flexível, tem uma desvantagem: não existe um método para armazenar informações (por exemplo, pontuação ou inventário de um jogador) que seja necessário em mais de uma cena.

É possível resolver isso com algumas soluções alternativas, mas elas vêm com suas próprias limitações:

- Embora seja possível que uma cena carregue e descarregue outras cenas como seus filhos para armazenar informações comuns a essas cenas filhas, não será mais possível executar essas cenas sozinhas e esperar que elas funcionem corretamente.

- Embora a informação possa ser armazenada no disco em "user: // " e esta informação possa ser carregada por cenas que a exijam, salvar e carregar continuamente esses dados ao alterar cenas é trabalhoso e pode ser lento.

O padrão Singleton é uma ferramenta útil para resolver o caso de uso comum em que você precisa armazenar informações persistentes entre as cenas. Em nosso caso, é possível reutilizar a mesma cena ou classe para vários singletons, desde que tenham nomes diferentes.

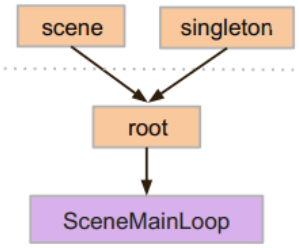
Estão sempre carregados, independentemente da cena em execução.

- Pode armazenar variáveis globais, como informações do jogador.
- Pode lidar com a troca de cenas e transições entre as cenas.
- Aja como um singleton, já que GDScript não oferece suporte a variáveis globais por design.

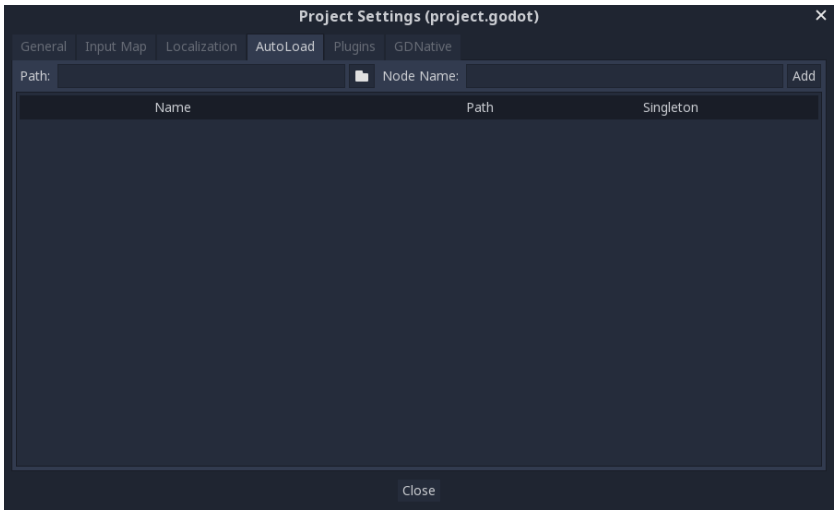
Carregamento automático e **NÓS** e scripts atendem essa necessidade.

Autoload

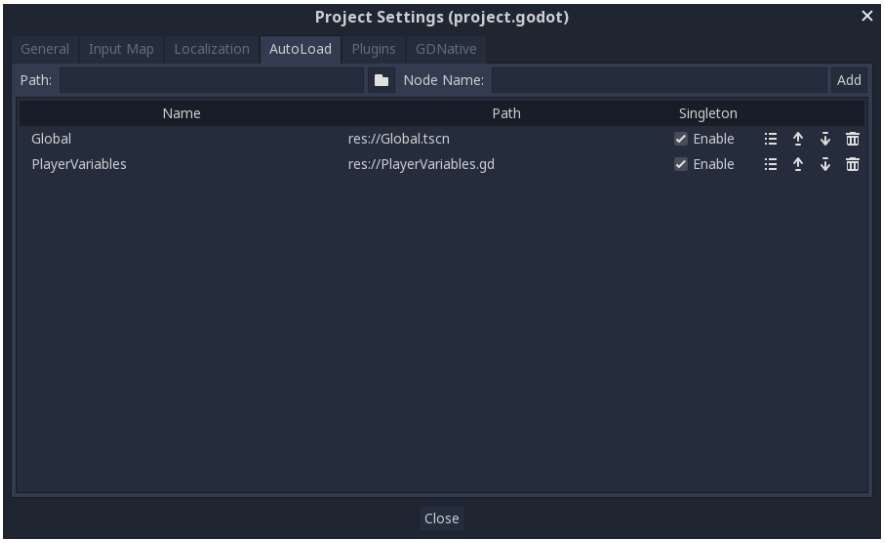
Você pode criar um AutoLoad para carregar uma cena ou um script que herda do Node.



Para carregar automaticamente uma cena ou script, selecione Projeto > Configurações do projeto no menu e mude para a guia Carregamento automático.



Aqui você pode adicionar qualquer número de cenas ou scripts. Cada entrada na lista requer um nome, designada como a propriedade nome do **NÓ**. A ordem em que as entradas são adicionadas à árvore da cena global pode ser manipulada usando as setas para cima e para baixo.



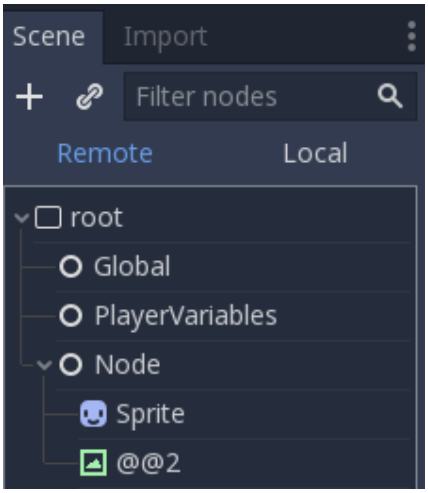
Isso significa que qualquer nó pode acessar um singleton chamado "PlayerVariables" com:

```
var player_vars = get_node("/root/PlayerVariables")

player_vars.health -= 10
```

Se a coluna ativar estiver marcada (que é o padrão), o singleton pode ser acessado diretamente sem a necessidade de get_node ():

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

Note que objetos carregados automaticamente (scripts e/ou cenas) são acessados como qualquer outro nó na árvore de cenas. Se você olhar numa cena executando, você verá os **NÓS** carregados automaticamente aparecerem:



ANOTAÇÕES