Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Leonardo Pereira Macedo

Desenvolvimento de um módulo de reconhecimento de voz para a *game engine Godot*

São Paulo
17 de dezembro de 2017

Desenvolvimento de um módulo de reconhecimento de voz para a *game engine Godot*

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Marco Dimas Gubitoso

São Paulo
17 de dezembro de 2017

Resumo

A área de jogos eletrônicos (*video games*) evoluiu muito desde o início da década da 70, quando começaram a ser comercializados. As principais causas estão relacionadas aos avanços em diferentes áreas da Computação.

Com o passar do tempo, surgiram as *game engines*: *frameworks* voltados especificamente para a criação de jogos, visando a facilitar o desenvolvimento e/ou algumas de suas etapas.

Focaremos em uma *game engine* em particular, *Godot* (Linietsky e Manzur, 2017a). Por possuir código aberto, este *software* permite a extensão de suas funcionalidades através da criação de novos módulos.

Este projeto busca implementar um módulo de reconhecimento de voz para *Godot*, depois demonstrando a nova capacidade em um jogo simples desenvolvido na própria plataforma.

Palavras-chave: *software*, *game engine*, *Godot*, desenvolvimento de módulo, extensão de funcionalidade, reconhecimento de voz.

Abstract

Video games have evolved considerably since the beginning of the 70's, when they started to be commercialized. The main reasons are related to several advances in different fields of Computer Science.

Over time, *game engines* were developed: *frameworks* designed specifically to assist on game creation, simplifying the process and/or some of its steps.

We will focus on a specific game engine, *Godot* (Linietsky e Manzur, 2017a). Since it is an open source project, it is possible to extend its functionalities by creating new modules.

This project's goal is to implement a speech recognition module for *Godot*, then presenting the new feature in a simple game, developed on the engine itself.

Keywords: software, game engine, *Godot*, module development, functionality extension, speech recognition.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Capítulo 1

Introdução

1.1 Motivação e objetivo

Hoje em dia, não há como negar que o mercado de *games* é um fenômeno mundial, gerando mais de US$ 91 bilhões em 2016 (SuperData Research, 2016). Comparado aos primeiros jogos, comercializados no início da década de 1970 (Wikipedia, 2017c), a evolução em diversas áreas da computação permitiu grandes avanços nos jogos criados. Inclui-se nisso a evolução dos computadores por conta da *Lei de Moore* (Wikipedia, 2017e), permitindo processamento mais rápido; *games* em 3D e gráficos cada vez mais sofisticados e realistas devido à Computação Gráfica; e adversários sofisticados e de raciocínio rápido com a Inteligência Artificial.

Junto aos próprios jogos, as tecnologias usadas para desenvolvê-los também tiveram progressos. Em especial, citamos as *game engines*, que podem ser descritas como "*frameworks* voltados especificamente para a criação de jogos" (Enger, 2013). Elas oferecem diversas ferramentas para acelerar o desenvolvimento de um jogo, como maior facilidade na manipulação gráfica e bibliotecas prontas para tratar colisões entre objetos. Além disso, como eficiência é um fator essencial para manter um bom valor de FPS (*Frames per Second*), as *engines* costumam ter sua base construída em linguagens rápidas e compiladas, como C e C++.

Focaremos em uma *game engine* em particular, ***Godot*** (Linietsky e Manzur, 2017a). O principal motivo de ter sido escolhida é por ser um *software* de código aberto, o que permite a qualquer pessoa baixar seu código fonte e fazer modificações. Em especial, a *engine* permite a criação de módulos para adicionar a ela novas funcionalidades.

Este trabalho visa a criar um novo módulo para *Godot*. Tal extensão adicionará funções simples de reconhecimento de voz, algo ainda inexistente no *software*. Feito isso, a nova funcionalidade será demonstrada em um jogo simples criado nessa *engine*.

1.2 Organização do trabalho

O capítulo 2 aborda resumidamente reconhecimento de voz através de um olhar teórico.

No capítulo 3, são realizados os primeiros passos para a concretização deste trabalho; busca-se a melhor biblioteca de reconhecimento de voz que possa ser usada no módulo. A biblioteca escolhida, *Pocketsphinx*, é estudada no capítulo 4.

A arquitetura do *Godot* é apresentada no capítulo 5 a fim de se entender a lógica por trás da construção do módulo de reconhecimento de voz no capítulo 6. O capítulo 7 apresenta a criação de jogo simples, feito na própria *game engine*, para demonstrar o módulo em funcionamento e suas capacidades.

O capítulo 8 apresenta as conclusões do trabalho. Por fim, há uma parte subjetiva contendo agradecimentos e uma descrição das matérias cursadas no Bacharelado em Ciência da Computação que mais ajudaram no desenvolvimento do projeto.

Capítulo 2

Reconhecimento de voz

Neste capítulo, abordaremos a parte teórica do reconhecimento de voz, sem nos preocuparmos com a forma de implementação ou sua aplicação no contexto deste trabalho. Em particular, analisaremos brevemente os principais parâmetros que influenciam seu uso.

2.1 Definição

Reconhecimento automático de voz (ou da fala), muitas vezes referido como *speech to text* (**STT**) ou *automatic speech recognition* (**ASR**), é um campo multidisciplinar que envolve as áreas de Inteligência Artificial, Estatística e Linguística. Busca-se desenvolver metodologias e tecnologias para que computadores sejam capazes de captar, reconhecer e traduzir a linguagem falada para texto (Wikipedia, 2017g).

2.2 História

Apresentamos uma breve visão histórica de sistemas de reconhecimento de voz, desde seu início até os dias atuais. Baseamo-nos principalmente em um artigo (Pinola, 2011).

2.2.1 Décadas de 50 e 60: Primeiros passos

O primeiro sistema de reconhecimento de voz conhecido foi o *Audrey*, construído em 1952 por três pesquisadores do *Bell Labs*. A máquina conseguia reconhecer apenas dígitos falados por um único usuário.

Dez anos depois, a IBM apresentou o *Shoebox*, que reconhecia 16 palavras em inglês, entre elas os dígitos de 0 a 9. Quando captava palavras como *plus*, *minus* ou *total*, *Shoebox* instruía outra máquina de adições a realizar cálculos ou imprimir o resultado. A entrada era feita por um microfone (figura 2.1), que convertia a voz do usuário em impulsos elétricos, classificados internamente por um circuito de medição (IBM, Ano Desconhecido).

Figura 2.1: *Máquina Shoebox sendo operada (Cassiopedia, Ano Desconhecido)*

Laboratórios nos EUA, URSS, Inglaterra e Japão começaram a desenvolver *hardware* para reconhecer uma maior variedade de sons. Conseguiu-se suporte para quatro vogais e nove consoantes; um avanço notável, considerando a tecnologia da época.

2.2.2 Décadas de 70 e 80: Grandes avanços

Na década de 70, o departamento de defesa dos EUA mostrou grande interesse em financiar a tecnologia de reconhecimento de voz. Tal impulso ajudou no desenvolvimento do sistema *Harpy* de reconhecimento de voz pela Universidade Carnegie Mellon.

Harpy usava um grafo para representar o domínio das palavras reconhecíveis. Um algoritmo de busca heurística, *Beam Search*, era aplicado para procurar a melhor interpretação para a voz de entrada. Este algoritmo assemelha-se ao *Best-First Search* (BFS), que explora um grafo através da expansão do estado mais promissor ao sair do estado presente. No entanto, sua otimização consiste em ordenar os próximos possíveis estados, através de uma heurística, antes de realizar uma expansão, o que permite prever o quão longe o estado presente está em relação ao estado meta. Com isso, o *Beam Search* é caracterizado como um algoritmo guloso, que gasta menos memória quando comparado ao BFS (Wikipedia, 2017b).

Através de uma forma de busca mais eficiente, *Harpy* conseguia entender 1011 palavras, aproximadamente o vocabulário de uma criança típica de três anos.

Sistemas de reconhecimento de voz só tiveram um avanço realmente significativo na década de 80, devido a um método estatístico denominado **Modelo Oculto de Markov** (ou **HMM**, sigla para *Hidden Markov Model*). Ao invés de procurar por modelos de palavras em padrões de som, considera-se a probabilidade de um som desconhecido possuir palavras, o que acelerou o processo e tornou possível usar um vocabulário maior nos computadores.

Outro modelo que ganhou bastante popularidade na mesma época foi o de redes neurais, que é efetivo para classificar palavras isoladas e fonemas individuais mas encontra problemas

em tarefas envolvendo reconhecimento contínuo. Ao contrário do HMM, este método não consegue modelar bem dependências temporais. No entanto, em ambos os casos, existia a necessidade de falar pausadamente para o sistema poder melhor interpretar o usuário.

Os progressos em sistemas de reconhecimento de voz começaram a se refletir no meio comercial. Destacamos a boneca *Julie* (figura 2.2), comercializada em 1987 como *"Finalmente, a boneca que te entende"*, pois era capaz de ser treinada para responder à voz de uma criança.

Figura 2.2: *Caixa da boneca Julie; note, na parte inferior, a frase "Ela entende o que você diz" (rrisner, 2016)*

2.2.3 Década de 90 até hoje: Popularização

Na década de 90, a popularização de computadores para uso pessoal e o desenvolvimento de processadores mais rápidos permitiu que o reconhecimento de voz ficasse viável para uma quantidade maior de pessoas.

Em 1996, surgiu o primeiro portal de voz, VAL, criado pela empresa de telecomunicações norte-americana BellSouth. O sistema atendia chamadas telefônicas e respondia de acordo com a informação proferida pelo cliente.

Até o final dos anos 2000, sistemas de reconhecimento de voz pareciam ter ficado estagnados em uma acurácia de aproximadamente 80%, e muitas aplicações eram caracterizadas pela complexidade ou dificuldade de uso se comparadas ao tradicional *mouse* e teclado.

A popularidade do conceito ressurgiu com força através do aplicativo de Busca por Voz, feito pela Google para *iPhone*. As duas razões para tal sucesso eram a facilidade de entrada de dados, se comparado ao teclado da plataforma, e o uso de *data centers* em nuvem da Google, o que retirava a necessidade de um poderoso processamento nos *iPhones* em si. Com isso, mostrava-se que era possível contornar duas das principais limitações de sistemas de voz: a disponibilidade de dados e a dificuldade de processá-los eficientemente.

A evolução na tecnologia de reconhecimento de voz foi tamanha que, atualmente, é inegável seu impacto em nosso dia a dia. Um celular moderno consegue captar palavras ou pequenas frases de seu usuário dentre um enorme vocabulário para fazer buscas na Internet, tocar uma música ou fazer uma ligação. Alguns países usam reconhecimento de voz para autenticar a identidade de alguém por telefone, com o objetivo de evitar fornecer dados pessoais pelo mesmo. Também há usos em transportes, na área médica e para fins educativos, muitas vezes acentuados pela maior facilidade em se falar um comando comparado ao uso de um teclado ou interface gráfica.

2.3 Componentes de um sistema genérico

A figura 2.3 apresenta os três componentes de um sistema genérico envolvendo STT (National Research Council, 1984):

- O **usuário** do sistema, que codifica um comando através de sua voz;
- O **dispositivo** de STT, que converte a mensagem falada para um formato interpretável;
- O **software de aplicação**, que recebe a saída do dispositivo e realiza uma ação apropriada.

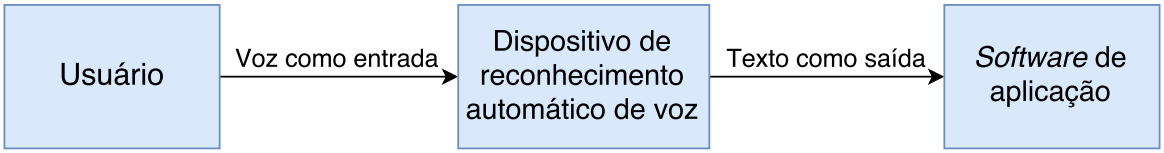

Figura 2.3: *Sistema genérico de reconhecimento automático de voz (National Research Council, 1984)*

2.4 Principais termos

De acordo com (National Research Council, 1984) e (Cook, 2002), apresentamos, a seguir, os termos mais recorrentes em sistemas de reconhecimento de voz. Também detalhamos seus parâmetros principais, que atuam sobre sua forma de funcionamento, eficiência e acurácia. A influência destes fatores varia de acordo com o tipo de aplicação que se deseja construir.

2.4.1 Fluência

A fluência está relacionada à forma de se comunicar com o sistema. Tipicamente, a fala do usuário pode ser feita através de:

- **Palavras isoladas**, com pausas entre elas;

- **Palavras conectadas**, que são concatenadas sem pausas;
- **Fala contínua**, onde o fluxo de palavras é semelhante a uma fala natural.

2.4.2 Dependência do usuário

A dependência ou não do usuário classifica os sistemas em dois grupos:

- Os sistemas **dependentes** (*speaker-dependent*), caracterizados pelo *treinamento* feito pelo usuário. Isto é, são computadores que analisam e se adaptam aos padrões particulares da fala captada, resultando em uma maior acurácia. Geralmente, o usuário deve ler algumas páginas de texto para a máquina antes de iniciar o uso do sistema. Esta variante é comumente escolhida em casos particulares, onde um número limitado de palavras deve ser reconhecido com bastante precisão (SpeechAngel, 2016).
- Os sistemas **independentes** (*speaker-independent*), que são desenvolvidos para reconhecer a voz de qualquer pessoa e não requerem treinamento. É a melhor opção para aplicações interativas que usam voz, já que não é viável fazer com que os usuários leiam páginas de texto antes do uso, ou para sistemas usados por diferentes pessoas. Sua desvantagem é a acurácia menor se comparado ao reconhecimento dependente; para contornar isso, costuma-se limitar o vocabulário reconhecido pelo sistema (SpeechAngel, 2016).

2.4.3 Vocabulário

O vocabulário representa as palavras reconhecidas pelo sistema. Seu tamanho pode ser pequeno (menor que 20 palavras) até muito grande (mais de 20 mil palavras), sendo diretamente proporcional à velocidade do reconhecimento. Além disso, a similaridade entre a pronúncia de algumas palavras pode afetar a acurácia, uma vez que a distinção entre elas torna-se mais complicada.

2.4.4 *Utterance*

O termo *utterance* não possui uma tradução exata no contexto de reconhecimento de voz, embora possa ser interpretado como *"pronunciamento, elocução"*. Refere-se à vocalização (fala) de uma ou mais palavras, pronunciadas de forma contínua e terminando com uma pausa clara, que possuem um significado único ao computador. Em outras palavras, *utterances* são o conteúdo entendido pelo sistema após receber a fala do usuário.

Ao voltarmos para o sistema genérico de reconhecimento de voz apresentado na seção 2.3, notaremos que a interpretação de *utterances* representa a saída produzida pelo dispositivo de STT.

2.4.5 Taxa de erro por palavra

A taxa de erro por palavra (*word error rate*, ou WER) é uma métrica de desempenho de um sistema de reconhecimento de voz (Wikipedia, 2017h).

Suponha que tenhamos o texto de referência τ, correspondente ao que o usuário falou, e o texto τ' de hipótese, que foi gerado pelo sistema STT. O reconhecimento de voz pode levar a diferentes interpretações nas N palavras de τ, levando a um τ' diferente. Em particular, costuma haver um certo número I de palavras inseridas, uma quantidade D de palavras removidas e um valor S de palavras substituídas.

Exemplificando, seja τ = *"Gosto de chocolate"* e τ' = *"Gosto chocolate"*. Houve uma remoção por conta da preposição *"de"* ter sido eliminada pelo sistema de reconhecimento de voz, então $D = 1$.

A taxa de erro WER, geralmente expressa como uma porcentagem, é dada pela equação 2.1. Um valor elevado para esta métrica significa que a qualidade do reconhecimento de voz está ruim.

$$WER = \frac{I + D + S}{N} \tag{2.1}$$

2.4.6 Parâmetros ambientais

Parâmetros ambientais referem-se a fatores externos ao sistema que podem interferir no reconhecimento de voz. Destacam-se:

- A **relação sinal/ruído**, que avalia a intensidade média do sinal recebido em relação ao ruído de fundo, tipicamente medido em decibéis (dB). Quanto menor a taxa, maior a dificuldade no reconhecimento de voz.
- O **próprio usuário**, o que inclui o volume de sua voz, a velocidade com que fala e até mesmo sua condição psicológica: o nível de estresse de um piloto sob ataque em uma aeronave é diferente de alguém simplesmente querendo ouvir uma música, por exemplo.

Capítulo 3

Bibliotecas para Reconhecimento de Voz

A seguir, veremos o primeiro item necessário para atingirmos nosso objetivo final: uma biblioteca que fará o reconhecimento de voz dentro do módulo.

Uma implementação do zero fugiria do tema deste trabalho, pois seria necessário aprender sobre reconhecimento de padrões voltado a sons e outros tópicos relacionados a Inteligência Artificial. A outra opção existente, e a que seguiremos, é procurar por uma biblioteca existente e aprender a manejá-la.

Analisaremos quais as características necessárias e desejáveis em uma biblioteca ideal, e estudaremos, no capítulo 4, a opção que melhor se adequa dentre as existentes.

3.1 Considerações iniciais

Recordemos os principais componentes para reconhecimento de voz, apresentados na seção 2.3. No contexto do módulo de reconhecimento de voz para *Godot*, as seguintes associações surgem naturalmente:

- O **usuário** representa tipicamente o **jogador**, que interage parcialmente ou totalmente com o jogo por meio de comandos de voz.
- O **dispositivo de STT** corresponde ao **módulo de reconhecimento de voz**, objetivo principal deste trabalho. Esta componente é usada pelo jogo para converter a fala do jogador em texto.
- O **software de aplicação** é o **jogo** em si, feito em *Godot*, que recebe indiretamente os comandos do usuário através do módulo e realiza ações apropriadas.

3.2 A biblioteca ideal

Realçamos novamente que o módulo de reconhecimento de voz será usado diretamente em jogos. Tal contexto automaticamente nos leva a pensar em diversas características que a

biblioteca ideal deve possuir.

3.2.1 Características obrigatórias

Em ordem decrescente de importância, temos:

1. **Ter código aberto e licença permissiva:** Justifica-se pela integração da biblioteca em uma *game engine* de código aberto. A importância é ainda maior se levarmos em conta que jogos com fins comerciais podem ser produzidos em *Godot*.

2. **Ser eficiente (rápida):** Já foi mencionado que o módulo de reconhecimento de voz será usado em uma *game engine*. Um jogo é um *software* onde tipicamente a eficiência é de extrema importância, pois costuma envolver a renderização de cenas várias vezes por segundo. Devido a isso, surge a necessidade da biblioteca ser *rápida* para não afetar negativamente a experiência do jogador.

3. **Reconhecer inglês:** O inglês possui presença constante em cenários de computação. Portanto, é a única língua que a biblioteca deve obrigatoriamente oferecer suporte.

4. **Não ser pesada:** Não é desejável ter uma biblioteca que ocupe muito espaço em disco (o que poderia aumentar o tamanho do jogo que a utiliza) e memória (aspecto relacionado diretamente à eficiência).

3.2.2 Características desejáveis

Em ordem decrescente de importância, temos:

1. **Ser multiplataforma:** *Godot* possibilita exportar jogos para diferentes plataformas, dentre elas *Windows*, *MacOS*, *Unix*, *Android* e *iOS* (Linietsky e Manzur, 2017b). Uma biblioteca que possa ser compatível com o maior número possível destes sistemas operacionais tornaria o módulo de reconhecimento de voz mais flexível para a produção de jogos em diferentes ambientes.

2. **Reconhecer diferentes línguas:** Apesar da obrigatoriedade do inglês, a possibilidade de usar diferentes línguas aumentaria a versatilidade do módulo. Tal característica é acentuada ao notarmos que muitos jogos, hoje em dia, oferecem a possibilidade de alterar a língua.

3. **Ser implementada em *C*/*C++*:** Conforme veremos no capítulo 5, *Godot* possui toda a sua base escrita em *C++*, linguagem também usada para a criação de módulos. A implementação da biblioteca na mesma linguagem ajudaria a simplificar problemas de compatibilidade. Eventualmente, *C* também é uma opção viável por ser aceita pela linguagem sucessora.

3.3 Bibliotecas viáveis

Realizou-se uma pesquisa por bibliotecas de reconhecimento de voz que sigam o máximo de características possíveis propostas na seção 3.2. O artigo (NeoSpeech, 2016) sintetiza razoavelmente bem os resultados da busca. A seguir, destacamos as quatro bibliotecas mais notáveis encontradas:

- **Kaldi** (Kaldi, 2017): É a biblioteca mais recente da lista, com seu código publicado em 2011. Escrita em C++, é tida como uma biblioteca para pesquisadores de reconhecimento de voz.
- **CMUSphinx** (CMUSphinx, 2015): Desenvolvida pela *Carnegie Mellon University*, possui diversos pacotes para diferentes tarefas e aplicações. O pacote principal é escrito em *Java*. Existe também a variante *Pocketsphinx*, com características interessantes para este trabalho: é escrita em *C*, possuindo maior velocidade e portabilidade que a biblioteca original.
- **HTK** (HTK, 2016): Desenvolvida pela *Cambridge University Engineering Department*, HTK é uma sigla para *Hidden Markov Model Toolkit*. É escrita em *C*, com novas versões sendo lançadas consistentemente.
- **Simon** (Simon, 2017): Popular para sistemas *Unix* e escrita em *C++*, *Simon* utiliza *CMUSphinx*, *HTK* e *Julius* internamente. Não havia suporte para *MacOS* até abril de 2017.

3.3.1 Escolha da biblioteca mais adequada

Um artigo de 2014 (Gaida *et al.*, 2014) comparou as versões mais recentes de *Kaldi*, *CMUSphinx* e *HTK* na época em relação à taxa de erro por palavra (*word error rate*, explicado na seção 2.4.5). Arquivos de áudio em inglês e alemão foram usados como entrada. Os resultados obtidos são apresentados na tabela 3.1.

Biblioteca	WER em inglês (%)	WER em alemão (%)
Kaldi	6,5	12,7
Pocketsphinx v0.8	21,4	23,9
HTK com *HDecode*[1] v3.4.1	19,8	22,9

Tabela 3.1: *Comparação do WER de bibliotecas STT para entradas em inglês e alemão (Gaida et al., 2014)*

A tabela mostra que *Kaldi* obteve resultados vastamente superiores, enquanto *Pocketsphinx* e *HTK* apresentaram desempenho parecido. O artigo comenta que, apesar dessa semelhança,

[1] Decodificador utilizado junto à biblioteca *HTK*.

a dificuldade bem maior em se configurar *HTK* leva esta biblioteca a ser a menos ideal dentre as citadas.

Os resultados apontam fortemente ao uso de *Kaldi*. No entanto, seu uso e documentação apresentaram ser bastante complexos, e o fato de ser pesada e demorada para compilar nos levou a escolher *Pocketsphinx* para uso no módulo de reconhecimento de voz.

Capítulo 4

Pocketsphinx

Neste capítulo, analisaremos mais a fundo a biblioteca *Pocketsphinx*, incluindo seu funcionamento, passos para compilação a partir do código fonte e instruções para usá-la de forma básica.

Supõe-se que o usuário esteja usando um sistema operacional *Unix*, e que possua acesso a privilégios administrativos para a realização de alguns passos. Recomenda-se que o leitor possua um microfone à disposição no computador, podendo ser embutido ou externo, para melhor aproveitamento.

Todas as instruções e comandos apresentados foram originalmente realizados no sistema `Ubuntu 16.04 LTS, 64-bit` do autor.

4.1 Funcionamento

O funcionamento das bibliotecas do projeto *CMUSphinx*, incluindo-se a *Pocketsphinx*, pode ser resumido por três grandes passos:

- A configuração inicial de arquivos a serem usados pela biblioteca, como o dicionário;
- A captura de áudio de voz, separando-a em *utterances*;
- A busca, para cada *utterance*, da melhor combinação de palavras do dicionário que se assemelhe a ele.

Definimos, abaixo, alguns conceitos numa ordem que nos proporcione um melhor entendimento das etapas descritas.

4.1.1 Fonema

Um **fonema** é a menor unidade de som em uma língua.

O leitor poderia pensar que uma palavra é uma sequência de fonemas, mas tal definição esconde diversas complexidades: há sons que surgem na transição entre palavras, além de

variantes linguísticas afetando a pronúncia do falante, por exemplo. Devido a isso, surgem termos como *difonemas* e *trifonemas*, que tratam de fonemas consecutivos para levar em conta o contexto em que o som é captado.

4.1.2 Vetor de características

Em aprendizado de máquina, uma **característica** é uma quantidade que descreve algum exemplo.

No contexto de reconhecimento de voz, CMUSphinx divide os *utterances* em quadros (*frames*) de aproximadamente 10 ms de comprimento. Através de uma função complexa, extraem-se 39 números – características – para representar o *utterance*; juntos, eles formam o **vetor de características**.

4.1.3 Modelo

Um **modelo** é uma simplificação, onde reduz-se um objeto de estudo às suas características mais importantes. Neste caso, falamos de um modelo de reconhecimento de voz: como tratar as transições entre os quadros em que se divide o áudio capturado?

A solução encontrada pelo projeto *CMUSphinx* foi utilizar o **Modelo Oculto de Markov** (*Hidden Markov Model*, ou HMM) para tratar a fala gravada como uma sequência de estados que transitam entre si com certa probabilidade.

Buscam-se os estados do HMM que levam à maior probabilidade no vetor de características. Para isso, três modelos, alimentados à biblioteca na forma de arquivos externos, são usados:

- **Modelo acústico**: Conjunto de arquivos que contém propriedades acústicas para detectores de fonemas. Define os vetores de características mais prováveis para cada unidade de som, além de determinar a criação de uma sequência de fonemas para um dado contexto. Este modelo costuma vir na forma de vários arquivos. Possui alta dependência com a língua na qual o reconhecimento de voz é realizado.

- **Dicionário fonético**: Arquivo texto responsável por mapear palavras em fonemas, sendo que estes últimos devem existir no modelo acústico. Um mapeamento perfeito é praticamente impossível; devido a variantes linguísticas e outros fatores, não há como adicionar todas as diferentes formas de se pronunciar uma palavra.

  Exemplificando, a palavra `yellow` poderia ser expressa da seguinte forma em um dicionário em inglês:

  ```
  yellow Y EH L OW
  ```

- **Modelo de linguagem**: Arquivo que formaliza uma sintaxe para a linguagem a ser reconhecida. Sua principal finalidade é diminuir o espaço de busca nas palavras, descartando-se palavras improváveis no áudio capturado e melhorando a acurácia.

4.1.4 Palavras-chave

Dentre várias formas diferentes de busca, *CMUSphinx* também oferece suporte para reconhecimento de voz por palavras-chave. Ao invés de usar um modelo de linguagem, fornece-se à biblioteca um arquivo de palavras ou frases a qual se quer detectar, juntamente com um limiar de detecção. Qualquer som capturado que não se encaixar no arquivo ou cujo limiar calculado for baixo demais será descartado.

Exemplificamos, a seguir, a palavra `yellow` como palavra-chave. Cada palavra do arquivo deve vir seguida de seu limiar, que deve ser escrito entre caracteres "`/`".

```
yellow /1e-6/
```

4.2 Compilação

Apresentamos instruções, em *Bash*, para baixar e compilar a biblioteca *Pocketsphinx*. Os passos foram baseados nas instruções em (CMUSphinx, 2016a).

Antes de começar, instale as seguintes dependências em seu sistema:

```
gcc, automake, autoconf, libtool, bison, swig, python-dev, pulseaudio
```

Em um sistema *Ubuntu*, por exemplo, digitaria-se no terminal:

```
$ sudo apt-get install gcc automake autoconf libtool bison swig \
  python-dev pulseaudio
```

4.2.1 Pacote *Sphinxbase*

O pacote **Sphinxbase** oferece funcionalidades comuns a todos os projetos *CMUSphinx*. Siga as instruções abaixo para compilá-lo.

1. Clone o repositório de *Sphinxbase*:

   ```
   $ git clone https://github.com/cmusphinx/sphinxbase
   ```

2. Dentro do diretório `sphinxbase/` criado pelo passo anterior, execute o *script* `autogen.sh` para gerar o arquivo `configure`:

   ```
   $ ./autogen.sh
   ```

3. Execute o *script* `configure` criado no último passo:

```
# Padrão
$ ./configure

# Plataformas sem aritmética de ponto flutuante
$ ./configure --enable-fixed --without-lapack
```

Note que qualquer dependência ausente no sistema (por exemplo, o pacote `swig`) será notificada ao usuário neste passo. Se a execução ocorrer sem problemas, um `Makefile` será gerado.

4. Compile *Sphinxbase* através do `Makefile`:

```
$ make
```

4.2.2 Pacote *Pocketsphinx*

O pacote **Pocketsphinx** contém as funcionalidades de reconhecimento de voz em si que nos interessam para este trabalho. Siga as instruções abaixo para compilá-lo.

1. Clone o repositório de *Pocketsphinx*, o que criará o diretório `pocketsphinx/`.

```
$ git clone https://github.com/cmusphinx/pocketsphinx
```

2. Certifique-se que as pastas `sphinxbase/` e `pocketsphinx/` estejam no mesmo diretório, pois *Pocketsphinx* usa o caminho `../` para procurar pelo pacote *Sphinxbase*.

3. Dentro do diretório `pocketsphinx/`, execute o *script* `autogen.sh` para gerar o arquivo `configure`:

```
$ ./autogen.sh
```

4. Execute o *script* `configure` criado no último passo:

```
$ ./configure
```

Note que qualquer dependência ausente no sistema será notificada ao usuário neste passo. Se a execução ocorrer sem problemas, um `Makefile` será gerado.

5. Compile *Pocketsphinx* através do `Makefile`:

```
$ make
```

4.2.3 Teste de verificação

Para verificar se a compilação feita nas seções 4.2.1 e 4.2.2 ocorreu corretamente, recomenda-se fazer um teste de reconhecimento de voz contínuo com o binário `pocketsphinx_continuous`, criado na compilação do *Pocketsphinx*. Nesta verificação, o usuário fala uma palavra ou uma frase curta, em inglês, em seu microfone. Quando um silêncio é detectado, o programa analisa o *utterance* obtido e imprime na tela o texto que calculou ser a melhor interpretação.

No diretório onde encontram-se as pastas `sphinxbase/` e `pocketsphinx/`, execute o conteúdo da listagem 4.1.

```
# Diretório contendo arquivos para reconhecimento de voz (modelos, etc.)
MODELDIR=pocketsphinx/model

./pocketsphinx/src/programs/pocketsphinx_continuous \
-inmic yes \                                    # Acionar uso do microfone
-hmm   $MODELDIR/en-us/en-us \                  # Diretório do modelo acústico
-dict  $MODELDIR/en-us/cmudict-en-us.dict \     # Arquivo do dicionário
-lm    $MODELDIR/en-us/en-us.lm.bin             # Arquivo do modelo da língua
```

Listagem 4.1: *Comandos para teste de reconhecimento de voz contínuo usando Pocketsphinx*

O programa imediatamente irá imprimir uma lista de seus parâmetros e seus respectivos valores. Depois, avisará ao usuário que está pronto para receber a entrada de voz por meio de uma linha terminada em `Ready....`

A listagem 4.2 representa uma saída resumida ao se falar "`one two three`" no microfone. Os caracteres `[..]` representam uma ou mais linhas omitidas.

```
1  INFO: continuous.c(275): Ready....
2  INFO: continuous.c(261): Listening...
3  [..]
4  INFO: ngram_search_fwdtree.c(1550):     3081 words recognized (15/fr)
5  INFO: ngram_search_fwdtree.c(1552):   703838 senones evaluated (3400/fr)
6  INFO: ngram_search_fwdtree.c(1556):  2241048 channels searched (10826/fr)
7  [..]
8  INFO: ngram_search_fwdflat.c(302): Utterance vocabulary contains 154 words
9  INFO: ngram_search_fwdflat.c(948):     2575 words recognized (12/fr)
10 INFO: ngram_search_fwdflat.c(950):   148022 senones evaluated (715/fr)
11 INFO: ngram_search_fwdflat.c(952):   209298 channels searched (1011/fr)
12 [..]
13 INFO: ngram_search.c(1381): Lattice has 317 nodes, 942 links
14 INFO: ps_lattice.c(1380): Bestpath score: -4833
15 [..]
16 one two three
```

Listagem 4.2: *Saída do `pocketsphinx_continuous` ao se falar "`one two three`"*

Uma interpretação detalhada de toda a saída exige um estudo maior em reconhecimento de voz e na biblioteca *Pocketsphinx* em si. No entanto, destacamos alguma informações,

como o número de palavras analisadas no dicionário (linhas 4 e 9) e a quantidade de *senones* (detectores curtos de sons para trifonemas) captadas (linhas 5 e 10).

4.3 Estruturas e tipos importantes

Feita a compilação da biblioteca, analisaremos as ferramentas que ela oferece para implementação de reconhecimento de voz. Ressaltamos que a implementação dos pacotes é feita na linguagem *C*.

Dentre os tipos de dados oferecidos por *Sphinxbase* e *Pocketsphinx*, destacamos três, a seguir, que serão importantes para um experimento que faremos em breve.

4.3.1 Configuração: `cmd_ln_t`

A *struct* `cmd_ln_t`, definida no pacote *Sphinxbase*, representa uma variável de configuração (CMUSphinx, 2016a). Ela é fornecida a outros tipos de dados em *Pocketsphinx*; informa-se, por exemplo, os arquivos a serem usados (dicionário, modelo acústico, etc.) e se o reconhecimento usará um arquivo de áudio ou será feito na hora.

Um ponteiro pode ser alocado com a função `cmd_ln_init()`, devendo-se liberá-lo posteriormente com uma chamada a `cmd_ln_free_r()`.

Um dos parâmetros existentes neste tipo de configuração é o nome do microfone, no contexto do sistema do usuário. Não conseguimos encontrar uma forma de se obter, em tempo de execução, os nomes dos microfones disponíveis em um sistema *Unix*. Percebemos, também, que seria bastante complicado desenvolver uma solução que funcionasse independentemente do sistema operacional. Portanto, escolhemos passar o nome do microfone como `NULL`, o que leva *Pocketsphinx* a sempre utilizar o microfone padrão do computador, seja qual for a plataforma em que ele está.

4.3.2 Gravação: `ad_rec_t`

A *struct* `ad_rec_t` está definida no pacote *Sphinxbase* e tem como objetivo gravar som de alguma entrada de voz (CMUSphinx, 2016a). Seu uso é vital na implementação de reconhecimento de voz contínuo, através do microfone do usuário.

Aloca-se um ponteiro para este tipo com a função `ad_open_dev()`, cujos parâmetros são um ponteiro para um tipo de configuração `cmd_ln_t` e a taxa de amostragem por segundo. O ponteiro do gravador deve ser liberado posteriormente com `ad_close()`.

As três funções mais importantes para manipulação de um `ad_rec_t` são explicadas abaixo. Todas elas retornam um inteiro diferente de 0 no caso de um erro ocorrer.

`int ad_start_rec(recorder)`

Inicia a gravação no seu argumento `recorder`.

`int ad_read(recorder, buffer, size)`

Lê o áudio gravado em `recorder` desde a última chamada desta função para este argumento. Guarda-se o áudio lido em um *buffer* do tipo inteiro, que possui o tamanho `size` especificado.

`int ad_stop_rec(recorder)`

Termina a gravação no seu argumento `recorder`.

4.3.3 Decodificação: `ps_decoder_t`

A *struct* `ps_decoder_t`, definida no pacote *Pocketsphinx*, representa um decodificador de áudio para texto (CMUSphinx, 2016a). Toda a lógica por trás de reconhecimento de voz, portanto, é tratada por funções ligadas a este tipo.

Cria-se um ponteiro para um decodificador com a função `ps_init()`, que recebe um tipo de configuração `cmd_ln_t` como seu único argumento. A memória alocada deve ser liberada após seu uso com a função `ps_free()`.

Suas funções mais importantes para manipulação são explicadas a seguir. Todas as funções do tipo `int` retornam um inteiro diferente de 0 no caso de um erro ocorrer.

`int ps_start_utt(decoder)`

Inicializa o processamento para *utterance* no `decoder` indicado.

`int ps_process_raw(decoder, buffer, size, no_search, full_utt)`

Usa-se `decoder` para decodificar o áudio guardado no *buffer* de tamanho especificado. Os dois parâmetros restantes representam pequenas otimizações possíveis: `no_search` realiza parcialmente o reconhecimento para usar menos processamento, e `full_utt` considera o *buffer* inteiro como um *utterance*.

`bool ps_get_in_speech(decoder)`

Retorna `true` se o `decoder` fornecido tiver decodificado algo, ou `false` caso contrário. Esta função só deve ser chamada após um `ps_process_raw()`.

`char * ps_get_hyp(decoder, &score)`

Retorna a hipótese (isto é, o termo mais provável) para um `decoder` que foi processado anteriormente com `ps_process_raw()`. Em outras palavras, retorna o termo, em texto, que melhor corresponde ao que o usuário queria dizer. Opcionalmente, se um modelo de língua está sendo usado, pode-se passar a referência de um inteiro como segundo argumento para obter a avaliação (*score*) recebida para a hipótese.

`int ps_stop_utt(decoder)`

Termina o processamento para *utterance* no `decoder` indicado.

4.4 Implementação de reconhecimento contínuo

O reconhecimento contínuo de palavras-chave foi implementado com base no código presente em (CMUSphinx, 2016b). Apresentamos, na listagem 4.3, a função que implementa o laço central do algoritmo, onde repetidamente capta-se a voz do usuário para processamento e imprime-se, na tela, as palavras mais próximas ao que foi pronunciado. Removeram-se as linhas relativas a tratamento de erro para não estender demais a função.

```
1  void mic_loop(ad_rec_t *recorder, ps_decoder_t *decoder) {
2      int buffer[BUFFER_SIZE];
3      bool utt_started = false;
4      int n;
5
6      ad_start_rec(recorder);
7      ps_start_utt(decoder);
8
9      puts("Ready (press Ctrl+C to exit)...");
10
11     while (true) {
12         // n stores how much data was read
13         n = ad_read(recorder, buffer, BUFFER_SIZE);
14         ps_process_raw(decoder, buffer, n, false, false);
15
16         // If speech was captured during silence, start new utterance
17         if (ps_get_in_speech(decoder) && !utt_started) {
18             utt_started = true;
19             puts("Listening...");
20         }
21
22         // Speech stopped while in an utterance
23         if (!ps_get_in_speech(decoder) && utt_started) {
24             ps_end_utt(decoder);
25
26             hyp = ps_get_hyp(decoder, NULL);
27             if (hyp != NULL) {
28                 printf(">> %s\n", hyp);
29             }
30
31             ps_start_utt(decoder);
32             utt_started = false;
33             puts("Ready (press Ctrl+C to exit)...");
34         }
35     }
36
37     ad_stop_rec(recorder);
38 }
```

Listagem 4.3: *Laço de reconhecimento de voz usando-se as ferramentas de Pocketsphinx*

As linhas 2 a 7 correspondem à inicialização de variáveis, do gravador e do decodificador. Assume-se, inicialmente, que não estamos no meio de um *utterance* através de `utt_started = false` (linha 3).

O laço de reconhecimento de voz começa na linha 11, e usa a seguinte lógica:

- **Linhas 12-14**: O áudio de entrada é capturado, e seus dados são processados pelo

decodificador.

- **Linhas 16-18**: Se foi possível decodificar alguma palavra do usuário, significa que estamos diante de um *utterance*. Em outras palavras, o usuário está falando algo. Coloca-se `utt_started = true`.
- **Linhas 22-34**: A condição da linha 23 é oposta à que acabamos de ver anteriormente, verificando se o gravador e decodificador não detectaram mais nenhuma palavra vinda do usuário. Caso seja verdade, encerramos o *utterance* atual e adquirimos sua interpretação (hipótese) através de `ps_get_hyp()` (linhas 26 a 28). Por fim, reiniciamos o decodificador para captar um novo *utterance*, alterando `utt_started` para `false` para refletir esta mudança (linhas 31 a 33).

O leitor talvez note que a chamada a `ad_stop_rec()` na linha 37 é inalcançável devido ao laço implementado não possuir condição de parada. A implementação foi feita desta forma porque a situação apresentada é um teste; em um caso mais específico, como na implementação do módulo de reconhecimento de voz, a presença de uma condição de parada é vital.

Capítulo 5

Godot

Voltaremos nossa atenção à *game engine Godot* (Linietsky e Manzur, 2017a) neste capítulo. Em particular, estamos interessados no estudo dos elementos principais que compõe sua arquitetura, a organização do seu código fonte e instruções para compilação.

Para referência, todas as informações relacionadas a *Godot* são referentes à versão **2.1.4**, que é a mais recente e estável no momento de escrita deste trabalho.

Todas as instruções e comandos apresentados foram originalmente realizados no sistema `Ubuntu 16.04 LTS, 64-bit` do autor.

5.1 História

O desenvolvimento de *Godot* começou em 2007, através de Juan Linietsky e Ariel Manzur. O nome foi escolhido em homenagem à peça *Waiting for Godot*, de Samuel Beckett, para representar uma biblioteca que cada vez mais ganha novas funcionalidades, mas nunca chegará a um produto definitivo (Wikipedia, 2017d).

Godot é notavelmente conhecido por possuir código aberto, que foi liberado ao público em fevereiro de 2014. Desde então, ganha constantes atualizações para se equiparar a *game engines* competidoras mais sofisticadas, como *Unity* e *Unreal Engine*. Atualmente, encontra-se na versão 2.1.4, com uma versão 3 em beta, e possui suporte para a produção de jogos em diversas plataformas, entre elas *Unix*, *Windows*, *MacOS*, *Android*, *iOS* e Web.

O logo da *game engine* é apresentado na figura 5.1.

Figura 5.1: *Logo da game engine Godot (Wikipedia, 2017d)*

5.2 Linguagens

Godot possui seu código fonte escrito primordialmente em ***C++***. Apesar de não possuir toda a versatilidade de uma linguagem de *script* como *Python* e *Ruby*, um código escrito em *C++* possui uma execução bem rápida, fator crítico para um *software* que produzirá jogos. Comparado ao antecessor, *C*, a linguagem oferece ferramentas mais poderosas, como Orientação a Objetos e bibliotecas para tipos abstratos de dados (pilhas, filas, etc.).

Um usuário da *game engine*, no entanto, raramente interage diretamente com *C++*. Ao invés disso, usa-se uma interface gráfica, na forma de um editor, com várias facilidades para a criação de jogos. Para programação, *Godot* disponibiliza uma linguagem de *script* nativa chamada ***GDScript***.

GDScript possui uma sintaxe extremamente simples, parecida com *Python*, e foi projetada para usufruir da arquitetura da *game engine*. O usuário pode programar classes, estruturas e partes de seu jogo com maior facilidade, utilizando as ferramentas oferecidas pelo *software* sem precisar se preocupar com detalhes internos de implementação (Godot Docs, 2017j). Em outras palavras, *GDScript* age como uma "linguagem intermediária" entre o usuário e as interfaces em *C++* que *Godot* disponibiliza.

5.3 Arquitetura

A arquitetura de *Godot* é bastante complexa, chegando a mais de 7 milhões de linhas de código. Uma análise criteriosa de todos os componentes fugiria do contexto deste trabalho. No entanto, é essencial entender as principais classes e conceitos da *game engine* para podermos implementar, com sucesso, o módulo de reconhecimento de voz.

A título de curiosidade, uma árvore de herança de classes é mostrada na figura 5.2. Os filhos das classes *Control*, *Node2D*, *Reference* e *Spatial* foram omitidos por serem muito numerosos.

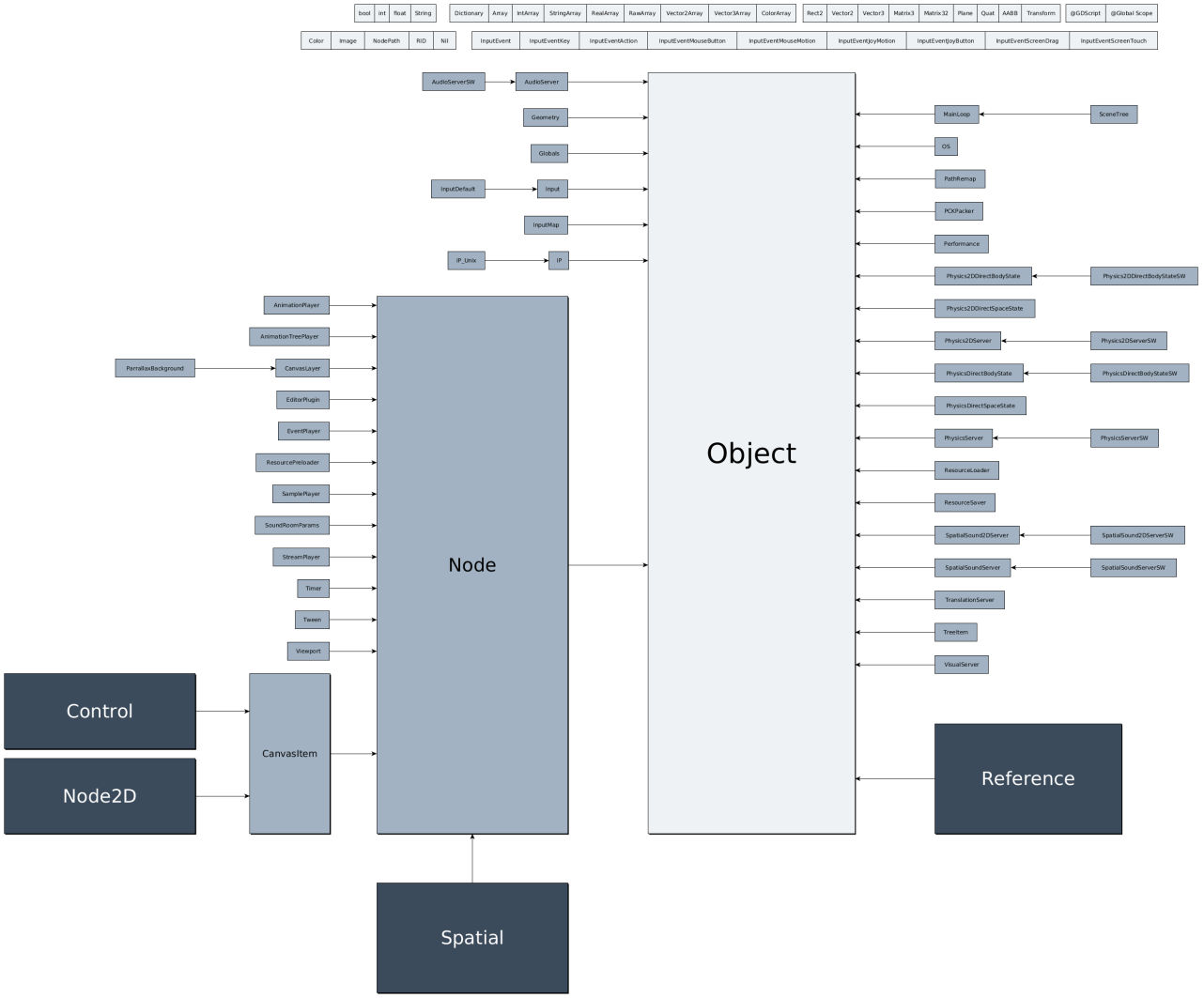

Figura 5.2: *Árvore de herança de classes em Godot (Godot Docs, 2017a)*

Nas próximas subseções, abordaremos as classes mais importantes, juntamente com as utilidades que trazem para o funcionamento de *Godot*.

5.3.1 *Object*

Object (objeto) é a classe base para todos os tipos que não estão embutidos na *game engine* (como os diferentes tipos de variáveis), conforme fica evidente na figura 5.2 apresentada anteriormente.

Algumas de suas características incluem a necessidade de liberar sua memória após o uso e a possibilidade de receber *notificações*, isto é, uma chamada assíncrona a um de seus métodos.

5.3.2 *Reference*

Reference (referência) é uma classe que herda de *Object*. Assim como sua classe pai, ela é mais evidente dentro do código C++, pois não aparece diretamente no contexto do editor *Godot*.

Sua principal propriedade está em uma forma de gerenciamento automático de memória, como encontrado na *coleta de lixo* em várias linguagens. Todo *reference* carrega um atributo para contar quantas referências externas possui. Quando não há mais nenhuma, sua memória é automaticamente liberada.

5.3.3 *Node*

Um ***node*** (nó) é um dos elementos mais básicos para a criação de jogos em *Godot*. Todo *node* herda de *Reference*, e possui um nome, propriedades (que podem ser alteradas/sobrescritas) e um *comportamento*: desenhar um modelo em 3D, mostrar uma interface gráfica, controlar o comportamento de um personagem, etc. (Godot Docs, 2017e). Podem-se criar classes que estendem um tipo de *Node*, atribuindo funcionalidades adicionais a elas.

Em termos práticos, a propriedade mais importante que oferecem é a adição a outros *nodes*, tornando-se filho deles. Com isso, cria-se uma hierarquia de árvore, deixando claro a dependência de funcionalidades.

A seguir, exemplificamos alguns tipos de nós existentes no editor *Godot*.

- ***BaseButton***: Oferece funcionalidades básicas a todos os nós do tipo *button* (botões). Internamente (em C++), é implementado como uma classe abstrata.
- ***Button***: É um botão padrão, que pode ser clicado pelo usuário. Estende o nó *BaseButton* (na implementação em C++, é uma classe que herda de *BaseButton*).
- ***Label***: Apresenta um texto formatado.
- ***Sprite***: É um *bitmap* bidimensional, tipicamente usado para representar personagens em um jogo 2D.

Suponha que desejamos criar um botão clicável, onde está escrito *"Começar Jogo"*. Ao pensarmos que este botão deverá *conter* um texto, a hierarquia de *nodes* fica intuitiva: iremos criar um nó *Button* que possui um filho *Label*. Além disso, desejamos alterar a propriedade de texto deste último para o valor *"Começar Jogo"*.

Teremos uma visão mais prática de nós no capítulo 7, onde criaremos um jogo em *Godot*.

5.3.4 *Scene*

Um ***scene*** (cena) é um grupo de *nodes* organizados em uma hierarquia de árvore. Toda cena possui apenas um nó raiz (Godot Docs, 2017h).

Em *Godot*, executar um jogo é equivalente a executar uma ou mais cenas, que podem ser salvas ou carregadas do disco no decorrer do programa. Ressalta-se que, para o jogo começar, um *scene* deve ser previamente configurada como a inicial, isto é, a primeira a ser executada quando o jogo é iniciado.

A maior vantagem de um *scene*, portanto, está em sua modularização. Ao invés de se criar um jogo grande com uma quantidade enorme de nós em hierarquia, podem-se fazer várias cenas, com uma *instanciando* outra durante a execução. Tal instância é adicionada na árvore do *scene* que fez a chamada. Quando a cena não é mais necessária, ela pode ser salva no disco, se necessário, e retirada da hierarquia.

A figura 5.3 exemplifica um instanciamento genérico de uma cena.

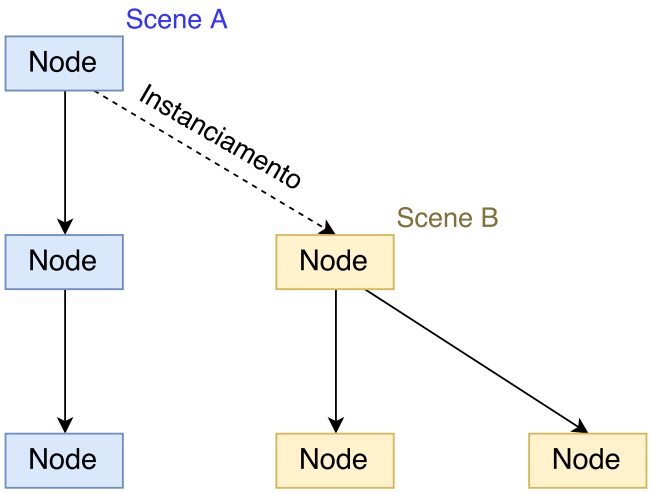

Figura 5.3: *Exemplo de instanciamento de um scene (Godot Docs, 2017h)*

5.3.5 *Resource*

Resources (recursos) são outro tipo de dados em *Godot* com uma importância tão grande quanto *nodes*. Todo recurso armazena algum dado, e portanto não realizam uma ação ou processamento por si só (Godot Docs, 2017g).

Uma característica importante de *resources* é que são carregados apenas uma vez do disco. Se um recurso que já está na memória for novamente carregado, será retornado a mesma cópia de antes; em detalhes internos, *Godot* guarda uma referência ao *resource* original.

A classe *Resource* herda de *Reference*, adquirindo sua característica de liberação automática de memória quando não há nenhuma referência à instância.

Exemplos de *resources* incluem:

- ***Texture***: Representa uma textura a ser aplicada em um objeto 2D ou 3D.
- ***Font***: Representa uma fonte a ser usada em um texto, interface gráfica, etc.
- ***AudioStream***: Usado para guardar um fluxo de áudio (uma música a ser tocada no jogo, por exemplo).

Exemplificamos, na figura 5.4, alguns *nodes* que tipicamente poderiam usar os *resources* citados.

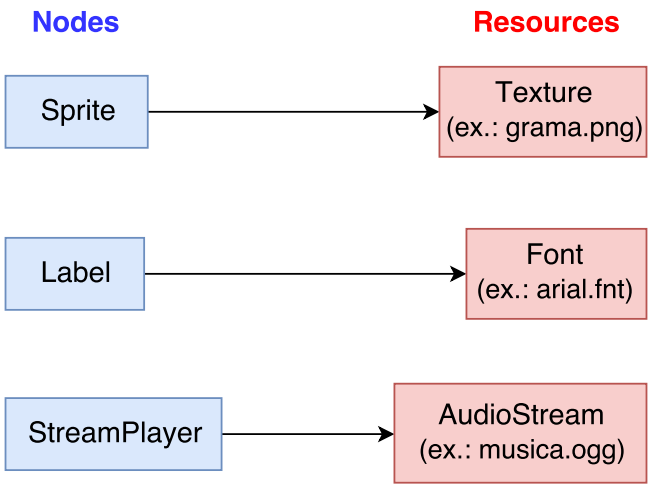

Figura 5.4: *Alguns resources e nodes que tipicamente os usam (Godot Docs, 2017g)*

5.3.6 Sistema de arquivos

Devido ao suporte a diferentes sistemas operacionais, *Godot* viu uma necessidade de criar um padrão interno para gerenciar arquivos em seus projetos (isto é, jogos). Duas convenções foram tomadas (Godot Docs, 2017b):

- O diretório raiz do projeto, que contém um arquivo denominado `engine.cfg`, é referido como **`res://`**. O usuário é encorajado fortemente a utilizar este prefixo ao invés do utilizado em sua plataforma.
- Caminhos que usem `res://` devem usar o caractere `/` para separar diretórios, independentemente do sistema operacional do usuário. Por exemplo: `res://fontes/arial.fnt`.

Por fim, muitos jogos possuem a necessidade de criar, ler, escrever e atualizar arquivos durante sua execução; uma situação comum é guardar o progresso do jogador. Em geral, não é aconselhável criar tais arquivos dentro do diretório do projeto; tal ação é impossível, aliás, se o jogo estiver no formato de um binário fechado.

A solução apresentada por *Godot* foi definir um outro prefixo, **`user://`**, que refere-se a algum diretório preestabelecido e externo ao projeto (Godot Docs, 2017b). Em sistemas *Unix*, o caminho tipicamente corresponde a `~/.godot/app_userdata`.

5.4 *SCons*

Godot pode ser compilado para diversas plataformas; entre elas, citamos *Windows*, *MacOS*, *Unix*, *Android*, *iOS* e *Web* (Linietsky e Manzur, 2017b). A necessidade de atender a uma

variedade tão grande de sistemas operacionais fez com que o projeto escolhesse ***SCons*** para simplificar sua compilação (Godot Docs, 2017i). Esta é uma ferramenta de construção de código aberto, como *automake* e *cmake*, mas apresenta algumas vantagens interessantes (SCons Foundation, 2017a):

- Compilação multiplataforma. Por exemplo, é possível compilar um executável para *Windows* em um sistema *Unix*.
- Todos os arquivos de configuração são *scripts* na linguagem *Python*.
- Análise de dependências automática para linguagens como *C*, *C++* e *Java*. Não é necessário, por exemplo, listar quais arquivos de código fonte são necessários para compilar o binário final, algo que ocorre em um sistema *Make*.

5.4.1 Instalação

SCons pode ser instalado pela linha de comando. Em um ambiente Ubuntu, por exemplo, digitaria-se:

```
$ sudo apt-get install scons
```

Também é possível baixar um binário ou o código fonte pelo site (SCons Foundation, 2017b).

5.4.2 Uso em *Godot*

Para usar *SCons* com os *scripts* de configuração em um diretório, basta invocá-lo:

```
$ scons
```

Godot oferece alguns argumentos por linha de comando para maior controle sobre a compilação, os quais usaremos na seção 5.5 e no capítulo 7. Veja a tabela 5.1 para maiores informações.

Parâmetro	Significado	Valores

`p, platform`	Plataforma alvo da compilação	Inclui:
		`x11` (*Unix*) `javascript` (Web)
		`windows` `android`
		`osx` `iphone`

`bits`	Nº de bits da plataforma alvo	`32`
		`64`
		`default`

`tools`	Deve-se incluir o editor *Godot* no binário?	`yes` `no`

`target`	Controla opções de otimização e *debug*	`debug`
		`release_debug`
		`release`

`j`	Nº de processadores usados para compilação em paralelo	Um inteiro compatível com o nº de processadores na máquina

Tabela 5.1: *Argumentos por linha de comando do SCons para Godot*

Algumas explicações devem ser dadas sobre a tabela 5.1. A opção `default` do parâmetro `bits` possui grande dependência do sistema operacional em uso: se for *Windows* ou *MacOS*, é equivalente a `32`; se for Unix, assume a mesma quantidade de bits do sistema.

O binário produzido pode ser usado para executar jogos na plataforma para a qual foi criado. A opção `tools` controla se ele deverá conter ou não o editor *Godot*. Executáveis gerados com o valor `no` são chamados de *templates de exportação*, pois são usados para exportar jogos para um determinado sistema operacional.

Por fim, a opção `target` apresenta três valores possíveis para controlar o uso de símbolos de depuração em *C++*, otimização e verificação de erros em tempo de execução (*runtime*). Sintetizamos as características de cada valor na tabela 5.2.

Valor	Símbolos de depuração	Otimização	Verificação em *runtime*
`debug`	✓	x	✓
`debug_release`	x	x	✓
`release`	x	✓	x

Tabela 5.2: *Valores possíveis do parâmetro `target` e seu significado*

5.5 Compilação

O código fonte de *Godot* está disponível no GitHub (GitHub, 2017). A criação de um módulo, conforme será visto no capítulo 6, exige a adição de código à *game engine* e sua recompilação, o que justifica a importância dos passos a seguir.

Antes de começar, verifique se o seu sistema *Unix* possui as seguintes dependências:

- *GCC* (versão 6 ou menor) ou *Clang*;

- *Python* 2.7+ (excluindo-se versões de *Python* 3);
- *SCons*;
- `pkg-config`;
- Bibliotecas de desenvolvimento *X11*, *Xcursor*, *Xinerama* e *XRandR*;
- Bibliotecas de desenvolvimento *MesaGL*;
- Bibliotecas de desenvolvimento *ALSA*;
- Bibliotecas de desenvolvimento *PulseAudio*;
- *Freetype*;
- *OpenSSL*;
- `libudev-dev`.

A página de documentação de *Godot* em (Godot Docs, 2017f) oferece comandos de terminal para baixar as dependências facilmente nas distribuições *Unix* mais populares. Em *Ubuntu*, por exemplo, faríamos:

```
$ sudo apt-get install build-essential scons pkg-config libx11-dev \
  libxcursor-dev libxinerama-dev libgl1-mesa-dev libglu-dev \
  libasound2-dev libpulse-dev libfreetype6-dev libssl-dev libudev-dev \
  libxrandr-dev
```

Feito isso, siga os passos a seguir para compilar *Godot* na versão 2.1.4.

1. Clone o repositório da *game engine*.

   ```
   $ git clone https://github.com/godotengine/godot
   ```

2. Dentro do diretório `godot/` criado pelo passo anterior, mude para a *tag* corresponde à versão 2.1.4.

   ```
   $ git checkout 2.1.4-stable
   ```

3. Compile *Godot* através da ferramenta *SCons*; deve levar em torno de 15 minutos.

   ```
   $ scons p=x11
   ```

5.5.1 Verificação

Se a compilação foi feita com sucesso, um diretório `bin/` será gerado dentro de `godot/`. Execute o binário nele contido para abrir o editor *Godot*.

```
$ ./bin/godot.x11.tools.32  # Em sistema Unix, 32 bits
$ ./bin/godot.x11.tools.64  # Em sistema Unix, 64 bits
```

O uso do editor em si será visto no capítulo 7, quando criarmos um jogo para demonstrar o módulo de reconhecimento de voz.

Capítulo 6

Módulo *Speech to Text* para *Godot*

Após adquirirmos conhecimento sobre a biblioteca *Pocketsphinx* e a *game engine Godot*, chegou o momento de construirmos o módulo de reconhecimento de voz.

Conforme descrito na seção 5.2, a linguagem *GDScript* é extremamente prática para programar estruturas em um jogo feito em *Godot*. No entanto, às vezes deseja-se otimizar alguma parte crítica através de *C++* ou adicionar uma nova funcionalidade inexistente. Os módulos servem justamente para este objetivo, pois não fazem parte do código essencial da *game engine*.

Este capítulo documenta os passos e decisões de projeto tomados na criação do módulo, a qual chamaremos de ***Speech to Text***. Pressupomos que o leitor esteja familiarizado com as instruções para compilação de *Godot*, vistas na seção 5.5, e com a biblioteca *Pocketsphinx* (capítulo 4), que será usada para a realização do reconhecimento de voz.

Os passos para criação do módulo foram baseadas no tutorial existente na documentação de *Godot* (Godot Docs, 2017d).

Todas as instruções e comandos apresentados foram originalmente realizados no sistema `Ubuntu 16.04 LTS, 64-bit` do autor.

6.1 Primeiros passos

Antes de começarmos a implementar o módulo em si, precisaremos tomar algumas medidas simples de preparação.

6.1.1 Criação do diretório do módulo

Todos os módulos ativos são encontrados como subdiretórios dentro da pasta `modules/` no código fonte de *Godot*. Começaremos, portanto, com a criação do diretório `speech_to_text`:

```
$ cd modules
$ mkdir speech_to_text
$ cd speech_to_text
```

6.1.2 Adição do pacote *Sphinxbase*

Usaremos a biblioteca *Pocketsphinx* para realizar o reconhecimento de voz no módulo. Um dos requisitos necessários para seu funcionamento, conforme visto na seção 4.2, é o pacote *Sphinxbase*. A seguir, apresentamos instruções para inserir os arquivos essenciais deste pacote no diretório do módulo.

1. Baixe o pacote *Sphinxbase* em sua versão atual mais estável, a **5-prealpha**.

```
$ SPHINXURL="https://sourceforge.net/projects/cmusphinx/files"
$ wget $SPHINXURL/sphinxbase/5prealpha/sphinxbase-5prealpha.tar.gz
```

2. Extraia e renomeie o pacote baixado.

```
$ tar -xvf sphinxbase-5prealpha
$ mv sphinxbase-5prealpha sphinxbase
```

3. Remova arquivos supérfluos, como `Makefiles`, arquivos de teste e *scripts* de compilação. Estes serviriam para aumentar, desnecessariamente, o tamanho do módulo. Em outras palavras, somente as interfaces e implementações nos pacotes serão mantidas. Veja a listagem 6.1 para maiores detalhes.

```
# Deletar a maioria dos arquivos/diretórios supérfluos de Sphinxbase
$ find . -not -name "src" \
  -a -not -name "include" \
  -a -not -wholename "./src/libsphinxbase*" \
  -a -not -wholename "./src/libsphinxad*" \
  -a -not -wholename "./include*" \
  -a -not -name "LICENSE" \
  -delete

# Eliminar arquivos supérfluos restantes
$ find "src" "include" -name "Makefile*" -delete \
  -o -name "*.in" -delete
```

Listagem 6.1: *Remoção de arquivos supérfluos no pacote Sphinxbase*

6.1.3 Adição do pacote *Pocketsphinx*

Precisamos, também, do pacote *Pocketsphinx* para a biblioteca homônima funcionar. A seguir, apresentamos instruções para inserir os arquivos essenciais deste pacote no diretório do módulo.

1. Baixe o pacote *Pocketsphinx* em sua versão atual mais estável, a **5-prealpha**.

```
$ SPHINXURL="https://sourceforge.net/projects/cmusphinx/files"
$ wget $SPHINXURL/pocketsphinx/5prealpha/pocketsphinx-5prealpha.tar.gz
```

2. Extraia e renomeie o pacote baixado.

```
$ tar -xvf pocketsphinx-5prealpha
$ mv pocketsphinx-5prealpha pocketsphinx
```

3. Remova arquivos supérfluos, isto é, que não sejam interfaces ou implementações. Veja a listagem 6.2 para maiores detalhes.

```
# Deletar a maioria dos arquivos/diretórios supérfluos de Pocketsphinx
$ find . -not -name "src" \
  -a -not -name "include" \
  -a -not -wholename "./src/libpocketsphinx*" \
  -a -not -wholename "./include*" \
  -a -not -name "LICENSE" \
  -delete

# Como sobra apenas um diretório dentro de src/, movemos seu conteúdo
$ mv src/libpocketsphinx/* src && rmdir src/libpocketsphinx

# Eliminar arquivos supérfluos restantes
$ find "src" "include" -name "Makefile*" -delete \
  -o -name "*.in" -delete
```

Listagem 6.2: *Remoção de arquivos supérfluos no pacote Pocketsphinx*

6.2 Planejamento

Quais os requisitos funcionais e não funcionais que queremos atender? O que um típico usuário do módulo *Speech to Text* desejaria para poder usar em seu jogo? Todo projeto deve começar com algum planejamento mínimo de onde se quer chegar para obter algum sucesso.

6.2.1 Requisitos não funcionais

Reconhecimento de voz em jogos geralmente é usado em um contexto de tempo real. Isto é, para uma dada fala do usuário, não desejamos que o jogo demore muito para dar alguma forma de resposta com o risco de comprometer seu aspecto lúdico.

Portanto, em termos dos principais parâmetros de reconhecimento de voz definidos na seção 2.4, projetaremos o módulo com a questão de **eficiência** em mente:

- **Fluência**: Idealmente, a forma de comunicação poderia chegar até *palavras conectadas*. Uma *fala contínua* demandaria um processamento muito pesado e comprometedor para o jogo.
- **Dependência do usuário**: Um sistema *independente* é mais flexível por atender a uma maior quantidade de pessoas sem a necessidade de um longo treinamento antes de começaram a jogar.
- **Vocabulário**: Deve ser tipicamente *pequeno* (não mais do que 40 palavras). Um vocabulário muito grande aumentaria o tempo de reconhecimento de voz, o que por sua vez afetaria a experiência do jogador.
- **Parâmetros ambientais**: Não é esperado que interfiram tanto no jogo. A relação sinal/ruído deve ser baixa, pois um ambiente muito barulhento comprometeria a jogabilidade. Por fim, desejamos que o usuário possa falar em um tom de voz normal, sem precisar "forçar" a pronúncia das palavras ou aumentar o volume de sua fala para o reconhecimento ser possível. Tais características são automaticamente tratadas pelo modelo acústico usado no *Pocketsphinx*.

Desejamos que o módulo seja **confiável** em relação à acurácia do reconhecimento de voz. Isto dependerá dos modelos usados na configuração do *Pocketsphinx* (descritos na seção 4.1.3).

Configurabilidade também é uma característica almejada: o usuário do módulo deverá ter controle sobre a língua do reconhecimento e o vocabulário reconhecido, por exemplo. *Pocketsphinx* permite isso facilmente com a alteração de seus arquivos de configuração. O uso de palavras-chave (comentado na seção 4.1.4) em vez de um modelo de língua é mais adequado para um jogo, uma vez que a interação exercida pelo usuário ocorre por meio de palavras específicas (comandos).

O módulo deverá ser de **propósito geral** em relação ao tipo de jogo em que é empregado (ação, terror, plataforma, etc.); portanto, não é possível prever características do típico usuário do jogo. Este requisito, em geral, não é tão preocupante quando levamos em conta o uso de um modelo acústico geral com a biblioteca *Pocketsphinx*.

É importante que o módulo seja **tolerante a erros**, isto é, que comunique ao restante do sistema quando um problema ocorre dentro de si.

Por fim, um requisito desejável, mas a princípio não estritamente necessário, é a **portabilidade**. Apesar das classes internas de *Godot* e a biblioteca *Pocketsphinx* terem sido projetados para funcionarem em diversos sistemas operacionais, colocaremos plataformas *Unix* como a meta principal. Suporte a outras plataformas poderá ser feito depois, dependendo da complexidade da implementação.

6.2.2 Requisitos funcionais

A princípio, desejamos que o reconhecimento de voz seja executado em paralelo com o restante do jogo. Isto é, gostaríamos que a execução não parasse totalmente até obter um comando por voz do usuário. Queremos, também, uma forma de verificar se o reconhecimento está ativo e de iniciá-lo/desligá-lo a qualquer momento.

As palavras reconhecidas pelo módulo *Speech to Text* não precisariam ser interpretadas imediatamente pelo jogo. Uma ideia mais flexível é guardá-las em um *buffer* e deixar o próprio jogo lê-las em seu ritmo.

Como o vocabulário deve ser tipicamente pequeno, o reconhecimento por *palavras-chaves* do *Pocketsphinx* (visto na seção 4.1.4) é bem mais viável do que por modelo de língua.

A configurabilidade desejada nos requisitos não funcionais nos leva a precisar de uma interface para ajustar parâmetros e arquivos do reconhecimento de voz. Em particular, um usuário estaria interessado em configurar o modelo acústico, o dicionário e as palavras-chave.

6.3 Implementação

Apresentamos, na figura 6.1, um diagrama de classes simplificado do módulo *Speech to Text*. Os atributos e métodos de cada classe serão indicados nas próximas subseções.

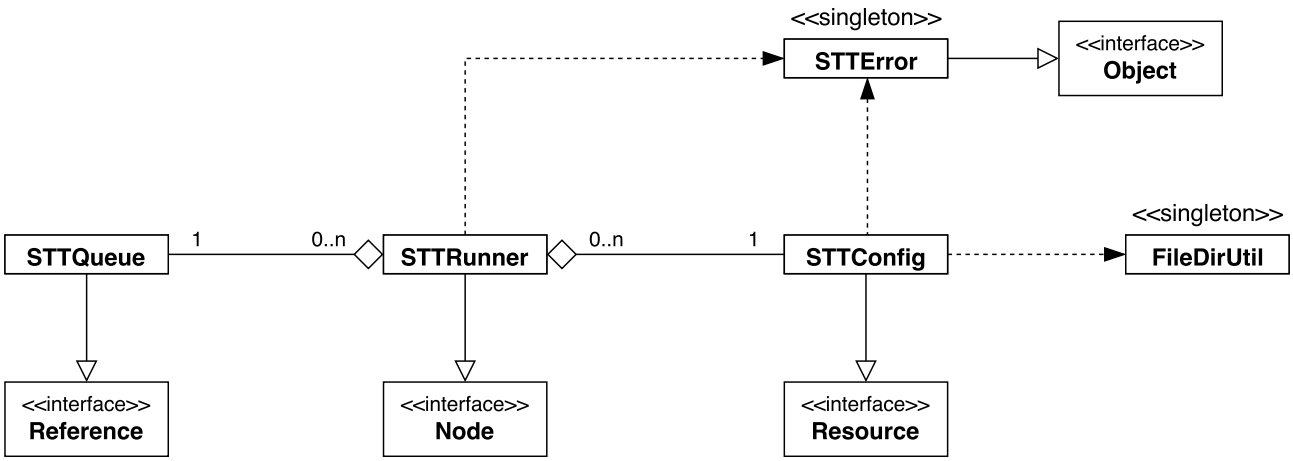

Figura 6.1: *Diagrama de classes simplificado do módulo Speech to Text*

Antes de nos aprofundarmos na arquitetura do módulo, algumas observações gerais devem ser feitas:

- *Godot* não permite que construtores e destrutores possuam argumentos como uma forma de padronizar o instanciamento e liberação de objetos. A restrição aos construtores, no entanto, pode ser contornada através de *setters* para atributos desejados.
- A *game engine* oferece implementações próprias de alguns tipos e estruturas de dados. Destacamos `String` para cadeias de caracteres e `Vector` como um vetor de uso geral, podendo representar uma fila, pilha, etc.
- O uso de duas classes *singleton* no módulo parece exagero. A justificativa é que *Godot* proíbe a exportação de classes que não podem ser instanciadas (em outras palavras, classes que só possuem métodos estáticos) para uso em *GDScript*. Recomenda-se, portanto, esta outra abordagem para contornar tal limitação (Linietsky, 2016).
- Quase todas as classes implementadas, exceto `FileDirUtil`, possuem um método especial chamado `_bind_methods()`. Esta função, de nome predefinido por *Godot*, é usada para ligar nomes a constantes ou referências de métodos, permitindo seu uso em *GDScript*. Todas as ligações são guardadas numa grande classe *singleton* denominada `ObjectTypeDB`. A listagem 6.3 exemplifica a adição de uma linha no método `_bind_methods()` de `STTConfig` para ser possível usar o método `init()` desta mesma classe em *GDScript*.

```
void STTConfig::_bind_methods() {
    ObjectTypeDB::bind_method("init", &STTConfig::init);
}
```

Listagem 6.3: *Adicionando o método `init()` de STTConfig para uso em GDScript*

6.3.1 Classe `STTConfig`

Como o nome sugere, `STTConfig` é uma classe de configuração para a realização do reconhecimento de voz. Por possuir a característica de servir mais como um armazenamento de informação, decidimos fazer a classe herdar de `Resource` (visto na seção 5.3.5).

A figura 6.2 apresenta os atributos, métodos e relacionamentos da classe `STTConfig`. O construtor e destrutor foram omitidos por simplicidade. Todos os atributos e métodos estáticos, neste e em futuros diagramas, estarão sublinhados.

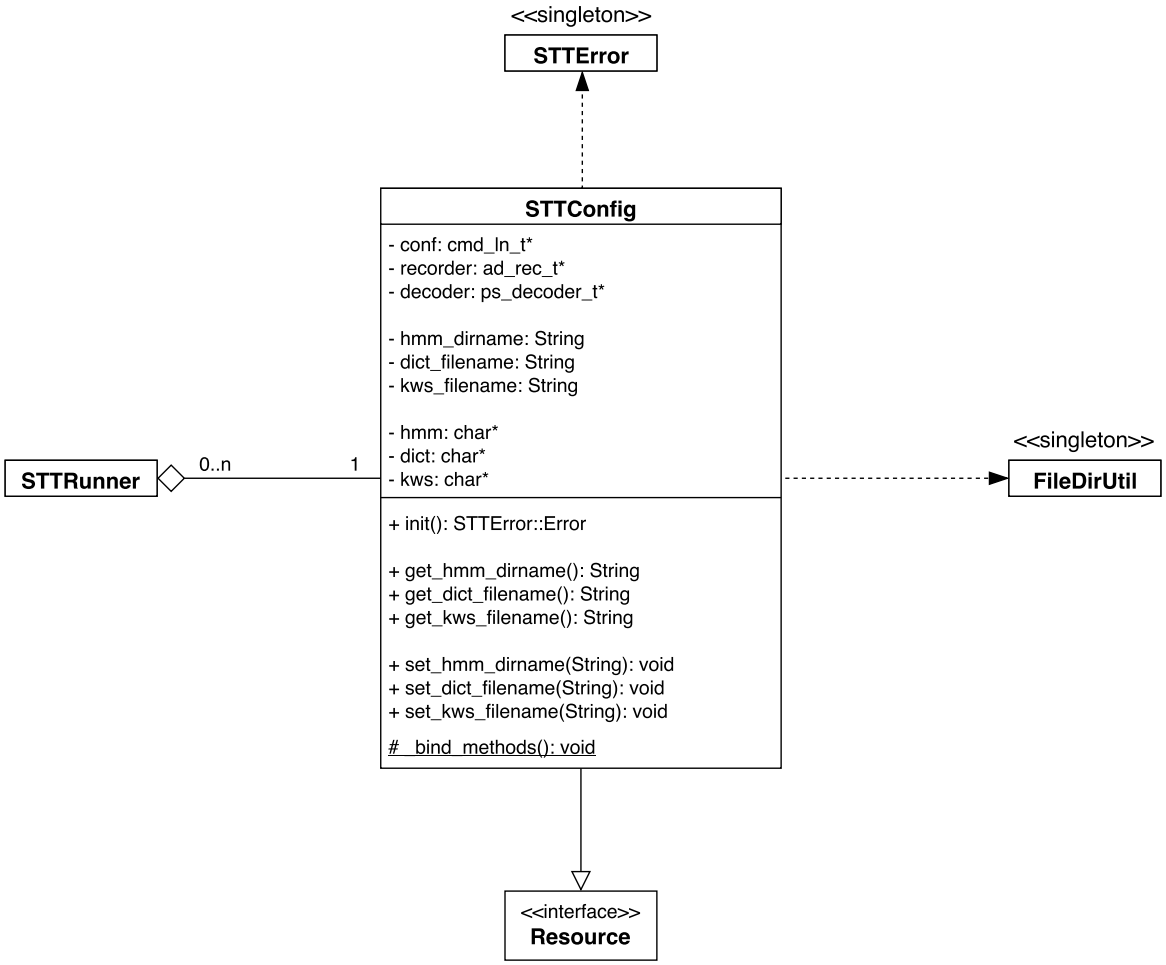

Figura 6.2: *Atributos, métodos e relacionamentos da classe* `STTConfig`

`STTConfig` possui duas funcionalidades principais:

Definir arquivos de configuração

Nomes de arquivos de configuração são tratados pelos *getters* e *setters* de `hmm_dirname`, `dict_filename` e `kws_filename`, que correspondem, respectivamente, ao diretório do modelo acústico, o arquivo de dicionário e o arquivo de palavras-chaves.

Verifica-se se os nomes passados como argumentos para os *setters* correspondem a arquivos/diretórios existentes. No entanto, infelizmente não há como checar facilmente, em tempo de execução, se o arquivo/diretório possui erros de sintaxe. O diretório do modelo acústico, por exemplo, contém arquivos binários que, a princípio, parecem impossíveis de serem verificados. Optamos, portanto, pela filosofia de "atirar primeiro, perguntar depois" ao avisar posteriormente que houve problemas no uso dos arquivos.

Por fim, resta um problema para tratarmos, resultante da distinção entre os sistemas de arquivo de *Godot* (visto na seção 5.3.6) e *Pocketsphinx*. Suponha que o desenvolvedor de um jogo tenha guardado seu arquivo de dicionário, o `dicionario.dict`, na raiz do projeto. Como referenciar este arquivo? *Godot* utilizaria o caminho `res://dicionario.dict`, mas

Pocketsphinx não entende este prefixo, necessitando do caminho absoluto usado no sistema.

Uma solução simples seria converter o caminho com `res://` para o caminho absoluto do sistema operacional; o método `globalize_path` da classe `Globals` de *Godot* faz justamente isso. No entanto, se o jogo estiver no formato de um binário fechado, é impossível referenciar qualquer arquivo dentro dele por meio da plataforma hospedeira.

A solução final implementada envolve usar o outro prefixo definido pela *game engine*, o **`user://`**. Copiam-se todos os arquivos e diretórios de configuração para este caminho e usa-se o método `get_data_dir()` da classe `OS` para buscar seu equivalente na plataforma hospedeira. Por precaução, adotou-se a prática de sobrescrever arquivos e diretórios copiados previamente.

Inicializar variáveis do *Pocketsphinx*

A inicialização de variáveis usadas por *Pocketsphinx* é feita através de `init()`. Os nomes do diretório do modelo acústico, do arquivo de dicionário e do arquivo de palavras-chave precisam ter sido definidos previamente com os apropriados *setters*, ou o método retornará um número de erro.

Deparamo-nos com outro problema de compatibilidade: os nomes dos arquivos são fornecidos com o tipo `String` adotado por *Godot*. No entanto, *Pocketsphinx* não conhece este tipo, utilizando o `char *` comumente encontrado em *C* para receber estes mesmos nomes. Felizmente, `String` possui um método `c_str()` para realizar a conversão.

O método `init()` irá inicializar as variáveis de configuração `cmd_ln_t`, de gravação de voz `ad_rec_t` e o decodificador `ps_decoder_t` (todos vistos na seção 4.3). Caso algum problema ocorra, retorna-se um número de erro, pertencente à classe *STTError*, relativo ao problema ocorrido.

A cópia de arquivos para o caminho `user://`, mencionada anteriormente, também ocorre neste método.

6.3.2 Classe `STTRunner`

`STTRunner` é a classe responsável por realizar o reconhecimento de voz em si. Por claramente implementar uma **funcionalidade** a ser usada pelo usuário do editor *Godot*, decidiu-se que uma herança de `Node` (visto na seção 5.3.3) seria bastante apropriada.

A figura 6.3 apresenta os atributos, métodos e relacionamentos da classe `STTRunner`. O construtor e destrutor foram omitidos por simplicidade.

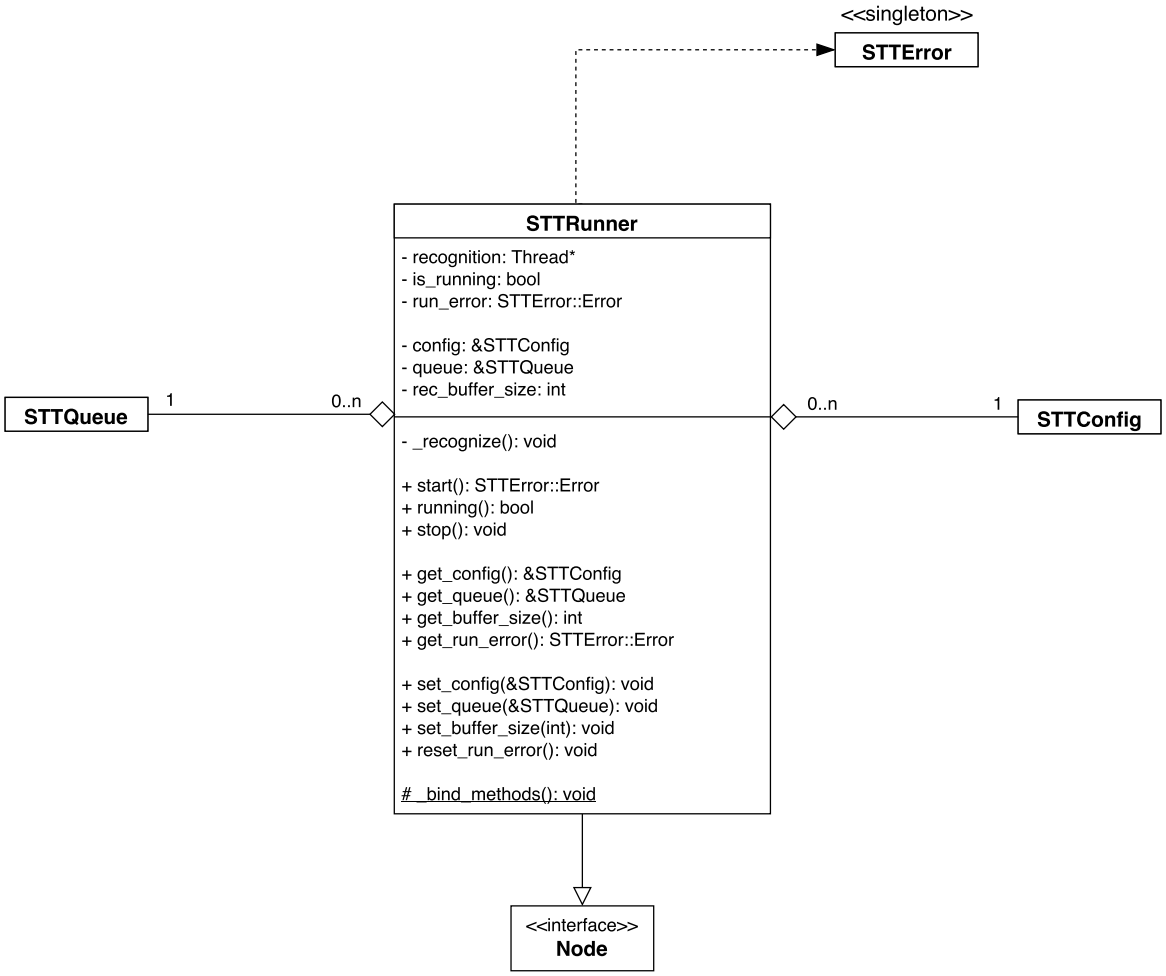

Figura 6.3: *Atributos, métodos e relacionamentos da classe* `STTRunner`

A classe usa um objeto `STTConfig` para obter as variáveis *Pocketsphinx* necessárias em sua tarefa, e um objeto `STTQueue` para guardar termos (palavras ou pequenas frases) gerados no reconhecimento de voz. Ambos podem ser obtidos e definidos através dos *getters* e *setters* para `config` e `queue`.

Comentamos, nos requisitos funcionais (seção 6.2.2), que o reconhecimento deveria ocorrer em paralelo com o restante do jogo. O uso de uma *thread* serve bem para tal propósito; *Godot*, inclusive, oferece uma implementação própria dessa estrutura em sua classe `Thread`. Seus dois métodos de maior importância para nós são:

- `Thread * create(função, argumento, configurações)`: Cria uma *thread* para executar uma função de assinatura `func(void *arg)`. O parâmetro **função** corresponde ao nome da função (neste caso, `func`) e **argumento**, a `arg`. Opcionalmente, podem ser passadas configurações adicionais como um terceiro parâmetro, mas não entraremos em detalhes por não terem sido necessárias. Retorna-se um ponteiro para a instância de `Thread` criada.

- `void wait_to_finish(thread)`: Recebe um ponteiro para uma instância de `Thread`. Espera a *thread* terminar, finalizando-a com segurança.

O método `start()` da nossa classe cria uma *thread* para realizar o reconhecimento de voz desde que o objeto `STTRunner` já possua instâncias de `STTConfig` e `STTQueue`. Chama-se `Thread::create()` com um método estático como parâmetro (`_thread_recognize()`, que não aparece na figura 6.3 por agir apenas como um intermédio) e a própria instância como argumento (isto é, `this`).

A listagem 6.4 mostra a implementação de `_thread_recognize()`, responsável por chamar o método privado `_recognize()` da instância passada como argumento. É nesta última função que o laço de reconhecimento de voz ocorre, implementado de forma bastante similar ao que vimos na seção 4.4.

```
void STTRunner::_thread_recognize(void *runner) {
    STTRunner *self = (STTRunner *) runner;
    self->_recognize();
}
```

Listagem 6.4: *Método estático `_thread_recognize()` de STTRunner*

Para controle sobre a *thread*, o usuário do módulo pode verificar sua execução com `running()` e ordenar sua parada imediata com `stop()`. Definiu-se que apenas uma *thread* pode existir por instância de `STTRunner`, pois o controle de várias operações em paralelo não traria nenhum benefício ao módulo.

Por fim, oferece-se controle sobre o tamanho do *buffer* de gravação de voz através do *getter* e *setter* de `buffer_size`. É importante ressaltar que o uso de um *setter* demanda a parada imediata da *thread* de reconhecimento para evitar possíveis problemas de acesso antes e depois da alteração (*race condition*, ou condição de corrida).

6.3.3 Classe `STTQueue`

`STTQueue` implementa um *buffer* para guardar palavras geradas no reconhecimento de voz, conforme planejado nos requisitos funcionais (seção 6.2.2). Fizemos que com herdasse da classe `Reference` (visto na seção 5.3.2) com o intuito de simplificar seu gerenciamento de memória.

É usado por `STTRunner` para guardar termos do reconhecimento de voz. Ao mesmo tempo, o usuário do módulo pode retirar estes termos do módulo e usá-los no jogo.

A figura 6.4 apresenta os atributos, métodos e relacionamentos da classe `STTQueue`. O construtor e destrutor foram omitidos por simplicidade.

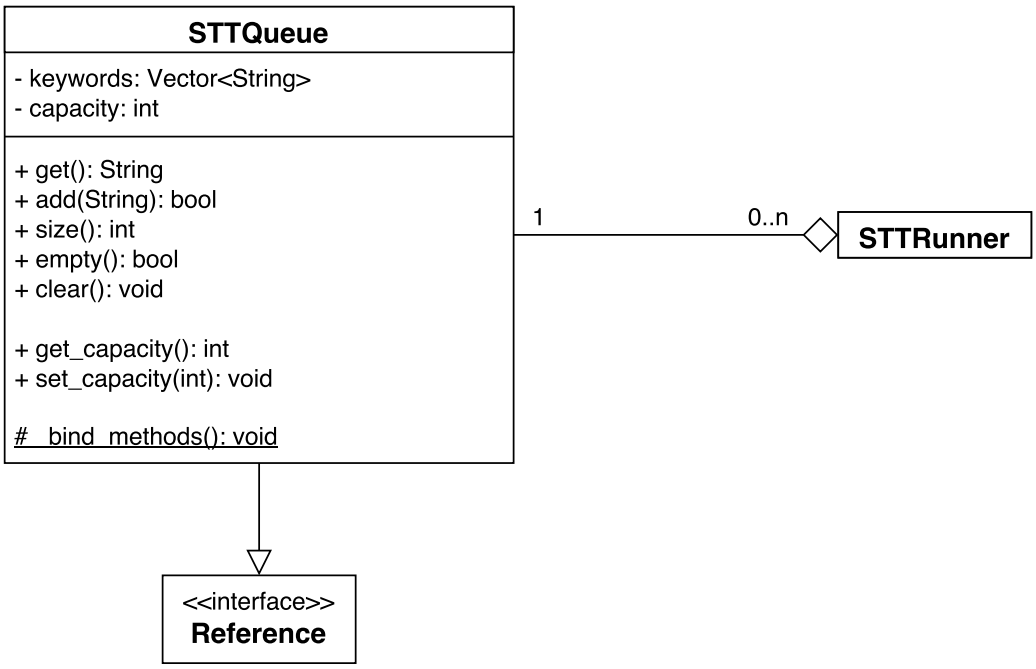

Figura 6.4: *Atributos, métodos e relacionamentos da classe* `STTQueue`

Conforme o nome sugere, o tipo abstrato de dados usado para o *buffer* da `STTQueue` é uma **fila**. *Godot* implementa esta estrutura, com diversas funcionalidade a mais, em sua classe `Vector`. Como muitos de seus métodos seriam desnecessários (não é necessário, por exemplo, que o usuário possa ordenar a fila de reconhecimento de voz), `STTQueue` age como um intermédio do que pode e não pode ser feito em `Vector`.

Os principais métodos de `STTQueue` estão resumidos na tabela 6.1.

Método	Descrição
`get()`	Retira e retorna a primeira `String` da fila
`add()`	Adiciona uma `String` no final da fila
`size()`	Retorna o número de elementos atualmente na fila
`empty()`	Retorna `true` se a fila estiver vazia
`clear()`	Esvazia a fila, removendo todos os seus elementos

Tabela 6.1: *Principais métodos de STTQueue*

O leitor pode se perguntar se não ocorre condição de corrida no acesso à fila: há risco de se perder algum dado quando `STTRunner` realiza um `add()` ao mesmo tempo em que o usuário utiliza `get()`, por exemplo? Felizmente, a implementação de `Vector` é *thread-safe*, garantindo que esses problemas não ocorram.

Uma decisão deveria ser tomada quanto a um caso particular de `get()`: o que fazer quando o método fosse chamado com uma fila vazia? Nesta situação, decidiu-se retornar uma `String` vazia (“”) e simultaneamente imprimir uma mensagem de *warning* devido ao uso impróprio. Recomenda-se, portanto, verificar o tamanho da fila com `empty()` antes do uso de `get()`.

Além dos métodos da tabela 6.1, há mais dois que foram criados para definir um limite superior para o número de elementos na fila. Este valor pode ser lido e alterado por `get_capacity()` e `set_capacity()`, respectivamente. O usuário, teoricamente, não deveria permitir um acúmulo muito grande de palavras na fila, mas se tal caso ocorrer, ao menos o consumo de memória fica controlado.

6.3.4 Classe `STTError`

`STTError` define constantes numéricas para possíveis erros que podem ocorrer no módulo (mais especificamente, em `STTConfig` e `STTRunner`). A utilidade da classe, portanto, está em ajudar o usuário a entender melhor a causa de um erro que possa vir a ocorrer no módulo. Por ser um *singleton* mas ter seu uso necessário no editor *Godot* (através de *GDScript*), decidiu-se fazer a classe herdar de `Object` (visto na seção 5.3.1).

A figura 6.5 apresenta os atributos, métodos e relacionamentos da classe `STTError`. O construtor e destrutor foram omitidos por simplicidade.

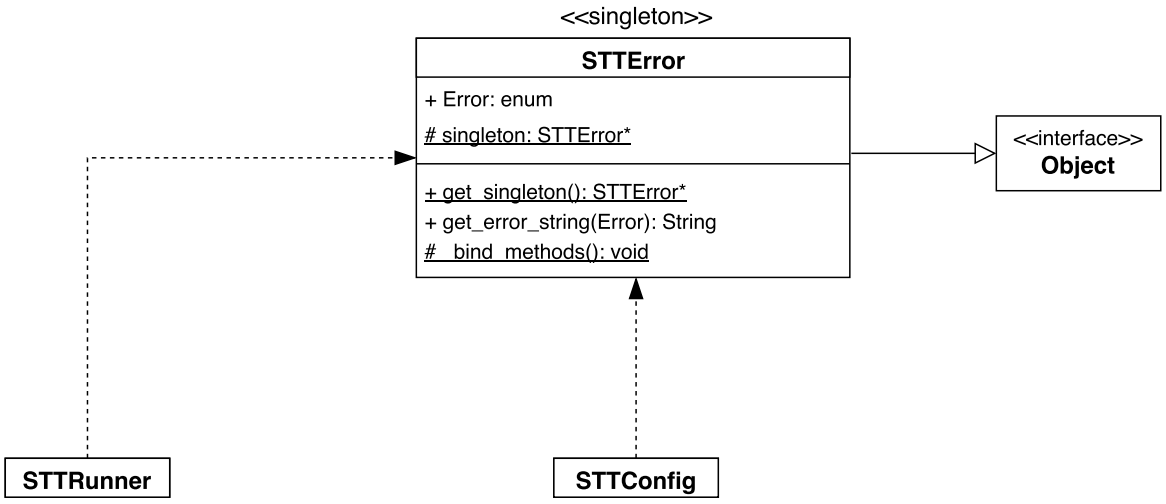

Figura 6.5: *Atributos, métodos e relacionamentos da classe `STTError`*

Alguns métodos de `STTConfig` e `STTRunner` retornam um número definido pelo `enum Error`. Uma `String` contendo a interpretação deste valor pode ser obtida chamando-se o método `get_error_string()` da classe.

A tabela 6.2 contém os valores definidos no `enum Error`, bem como suas respectivas interpretações.

Erro	Interpretação
`OK`	Nenhum erro ocorreu
`UNDEF_FILES_ERR`	Um ou mais nomes de arquivos/diretórios de configuração não foram definidos
`UNDEF_CONFIG_ERR`	Um objeto `STTConfig` não foi definido
`UNDER_QUEUE_ERR`	Um objeto `STTQueue` não foi definido
`USER_DIR_MAKE_ERR`	Erro ao criar o diretório STT em `user://`
`USER_DIR_COPY_ERR`	Erro ao copiar arquivos de configuração para `user://`
`MULTIBYTE_STR_ERR`	Erro ao converter o nome do arquivo para uma sequência de vários bytes
`MEM_ALLOC_ERR`	Não há memória disponível para alocação
`CONFIG_CREATE_ERR`	Erro ao criar a variável de configuração de *Pocketsphinx*
`REC_CREATE_ERR`	Erro ao se conectar ao aparelho de áudio (microfone)
`DECODER_CREATE_ERR`	Erro ao criar a variável de decodificação de *Sphinxbase*
`REC_START_ERR`	Erro ao começar a gravar a voz do usuário
`REC_STOP_ERR`	Não foi possível parar a gravação da voz do usuário
`UTT_START_ERR`	Erro ao iniciar a captura de *utterance* durante o reconhecimento de voz
`UTT_RESTART_ERR`	Erro ao reiniciar a captura de *utterance* durante o reconhecimento de voz
`AUDIO_READ_ERR`	Erro ao ler dados da gravação de áudio

Tabela 6.2: *Valores de erro definidos em `enum Error` e suas interpretações*

6.3.5 Classe `FileDirUtil`

`FileDirUtil` é uma classe auxiliar, não possuindo relação alguma com reconhecimento de voz. Como o nome sugere, ela possui métodos para manipular arquivos e diretórios.

A figura 6.6 apresenta os métodos e relacionamento da classe `FileDirUtil`. O construtor e destrutor foram omitidos por simplicidade.

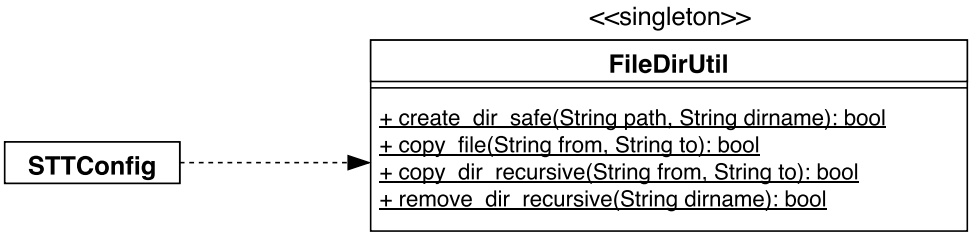

Figura 6.6: *Atributos, métodos e relacionamentos da classe `FileDirUtil`*

As ferramentas oferecidas por *Godot* para gerenciamento de arquivos e diretórios são bem simples, limitando-se ao seguinte: criação de arquivos/diretórios, escrita/leitura/cópia de arquivos, remoção de arquivos individuais e diretórios vazios.

Vimos que `STTConfig` necessita copiar os arquivos de configuração para `user://`. `FileDirUtil` meramente combina os métodos já implementados pela *game engine* para

realizar operações como cópia e remoção recursiva de diretórios.

6.4 Arquivos e *scripts* de configuração

Com a implementação feita, faltam apenas alguns arquivos e *scripts* que informam a existência das classes a *Godot*, bem como a forma de compilar o código fonte do módulo.

6.4.1 Registro de tipos

Godot exige dois arquivos adicionais, `register_types.h` e `register_types.cpp`, para controlar quais classes do módulo poderão ser usadas em *GDScript*. Duas funções obrigatoriamente devem ser implementadas, com o nome do módulo no meio de seu nome:

`register_speech_to_text_types()`

Serve para registrar classes no banco de dados `ObjectTypeDB`. A implementação desta função no módulo *Speech to Text* é apresentada na listagem 6.5.

```
static STTError *stt_error = NULL;

void register_speech_to_text_types() {
    ObjectTypeDB::register_type<STTConfig>();
    ObjectTypeDB::register_type<STTQueue>();
    ObjectTypeDB::register_type<STTRunner>();
    ObjectTypeDB::register_virtual_type<STTError>();

    stt_error = memnew(STTError);
    Globals::get_singleton()->add_singleton(
        Globals::Singleton("STTError", STTError::get_singleton()));
}
```

Listagem 6.5: *Implementação de `register_speech_to_text_types()`*

O método `register_type` de `ObjectTypeDB` faz o registro da classe. Há um caso especial em `STTError`, que usa `register_virtual_type` para não ser instanciável. Por fim, como esta mesma classe é um *singleton*, aproveitamos para criar sua única instância com `memnew()` (o equivalente de `new` para *Godot*) e guardar uma referência junto a outros *singletons* em `Globals`.

`unregister_speech_to_text_types()`

Esta função é chamada após o jogo ser finalizado (fechado). Tipicamente é usada para desalocar memória e/ou realizar tarefas de término no módulo. A implementação desta função no módulo *Speech to Text* é apresentada na listagem 6.6.

```
void unregister_speech_to_text_types() {
    if (stt_error) memdelete(stt_error);

    // Remove all STT data in user://
    String user_dirname = "user://" + String(STT_USER_DIRNAME);
    if (DirAccess::exists(user_dirname))
        FileDirUtil::remove_dir_recursive(user_dirname);
}
```

Listagem 6.6: *Implementação de* `unregister_speech_to_text_types()`

Em nosso caso, basta liberar o *singleton* de `STTError` alocado previamente; note que `memdelete` é o equivalente de `delete` em *Godot*. Também removemos todos os arquivos de configuração copiados para `user://` pela classe `STTConfig`.

6.4.2 `SCsub`

Já vimos que *SCons* é a ferramenta usada na compilação de *Godot* (seção 5.4), utilizando-se de *scripts* escritos em *Python* para definir seu comportamento. Com módulos não é diferente, necessitando-se de um *script* de nome `SCsub`.

A implementação deste script envolve listar quais os arquivos a serem compilados e quais *flags* o compilador deve usar.

Percebemos que uma compilação em *Windows* era possível devido à facilidade da ferramenta; bastou adicionar um arquivo próprio de *Pocketsphinx*, `ad_win32.c`, que opera especificamente sobre este sistema operacional.

6.4.3 `config.py`

A última configuração necessária é dada pelo *script* em *Python* `config.py`. Ele possui duas funções com propósitos diferentes:

`can_build(platform)`

Esta função recebe o nome da plataforma em que *Godot* está sendo compilado. Retorna-se `True` se o módulo pode ser compilado nela ou `False` caso contrário.

A listagem 6.7 apresenta a implementação da função.

```
import os

def can_build(platform):
    if platform == "x11":
        has_pulse = os.system("pkg-config --exists libpulse-simple") == 0
        has_alsa = os.system("pkg-config --exists alsa") == 0
        return has_pulse or has_alsa
    elif platform == "windows":
        return True
    else:
        return False
```

Listagem 6.7: *Função* `can_build()` *em* `config.py`

Em nosso caso, *Speech to Text* deve verificar se a plataforma *Unix* (referida como *x11*) possui sistema de som *PulseAudio* (Wikipedia, 2017f) ou *ALSA* (Wikipedia, 2017a), pois ambos são compatíveis tanto com *Godot* quanto com *Pocketsphinx*. Sistemas *Windows* são, teoricamente, sempre compatíveis devido a um arquivo de *Pocketsphinx* que trata seu sistema de som. Outros sistemas operacionais não foram testados, então preferiu-se retornar `False`.

`configure(env)`

Recebe um objeto caracterizando o ambiente de compilação, podendo realizar ajustes nele independentemente dos arquivos existentes no módulo.

Nenhuma mudança foi necessária em *Speech to Text*, o que levou esta função a ficar sem conteúdo (listagem 6.8).

```
def configure(env):
    pass
```

Listagem 6.8: *Função* `configure()` *em* `config.py`

6.5 Divulgação

Todo o código fonte do módulo *Speech to Text* encontra-se em um repositório no GitHub do autor (Macedo, 2017f), juntamente com instruções para compilação. Um tutorial para seu uso no editor, baseado no formato usado na documentação da própria *game engine*, encontra-se em outro repositório (Macedo, 2017i). Também foram disponibilizados binários dos editores *Godot* e *templates de exportação*, ambos compilados previamente com o módulo, para *Windows* e *Unix* (Macedo, 2017e).

A compilação com o módulo é bastante simples: basta baixar o diretório de *Speech to Text*, que deve ser inserido em `modules/`, e recompilar *Godot*, seguindo os mesmos passos apresentados na seção 5.5.

Speech to Text foi divulgado em dois fóruns de *Godot*, onde obteve algumas poucas aprovações:

- **Godot Engine Q&A** (Macedo, 2017h): O site oficial de *Godot* disponibiliza um fórum para perguntas e respostas, onde existe uma seção para publicação de projetos.
- **Godot Developers** (Macedo, 2017g): Embora seja mais voltado para jogos produzidos na *game engine*, este fórum possui uma seção para compartilhamento de recursos e ferramentas.

Capítulo 7

Jogo *Color Clutter*

Dado que o módulo *Speech to Text* está agora pronto, desenvolveremos um pequeno jogo com ele através do editor *Godot* para demonstrar seu uso e analisar a qualidade do reconhecimento de voz na prática.

Ao longo deste capítulo, relatamos os passos realizados na criação do jogo *Color Clutter*. Para melhor aproveitamento, o leitor precisará do editor *Godot*, na versão 2.1.4, com o módulo instalado. Mencionamos, na seção 6.5, alguns *links* para baixar uma versão já pronta e para obter instruções de como compilar *Speech to Text* com a *game engine*.

7.1 Uso do editor *Godot*

Antes de começarmos o desenvolvimento do jogo, iremos descrever alguns elementos da interface do editor *Godot* para melhor familiarizar o leitor com esta ferramenta.

7.1.1 Gerenciamento de projetos

Ao executar o editor *Godot*, o leitor é confrontado com a tela apresentada na figura 7.1, na qual pode gerenciar seus projetos (jogos).

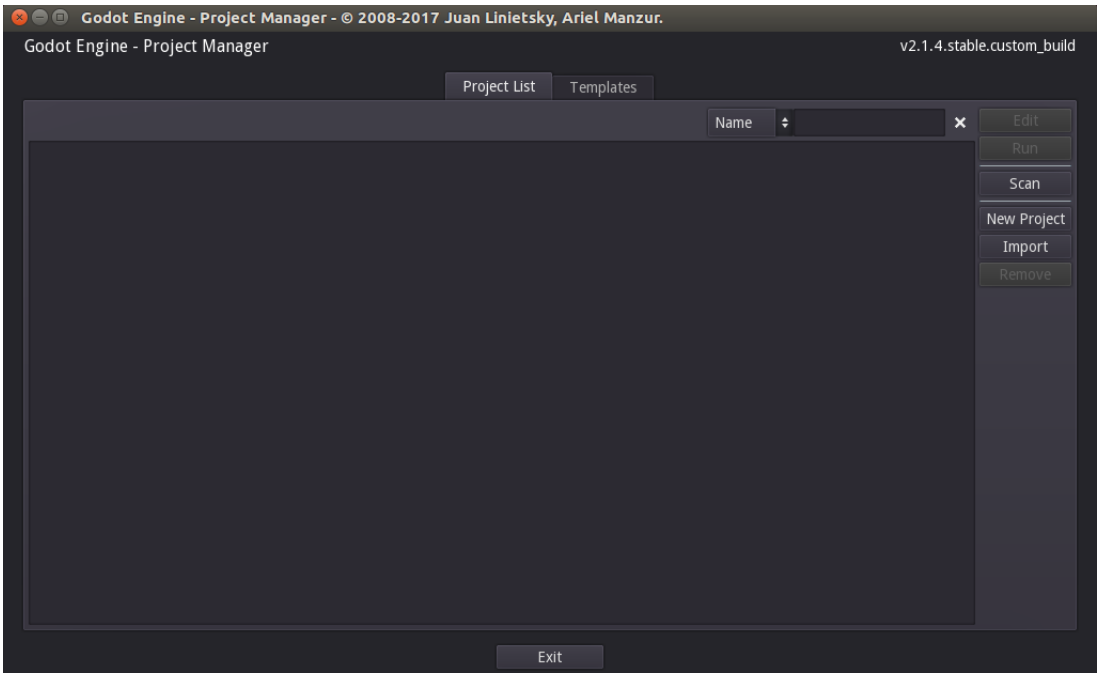

Figura 7.1: *Gerenciamento de projetos no editor Godot*

Projetos são listados na aba *Project List*, onde podem ser abertos para edição ou removidos. Também é possível criar um novo jogo ou importar um já existente. A criação de um projeto, por exemplo, é feita pelo botão *New Project* e requer a especificação do caminho na qual será criado (*Project Path*) e seu nome (*Project Name*). Já a importação exige que o usuário escolha um caminho que contenha o arquivo `engine.cfg`, responsável por definir o diretório raiz do jogo.

Na figura 7.2, exemplificamos a criação do projeto **teste** no diretório `~/Desktop/teste`. Ao clicar no botão *Create*, o projeto é criado e o usuário é automaticamente encaminhado para a tela de edição de **teste**.

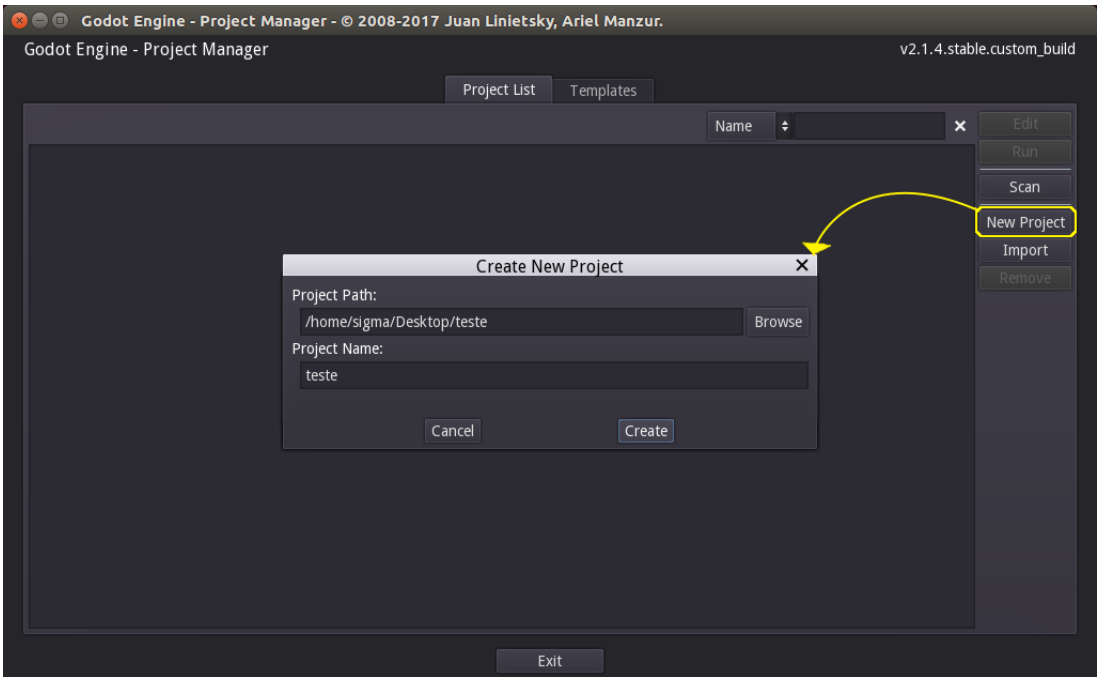

Figura 7.2: *Criação do projeto **teste** no editor Godot*

7.1.2 Interface de edição de um projeto

A criação de um novo projeto leva o usuário ao editor do mesmo. Continuando com nosso exemplo **teste**, sua tela de edição é apresentada na figura 7.3.

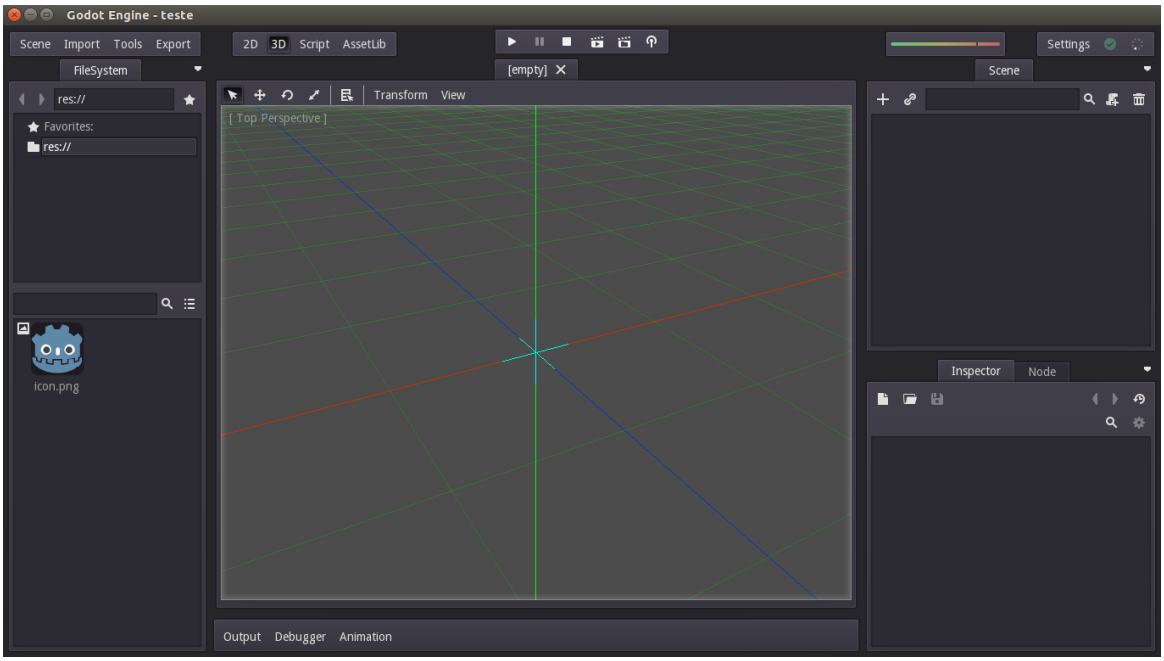

Figura 7.3: *Editor de um projeto Godot (neste caso, **teste**)*

Uma explicação detalhada de todas as funcionalidades do editor fugiria do tema deste trabalho. Iremos, portanto, nos atentar às ferramentas principais, apresentadas a seguir.

7.1.3 Uso de *nodes* e *scenes*

No canto superior direito do editor (conforme a figura 7.3, o *scene* (explicado na seção 5.3.4) atual do projeto é indicada na aba de mesmo nome. É neste espaço que a hierarquia de *nodes* (definidos na seção 5.3.3) pertencentes a esta cena é apresentada.

A criação de um *node* para a cena é feita através do símbolo **+** desta aba, que leva o usuário a uma lista de nós existentes (figura 7.4). Por curiosidade, recomenda-se que o leitor procure o nó ***STTRunner*** (descrito na seção 6.3.2), fornecido pelo módulo *Speech to Text*.

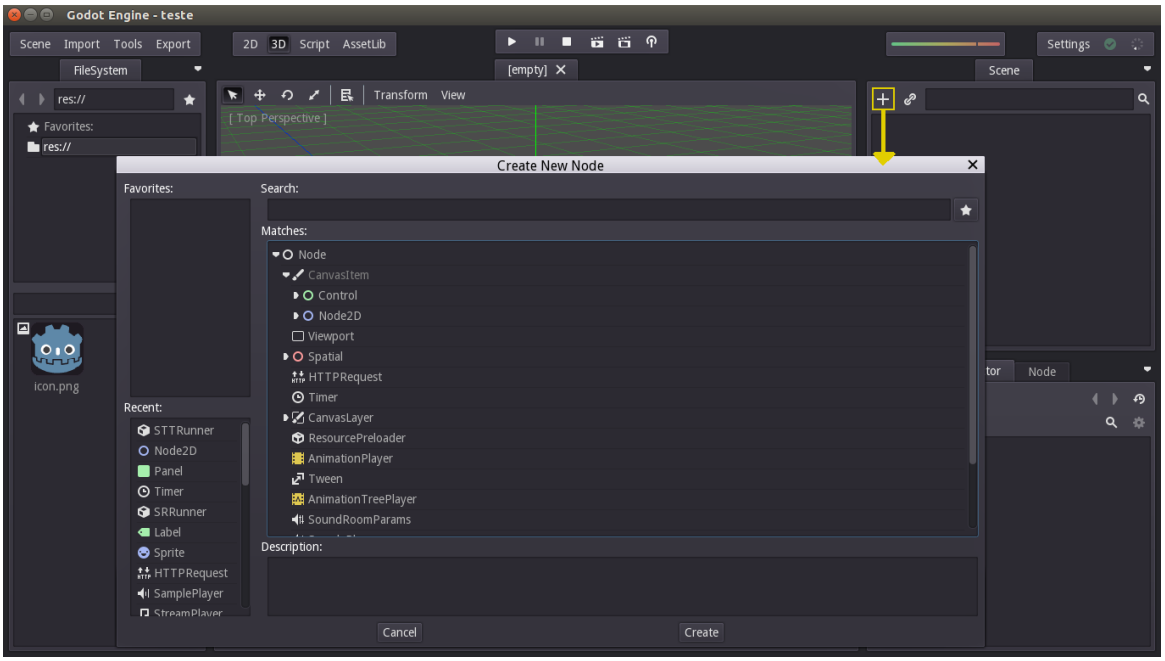

Figura 7.4: *Lista de nodes disponíveis no editor do projeto*

Como exemplo, escolheremos, nesta lista, o *node* ***Label*** para representar um texto. A aba *Scene* mostrará que foi adicionada à cena. Além disso, se selecionarmos este nó, a aba debaixo, *Inspector*, indicará quais as suas propriedades editáveis, enquanto a janela central do editor mostrará seu posicionamento em relação a outros elementos do jogo (figura 7.5).

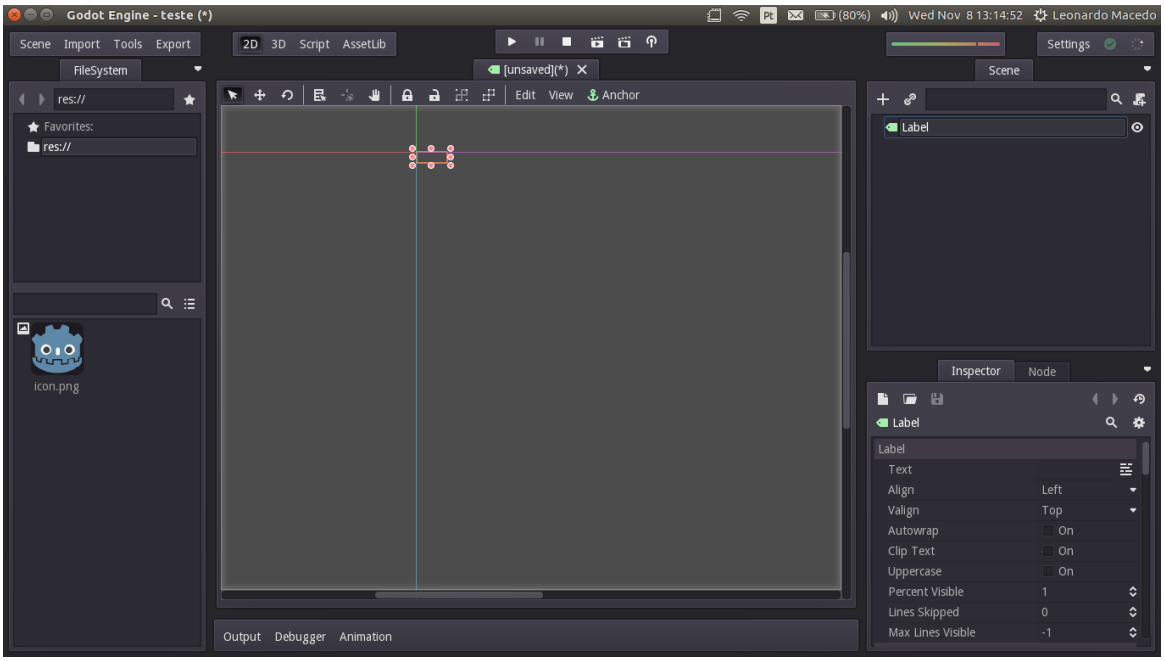

Figura 7.5: *Edição do node **Label**; note suas propriedades na aba Inspector (canto inferior direito)*

Suponha que desejamos executar o jogo com apenas o nó *Label*, com ele apresentando o texto *"Hello World!"*. Para tanto, seguiremos os seguintes passos, ilustrados melhor pela figura 7.6.

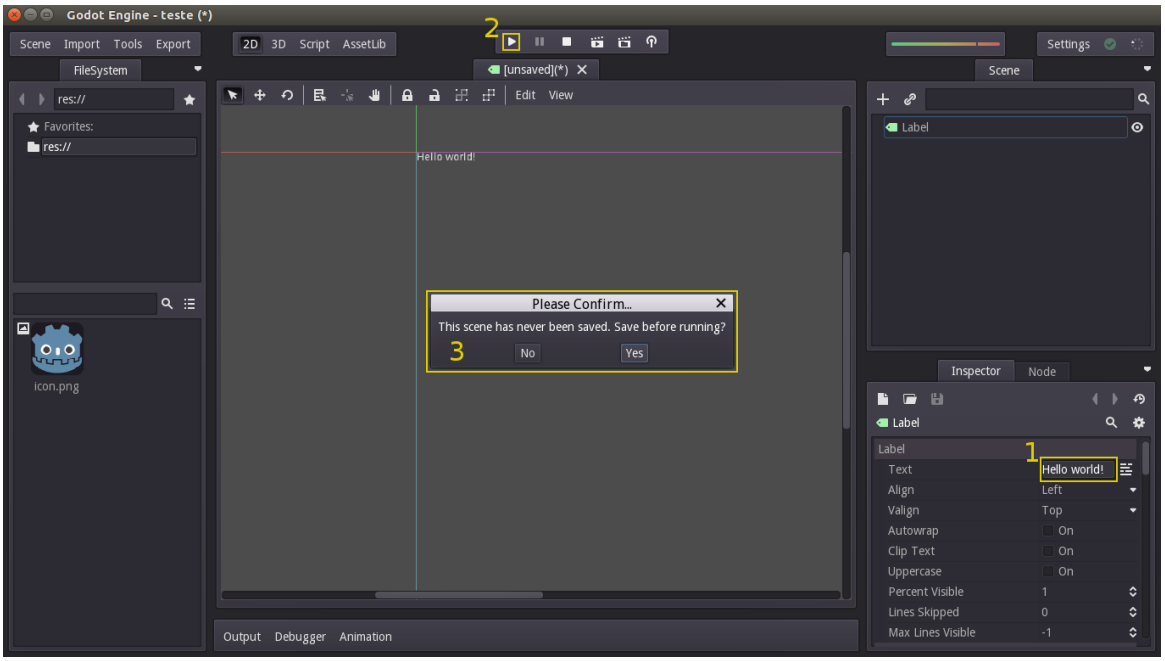

Figura 7.6: *Passos para execução da scene contendo um Label escrito "Hello World!"*

- Alterar o texto do *Label* para *"Hello World!"*. Isto é facilmente feito pela propriedade ***Text*** na aba *Inspector* (**1** na figura 7.6).
- Salvar o *scene* atual. Uma forma de fazer isso para depois executar o jogo é clicar no

botão *Play the project* (triângulo no centro superior do editor; vide **2** na figura 7.6). O editor apresentará uma janela para salvar a cena (**3** na figura 7.6) como um arquivo cujo formato padrão é `.tscn`.

- Se *Play the project* foi usado para salvar a cena, imediatamente outra janela será exibida, solicitando a escolha do *scene* a ser usado quando o jogo é iniciado. Basta escolher o arquivo `.tscn` que foi salvo no passo anterior.

O resultado, apresentado na figura 7.7, será a abertura de uma nova janela para a execução do jogo. Note o texto *"Hello World!"* em seu canto superior esquerdo.

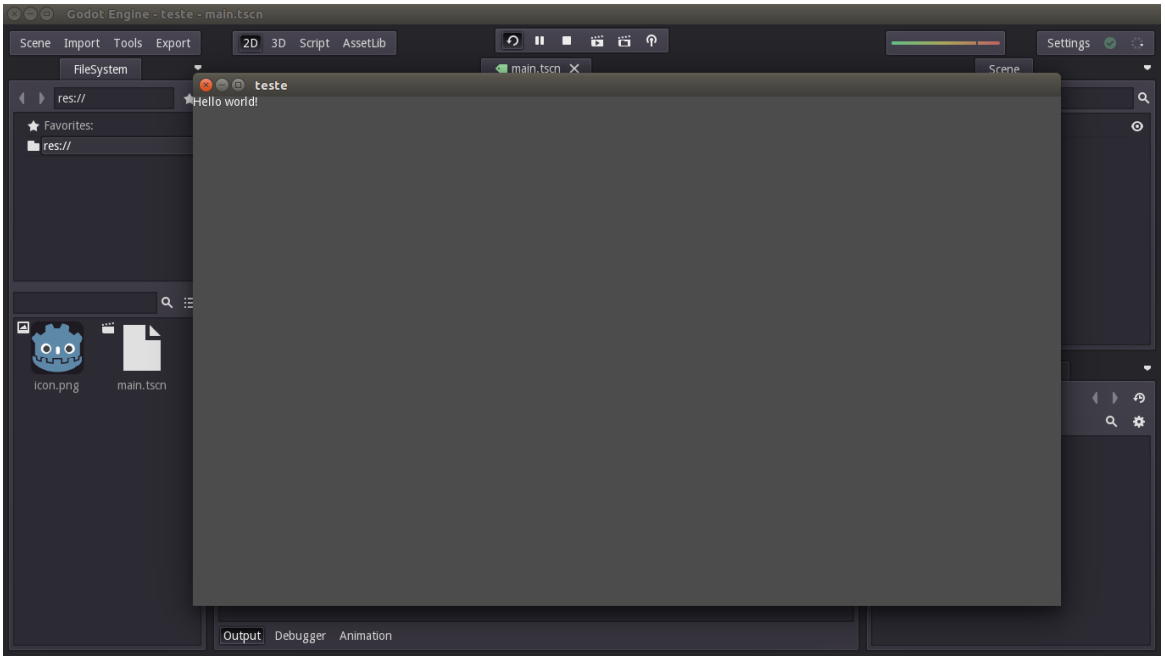

Figura 7.7: *Execução da scene contendo um Label de texto "Hello world!"*

7.1.4 Uso de *GDScript*

A linguagem *GDScript* (introduzida na seção 5.2) é usada para escrever trechos de código que definem o comportamento de *nodes* (Godot Docs, 2017j).

Para adicionar um *script* a um nó, basta clicar em seu nome, na aba *Scene*, com o botão direito e escolher ***Add Script*** (figura 7.8). Uma janela irá surgir, onde é necessário preencher apenas o campo *Path* para definir o nome do *script* (figura 7.9).

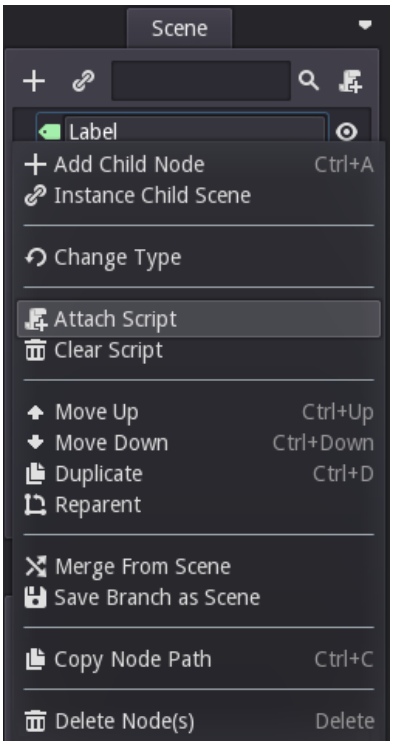

Figura 7.8: *Opção **Attach Script** para o node Label*

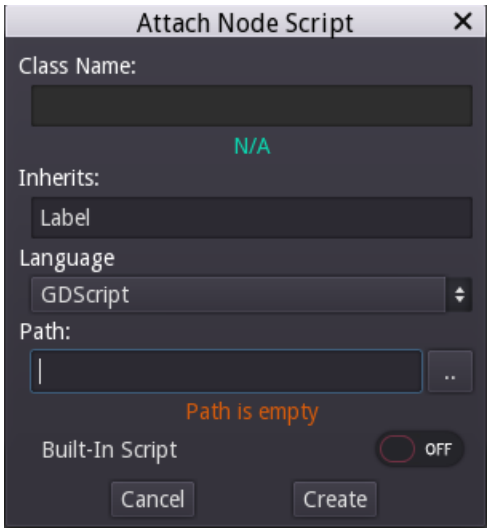

Figura 7.9: *Janela de criação do arquivo script*

Suponha que definimos *Path* como `res://script.gd` na figura 7.9. Ao clicarmos em *Create*, a aba central do editor mudará para ***Script***, onde o código do arquivo pode ser modificado. Ilustramos a modificação ocorrida na figura 7.10. Por curiosidade, veja, neste mesma imagem, que todos os recursos (como *scenes*, *scripts* e gráficos) existentes dentro do diretório do projeto são mostrados no canto inferior esquerdo do editor.

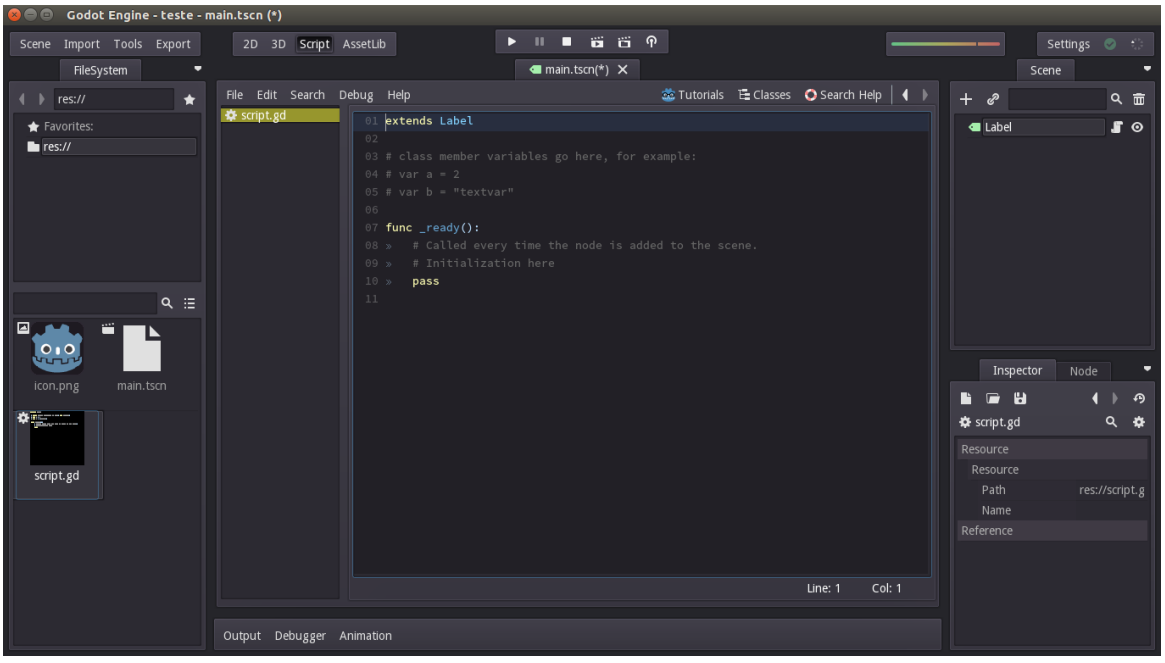

Figura 7.10: *A aba **Script** permite modificar arquivos homônimos*

Não iremos ensinar, neste trabalho, a linguagem *GDScript* porque o ganho nisso seria mínimo. Um tutorial detalhado, apresentado por *Godot* em (Godot Docs, 2017c), já existe com esta finalidade. No entanto, explicaremos duas funções predefinidas pela linguagem e que possuem bastante importância no controle de *nodes*.

`_ready()`

A função `_ready()` é chamada sempre que o *node* é adicionado à cena. É geralmente usada para inicializar atributos e preparar configurações de comportamento.

É importante lembrar que nós são organizados em árvore. Portanto, o `_ready()` de um *node* pai é sempre chamado antes da função respectiva nos filhos.

`_process(delta)`

`_process()` é uma função chamada repetidamente enquanto o nó existir na cena e `set_process(true)` tiver sido chamada previamente (em `_ready()`, por exemplo) para solicitar seu uso. Seu único argumento, `delta`, representa o tempo desde a última atualização de *frame* do jogo.

Esta função é muito útil para executar trechos de código periodicamente durante um jogo. Por exemplo, imagine um jogo em que o personagem controlado pelo usuário possua um certo número de pontos de vida. Poderia-se utilizar `_process()` para checar se este valor chegou a zero e, se afirmativo, alterar o *scene* atual para o do fim de jogo.

7.2 Planejamento

Além da escrita de código, jogos costumam usar diversos materiais, ou *assets*, como modelos gráficos, música, sons, fontes e texturas. A criação destes demanda bastante tempo e em geral exige uma equipe diversificada e qualificada. Portanto, gostaríamos de produzir um jogo que utilize poucos *assets*; buscaremos quaisquer materiais necessários em páginas Web que os disponibilizem para uso grátis e sem licença.

Quanto à jogabilidade, desejamos colocar ênfase na funcionalidade de reconhecimento de voz; este é o objetivo da criação do jogo, afinal. O uso de alguns poucos comandos orais em um projeto curto é o ideal para deixar as regras simples, mas com interação razoável com o usuário.

Por fim, definiremos que o jogo e reconhecimento de voz usarão **inglês americano** (*US English*), pois sabemos que um modelo acústico e arquivo de dicionário para *Pocketsphinx* existem e são de uso livre para esta língua.

Com estas características em mente, planejamos o jogo ***Color Clutter***.

7.2.1 Descrição de *Color Clutter*

Conforme o nome sugere, *Color Clutter* é um jogo cuja temática envolve uma "confusão" entre cores.

Uma típica tela do jogo consiste em um fundo totalmente preenchido com alguma cor X. Em alguma posição da tela, uma outra cor Y aparece escrita em um tom Z. O objetivo do usuário é falar a cor correta (X, Y ou Z), de acordo com o que é solicitado em uma legenda apresentada na tela.

Para acrescentar um aspecto competitivo e tornar *Color Clutter* mais lúdico, o jogo será no formato de rodadas, onde marcaremos quantos acertos o usuário consegue em 1 minuto. Cada resposta correta altera aleatoriamente as cores e a posição da palavra na tela; não há penalidade para uma resposta errada, exceto a perda de tempo acarretada pela mesma.

A figura 7.11 apresenta a tela do jogo durante uma rodada. O canto superior esquerdo informa qual cor deve ser pronunciada, enquanto o canto superior direito exibe o tempo restante e o *score* (pontuação) atual do jogador. Na situação apresentada, o usuário precisaria falar a cor correspondente ao *background* (fundo): ***blue***.

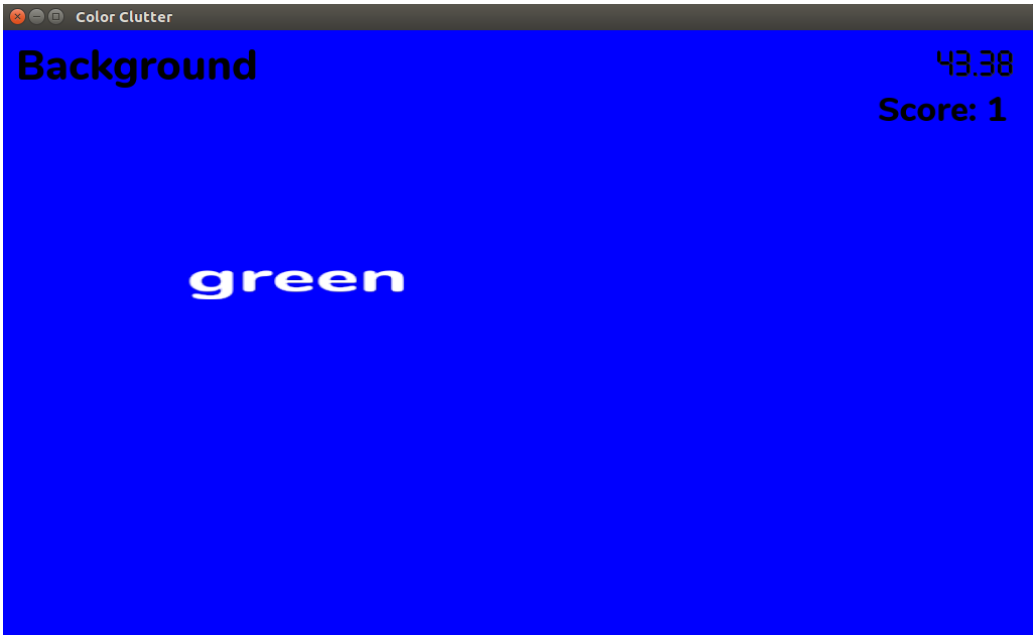

Figura 7.11: *Tela do jogo Color Clutter durante uma rodada*

7.3 Desenvolvimento

A simplicidade de *Color Clutter* nos levou a implementá-lo com apenas um *scene*. A árvore de *nodes* que o compõe é apresentada na figura 7.12.

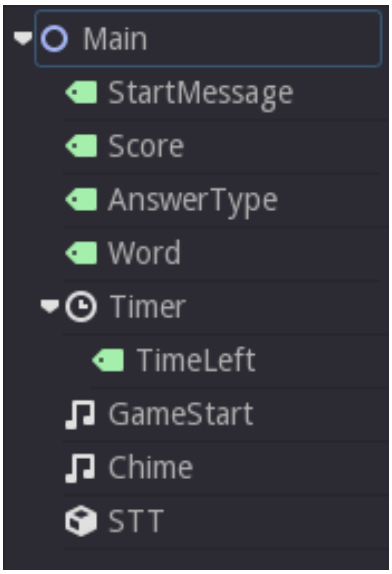

Figura 7.12: *Árvore de nodes utilizada em Color Clutter*

Explicamos, a seguir, o funcionamento geral do jogo e detalhamos alguns dos *nodes* utilizados.

7.3.1 Máquina de estados

Em termos de funcionamento geral, o jogo transita entre dois estados, *Start* e *Running*. O primeiro caso define o início do jogo, onde espera-se o usuário falar *start* para começar uma partida. Ao reconhecer tal palavra, transita-se para o estado *Running*, correspondente à partida em si, onde o usuário tem 1 minuto para acertar o máximo de cores possíveis. Terminado este tempo, exibe-se a pontuação do jogador e volta-se para o estado *Start*.

A qualquer momento, o usuário pode sair do jogo através do comando de voz *exit*, imediatamente finalizando *Color Clutter*.

7.3.2 *Node* `Main`

`Main` é um *Node2D*, que é um nó base para jogos em 2D; por isso seu uso como o *node* raiz da cena.

Conforme indicado na figura 7.12, este nó contém `STT`, que é do tipo *STTRunner*. Para poder utilizá-lo, `Main` prepara dois objetos na função `_ready()` de seu *script*:

- Um objeto da classe *STTConfig* (explicado na seção 6.3.1). Definiram-se os arquivos de configuração necessários: um modelo acústico em inglês, um dicionário de palavras em inglês e um arquivo de palavras-chave contendo os comandos usados no jogo. Neste último, os valores de limiar escolhidos foram, em geral, baseados no número de fonemas existentes no comando. Por fim, utilizou-se o método `init()` para inicializar a instância, deixando-a pronta para uso no *STTRunner*.

- Um objeto da classe *STTQueue* (explicado na seção 6.3.3). Esta fila guarda os comandos de voz falados pelo usuário.

O laço geral do jogo (*game loop*) está implementado na função `_process(delta)`, que é apresentada em pseudocódigo a seguir:

```
1:  function _process(delta)
2:      if not stt_queue.empty() then
3:          command ← stt_queue.get()
4:      end if
5:
6:      if game_state == START then
7:          if command == "start" then
8:              start_game()
9:          end if
10:     else if game_state == RUNNING then
11:         if command == right_answer then
12:             update_score()
13:             update_round()
14:         end if
15:     end if
16:
17:     if command == "exit" then
18:         exit_game()
19:     end if
20: end function
```

A função `update_score()` é responsável por atualizar a pontuação guardada no *Label* `Score`, enquanto `update_round()` altera os *Labels* `Word` (a palavra-cor espalhada na tela) e `AnswerType` (que descreve se a próxima resposta é a palavra, sua cor ou o fundo).

7.3.3 *Node* `Word`

`Word` é um nó do tipo *Label*, correspondendo à palavra gerada aleatoriamente pela tela. Quando o usuário acerta a cor da rodada, sua posição é alterada para algum valor dentro dos limites da tela. Garante-se que a cor em que `Word` é escrita não seja igual à cor de fundo (uma palavra azul em um fundo azul, por exemplo, seria de difícil visualização).

7.3.4 *Node* `Timer`

`Timer` é um nó de tipo homônimo. Como o nome indica, ela é usada como um relógio: após atribuir um valor de tempo e acioná-lo, exerce-se uma contagem regressiva, realizando automaticamente uma chamada de função (*callback*) quando chegar a zero.

No contexto de *Color Clutter*, este *node* automaticamente atualiza seu filho `TimeLeft`, que é um *Label* usado para indicar, na tela, o tempo restante de jogo. `Timer` chama uma função em `Main` quando a partida acaba, com o objetivo de alterar o estado do jogo para *Start*.

7.3.5 *Nodes* `GameStart` e `Chime`

`GameStart` e `Chime` são nós do tipo *StreamPlayer*. Guardam um efeito sonoro tocado no início do jogo e quando o usuário acerta a cor, respectivamente.

7.4 Testes

Color Clutter foi testado por 10 pessoas com boa fluência em inglês. A maioria dos testadores utilizavam alguma versão de *Windows* (versões 7, 8 ou 10), com três testes sendo feitos nas distribuições *Ubuntu*, *Debian* e *Linux Mint* do sistema operacional *Unix*.

Um dos primeiros resultados visualizados é que não houve diferença perceptível no reconhecimento de voz entre os sistemas operacionais. Todos os testadores comentaram que a velocidade de reconhecimento estava rápida (menos de 1 segundo).

No entanto, praticamente todas as pessoas envolvidas no teste comentaram que alguns comandos não eram reconhecidos na primeira vez em que eram pronunciados, exigindo-se repeti-los 2 ou 3 vezes até serem aceitos. Destacam-se as cores *green*, *orange* e *purple* como as palavras que, em geral, geravam maior dificuldade de reconhecimento.

A efeito de comparação, executou-se uma API de reconhecimento de voz desenvolvida pelo Google (Google, Ano Desconhecido), configurada para o inglês americano, simultaneamente em que *Color Clutter* era jogado. Este programa reconheceu todas as palavras proferidas enquanto *Color Clutter*, por vezes, não entendia algum comando. Duas respostas possíveis surgem para esta diferença: a eficiência da API do Google é bem maior do que a de *Pocketsphinx*, ou é necessário um ajuste no limiar das palavras-chave do jogo (ou seja, é possível que uma pronúncia muito parecida com a da fonética do dicionário esteja sendo exigida).

7.5 Divulgação

Todo o código fonte de *Color Clutter* encontra-se em um repositório no GitHub do autor (Macedo, 2017b), juntamente com as instruções do jogo. Também foram disponibilizados binários para *Windows* e *Unix* (Macedo, 2017a).

Color Clutter foi divulgado nos mesmos dois fóruns de *Godot* onde publicou-se sobre *Speech to Text*:

- **Godot Engine Q&A** (Macedo, 2017d): Embora seja mais voltado para tirar dúvidas, publicou-se o jogo no fórum oficial de *Godot*.
- **Godot Developers** (Macedo, 2017c): Voltado principalmente para jogos produzidos na *game engine*.

Capítulo 8

Conclusão

Este Trabalho de Conclusão de Curso envolveu o desenvolvimento de um módulo de reconhecimento de voz para a *game engine Godot*. Para tanto, dedicou-se um tempo razoável do trabalho para o estudo das principais características de sistemas de reconhecimento de voz. Ficou evidente o vasto uso que esta tecnologia possui hoje, e o futuro indica que seu uso poderá crescer ainda mais. A tendência de ideologias como *Internet das Coisas* é trazer cada vez mais poder computacional para automatizar tarefas repetitivas na vida do ser humano, e interação por voz se encaixaria bem para tais situações.

A procura por bibliotecas de código aberto que atendam a certas expectativas veio a seguir, levando-nos a estudar *Pocketsphinx* como a opção mais viável no contexto de usabilidade dentro de um jogo. Apesar da documentação da biblioteca ser por vezes incompleta, vários exemplos em fóruns possibilitaram um aprendizado satisfatório de seu uso.

Godot possui uma documentação bastante completa quanto a seu uso para jogos. No entanto, não há muito material disponível relacionado a suas classes em *C++*. A leitura e entendimento de código, portanto, foi uma habilidade bastante exercida nesta parte, uma vez que certas implementações no módulo *Speech to Text* foram baseadas em algo já feito em outra classe.

A produção do jogo demonstrativo *Color Clutter* teve importância considerável no final deste trabalho, pois mostrou que o módulo correspondeu às expectativas referentes a eficiência e simplicidade de uso.

Em geral, o resultado final foi bastante satisfatório, uma vez que o módulo *Speech to Text* foi até divulgado, com seu código aberto, em fóruns da *game engine*, junto ao jogo *Color Clutter* produzido. Deseja-se continuar o suporte ao módulo no futuro, possivelmente estudando-se mais sobre *Android* e *MacOS* para tentar portá-lo a estes sistemas operacionais.

Agradecimentos

Gostaria de agradecer muito a minha família, principalmente a meus pais, pelo constante apoio durante toda a minha vida universitária. Dificilmente teria conseguido arranjar forças para chegar até o final do curso se não fosse por eles.

Sou muito grato ao IME e aos professores com os quais tive aula. Pretendo levar a dedicação e ensinamentos deles pelo resto de minha vida profissional. Em especial, agradeço ao meu orientador, o professor Gubi, pelas várias reuniões e conversas ao longo do ano sobre este trabalho.

Por fim, agradeço o apoio dos poucos mas valiosos amigos que fiz durante a graduação, pois a experiência da universidade não teria sido a mesma sem a presença deles.

Disciplinas importantes para este trabalho

Dentre as disciplinas que cursei durante a graduação, as seguintes se destacam pela relação maior com a confecção deste trabalho:

(MAC0110) Introdução a Computação
(MAC0122) Princípios de Desenvolvimento de Algoritmos
(MAC0323) Estrutura de Dados

Entrei no Bacharelado em Ciência da Computação sem conhecimento algum de programação, mas com o intuito de aprender bastante sobre o assunto. Estas três disciplinas fornecerem a base que eu desejava, e mostraram que programar não é apenas redigir código. Escrever um programa eficiente, robusto e correto exige raciocínio lógico, capacidade de abstração e domínio das estruturas de dados disponíveis.

(MAC0211) Laboratório de Programação I
(MAC0242) Laboratório de Programação II

Estas disciplinas possuem um foco mais prático de ferramentas e concepções para programação em si. Destacam-se o uso de linguagens de *script*, expressões regulares e Programação Orientada a Objetos. Meu primeiro contato com UML também ocorreu nestas disciplinas.

(MAC0332) Engenharia de Software

Apresentou metodologias interessantes para a produção de *software*. Também tive um forte contato com diagramas UML. Sinto que a classificação de aplicações de acordo com sua finalidade e técnicas importantes para planejamento me ajudaram, de certa forma, na preparação dos requisitos do módulo de reconhecimento de voz.

(MAC0441) Programação Orientada a Objetos

Minha primeira experiência com a produção de um *software* de proporção bem maior que um típico Exercício-Programa, pois fora realizado um projeto sobre Cidades Inteligentes que englobou a classe inteira. Ajudou a aprofundar meus conhecimentos sobre Orientação a Objetos e a produção de código de qualidade.

(MAC0422) Sistemas Operacionais

Apresentou a ideia de *threads* pela primeira vez no curso, além de mostrar as componentes principais de sistemas operacionais. Em especial, destacamos o gerenciamento de arquivos, que apareceu, neste trabalho, na implementação própria do sistema de arquivos de *Godot*.

(MAC0425) Inteligência Artificial

Alguns tópicos, como *Markov Decision Process* (MDP), lembram o procedimento estocástico comentado em *Hidden Markov Model* (HMM).

Referências Bibliográficas

Cassiopedia(Ano Desconhecido) Cassiopedia. *Computing History Timeline*. http://www.cassiopeia.it/resources-2/computing-history-timeline, Ano Desconhecido. Acessado: 2017-09-29. ix, 4

CMUSphinx(2015) CMUSphinx. *About the CMUSphinx*. http://cmusphinx.sourceforge.net/wiki/about, Fevereiro 2015. Acessado: 2017-04-03. 11

CMUSphinx(2016a) CMUSphinx. *Building application with Pocketsphinx*. http://cmusphinx.sourceforge.net/wiki/tutorialpocketsphinx, 2016a. Acessado: 2017-04-17. 15, 18, 19

CMUSphinx(2016b) CMUSphinx. `continuous.c`. https://github.com/cmusphinx/pocketsphinx/blob/master/src/programs/continuous.c, Janeiro 2016b. Acessado: 2017-10-20. 20

Cook(2002) Stephen Cook. *Speech Recognition HOWTO*. http://www.tldp.org/HOWTO/Speech-Recognition-HOWTO/introduction.html, Abril 2002. Acessado: 2017-09-30. 6

Enger(2013) Michael Enger. *Game Engines: How do they work?* https://www.giantbomb.com/profile/michaelenger/blog/game-engines-how-do-they-work/101529/, Junho 2013. Acessado: 2017-04-03. 1

Gaida *et al.***(2014)** Christian Gaida, Patrick Lange, Rico Petrick, Patrick Proba, Ahmed Malatawy e David Suendermann-Oeft. *Comparing Open-Source Speech Recognition Toolkits*. http://suendermann.com/su/pdf/oasis2014.pdf, 2014. Acessado: 2017-04-03. xi, 11

GitHub(2017) GitHub. *Repositório Godot*. https://github.com/godotengine/godot, 2017. Acessado: 2017-10-01. 30

Godot Docs(2017a) Godot Docs. *Inheritance class tree*. http://docs.godotengine.org/en/stable/development/cpp/inheritance_class_tree.html, 2017a. Acessado: 2017-10-02. ix, 25

Godot Docs(2017b) Godot Docs. *File system*. http://docs.godotengine.org/en/stable/learning/step_by_step/filesystem.html, 2017b. Acessado: 2017-10-03. 28

Godot Docs(2017c) Godot Docs. *GDScript*. http://docs.godotengine.org/en/stable/learning/scripting/gdscript/gdscript_basics.html, 2017c. Acessado: 2017-10-29. 58

Godot Docs(2017d) Godot Docs. *Custom modules in C++*. http://docs.godotengine.org/en/stable/development/cpp/custom_modules_in_cpp.html, 2017d. Acessado: 2017-10-15. 33

Godot Docs(2017e) Godot Docs. *Nodes*. http://docs.godotengine.org/en/stable/learning/step_by_step/scenes_and_nodes.html#nodes, 2017e. Acessado: 2017-10-01. 26

Godot Docs(2017f) Godot Docs. *Distro-specific oneliners*. http://docs.godotengine.org/en/stable/development/compiling/compiling_for_x11.html#distro-specific-oneliners, 2017f. Acessado: 2017-10-01. 31

Godot Docs(2017g) Godot Docs. *Resources*. http://docs.godotengine.org/en/stable/learning/step_by_step/resources.html, 2017g. Acessado: 2017-10-01. ix, 27, 28

Godot Docs(2017h) Godot Docs. *Scenes*. http://docs.godotengine.org/en/stable/learning/step_by_step/scenes_and_nodes.html#scenes, 2017h. Acessado: 2017-10-01. ix, 26, 27

Godot Docs(2017i) Godot Docs. *Introduction to the buildsystem*. http://docs.godotengine.org/en/stable/development/compiling/introduction_to_the_buildsystem.html, 2017i. Acessado: 2017-10-01. 29

Godot Docs(2017j) Godot Docs. *Scripting*. http://docs.godotengine.org/en/stable/learning/step_by_step/scripting.html, 2017j. Acessado: 2017-10-01. 24, 56

Google(Ano Desconhecido) Google. *Web Speech API Demonstration*. https://www.google.com/intl/en/chrome/demos/speech.html, Ano Desconhecido. Acessado: 2017-10-30. 62

HTK(2016) HTK. *What is HTK?* http://htk.eng.cam.ac.uk, 2016. Acessado: 2017-04-03. 11

IBM(Ano Desconhecido) IBM. *IBM Shoebox*. https://www-03.ibm.com/ibm/history/exhibits/specialprod1/specialprod1_7.html, Ano Desconhecido. Acessado: 2017-09-29. 3

Kaldi(2017) Kaldi. *About the Kaldi project*. http://kaldi-asr.org/doc/about.html, Março 2017. Acessado: 2017-04-03. 11

Linietsky(2016) Juan Linietsky. *Using static functions in a module*. https://godotdevelopers.org/forum/discussion/15686/using-a-static-functions-in-a-module, Julho 2016. Acessado: 2017-10-21. 38

Linietsky e Manzur(2017a) Juan Linietsky e Ariel Manzur. *Godot Engine*. https://godotengine.org, 2017a. Acessado: 2017-04-03. i, iii, 1, 23

Linietsky e Manzur(2017b) Juan Linietsky e Ariel Manzur. *Godot Features*. https://godotengine.org/features#multiplatform-deploy, 2017b. Acessado: 2017-09-20. 10, 28

Macedo(2017a) Leonardo Pereira Macedo. *Color Clutter Releases*. https://github.com/SamuraiSigma/color-clutter/releases/latest, Setembro 2017a. Acessado: 2017-10-21. 62

Macedo(2017b) Leonardo Pereira Macedo. *Color Clutter*. https://github.com/SamuraiSigma/color-clutter, Setembro 2017b. Acessado: 2017-10-21. 62

Macedo(2017c) Leonardo Pereira Macedo. *Color Clutter game (Speech to Text module demo)*. https://godotdevelopers.org/forum/discussion/18660/color-clutter-speech-to-text-module-demo, Setembro 2017c. Acessado: 2017-10-22. 63

Macedo(2017d) Leonardo Pereira Macedo. *Color Clutter game (Speech to Text module demo)*. https://godotengine.org/qa/18323/color-clutter-game-speech-to-text-module-demo, Setembro 2017d. Acessado: 2017-10-22. 63

Macedo(2017e) Leonardo Pereira Macedo. *Speech to Text Releases*. https://github.com/SamuraiSigma/speech-to-text/releases/latest, Setembro 2017e. Acessado: 2017-10-21. 48

Macedo(2017f) Leonardo Pereira Macedo. *Speech to Text*. https://github.com/SamuraiSigma/speech-to-text, Setembro 2017f. Acessado: 2017-10-21. 48

Macedo(2017g) Leonardo Pereira Macedo. *Speech to Text module for Godot 2.1.3/2.1.4*. https://godotdevelopers.org/forum/discussion/18659/speech-to-text-module-for-godot-2-1-, Setembro 2017g. Acessado: 2017-10-21. 49

Macedo(2017h) Leonardo Pereira Macedo. *Speech to Text module for Godot 2.1.3/2.1.4*. https://godotengine.org/qa/18322/speech-to-text-module-for-godot-2-1-3-2-1-4, Setembro 2017h. Acessado: 2017-10-21. 49

Macedo(2017i) Leonardo Pereira Macedo. *Godot Docs - Speech to Text*. https://github.com/SamuraiSigma/godot-docs, Setembro 2017i. Acessado: 2017-10-21. 48

National Research Council(1984) National Research Council. *Automatic Speech Recognition in Severe Environments*. The National Academies Press. ix, 6

NeoSpeech(2016) NeoSpeech. *Top 5 Open Source Speech Recognition Toolkits*. http://blog.neospeech.com/top-5-open-source-speech-recognition-toolkits, 2016. Acessado: 2017-04-03. 11

Pinola(2011) Melanie Pinola. *Speech Recognition Through the Decades: How We Ended Up With Siri*. http://www.pcworld.com/article/243060/speech_recognition_through_the_decades_how_we_ended_up_with_siri.html, Novembro 2011. Acessado: 2017-09-30. 3

rrisner(2016) rrisner. http://www.ebay.com/itm/Vintage-1987-Worlds-of-Wonder-Muneca-Interactiva-Julie-mas-inteligente-hablando-/272259248680?_ul=BO, Junho 2016. Acessado: 2017-09-30. ix, 5

SCons Foundation(2017a) SCons Foundation. *SCons: A software construction tool*. https://scons.org, 2017a. Acessado: 2017-10-01. 29

SCons Foundation(2017b) SCons Foundation. *SCons Download*. http://scons.org/pages/download.html, 2017b. Acessado: 2017-10-01. 29

Simon(2017) Simon. *About Simon*. https://simon.kde.org/, 2017. Acessado: 2017-04-03. 11

SpeechAngel(2016) SpeechAngel. *The difference between speaker-dependent and speaker-independent recognition software*. https://speechangel.com/2016/05/04/difference-speaker-dependent-speaker-independent-recognition-software, Maio 2016. Acessado: 2017-09-29. 7

SuperData Research(2016) SuperData Research. *Worldwide game industry hits $91 billion in revenues in 2016, with mobile the clear leader*. https://venturebeat.com/2016/12/21/worldwide-game-industry-hits-91-billion-in-revenues-in-2016-with-mobile-the-clear-leader, 2016. Acessado: 2017-04-02. 1

Wikipedia(2017a) Wikipedia. advanced linux sound architecture. https://en.wikipedia.org/wiki/Advanced_Linux_Sound_Architecture, Julho 2017a. Acessado: 2017-10-10. 48

Wikipedia(2017b) Wikipedia. *Beam Search*. https://en.wikipedia.org/wiki/Beam_search, Julho 2017b. Acessado 2017-09-29. 4

Wikipedia(2017c) Wikipedia. *The commercialization of video games*. https://en.wikipedia.org/wiki/History_of_video_games#The_commercialization_of_video_games, 2017c. Acessado: 2017-04-03. 1

Wikipedia(2017d) Wikipedia. *Godot (game engine)*. https://en.wikipedia.org/wiki/Godot_(game_engine), Outubro 2017d. Acessado: 2017-10-22. ix, 23

Wikipedia(2017e) Wikipedia. *Moore's Law*. https://en.wikipedia.org/wiki/Moore%27s_law, 2017e. Acessado: 2017-04-03. 1

Wikipedia(2017f) Wikipedia. *PulseAudio*. https://en.wikipedia.org/wiki/PulseAudio, Setembro 2017f. Acessado: 2017-10-10. 48

Wikipedia(2017g) Wikipedia. *Speech Recognition*. https://en.wikipedia.org/wiki/Speech_recognition, Setembro 2017g. Acessado: 2017-09-29. 3

Wikipedia(2017h) Wikipedia. *Word error rate*. https://en.wikipedia.org/wiki/Word_error_rate, Setembro 2017h. Acessado: 2017-11-04. 8