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

Leonardo Pereira Macedo

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

São Paulo
3 de novembro 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
3 de novembro de 2017

Agradecimentos

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* (Juan Linietsky, Ariel 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* started appearing: *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* (Juan Linietsky, Ariel Manzur, 2017a). Since it is an open source project, it is possible to extend its funcionalities by creating new modules.

This project's goal is to implement a speech recognition module for *Godot*, then showing 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, 2017b), 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, 2017d), 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, temos 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*** (Juan Linietsky, Ariel 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 novos módulos para adicionar a ele 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. A seguir, no capítulo 3, apresenta-se uma forma de realizar reconhecimento de voz por meio do *Modelo Oculto de Markov*.

No capítulo 4, 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 5.

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

O capítulo **??** apresenta as conclusões do trabalho. Por fim, há uma parte subjetiva contendo a apreciação pessoal do TCC e uma descrição das matérias 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**), 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, 2017e).

2.2 História

Apresentamos uma breve visão histórica de sistemas de reconhecimento de voz, baseado principalmente em (Melanie Pinola, 2011), desde seu início até os dias atuais.

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.

10 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).

Figura 2.1: *Máquina Shoebox sendo operada (Cassiopedia)*

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.

Usava-se 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, 2017a).

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. Veremos HMM com mais detalhes no capítulo 3.

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 ca-

racterizadas 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 o sucesso dessa forma de busca 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: 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 chegam até a usar 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.

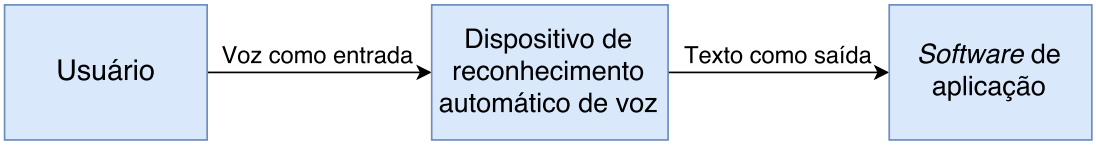

Figura 2.3: *Sistema genérico de reconhecimento automático de voz (National Research Council, 1984)*

2.4 Principais termos

De acordo com (National Research Council, 1984) e (Stephen Cook, 2002), apresentamos, a seguir, os termos mais recorrentes em sistemas de reconhecimento de voz. Também entramos em detalhes nos tipos de parâmetros que caracterizam as capacidades de um sistema de reconhecimento de voz, influenciando 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; ou *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 começar a usar o sistema. Esta variante é comumente usada 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 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

Modelo Oculto de Markov

Capítulo 4

Bibliotecas para Reconhecimento de Voz

A seguir, veremos o primeiro item necessário para atingirmos o 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 na biblioteca ideal, e estudaremos a que melhor se adequa ao nosso objetivo dentre as opções existentes.

4.1 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 e realiza ações apropriadas.

4.2 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.

4.2.1 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).

4.2.2 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 (Juan Linietsky, Ariel 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 na seção 6, *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.

4.3 Bibliotecas viáveis

Realizou-se uma pesquisa por bibliotecas de reconhecimento de voz que sigam o máximo de características possíveis propostas na seção 4.2. 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 Linux e escrita em C++, Simon utiliza *CMUSphinx*, *HTK* e *Julius* internamente. Não havia suporte para *MacOS* até abril de 2017.

Um artigo de 2014 comparou *Kaldi*, *CMUSphinx* e *HTK* em relação a precisão e tempo gasto (Gaida, 2014). *Kaldi* obteve resultados vastamente superiores; *CMUSphinx* obteve bons resultados em pouco tempo; *HTK* precisou de muito mais tempo e treino para conseguir resultados na ordem dos outros dois.

4.4 *Pocketsphinx*, a biblioteca escolhida

Capítulo 5

Pocketsphinx

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

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.

5.1 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 ela.

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

5.1.1 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 e variantes linguísticas na 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.

5.1.2 Vetor de características

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

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

5.1.3 Modelo

Um **modelo** é uma simplificação, onde reduz-se o que se quer modelar 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, conforme visto na seção 3) 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 se realiza o reconhecimento de voz.

- **Dicionário fonético**: Arquivo texto responsável por mapear palavras em fonemas, que devem existir segundo o 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.

  Um exemplo de uma linha de um dicionário em inglês seria:

```
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.

5.1.4 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.

Um exemplo de linha no arquivo de palavras-chave está a seguir: cada palavra deve vir seguida de seu limiar. Destaca-se que este valor deve vir isolado entre caracteres "/".

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

5.2 Compilação

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

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
```

5.2.1 Pacote *Sphinxbase*

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

1. Clone o repositório do *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 o *Sphinxbase* através do `Makefile`:

```
$ make
```

5.2.2 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 do *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 o *Pocketsphinx* através do `Makefile`:

```
$ make
```

5.2.3 Teste de verificação

Para verificar se a compilação feita nas subseções 5.2.1 e 5.2.2 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 5.1.

```
# 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/mdef \         # 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 5.1: *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 5.2 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 5.2: *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, algumas linhas mostradas na listagem 5.2 apresentam resultados compreensíveis:

- As linhas 4 a 6 apresentam resultados anteriores
- O número de *senones*, que são detectores curtos de sons para trifonemas

Capítulo 6

Godot

Voltaremos nossa atenção à *game engine Godot* 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.

6.1 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, 2017c).

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.

6.2 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, 2017c). Em outras palavras, *GDScript* age como uma "linguagem intermediária" entre o usuário e as interfaces em *C++* que *Godot* disponibiliza.

6.3 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 6.1. Os filhos das classes *Control*, *Node2D*, *Reference* e *Spatial* foram omitidos por serem muito numerosos.

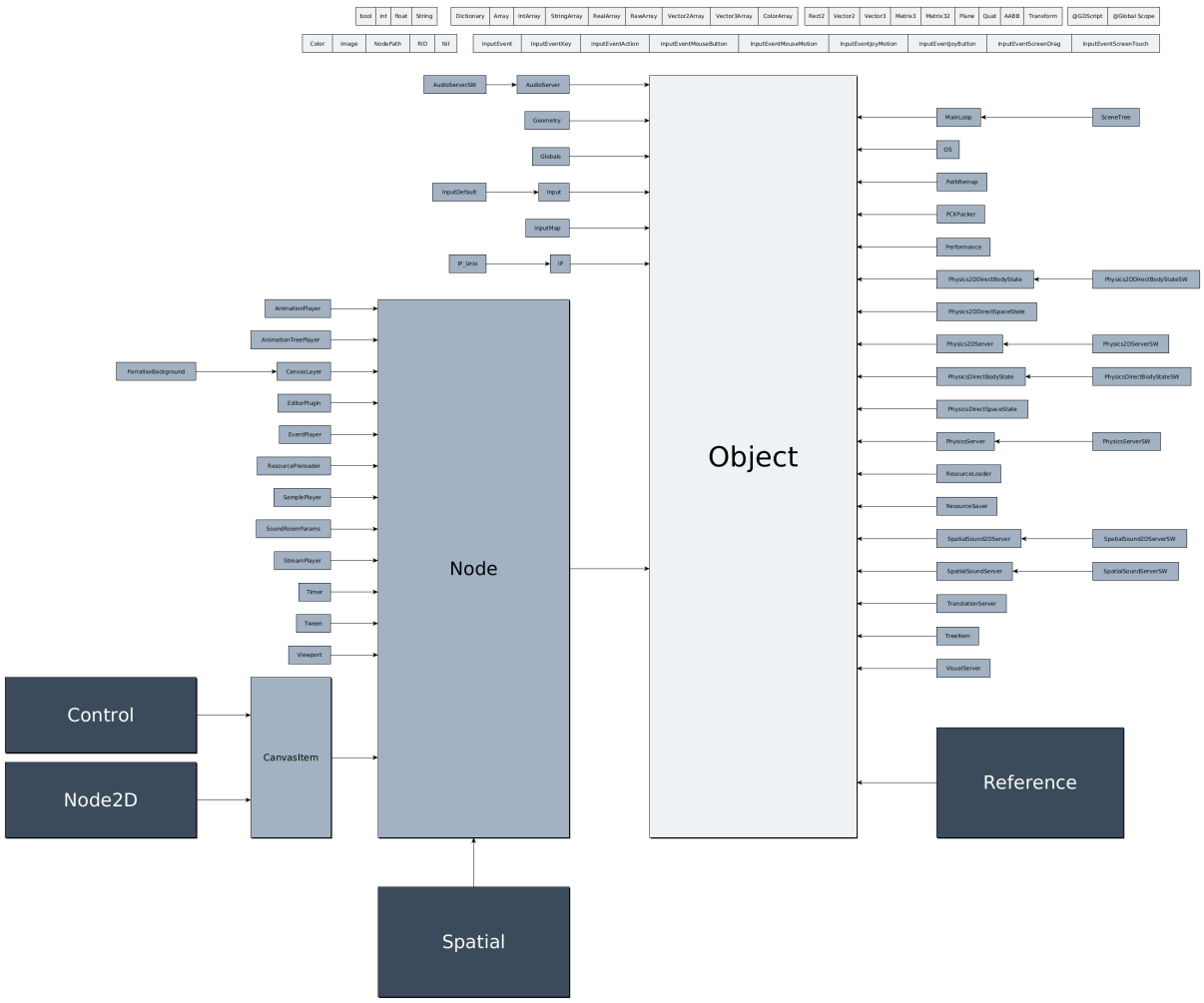

Figura 6.1: *Árvore de herança de classes em Godot (Godot Docs, 2017a)*

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

6.3.1 *Object*

Object (objeto) é a classe base para todos os tipos que não estão já embutidos na *game engine* (como os diferentes tipos de variáveis), conforme fica evidente na figura 6.1 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, transportando alguma variável, a um de seus métodos.

6.3.2 *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.

6.3.3 *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, 2017d). 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 na seção **??**, onde criaremos um jogo em *Godot*.

6.3.4 *Scene*

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

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, uma *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 da *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 6.2 exemplifica um instanciamento genérico de uma cena.

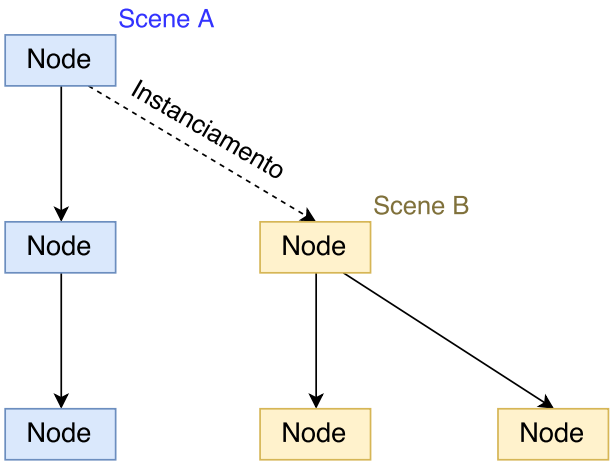

Figura 6.2: *Exemplo de instanciamento de uma scene (Godot Docs, 2017g)*

6.3.5 *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, 2017f).

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 6.3, alguns *nodes* que tipicamente poderiam usar os *resources* citados.

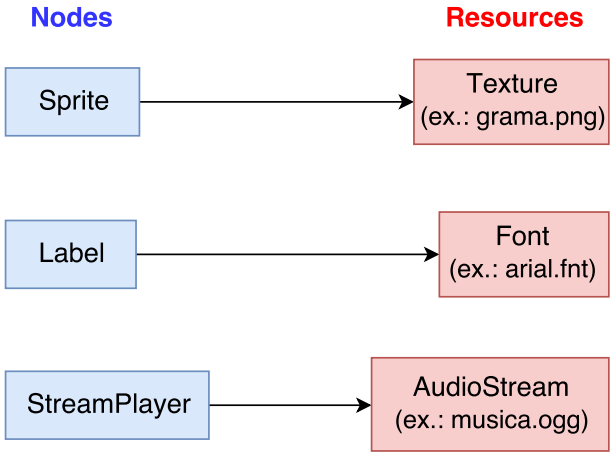

Figura 6.3: *Alguns resources e nodes que tipicamente os usam (Godot Docs, 2017f)*

6.3.6 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 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 criair 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`.

6.4 *SCons*

Godot pode ser compilado para diversas plataformas; entre elas, citamos Windows, MacOS, Unix, Android, iOS e HTML5. A necessidade de atender a uma variedade tão grande de sistemas operacionais fez com que o projeto escolhesse ***SCcons*** para simplificar sua compilação (Godot Docs, 2017h). 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*.

6.4.1 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).

6.4.2 Uso em *Godot*

Para usar o 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, que usaremos na seção 6.5 e no capítulo **??**. Veja a tabela 6.1 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 6.1: *Argumentos por linha de comando do SCons para Godot*

Algumas explicações devem ser dadas sobre a tabela 6.1. 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 criados 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 símbolos de depuração em *C++*, otimização e verificação de erros em tempo de execução (*runtime*). Sintetizamos as características de cada valor na tabela 6.2.

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

Tabela 6.2: *Valores possíveis do parâmetro `target` e seu significado*

6.5 Compilação

O código fonte de *Godot* está disponível no *GitHub* (GitHub, 2017). A criação de um módulo, conforme será visto no capítulo 7, 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 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, 2017e) 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
   ```

6.5.1 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 **??**, quando criarmos um jogo para demonstrar o módulo de reconhecimento de voz.

Capítulo 7

Módulo *Speech to Text* para *Godot*

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

Este capítulo documenta as instruções necessárias para adicionar novas funcionalidades a *Godot*, bem como as decisões de projeto tomadas 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 6.5.

7.1 Módulos em *Godot*

Conforme descrito na seção 6.2, 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 em *Godot*. Os módulos servem justamente para este objetivo, pois não fazem parte do código essencial da *game engine*.

Todos os módulos ativos são encontrados como subdiretórios dentro da pasta `modules/` no código fonte. A seguir, forneceremos as instruções necessárias para a criação de um módulo genérico.

7.1.1 Instruções para criação

Como exemplo, mostraremos como criar o módulo *sumator* no decorrer das instruções.

1. Crie um diretório, dentro da pasta `modules/` no código fonte, com o nome do módulo. Neste caso, seria o diretório `sumator/`.

```
$ cd modules
$ mkdir sumator
```

2. Dentro do diretório criado para o módulo, escreva o código C++ para quaisquer interfaces (arquivos `.h`) a serem usadas. Como nosso exemplo *Sumator* é bastante simples, criaremos apenas um arquivo: `sumator.h`. Seu conteúdo é apresentado na listagem 7.1.

```
1  #ifndef SUMATOR_H
2  #define SUMATOR_H
3
4  #include "node.h"
5
6  class Sumator : public Node {
7      OBJ_TYPE(Sumator, Node);
8      int count;
9
10 protected:
11     static void _bind_methods();
12
13 public:
14     void add(int value);
15     void reset();
16     int get_total() const;
17
18     Sumator();
19 };
20
21 #endif
```

Listagem 7.1: *Arquivo de interface* `sumator.h` *para o módulo Sumator*

Na linha 6 da listagem 7.1, fizemos a classe *Sumator* herdar de *Node* (visto na seção 6.3.3) para que o módulo possa ser usado de forma direta no editor *Godot*.

Toda classe que herda de alguma classe essencial da *game engine* exige uma chamada a `OBJ_TYPE()` (linha 7), que recebe como argumentos o nome da classe e de quem ela herda.

Por fim, vamos nos atentar a `_bind_methods()`, definido na linha 11: este método define quais métodos da classe poderão ser usados em *GDScript*. Seu nome é uma convenção, definido por *Godot* para facilmente localizar o método em uma classe.

3. Escreva o código C++ para quaisquer implementações (arquivos *.cpp*) necessárias. O exemplo do *Sumator* exige um arquivo `sumator.cpp` para implementar os métodos definidos no passo anterior. Apresentamos seu conteúdo na listagem 7.2.

```
1  #include "sumator.h"
2
3  void Sumator::add(int value) {
4      count += value;
5  }
6
7  void Sumator::reset() {
8      count = 0;
9  }
10
11 int Sumator::get_total() const {
12     return count;
13 }
14
15 void Sumator::_bind_methods() {
16     ObjectTypeDB::bind_method("add",       &Sumator::add);
17     ObjectTypeDB::bind_method("reset",     &Sumator::reset);
18     ObjectTypeDB::bind_method("get_total", &Sumator::get_total);
19 }
20
21 Sumator::Sumator() {
22     count=0;
23 }
```

Listagem 7.2: *Arquivo de implementação* `sumator.cpp` *para o módulo Sumator*

A implementação de `_bind_methods()` tipicamente envolve chamadas ao *singleton ObjectTypeDB*, que contém um banco de dados para todas as classes e funcionalidades utilizáveis em *GDScript*.

Nas linhas 16 a 18 da listagem 7.2, usamos `bind_method()` para amarrar nomes (1º argumento) à referência de métodos (2º argumento), permitindo seu uso em *GDScript*.

7.2 Organização de *Speech to Text*

7.3 Divulgação

Referências Bibliográficas

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

CMUSphinx(2015) CMUSphinx. *About the CMUSphinx*. http://cmusphinx.sourceforge.net/wiki/about, Fevereiro 2015. Acessado: 2017-04-03. 13

CMUSphinx(2016) CMUSphinx. Building application with pocketsphinx. http://cmusphinx.sourceforge.net/wiki/tutorialpocketsphinx, 2016. Acessado: 2017-04-17. 17

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(2014) Christian Gaida. *Comparing Open-Source Speech Recognition Toolkits*. http://suendermann.com/su/pdf/oasis2014.pdf, 2014. Acessado: 2017-04-03. 13

GitHub(2017) GitHub. *Repositório Godot*. https://github.com/godotengine/godot, 2017. Acessado: 2017-10-01. 28

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. xi, 23

Godot Docs(2017b) Godot Docs. *File system*. http://docs.godotengine.org/en/stable/learning/step_by_step/filesystem.html, 2017b. Acessado: 2017-10-03. 26

Godot Docs(2017c) Godot Docs. *GDScript*. http://docs.godotengine.org/en/stable/learning/step_by_step/scripting.html#gdscript, 2017c. Acessado: 2017-10-01. 22

Godot Docs(2017d) Godot Docs. *Nodes*. http://docs.godotengine.org/en/stable/learning/step_by_step/scenes_and_nodes.html#nodes, 2017d. Acessado: 2017-10-01. 24

Godot Docs(2017e) Godot Docs. *Distro-specific oneliners*. http://docs.godotengine.org/en/stable/development/compiling/compiling_for_x11.html#distro-specific-oneliners, 2017e. Acessado: 2017-10-01. 29

Godot Docs(2017f) Godot Docs. *Resources*. http://docs.godotengine.org/en/stable/learning/step_by_step/resources.html, 2017f. Acessado: 2017-10-01. xi, 25, 26

Godot Docs(2017g) Godot Docs. *Scenes*. http://docs.godotengine.org/en/stable/learning/step_by_step/scenes_and_nodes.html#scenes, 2017g. Acessado: 2017-10-01. xi, 24, 25

Godot Docs(2017h) Godot Docs. *Introduction to the buildsystem*. http://docs.godotengine.org/en/stable/development/compiling/introduction_to_the_buildsystem.html, 2017h. Acessado: 2017-10-01. 27

HTK(2016) HTK. *What is HTK?* htk.eng.cam.ac.uk, 2016. Acessado: 2017-04-03. 13

IBM() IBM. *IBM Shoebox*. https://www-03.ibm.com/ibm/history/exhibits/specialprod1/specialprod1_7.html. Acessado: 2017-09-29. 4

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

Juan Linietsky, Ariel Manzur(2017b) Juan Linietsky, Ariel Manzur. *Godot Features*. https://godotengine.org/features#multiplatform-deploy, 2017b. Acessado: 2017-09-20. 12

Kaldi(2017) Kaldi. *About the Kaldi project*. http://kaldi-asr.org/doc/about.html, Março 2017. Acessado: 2017-04-03. 13

Melanie 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

National Research Council(1984) National Research Council. Automatic Speech Recognition in Severe Environments. The National Academies Press. xi, 6, 7

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. 13

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. xi, 5

SCons Foundation(2017a) SCons Foundation. *SCons: A software construction tool*. https://scons.org, 2017a. Acessado: 2017-10-01. 27

SCons Foundation(2017b) SCons Foundation. *SCons Download*. http://scons.org/pages/download.html, 2017b. Acessado: 2017-10-01. 27

Simon(2017) Simon. *About Simon*. https://simon.kde.org/, 2017. Acessado: 2017-04-03. 13

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

Stephen Cook(2002) Stephen Cook. Speech recognition howto. http://www.tldp.org/HOWTO/Speech-Recognition-HOWTO/introduction.html, Abril 2002. Acessado: 2017-09-30. 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. *Beam Search*. https://en.wikipedia.org/wiki/Beam_search, Julho 2017a. Acessado 2017-09-29. 4

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

Wikipedia(2017c) Wikipedia. *Godot (game engine)*. https://en.wikipedia.org/wiki/Godot_(game_engine), Outubro 2017c. Acessado: 2017-10-22. 21

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

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