Instituto de Matemática e Estatística
Universidade de São Paulo

OpenDA : Open Dynamic Audio

Middleware para manipulação de trilhas sonoras
dinâmicas em jogos digitais

Lucas Dário
Orientador: Prof. Dr. Marcelo Queiroz
Co-orientador: Wilson Kazuo Mizutani

São Paulo, Janeiro de 2016

Conteúdo

Capítulo 1

Introdução

Jogos digitais são programas de entretenimento. Uma de suas características é o fato de os eventos que ocorrem em um jogo serem diretamente influenciados pelos comandos que o usuário envia ao programa. Desse modo, a experiência do usuário é construída através de sua interação com o sistema. Ao longo da história, os jogos foram evoluindo e, junto com eles, cresceram as buscas por maneiras de alimentar essa experiência. Os principais recursos utilizados para aumentar a imersão do usuário nos jogos são os recursos visuais e, igualmente importantes, os recursos auditivos. É através de imagens e sons que o jogo consegue transmitir ao usuário quais são as consequências que suas ações produzem no ambiente virtual.

No âmbito do som, podemos dizer que as trilhas sonoras desempenham grande parte desse papel imersivo, dado que alterações sonoras podem sugerir diversos acontecimentos dentro de um jogo, como por exemplo alterações no estado emocional de um personagem, mudanças de ambientes, acentuar a ação de uma cena, entre outros exemplos. O objetivo do trabalho, portanto, é estudar e desenvolver soluções computacionais que facilitem a representação e reprodução de trilhas sonoras que exercem esse tipo de comportamento.

Nas próximas seções serão apresentados a motivação por trás do trabalho, os objetivos almejados, a metodologia adotada e a organização do texto.

1.1 Motivação

A produção de jogos, por ser um processo fortemente interdisciplinar, costuma envolver pessoas de diversas áreas. São necessários: projetistas de jogo, que criarão os aspectos gerais do jogo, como a decisão de ser multijogador ou não, as regras do jogo, os desafios a serem enfrentados pelo jogador etc; compositores e projetistas de áudio, que irão produzir e descrever o comportamento da trilha sonora durante

todo o jogo; roteiristas, que escreverão todo o enredo e história do jogo; artistas gráficos, que serão responsáveis por definir a estética do jogo; programadores, que implementarão todas essas descrições do jogo através do código, transformando-o em um software utilizável pelos clientes; entre outros.

No processo de produção de um jogo, primeiramente ocorre uma fase criativa, onde a equipe levantará as principais características que o jogo possuirá, funcionalidades e diferenciais que o colocarão acima dos outros títulos da indústria. Todas essas características são organizadas e escritas em um documento, chamado de Documento de Projeto de Jogo (do inglês "Game Design Document"). Após essa fase inicial, então, é que começa o desenvolvimento do jogo.

A partir desse momento, o trabalho começa a ser separado. Os compositores e *sound designers*, que são encarregados da parte sonora do jogo, ficam responsáveis por produzir, geralmente no formato de vários arquivos de áudio, toda a parte audível que deve estar presente no jogo. Mais ainda, eles devem explicar aos programadores quais são os momentos em que esses sons devem ser tocados. Para áudio não dinâmico, essa explicação pode ser feita inclusive antes da composição dos sons, porém quando o áudio é dinâmico, o comportamento do áudio só é definido quando os músicos trabalham nele. Nesse caso, a tecnologia usada é que determina o quão complexa será a intervenção necessária pelos programadores no código fonte, para que o áudio seja reproduzido da maneira desejada. A ausência de um *middleware* implica que o trabalho de compreender as demandas dos compositores e, assim, poder implementar caso a caso o controle sobre a execução da mídia sonora no jogo recaia todo sobre os ombros dos programadores. Essas dificuldades acabam comprometendo o trabalho de todos os profissionais envolvidos.

A dependência de uma implementação direta no jogo acaba sendo ineficiente, pois se olharmos o intervalo de tempo entre a submissão de áudio do compositor ao programador e a análise do resultado proporcionado por esse áudio, vemos que muito tempo é perdido. Ou seja, o compositor só verifica se sua composição ficou boa no jogo após os programadores terminarem de implementar todos os casos onde esse som aparece. E quanto mais complexo for o áudio, mais casos os programadores devem cobrir e, assim, mais tempo o compositor fica esperando, como por exemplo no cenário em que deseja-se fazer um jogo cujo áudio é reativo a mudanças no ambiente de jogabilidade e a ações do jogador, o que mais adiante no trabalho será apresentado como um áudio dinâmico.

Além disso, o problema pode se agravar caso a tecnologia utilizada não forneça manutenibilidade suficiente ao áudio, pois cada nova alteração a ser feita pelo programador poderá demandar mais trabalho. Um modo de contornar diversas dessas

limitações é a utilização de vários arquivos de áudio que cubram todas as variações necessárias. Essa técnica, porém, se mostra pouco eficiente, já que exige um consumo muito maior de espaço virtual, o que é uma limitação considerável em plataformas móveis (como smartphones e tablets).

Desse modo, esses problemas demandam a existência de alguma ferramenta que permita alguma representação de uma trilha sonora, tal que esta possa assumir caráter dinâmico, ser facilmente manipulável pelo projetista de som e, ao mesmo tempo, facilmente reprodutível dentro de um jogo.

1.2 Objetivos

Este trabalho, portanto, tem como objetivo a implementação de um *middleware* capaz de permitir a definição, manipulação e reprodução de trilhas sonoras dinâmicas e reativas a parâmetros, de modo a facilitar o trabalho tanto dos projetistas de áudio quanto dos programadores durante o processo de produção de um jogo digital.

Para isso, o sistema implementado foi dividido em 2 grandes blocos:

1. um Motor de áudio, disponibilizado aos programadores através de uma API, e que é responsável pelo processamento e reprodução do áudio dinâmico;

2. uma Interface, oferecida aos projetistas de som, que proporciona recursos para que esses possam produzir e representar áudio dinâmico.

1.3 Metodologia

O projeto foi feito em conjunto com Wilson Kazuo Mizutani, aluno de mestrado em Ciência da Computação do Instituto de Matemática e Estatística da USP. Desde o início até o fim do projeto, foram realizadas reuniões semanais entre seus integrantes, a fim de, primeiramente, decidir toda a estrutura inicial do sistema. Após as decisões iniciais, as reuniões foram feitas para coordenar o trabalho entre os integrantes e promover discussões sobre as funcionalidades a serem implementadas, o andamento do projeto, tecnologias a serem utilizadas, problemas de implementação etc.

Para a execução do projeto, primeiramente foi realizado um levantamento sobre os principais problemas relativos a áudio no processo de produção de um jogo. Em seguida, deu-se início à implementação do sistema, visando a resolução dos problemas encontrados e, por fim, a validação do projeto foi realizada através de aplicações demonstrativas que fizessem uso do sistema proposto.

1.4 Organização do Texto

No capítulo 2, serão apresentados alguns conceitos importantes e que serão utilizados no decorrer do trabalho. No capítulo 3 serão introduzidas as tecnologias utilizadas no projeto. No capítulo 4 encontra-se uma descrição mais detalhada das funcionalidades do *middleware* e sua implementação. No capítulo 5 serão expostos alguns resultados obtidos ao utilizar a ferramenta criada na produção de jogos.

Capítulo 2

Conceitos

Neste capítulo, serão introduzidos alguns conceitos básicos que serão utilizados durante o decorrer do trabalho e que são essenciais para se entender o objetivo do projeto. São eles: trilha sonora, jogo digital e áudio dinâmico.

2.1 Trilha Sonora

Uma trilha sonora é o conjunto de elementos sonoros gravados ou sintetizados que se manifestam como fatores fundamentais na criação de uma narrativa audiovisual e que são, tradicionalmente, segmentos de áudio de reprodução linear, ou seja, sem interrupções, sequencial. A obra da qual essa trilha sonora faz parte pode ser um filme, uma peça teatral, um programa de TV, um jogo digital, entre outras. Como elementos sonoros, temos músicas, os sons de efeitos especiais e falas de personagens. No contexto deste trabalho, serão considerados como parte da trilha sonora de um jogo digital basicamente todo e qualquer elemento sonoro que possa ser reproduzido durante a execução desse jogo.

As trilhas sonoras têm um papel relevante, seja qual for o contexto do qual ela faça parte. O áudio dessas trilhas é responsável por tornar uma narrativa muito mais densa e rica, ao adicionar a ela elementos que não são possíveis de se transmitir apenas por imagens. No contexto de um jogo digital, por exemplo, elas desempenham grande parte da função de inserir o jogador dentro do ambiente proposto pela narrativa do jogo, seja através da música de fundo (que permite aos jogadores, por exemplo, a percepção sobre o local físico onde seu personagem está situado), através das falas (proporcionando os diálogos e conferindo personalidade às personagens do jogo) ou de efeitos sonoros (caracterizando os eventos que acontecem no jogo, como por exemplo os disparos em um jogo de guerra).

2.2 Jogo Digital

Segundo Battaiola[1], um jogo de computador pode ser definido como um sistema composto por três partes básicas: Enredo, Motor[1] e Interface Interativa. O enredo define o tema, a trama, os objetivos do jogo (a serem atingidos pelo usuário ou jogador). O motor é o sistema de controle, o mecanismo que administra a reação do jogo em função de uma ação do usuário. A interface interativa fica responsável por controlar a comunicação entre o motor e o usuário, apresentando graficamente um novo estado do jogo.

Dada essa definição e, encarando-a como uma definição geral de jogo digital ao considerá-la válida também para jogos de plataformas que não o computador, como consoles e dispositivos móveis, pode-se concluir que jogos digitais são extremamente interativos, onde a experiência final do usuário é influenciada principalmente pelas decisões que ele toma durante a execução do jogo. Para tanto, um jogo deve ser capaz de refletir as ações do jogador no estado atual do jogo e apresentar esse estado ao jogador, o que pode ser alcançado através de recursos tanto visuais quanto auditivos, sendo esse último um elemento de extrema importância para esse projeto. Outro detalhe importante e aproveitado nesse projeto é o fato de um estado do jogo ser nada mais do que um apanhado de parâmetros, os quais seriam interessantes se pudessem ser aproveitados na definição do que será reproduzido como elemento visual ou sonoro.

Motivado pelo caráter interativo e dinâmico dos jogos, pode-se verificar que boa parte das implementações desses jogos seguem decisões recorrentes de projeto. A grande maioria dos jogos utiliza-se de um padrão arquitetural chamado de *"Main Loop"*, onde toda a lógica do jogo fica inserida dentro de um grande laço, que é executado uma vez a cada instante de tempo, o qual deve ser suficientemente pequeno, de modo que dê ao jogador a sensação de que o jogo é executado em tempo real. Durante a execução desse laço, todos os comandos emitidos pelo jogador são capturados, processados, as alterações causadas por esses comandos são inseridas no estado do jogo e, ao final do laço, esse estado é apresentado ao usuário através dos recursos visuais e sonoros. Após essa apresentação, o laço recomeça. Esse detalhe de implementação, por ser extremamente presente no desenvolvimento de jogos, foi levado fortemente em consideração durante o planejamento da ferramenta produzida no projeto.

[1] Motores de Jogo são ferramentas utilizadas na produção de jogos digitais e que são responsáveis por simplificar e abstrair elementos comuns no desenvolvimento de vários jogos, como simulações físicas e a adição de recursos gráficos

2.3 Áudio Dinâmico

Pela definição dada por Collins[2], áudio dinâmico é o "áudio que reage a mudanças no ambiente de jogabilidade ou em resposta ao usuário". Desse modo, o áudio abandona uma característica linear e sequencial de execução (como os segmentos de áudio de reprodução linear presentes nas trilhas sonoras comuns) e começa a adquirir um aspecto mais interativo ou adaptativo.

Segundo o trabalho apresentado por Meneguette[3], há diversos níveis de dinamicidade. Existem, por exemplo, alguns jogos de baixa dinamicidade como *Sonic the Hedgehog*(Sega 1991), no qual a música de fundo consiste apenas de um *loop* que fica tocando repetidas vezes a mesma música durante a fase, simplesmente para permitir que o jogador diferencie uma fase da outra. Há, porém, outros jogos como *Dead Space*(Electronic Arts 2008) que sincronizam tanto o áudio quanto os recursos visuais com as ações narrativas, conferindo ao áudio um alto grau de dinamicidade.

Além disso, Meneguette e Collins também apresentam a separação do áudio dinâmico em dois tipos: Áudio Interativo, que é caracterizado pelo áudio que responde aos comandos emitidos pelo jogador, como o som de alguma ação que é efetuada pelo personagem após o jogador pressionar um botão (o barulho de um disparo de uma arma de fogo em um jogo de guerra seria um exemplo); e Áudio Adaptativo, que é o áudio que reage a parâmetros que estão fora do alcance direto do usuário, como em *Slender: The Eight Pages*(Parsec Productions 2012), em que a trilha sonora se intensifica conforme aumenta o número de páginas coletadas. Meneguette ainda fala sobre um terceiro tipo, o Áudio Gerativo/Procedural, que não é um áudio pré-concebido nem reproduzido, mas sim gerado durante a execução do jogo através de algoritmos, e que possua certo grau de indeterminação.

Capítulo 3

Tecnologias

Antes de iniciar a implementação do projeto, foi necessário definir as tecnologias que seriam utilizadas. Um dos critérios levados em consideração foi o fato de que, para que o design de *"Main Loop"* seja implementado, o código deve ser eficiente o suficiente para que o programa pareça ser executado em tempo real. Tendo isso em mente, as tecnologias foram escolhidas levando em consideração simplicidade e eficiência. Nesse capítulo serão apresentadas essas tecnologias e uma breve descrição delas.

3.1 Linguagem C++

A linguagem C++, como definida por Bjarne Stroustrup, em seu livro "The C++ Programming Language"[4], é uma linguagem de programação de propósito geral, com ênfase na concepção e utilização de abstrações leves e ricas em tipos. Stroustrup também diz que a linguagem é particularmente apropriada para aplicações com recursos limitados, portanto ela se torna ideal quando se quer desenvolver uma aplicação que utilize o menor número de recursos possível, priorizando sua eficiência tanto em consumo de espaço quanto em tempo de processamento.

Outra característica que motivou o uso dessa linguagem é o fato de a grande maioria dos jogos desenvolvidos hoje em dia utilizar a linguagem C++ em sua implementação, assim como os Motores de Jogo, que também fazem uso dessa linguagem devido a sua alta eficiência e flexibilidade.[5]

Durante o projeto, a linguagem C++ foi utilizada para a implementação de todo o código do Motor de Áudio.

3.2 Pure Data e `libpd`

Como o projeto envolve áudio e sua manipulação, era necessária a utilização de alguma linguagem que trabalhasse com processamento digital de sinal em tempo real.

Para isso, foi escolhido o Pure Data[6], que é uma linguagem de programação visual. Ela permite análise, modificação e geração de sinais através de algoritmos e funções presentes na linguagem. Além disso, por ser uma linguagem de programação visual, ela tem sua utilização simplificada, permitindo que compositores e projetistas de som façam uso dessa linguagem sem precisarem escrever linhas de código.

A linguagem Pure Data é uma linguagem orientada a fluxos de dados, onde os programas na linguagem são desenvolvidos graficamente em estruturas chamadas de *patches*. Nesses *patches*, cada algoritmo, função ou estrutura de dados são representados por objetos, inseridos manualmente na tela, chamada de *canvas*. Os objetos são ligados por cordas, e é por elas que os dados passam de um objeto a outro. Cada um desses objetos realizará uma tarefa específica, seja uma tarefa simples como uma operação matemática básica até tarefas mais complexas, como uma Transformada Rápida de Fourier (FFT).

Desse modo, foram estudadas alternativas para que a interface do Pure Data fosse aproveitada no projeto, de modo que pudesse ser utilizada para a criação de arquivos que oferecessem uma representação de áudio dinâmico.

Uma vez possível essa representação de áudio dinâmico dentro de um *patch* de Pure Data, o *middleware* proposto deveria ser capaz de lizá-lo. Mais especificamente, o Motor de Áudio do *middleware* deveria ser capaz de ler e processar as informações contidas no arquivo de Pure Data. Para isso, foi utilizada a biblioteca `libpd`[7] dentro do motor de áudio. A `libpd` é uma biblioteca que contém o núcleo da linguagem Pure Data, dispensando a interface gráfica, e que conta com um conjunto de funções que a torna incorporável em múltiplas linguagens de programação. Desse modo, como o motor é feito em C++, basta utilizar as chamadas de funções em C++ oferecidas pela `libpd` que permitem acionar todas as funcionalidades do Pure Data, como ativar o processamento digital de sinais, abrir um patch e processá-lo, entre outras. Outra função que a `libpd` oferece é a de recebimento e envio de mensagens para os patches que foram abertos. Essa funcionalidade terá um papel importantíssimo no modo como proporcionaremos a representação de áudio dinâmico.

Outras motivações que levaram à escolha do Pure Data e da `libpd` são o fato da linguagem e a biblioteca serem Open Source, assim como a ferramenta proposta, e a linguagem Pure Data ser bem conhecida dentro da comunidade de processamento de sinais, ou seja, não exigiria que o público alvo do *middleware* (programadores,

compositores e projetistas de áudio) utilizasse uma linguagem desconhecida ou com uma comunidade muito restrita.

3.3 Biblioteca OpenAL e OpenAL Soft

Tendo como objetivo a utilização de elementos sonoros dentro do jogo, após todo o processamento e manipulação do áudio, é necessário reproduzí-lo através do *hardware* de áudio do computador. Para isso, optou-se pela utilização da biblioteca OpenAL[8].

A OpenAL é uma especificação de biblioteca de áudio desenvolvida originalmente pela empresa Loki Software[9] em 2000 com o objetivo de ajudar a empresa a portar jogos do sistema Windows para Linux. Atualmente, a especificação está sob responsabilidade da empresa Creative Technology Ltd.[10] A implementação da OpenAL que foi utilizada durante o projeto é a OpenAL Soft[11], que é uma implementação de código aberto, multi-plataforma e atualizada constantemente.

A OpenAL é projetada para ser uma interface para o *hardware* de áudio, planejada para ser usada principalmente no desenvolvimento de jogos, e capaz de renderizar áudio multi-canal em ambientes tridimensionais, simulando situações que contenham fontes de som ao redor de um único ouvinte.

Essa biblioteca funciona utilizando 3 objetos principais: `Source` (ou fontes de som), um único `Listener` (o ouvinte) e diversas instâncias de `Buffer` (estrutura de dados que irá armazenar os sons que deverão ser reproduzidos). Um objeto `Source` guarda informações sobre sua posição, velocidade, direção do som que será emitido por ele, intensidade desse som e um ponteiro para o `Buffer` que conterá o som que ele irá reproduzir. Um objeto `Listener` contém informações sobre sua posição, velocidade e direção para a qual está virado, além de um ganho geral aplicado ao som recebido. Já o `Buffer` armazena áudio no formato de Modulação por Código de Pulso[2].

Ao utilizar a OpenAL, o programador tem a possibilidade de manipular todos esses valores, tornando possível posicionar as fontes de áudio e o ouvinte dentro de um ambiente 3D. O programador deve, então, preencher os *buffers* com os dados sobre o áudio que deverá ser reproduzido. Após isso ele escolhe uma fonte de som, adiciona esse *buffer* à fila de *buffers* a serem reproduzidos pela fonte e manda ela executar, o que fará com que a fonte de som comece a reproduzir o som contido

[2]Modulação por Código de Pulso (MCP) é um método usado para representar digitalmente amostras de sinais analógicos. Em um fluxo de MCP a magnitude do sinal analógico é amostrada regularmente em intervalos uniformes, com cada amostra quantizada para o valor mais próximo dentro de um intervalo de passos digitais

nos *buffers* presentes em sua fila. A biblioteca, então, simulará os efeitos sonoros causados pelo fenômeno de atenuação por distância, devido à diferença de posição entre as fontes e o ouvinte, e somará as componentes de áudio incidentes no ouvinte provindas de todas as fontes. O resultado final é, por fim, reproduzido através do *hardware* de áudio da máquina que está executando o programa.

Capítulo 4

Desenvolvimento

Durante o desenvolvimento do projeto, inicialmente foi realizado um levantamento de informações através de entrevistas feitas com projetistas de som profissionais e experimentos utilizando algumas ferramentas. Após isso, elaborou-se uma proposta para o projeto, com escolha das tecnologias a serem utilizadas e funcionalidades a serem implementadas.

Neste capítulo serão explicadas as etapas acima e, em seguida, serão descritos detalhes sobre a organização dos integrantes do projeto e implementação do *middleware*.

4.1 Levantamento de informações

Com a finalidade de conhecer melhor os problemas enfrentados por projetistas de som e compositores, foram realizadas entrevistas com profissionais com experiência na elaboração de áudio para jogos digitais. Das quatro entrevistas realizadas, duas ocorreram no 1º semestre de 2015, enquanto que as outras duas se passaram no 2º semestre do mesmo ano. Quanto ao foco delas, as duas primeiras foram realizadas com o objetivo de entender como era o trabalho dos músicos em equipes de desenvolvimento de jogos, enquanto que as duas últimas almejavam descobrir requisitos para o projeto em desenvolvimento.

Os entrevistados foram Rodolfo Santana, em 13/04/2015, Sound Designer da empresa de jogos mobile Tapps Games[12], José Guilherme Allen Lima, 16/4/2015, Pesquisador do NuSom[13], Kaue Lemos, 27/8/2015, Diretor de Áudio na 7Sounds - Game Audio Solutions[14] e Dino Vicente, 21/10/2015, compositor de Trilhas Sonoras na DVMúsica[15].

Durante a primeira entrevista, Rodolfo falou sobre como é seu trabalho dentro da Tapps, onde possui a seu dispor um estúdio no qual pode compor, gravar e remixar

seus áudios. Ele disse que costuma basear seu trabalho principalmente nas *concept arts*[3] que os artistas fazem no começo do projeto, desenvolvendo a trilha sonora ao mesmo tempo em que os programadores desenvolvem o jogo em si. Durante o processo, quando Rodolfo precisa que a trilha assuma algum comportamento diferente em tempo real, recorre a duas possibilidades que irão depender do efeito que ele quer obter. Um dos exemplos que ele deu foi o som do passos. Para evitar que o som fique repetitivo, ele pode utilizar uma opção chamada de *pitch stretching*, disponível no framework de desenvolvimento que a empresa usa. Para isso ele recorre aos programadores, para que esses alterem a altura do efeito sonoro de maneira aleatória a cada vez que esse efeito for reproduzido. Outro exemplo dado por ele é quando o jogo tem uma música de fundo que precisa refletir o ambiente virtual do jogo, como quando as únicas frequências a serem ouvidas são as frequências mais baixas, caso que ocorre quando o som escutado vem de trás de uma parede. O problema, nesse último caso, é que o efeito não está disponível dentro do framework de desenvolvimento, desse modo, é necessário que ele grave diferentes versões da música e depois entregue múltiplos arquivos de áudio aos programadores. No entanto, como ele possui apenas uma quota de no máximo 1 megabyte para entregar em arquivos de áudio, isso se torna uma complicação adicional. Vemos, portanto, que em ambos os casos apresentados Rodolfo possui alguma limitação: em um deles fica dependente dos programadores, enquanto que no outro seu trabalho é restrito ao espaço de memória disponível.

Na segunda entrevista, José relatou sobre a época em que foi contratado para fazer a trilha sonora de alguns projetos de jogo. Diferentemente de Rodolfo, que era um projetista de som da própria empresa, José foi contratado como terceiro. Em seus trabalhos, a equipe de desenvolvimento apresentava a ele algumas especificações de como queriam as músicas e efeitos sonoros, e José, então, as compunha e entregava. Nos casos em que o áudio não ficava do jeito esperado ou não combinava com os elementos do jogo, ele tinha de refazer o áudio, inserindo as mudanças necessárias. O teste da trilha sonora dentro do jogo só podia ser realizado após todos os arquivos de áudio serem entregues aos desenvolvedores, que então utilizavam esses arquivos dentro do jogo e verificavam o resultado das mudanças. Pode-se perceber, nesse caso, todo o tempo desperdiçado em que o compositor fica esperando até que os desenvolvedores insiram os arquivos de áudio no jogo até que seja possível verificar o resultado das composições dentro do jogo.

[3]Concept Art trata de uma forma de ilustração que é usada para representar a idéia geral de algum elemento visual a ser inserido em um projeto antes que sua forma final esteja pronta. Serve, portanto, como um esboço do modelo final que será apresentado quando o projeto que o possui estiver pronto. É uma técnica muito usada em filmes, jogos, animações e histórias em quadrinhos.

Durante a segunda metade das entrevistas, focadas nas funcionalidades interessantes para a ferramenta, a entrevista com Kaue revelou que ele é um usuário avançado de *middlewares* de áudio dinâmico, em especial o FMOD Studio[16], sua preferida. Ele mostrou, durante a entrevista, seu fluxo de trabalho utilizando a ferramenta e apontou as principais funcionalidades que, na opinião dele, um *middleware* assim deve ter. Algumas funcionalidades interessantes citadas foram:

- Controle do volume dos áudios de maneira fácil
- Randomização ou controle da Altura (do inglês Pitch) dos sons
- Mudar facilmente um evento de 3D para 2D
- Permitir reprodução não-linear da música (utilizando saltos condicionais)
- Poder repetir o mesmo som (looping) diversas vezes
- Organizar os eventos por prioridade
- Possibilidade de profiling[4]

Como Kaue utiliza um *middleware* em seu trabalho, ele possui controle total sobre a trilha sonora dos jogos e, desse modo, os contatos com a equipe de programação se reduzem apenas a acertar os nomes de alguns gatilhos e parâmetros-chave que o jogo deve fornecer através da API do FMOD. Kaue, durante a entrevista, defendeu o uso desse tipo de *middleware* durante a produção de jogos digitais.

A última entrevista foi com Dino, que possui vasta experiência na composição de trilhas sonoras. Ele possui um estúdio repleto de instrumentos e aparelhos que lhe possibilitam as mais diversas criações sonoras. Seus trabalhos com trilhas sonoras de jogos, porém, são relativamente poucos, mas faziam uso da linguagem MAX[17] por exemplo. Um dos trabalhos era composto de uma apresentação ao vivo no qual Dino utilizou um boxeador para "gamificar" a interatividade sonora. Outro exemplo foi uma exposição interativa que apresentava um jogo para cada um dos cincos sentidos, cada um com sua própria trilha sonora. Quanto à opinião de Dino sobre a utilização de um *middleware*, ele disse não concordar com a autonomia concedida ao compositor e projetista de áudio pela ferramenta, argumentando que a utilização da ferramenta caracterizaria, na verdade, mais uma sobrecarga de sua profissão. Ele

[4]Em engenharia de software, Profiling é uma forma de medir o desempenho de um programa através, por exemplo, da complexidade de tempo ou de espaço, da frequência ou duração das chamadas de função, utilização de certas funções etc. Essa técnica geralmente é utilizada para auxiliar na otimização de código.

acredita que o ideal seria que cada profissional pudesse fazer a sua parte sem ter de depender dos outros.

Após as entrevistas, foi possível perceber alguns problemas comuns apresentados pelos entrevistados:

- Ter de fazer várias versões da mesma música
- Tamanho limitado para os arquivos de áudio
- Não poder testar o som junto do jogo
- Dependência, em alguns casos, entre o compositor e os programadores

Quanto às funcionalidades, além das apresentadas pelo Dino em sua entrevista, podemos citar, como funcionalidades interessantes à ferramenta, a inserção de arquivos de áudio, a possibilidade de reprodução de efeitos (como aplicação de filtros passa alta/baixa), síntese de áudio no próprio *middleware* e recebimento de parâmetros vindos do jogo.

Além das entrevistas realizadas, antes de pensar na proposta do *middleware*, foi realizado um experimento para testar a possibilidade de utilizar a linguagem Pure Data dentro do projeto. Tendo o código fonte do jogo Mari0[18] em mãos, o código foi adaptado para que enviasse, via protocolo OSC, mensagens para um patch de Pure Data, o qual reproduziria um som de acordo com o valor recebido por esse protocolo. Dentro da mensagem enviada no protocolo, estaria contido um valor indicando o número de personagens que estavam presentes na tela do jogador, e o patch executaria, para cada intervalo de valores diferentes, um novo padrão de percussão. O intuito era que a percussão se intensificasse quanto maior fosse o número de inimigos na tela. Esse teste serviu como uma primeira experiência mostrando que seria possível a implementação de um patch que reagisse a parâmetros vindos de um jogo.

4.2 Proposta da ferramenta

Tendo em vista os problemas apresentados na seção acima, começou a ser elaborada uma proposta de ferramenta que pudesse resolver esses problemas e que fosse capaz de suprir as necessidades dos projetistas de som e ao mesmo tempo auxiliar os programadores na adição de áudio aos jogos. Pensando nisso, optou-se por dividir o sistema produzido em dois grandes módulos: a interface autoral, que é oferecida para os projetistas de som produzirem o conteúdo sonoro a ser adicionado aos jogos,

e o motor de áudio, que oferece diversas funções aos programadores para facilitar a manipulação do áudio produzido na interface autoral.

Desse modo, a interface autoral deve ser capaz de permitir a criação de um arquivo de áudio que possa ter seu comportamento alterado mediante eventos ou mudanças ocorridas dentro do jogo. Deve, portanto, existir um meio de comunicação entre o jogo e o áudio que está sendo reproduzido. É necessário também que esse canal de comunicação possa ser simulado pelo próprio compositor ou projetista de som, durante a composição, utilizando a própria interface autoral, para que o resultado final do áudio possa ser testado antes de ser enviado para implementação no jogo.

Além disso, a interface deve oferecer ao projetista de som e ao compositor recursos de processamento de sinais, como por exemplo síntese de áudio em tempo real, sincronização de sons independentes e aplicação de efeitos como filtros passa alta/baixa, *delay*, *reverb* etc, a fim de facilitar o processo de composição do áudio e produção do resultado final.

O motor de áudio, por sua vez, é responsável por oferecer ao programador um arcabouço de funções que facilite a inserção e reprodução de áudio dentro do jogo. O motor ainda deve ser capaz de reproduzir os arquivos gerados pela interface autoral, bem como transmitir a esses arquivos informações sobre eventos e acontecimentos do jogo, em tempo real, para que os elementos sonoros possam ser alterados durante a execução do jogo.

Não apenas deve ser apto a reproduzir os sons dentro do jogo, o motor de áudio também deve fazer isso de maneira eficiente, sem que a sua execução comprometa a performance dos outros elementos do jogo.

Ambos os módulos devem ser implementados pensando no usuário final, ou seja, a interface autoral deve ser o mais intuitiva possível para os projetistas de som e compositores, enquanto que as funções oferecidas pelo motor de áudio devem apresentar utilização simples por parte dos programadores, com nomes informativos e exigindo apenas os parâmetros necessários, reduzindo a complexidade na execução de qualquer ação envolvendo reprodução e manipulação de áudio dentro do jogo.

Seguindo a descrição acima, o modelo esperado para utilização do sistema é o seguinte:

- Trechos de áudio gravados previamente ou sintetizados são adicionados ao arquivo de áudio através da interface autoral
- Efeitos desejados são aplicados utilizando recursos oferecidos pela própria interface

- Parâmetros vindos do jogo podem ser utilizados pelos projetistas, dentro da interface, para gerar alterações no áudio de acordo com o que foi combinado entre a equipe de áudio e os programadores
- Simultaneamente, os programadores começam a implementação do jogo e dos elementos de áudio dentro dele. Para a parte do áudio, basta que utilizem as funções oferecidas pelo motor de áudio, fazendo com que ele execute junto do jogo e possa, com isso, reproduzir o áudio e enviar parâmetros a ele em tempo real.
- Após finalizado o trabalho da equipe de som e dos programadores, basta que os programadores passem a utilizar os arquivos de áudio produzidos pelos projetistas de som no código

4.3 Implementação do middleware

Para falar sobre a implementação do *middleware*, iremos dividir esta seção em duas partes: uma parte gerencial, na qual serão apresentadas as ferramentas adotadas para organização dos alunos presentes no projeto durante sua realização; e uma parte técnica, na qual serão apresentados os detalhes sobre a implementação do sistema em si.

4.3.1 Parte Gerencial

Durante a realização do projeto, foram organizadas reuniões semanais entre seus integrantes. Nessas reuniões, eram discutidos os avanços e problemas durante a implementação do sistema, testes efetuados, resultados intermediários e eram decididos os próximos passos e funcionalidades a serem implementadas. Além disso, também eram realizadas, durante as reuniões, sessões de implementação envolvendo os dois integrantes do projeto, nas quais ambos trabalhavam juntos, através da técnica denominada de programação em par, para programar novas funções ao sistema proposto.

Semanalmente, os integrantes do projeto também participavam de reuniões do grupo de Computação Musical do IME[19], durante as quais era apresentado o andamento do projeto e eram discutidas dúvidas tanto conceituais quanto de implementação.

Para organização das responsabilidades e tarefas, optou-se inicialmente pela utilização da plataforma online Trello[20]. Trello é um aplicativo para administrar projetos, no qual os usuários têm acesso a diferentes tabelas, nas quais podem ser adicionados cartões, anotações, e-mails entre outras funcionalidades, que podem ser

editadas e receberem comentários e atualizações. Decidiu-se que as tabelas dividiriam o projeto em tarefas a serem feitas, tarefas em execução e tarefas já terminadas, e em cada uma dessas tabelas, seriam adicionados cartões, os quais conteriam as tarefas em si que estariam sendo acompanhadas. A equipe acabou não se adaptando à plataforma, o que posteriormente fez com que os integrantes concordassem em abandonar seu uso no projeto, já que as anotações pessoais de cada membro aliadas às reuniões semanais se mostraram suficientes para manter a organização e andamento do trabalho.

Com relação ao armazenamento, compartilhamento e versionamento do projeto, optou-se pela utilização do sistema de controle de versão Git[21] e da plataforma de armazenamento de repositórios Github[22]. Foi criado um repositório no Github, no qual foram armazenados todos os códigos-fonte do projeto, de modo que este fosse acessível a todos os membros do projeto e, através das funções oferecidas pelo Git, foi possível que os membros editassem o mesmo código independentemente e que as alterações fossem unidas, posteriormente, sem problemas de versionamento nem de conflito entre as alterações. Além disso, o fato do código estar armazenado em um serviço online permitia a recuperação do código caso alguma alteração indesejada ocorresse no código ou caso algum membro perdesse os arquivos contendo todo o código-fonte do projeto. Ambos os serviços continuam sendo utilizados até hoje, e o repositório contendo o projeto[23] encontra-se aberto e disponível a todos que quiserem acessá-lo.

4.3.2 Parte Técnica

Durante as discussões semanais, muito se discutiu sobre como seria implementada a interface autoral para os sound designers, a tecnologia utilizada, o visual final etc. Foi decidido, porém, pelo aproveitamento da interface gráfica oferecida pela própria linguagem Pure Data. Desse modo, toda a parte de composição, aplicação de efeitos, processamento de sinais digitais e inserção de arquivos de áudio poderia ser aproveitada e, consequentemente, bastava apenas desenvolver uma maneira de permitir o recebimento de parâmetros vindos do jogo e algumas funcionalidades que facilitassem aos profissionais de áudio o trabalho utilizando a ferramenta.

Pensando nisso, a solução encontrada foi a implementação de submódulos para o Pure Data, que, após implementados, se tornam objetos utilizáveis dentro da linguagem, como se fossem novas funções implementadas dentro de um programa e que pudessem ser chamadas a qualquer instante. Os submódulos desenvolvidos foram:

- ODA_PANEL - Consiste em um objeto cuja entrada deve ser composta por

todo o áudio dentro do patch que o compositor ou projetista espera que seja reproduzido. Esse objeto, então, conterá em seu interior um vetor que armazenará todo esse áudio que deverá ser reproduzido pelo jogo. Esse vetor ficará acessível ao Motor de Áudio que poderá, assim, reproduzir esse conteúdo no momento em que for requisitado.

- ODA_SOUND - Trata-se de um objeto que permite ao compositor carregar um arquivo de áudio dentro desse patch, iniciar e parar sua reprodução tanto do início ao fim do arquivo quanto de trechos arbitrários do mesmo, escolhidos pelo compositor durante a utilização desse objeto. A saída do objeto é o sinal correspondente ao trecho atual do arquivo de áudio que está sendo reproduzido.

Antes de comentar sobre a maneira com que o Motor de Áudio envia comandos e faz acesso ao áudio contido nos patches criados na interface autoral, é preciso descrever duas outras classes existentes, que são elas a classe Player e a classe DSPServer.

A classe Player tem como função principal encapsular a OpenAL. Ou seja, essa classe oferece funções para reprodução de áudio, as quais fazem uso, em sua implementação, dos métodos oferecidos pela biblioteca OpenAL. Através desse encapsulamento, essa classe livra todas as outras classes e o programador também de ter de chamar métodos da biblioteca, monopolizando o acesso a ela e fazendo com que, toda vez que alguém precise reproduzir áudio, deva solicitar isso à classe Player. Desse modo, toda a complexidade fica escondida dentro da implementação dessa classe, enquanto que e as outras classes, para executar algum áudio, precisam apenas realizar uma chamada de função a alguma instância de objeto da classe Player. Um método interessante a se comentar sobre a classe Player é a função `streamData` que, utilizando-se das funções da OpenAL, administra diversos buffers para permitir que um trecho longo de áudio seja executado sem interrupção e nem causar redução de desempenho no Motor.

A classe DSPServer, por sua vez, é semelhante à classe Player, porém é a libpd quem é encapsulada por ela. Desse modo, a classe DSPServer fica responsável, dentro da OpenDA, por tudo que envolve processamento digital de sinal. Como a libpd é formada, na verdade, pelo código do Pure Data excluindo apenas a interface visual, tem-se, através dela, acesso a todas as funcionalidades da linguagem, e uma importante funcionalidade é que, através da libpd, é possível abrir patches de Pure Data dentro dela durante a execução do Motor de Áudio. Outra função implementada na classe DSPServer e que é essencial é a função `process`. Através dela, podemos fazer com que todos os patches executem um determinado número de *ticks*[5] passado

[5]Um tick é o relativo a uma unidade de ciclo de processamento dentro da linguagem Pure

como argumento e que o valor contido no array presente no objeto ODA_PANEL de cada um dos patches seja, após cada tick, somado em um vetor de saída também passado como argumento para essa função.

Outra função interessante presente na libpd é enviar mensagens para os patches abertos. É através dessas mensagens que a passagem de parâmetros do jogo para o arquivo que define o áudio se faz possível. Para a utilização dessa funcionalidade, foi criado o método `handleCommands` dentro da classe DSPServer, que garante que todas as mensagens dentro de uma fila de mensagens serão enviadas a seus respectivos patches.

Tendo em mãos essas classes auxiliares, era necessária alguma outra estrutura que oferecesse suporte ao programador para manipular o conteúdo criado através da interface autoral. Para isso, desenvolveu-se a classe Evento. Um evento é uma estrutura que armazena patches de Pure Data (utilizando-se, para isso, das funções presentes na classe DSPServer) e que é capaz de reproduzir esses patches, pausar ou parar sua reprodução, além de enviar comandos a eles. A grande vantagem na utilização dessa classe, portanto, é a possibilidade de enviar comandos aos patches através da função `pushCommand`, pela qual o Motor de Áudio é capaz de enviar um comando, que nada mais é do que uma sequência de parâmetros. Essa última funcionalidade se mostra essencial pois é através desses comandos, enviados em tempo de execução, que o áudio pode mudar seu comportamento mediante eventos que ocorram dentro do jogo.

Tem-se, por fim, a classe Engine, que funciona como o coração do Motor. Essa classe é responsável primeiramente por inicializar todos os módulos do Motor de Áudio, preparando o ambiente para que as funções tanto da OpenAL quanto da libpd possam ser executadas, através da função `start`. Ela encontra os dispositivos nos quais o áudio é reproduzido e inicializa um contexto no qual os efeitos de espacialização de áudio serão simulados (ambos os procedimentos realizados através de funções da OpenAL). Além disso, essa função também inicializa uma instância da classe DSPServer, o que permite a realização de qualquer atividade que envolva processamento digital de sinais (em inglês Digital Signal Processing ou DSP). A Engine também inicializa, nessa função, uma instância de Player que será utilizado para reproduzir todo o áudio produzido pelos Eventos. Assim como a função `start`, também existe a função `finish` que serve para finalizar o ambiente gerado no início da utilização da Engine, liberando a memória utilizada pelas bibliotecas abertas.

Outra função importante da Engine é a função `eventInstance`. A Engine é

Data, no qual são processados por completo um número específico de amostras (a quantidade de amostras processadas é chamada de "tamanho de bloco" e é um valor que pode ser alterado e definido arbitrariamente pelo programador)

a única entidade capaz de instanciar eventos, desse modo, um evento só pode ser criado pelo programador a partir dessa função da Engine. A função mais importante da Engine, porém, é `tick`. Ao passar um valor "dt" de tempo em segundos como argumento, essa função calcula quantos ticks deverão ser realizados pela classe DSP-Server em todos os patches para que se produza "dt" segundos de áudio processado. Esse número de ticks é, então, executado através da função `process` e o áudio processado é reproduzido pela instância de Player que foi criada na inicialização da Engine.

Outras funções presentes nas classes, mas menos importantes, são por exemplo as pertencentes à classe Player e responsáveis por modificar a posição das fontes de áudio, além de outras funções de controle, que servem para alterar a taxa de amostragem utilizada, número de bytes utilizados para representar uma amostra ou o formato de reprodução do áudio (Mono ou Estéreo). Quanto à classe Engine temos a função `registerPath`, que invoca a função `addPath` da classe DSPServer, cuja função é adicionar novos caminhos válidos à lista de caminhos que a libpd utiliza para encontrar os patches feitos pelos projetistas de som.

Capítulo 5

Resultados

Ao fim do projeto, foi possível implementar um *middleware* que cumprisse com o proposto. Através da interface autoral, é possível desenvolver um arquivo (ou patch de Pure Data) contendo uma definição de áudio que assume comportamento dinâmico de acordo com parâmetros que podem ser recebidos de um jogo. O Motor de Áudio, por sua vez, é capaz de executar esse patch produzido na interface autoral e reproduzir o áudio gerado por ele através do *hardware* de áudio do computador que executa o motor. Além disso, o Motor de Áudio também é capaz de enviar parâmetros em tempo de execução para esses patches. Com isso, o desenvolvedor pode utilizar essa ferramenta durante a produção de um jogo digital, utilizando o Motor para reprodução de áudio e para comunicação entre o jogo e o áudio contido nos patches criados pela interface autoral.

A utilização da ferramenta, portanto, ocorre como apresentado na seguinte figura:

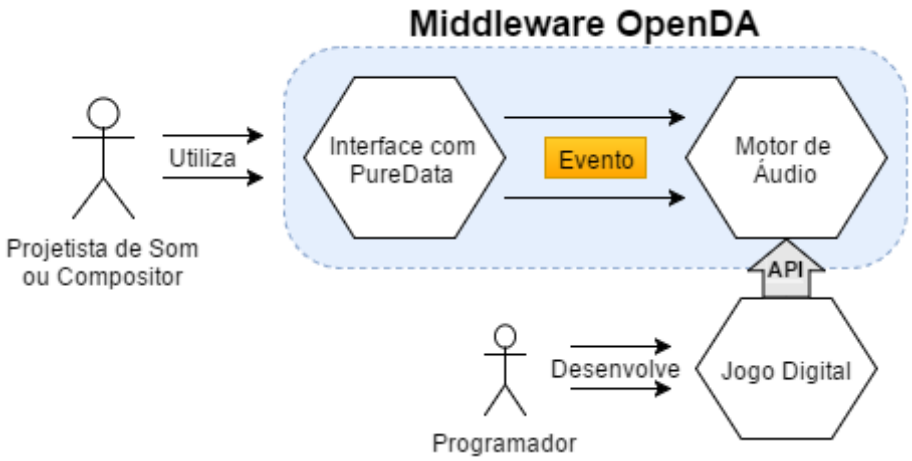

Figura 5.1: *Utilização do Middleware OpenDA*

Inicialmente, duas ações ocorrem ao mesmo tempo: Os projetistas de som e compositores começam a desenvolver a trilha sonora através da interface autoral en-

quanto que os programadores começam a implementação do jogo utilizando o Motor de Áudio para reprodução do Áudio do jogo. Durante a composição na interface autoral, o objeto que será utilizado dentro do Pure Data para receber os parâmetros do jogo é o objeto `[receive $0-command]`. Todos os parâmetros enviados a esse patch pelo jogo utilizando o Motor de Áudio serão recebidos através desse objeto, portanto o compositor já pode utilizá-lo para montar a estrutura de áudio. Já o programador pode utilizar as funções oferecidas pela API do Motor de Áudio para adicionar o áudio a seu jogo. Toda a interação entre a interface com o Pure Data (ou interface autoral) e o Motor de Áudio é realizada através da classe Evento, implementada dentro do Motor, que será responsável por reproduzir os patches criados pela interface e enviar a eles os parâmetros vindos do jogo.

Enquanto isso, dentro da interface o compositor pode simular a entrada de parâmetros através de objetos do próprio Pure Data, enquanto que o programador pode simular os áudios utilizando patches simples e facilmente produzidos no Pure Data (para fins de depuração de código). Ao fim da implementação do jogo e da composição da trilha, basta que os programadores substituam os patches de teste pelos patches contendo a trilha sonora feita pelos compositores e o jogo está pronto.

Para testar, portanto, o funcionamento da ferramenta em situações reais, foram realizados dois experimentos, que serão apresentados nas próximas seções.

5.1 Demonstração "Demo" da OpenDA

A fim de testar o funcionamento do *middleware* em um jogo, os integrantes do projeto desenvolveram uma demonstração da OpenDA, implementando, para isso, um pequeno jogo e utilizando a OpenDA para reprodução de áudio do jogo. Como os membros do projeto estão mais familiarizados com a implementação de jogos utilizando a linguagem de programação Lua, e com auxílio de um motor de jogos 2D em Lua chamado Löve[24], foi implementado um *wrapper* que irá traduzir chamadas de funções escritas em Lua em chamadas de funções em C++ para que o *middleware* consiga entender o código que foi implementado em Lua. O código do *wrapper* encontra-se no repositório do projeto, cujo link está apresentado no capítulo sobre a bibliografia.

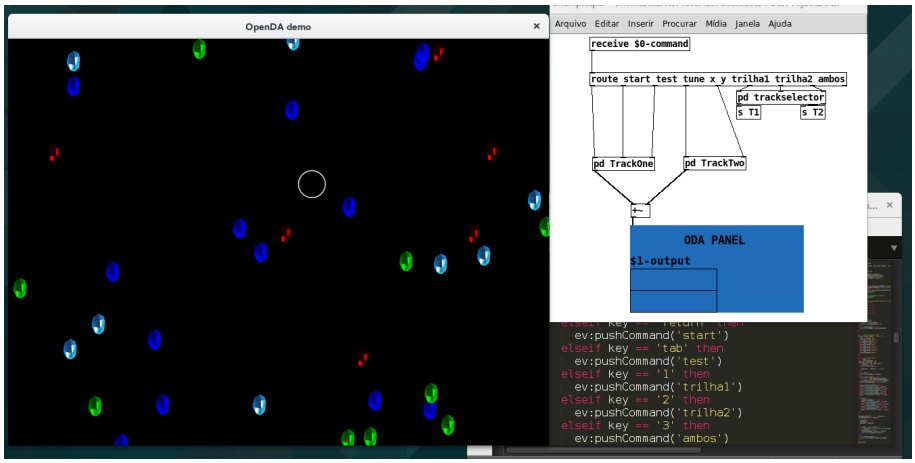

Figura 5.2: *Protótipo de demonstração utilizando a OpenDA*

A demonstração implementada consiste em um exemplo pequeno de jogo, feito para averiguar a usabilidade da biblioteca OpenDA. Para o áudio do jogo foram utilizados Patches simples de Pure Data. O jogo, cujo código também encontra-se no repositório do projeto, consiste de um círculo que deve capturar diferentes cristais que aparecerão na tela. A movimentação do círculo é Vertical e Horizontal e pode ser realizada através de um *joystick* ou do teclado da máquina.

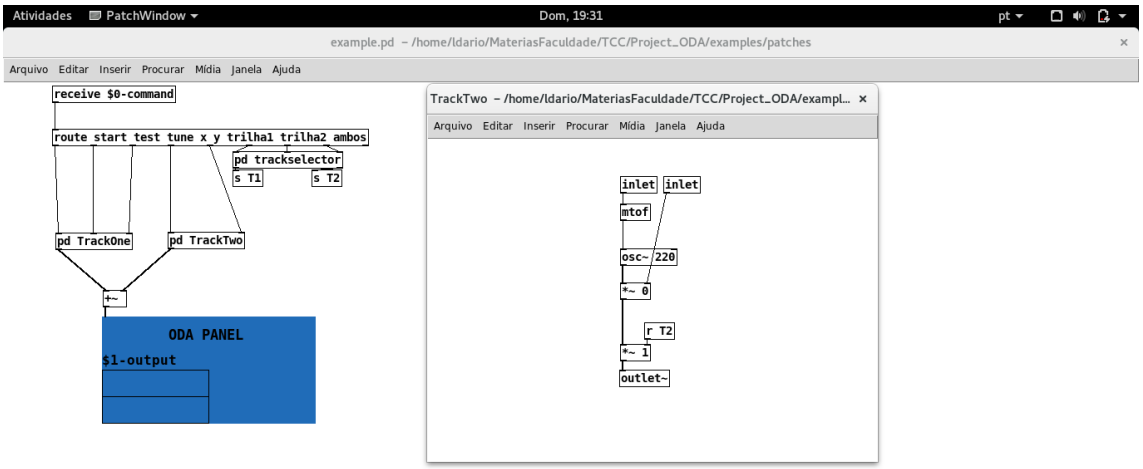

Figura 5.3: *Patch de Pure Data responsável pela música de fundo do protótipo*

Como música de fundo, um oscilador executa uma senóide cuja frequência e volume variam de acordo com a posição do círculo na tela. O valor da posição do círculo nos eixos X e Y da tela são os valores que foram utilizados como parâmetros

para serem enviados do jogo para o patch em Pure Data. Outros parâmetros enviados para o patch são algumas teclas pressionadas que permitem o jogador escolher entre ligar ou desligar o áudio de fundo. Vemos, na figura 5.3, uma imagem do patch utilizado na demonstração. Os parâmetros chegam através do objeto `[receive $-0command]`, enquanto o objeto `[route]` será responsável por redirecionar os parâmetros recebidos para suas respectivas funções. A principal trilha executada é a presente na sub-janela TrackTwo, que aparece em destaque na imagem. Os valores correpsondentes às posições X e Y são recebidas pelo inlet dessa sub-janela e são utilizados para alimentar o oscilador com uma frequência e o multiplicador `[* ]` com um valor de 0 a 1, que irá alterar o volume do áudio reproduzido. A sub-janela TrackOne contém uma outra trilha que pode ser reproduzida durante a execução do jogo (acessada através de teclas pressionadas dentro do jogo) e a sub-janela trackselector é a responsável por decidir qual trilha sonora será reproduzida, de acordo com parâmetros recebidos pelo jogo.

Com relação ao código em Lua, deixando de lado todas as linhas contendo a lógica do jogo e chamadas às funções presentes na LÖVE, poucas linhas foram necessárias para que as funcionalidades descritas acima fossem implementadas.

```
local oda = require 'oda'
```

A linha acima foi utilizada para incluir a biblioteca OpenDA dentro do código em Lua.

```
oda.start()
oda.registerPath("../patches")
oda.registerPath(ODA_PATCHES_PATH)
ev = oda.eventInstance "example"
```

Através das funções acima, o motor de áudio é inicializado (`oda.start()`), as funções `registerPath` atualizarão os caminhos para onde os patches estão e a linha `ev = oda.eventInstance "example"` instanciará um evento nomeado de "ev" e irá inicializá-lo com o patch "example.pd". Essas funções foram adicionadas dentro da função `love.load` para que sejam executadas logo que o jogo for iniciado.

Para o envio de comandos, bastou que dentro da função `love.update` fossem adicionadas as seguintes linhas:

```
ev:pushCommand('x', 70 + 57*ballX/w)
ev:pushCommand('y', ballY/h)
```

Na primeira linha, enviamos 2 parâmetros para o patch: o caractere 'x' indicando que o parâmetro enviado é relativo à posição X (horizontal) do círculo, e um valor dado pela expressão (70+57*ballX/w) . Esse valor varia de 70 a 127, sendo 70 a

extrema esquerda e 127 a extrema direita da tela, e será utilizado como o valor MIDI. Esse valor MIDI é traduzido para um valor em frequência utilizando objetos do Pure Data e então é usado para alimentar o oscilador que está em reprodução. Na segunda linha, a ideia é a mesma, porém o caractere enviado é 'y', indicando que o parâmetro enviado será relativo à altura Y do círculo na tela. O valor enviado é (ballY/h) que retornará um valor de 0 a 1 indicando em que fração da tela está o objeto verticalmente, sendo a parte inferior da tela representada pelo valor 1 e a parte superior da tela representada pelo valor 0. Esse valor será utilizado para modificar o volume do som, sendo 0 silencioso e 1 o volume máximo.

Dentro do `love.update` também chamamos a função:

```
oda.tick()
```

Ou seja, a cada *tick* executado pela engine LÖVE, executamos um *tick* da OpenDA, processando e reproduzindo os sons dos patches contidos nos eventos instanciados no programa. Por fim, para encerrar o funcionamento da OpenDA, chamamos, dentro da função `love.quit` a função:

```
oda.finish()
```

Utilizando poucas linhas de código, foi possível adicionar a biblioteca OpenDA dentro do código e executar suas funções, bem como reproduzir o áudio contido nos patches feitos para o jogo. Além disso, a execução do áudio não comprometeu a performance do jogo e o áudio reproduzido foi o esperado.

5.2 Alteração do áudio do Mari0

Após a implementação do "Demo" da OpenDA, comprovou-se que a ferramenta implementada funcionava e cumpria com o objetivo para a qual foi projetada. Decidiu-se, portanto, fazer um teste que experimentasse a utilização da OpenDA em um projeto maior, em um jogo que já tivesse muitas linhas de código implementadas, a fim de verificar se a biblioteca comprometeria o funcionamento do jogo, causando lentidão ou algum tipo de erro.

Desse modo, decidiu-se por testar a OpenDA dentro do jogo Mari0, o mesmo que foi utilizado nos primeiros testes com Pure Data apresentados na seção 4.1. Essa escolha se deu pelo fato do jogo apresentar código aberto, permitindo à equipe do projeto editá-lo, fazendo com que ele reproduza a música de fundo através da OpenDA e não da maneira originalmente implementada, além do fato de que o Mari0 é um jogo relativamente grande, contendo milhares de linhas de código implementadas. O jogo é implementado na linguagem de programação Lua.

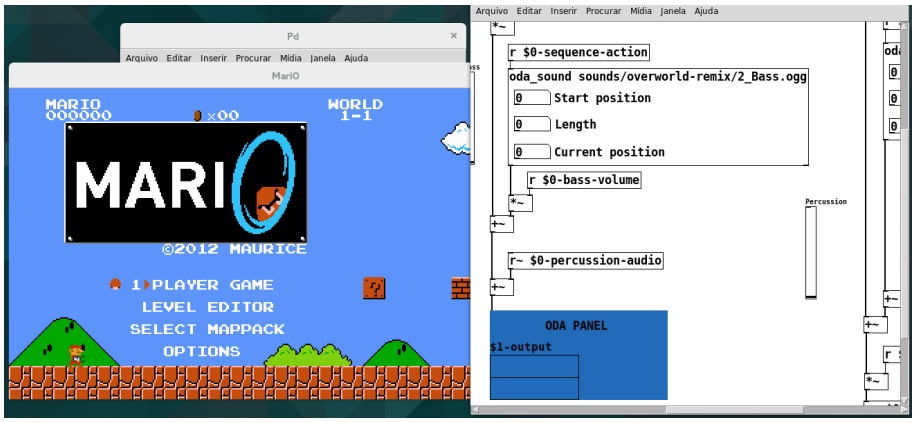

Figura 5.4: *Jogo Mari0 utilizando OpenDA. Em destaque, parte do patch que manipula a música de fundo do jogo*

Quanto ao código, dentro as funções `love.load`, `love.update` e `love.finish` foram adicionadas as mesmas linhas que no caso do "Demo", a única diferença é que, ao invés de criar um evento chamado "ev" e carregar o patch "example.pd", temos:

```
bgm_event = oda.eventInstance "bgm"
```

Ou seja, criamos o evento "bgm_event" (BGM do inglês Background Music ou música de fundo) que utiliza o patch "bgm.pd". Sempre que o áudio deve ser reproduzido, utiliza-se a linha

```
bgm_event:pushCommand "start"
```

E sempre que ele deve encerrar sua reprodução, utiliza-se a linha

```
bgm_event:pushCommand "stop"
```

Durante a execução do jogo, dentro da função `love.update` o seguinte trecho de código foi adicionado:

```
do -- report level progress
    local player_x = objects["player"][1].x
    bgm_event:pushCommand("progress", player_x/mapwidth)
end
```

Basicamente, a cada atualização realizada pela engine LÖVE, captura-se a posição do jogador, calcula-se em que fração da fase atual o jogador está e, então, envia-se esse valor precedido pela string "progress" para o patch contido no evento "bgm_event". Desse modo, o patch recebe esse valor e consegue saber em que parte do mapa encontra-se o jogador e, utilizando dessa informação, o patch pode tomar a ação que preferir.

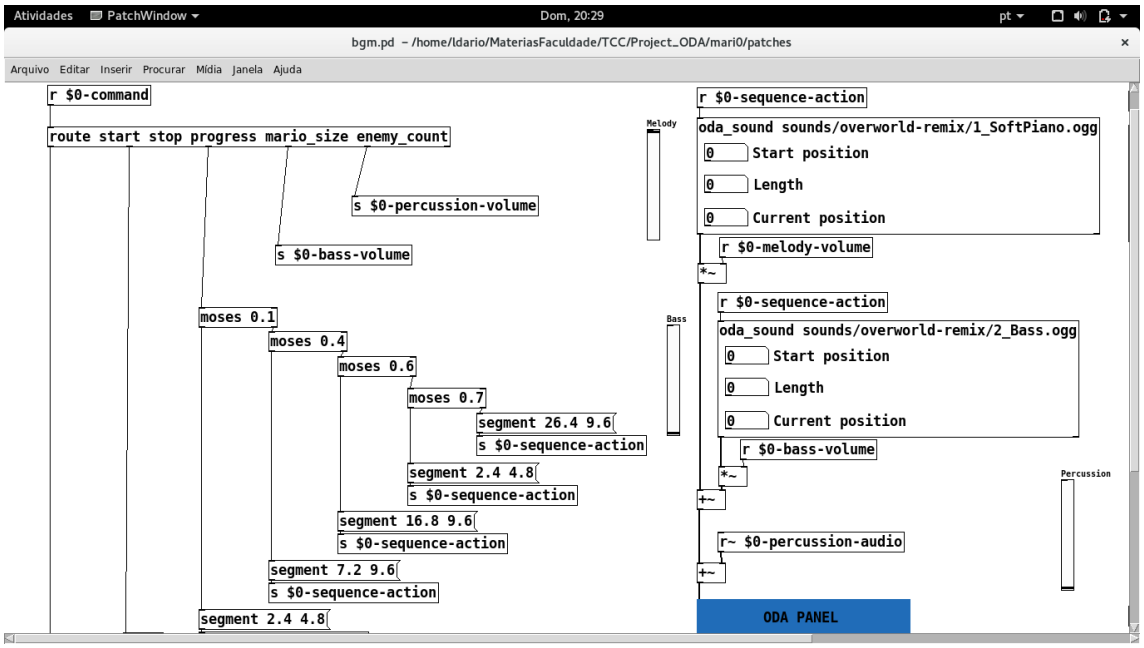

Figura 5.5: *Parte do patch de Pure Data responsável pela música de fundo utilizada no experimento utilizando o jogo Mari0*

No caso do patch "bgm.pd", apresentado na figura 5.5, ele receberá esse valor através do objeto `[receive $0-commando]` e, a partir dele, decidirá qual trecho da música contida no patch será executado. O objeto `[route]` é usado, nesse patch, da mesma maneira com que foi utilizado no patch "example.pd" da seção 5.1. O valor recebido através do parâmetro "progress" será redirecionado a uma sequência de objetos `[moses]` para descobrir a que intervalo de valores ele pertence. Depois disso, forma-se uma mensagem a ser utilizada pelos objetos `[ODA_SOUND]`, contendo a definição de que trecho da trilha deve ser reproduzido. Após o envio dessas mensagens, o trecho específico é selecionado dentro desse objeto e a saída de áudio é enviada ao objeto `[ODA_PANEL]`, que irá disponibilizar o áudio final para que o Motor de Áudio possa reproduzí-lo.

Os arquivos de áudio contidos no patch "bgm" são uma réplica da música de fundo original do jogo Mari0 e foram compostos pelos membros do projeto utilizando a ferramenta LMMS[25]. O LMMS é um software para compor música nos sistemas operacoinais Windows, Linux ou Apple. Essa ferramenta possui diversas funcionalidades que auxiliam no processo de composição, como por exemplo suporte a arquivos MIDI, sintetizadores implementados, simulações de instrumentos musicais, equalizadores, diversos efeitos como compressor, delay, reverb entre muitas outras funcionalidades. Além disso, LMMS é um software open source, poderoso e de fácil utilização, justificando sua utilização no projeto.

As modificações no código fonte do jogo exigidas para utilização do patch criado

foram muito pequenas. Além disso, adicionar ao áudio um comportamento dinâmico foi simples por conta da utilização do objeto ODA_SOUND, do recebimento de parâmetros vindos do jogo e dos objetos presentes na linguagem Pure Data. O áudio contido dentro do patch foi reproduzido de acordo com o esperado e sua reprodução não causou nenhuma perda de eficiência ao jogo, que continuou rodando com o mesmo desempenho que o apresentado na implementação original.

Capítulo 6

Conclusão

Ao fim do projeto, foi possível concluir boa parte das implementações esperadas para o *middleware* proposto e apresentado na seção 4.2. O fato de grande parte das funcionalidades essenciais estarem completas é confirmado pelos testes feitos através das implementações que utilizavam o OpenDA, pois todos eles mostram que o OpenDA serviu o propósito do sistema idealizado pelo projeto, que era de ser uma ferramenta capaz de permitir a definição, manipulação e reprodução de áudio dinâmico, a ser utilizada na produção de jogos digitais.

Através da interface autoral do OpenDA, que é um aproveitamento da interface gráfica existente do Pure Data, o compositor e projetista de áudio pode definir um arquivo, construído como um patch dessa linguagem, que seja uma representação de áudio dinâmico, ou seja, cujo comportamento é alterado mediante recebimento de parâmetros vindos de alguma outra aplicação, como por exemplo um jogo.

Seu Motor de Áudio, por sua vez, é capaz de executar duas funções importantes: A primeira é a de reproduzir áudio através do *hardware* de áudio da máquina que esteja executando o motor, e a segunda é ser capaz de manipular os arquivos produzidos pela interface autoral, devido à capacidade de abrí-los, enviar parâmetros a esses arquivos em tempo de execução, e ainda reproduzir o áudio definido neles.

Além de ser capaz de realizar todas essas ações, o *middleware* ainda as faz de maneira eficiente, pois consegue executar todas essas tarefas sem comprometer o desempenho do aplicativo para o qual oferece seus serviços. Isso mostra que o OpenDA é flexível o suficiente para ser utilizado tanto em projetos pequenos e simples (como o "Demo" apresentado na seção 5.1) quanto em projetos de porte relativamente maior (como o jogo Mari0, cujo experimento é apresentado na seção 5.2).

Boa parte das funcionalidades interessantes a serem implementadas no *middleware* e que foram apresentadas na seção 4.1 podem ser reproduzidas utilizando apenas os elementos presentes na linguagem Pure Data, o que torna o *middleware*

mais leve e flexível e aproveita melhor as competências do sound designer, que não precisará aprender elementos novos externos à linguagem Pure Data. É possível, por exemplo, controlar o volume dos áudios, reproduzir os arquivos de áudio de maneira não linear, repetir o mesmo som diversas vezes, além de aplicar diversos efeitos, todas ações que são facilmente reproduzidas utilizando os objetos da linguagem. Algumas funcionalidades, como por exemplo o "Profiling", acabaram não sendo implementadas, mas são um ótimo candidato a possíveis atualizações futuras na ferramenta. Alguns exemplos que correspondem a perspectivas de trabalho futuro, e que são de grande interesse dos membros do projeto, são novos módulos para a interface autoral que facilitem o trabalho dos projetistas de som, oferecendo objetos poderosos e que agilizem o trabalho de composição, como por exemplo Mixers, efeitos pré-implementados, uma pequena interface possuindo botões clicáveis e que ofereçam ao compositor uma maneira simples e rápida de adicionar novos arquivos de áudio e entrada de parâmetros, removendo do usuário da interface a necessidade de possuir conhecimento profundo da linguagem Pure Data para utilizá-la.

Bibliografia

[1] A. L. Battaiola. Jogos por Computador – Histórico, Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação, 2000. 6

[2] K. Collins. An Introduction to the Participatory and NonLinear Aspects of Video Games Audio. 2007. 7

[3] L. C. Meneguette. Áudio dinâmico para games: conceitos fundamentais e procedimentos de composição adaptativa. SBGames 2011, 2011. 7

[4] B. Stroustrup. **The C++ Programming Language**. Addison-Wesley, 4th edition, 2013. 8

[5] R. Nystrom. **Game Programming Patterns**. Online edition, 2015. 8

[6] Pure Data website. http://puredata.info/. Acesso em: 2015-11-29. 9

[7] libpd website. http://libpd.cc/. Acesso em: 2015-11-16. 9

[8] OpenAL website. http://openal.org/. Acesso em: 2016-01-27. 10

[9] Loki Software website. http://www.lokigames.com/. Acesso em: 2016-01-27. 10

[10] Creative Technology Ltd. website. http://www.creative.com/. Acesso em: 2016-01-27. 10

[11] OpenAL Soft website. http://kcat.strangesoft.net/openal.html. Acesso em: 2016-01-27. 10

[12] Tapps Games website. http://tappsgames.com/. Acesso em: 2016-01-28. 12

[13] Nucleo de Pesquisas em Sonologia website. http://www2.eca.usp.br/nusom/. Acesso em: 2016-01-28. 12

[14] 7Sounds - Game Audio Solutions website. https://www.linkedin.com/company/7sounds. Acesso em: 2016-01-28. 12

[15] DVMúsica website. http://dvmusica.wix.com/dvmusicaapresentacao. Acesso em: 2016-01-28. 12

[16] FMOD Studio website. http://www.fmod.org/fmod-studio/. Acesso em: 2016-01-30. 14

[17] Max Programming Language website. https://cycling74.com/products/max/. Acesso em: 2016-01-30. 14

[18] Mari0 website. http://stabyourself.net/mari0/. Acesso em: 2016-01-30. 15

[19] Grupo de Computação Musical do IME website. http://compmus.ime.usp.br/. Acesso em: 2016-01-28. 17

[20] Trello website. https://trello.com/. Acesso em: 2016-01-28. 17

[21] Git website. https://git-scm.com/. Acesso em: 2016-01-28. 18

[22] Github website. https://github.com/. Acesso em: 2016-01-28. 18

[23] Repositório do projeto OpenDA. https://github.com/open-dynamic-audio. Acesso em: 2016-01-28. 18

[24] LÖve website. https://love2d.org/. Acesso em: 2016-01-30. 23

[25] LMMS website. https://lmms.io/. Acesso em: 2016-01-30. 28