UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

IVO MATHEUS DE GOES LOPES

GINGA WINGS - UM MOTOR DE JOGOS PARA AUTORIA DE APLICAÇÕES DE HIPERMÍDIA COM NCLUA

São Luís
2015

IVO MATHEUS DE GOES LOPES

GINGA WINGS - UM MOTOR DE JOGOS PARA AUTORIA DE APLICAÇÕES DE HIPERMÍDIA COM NCLUA

Monografia apresentada ao curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Carlos de Salles Soares Neto

São Luís
2015

Lopes, Ivo Matheus de Goes

Ginga Wings - Um Motor de Jogos para Autoria de Aplicações de Hipermídia com NCLua/ Ivo Matheus de Goes Lopes. – São Luís, 2015.

81 f.

Orientador: Prof. Dr. Carlos de Salles Soares Neto

Monografia(Graduação) – Universidade Federal Do Maranhão , Curso de Ciência da Computação, 2015.

1. Desenvolvimento de Jogos 2. Motor de Jogos 3. TV digital 4.3, Lua

CDU 004.55

IVO MATHEUS DE GOES LOPES

GINGA WINGS - UM MOTOR DE JOGOS PARA AUTORIA DE APLICAÇÕES DE HIPERMIDIA COM NCLUA

Monografia apresentada ao curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Carlos de Salles Soares Neto

Aprovada em 14 de Julho de 2015

BANCA EXAMINADORA

Carlos de Salles Soares Neto

Dr. Carlos de Salles Soares Neto
Departamento de Informática / UFMA - Orientador

Portela

Me. Carlos Eduardo Portela Serra de Castro
Departamento de Informática / UFMA

Dr. Luciano Reis Coutinho
Departamento de Informática / UFMA

Resumo

A presente monografia tem como objetivo desenvolver um motor de jogos (game engine), flexível, extensível e de propósito genérico para o Sistema Brasileiro de Televisão Digital (SBTVD), através do middleware Ginga, utilizando-se NCLua, que é a integração da linguagem NCL e Lua. Por flexível, entende-se um software que, apesar de apresentar restrições de uso pode ser adaptada para resolver problemas específicos. Por extensível, entende-se que o software pode ser evoluído em suas funcionalidades internas pelo usuário para atender necessidades não supridas pelo funcionamento original. E por propósito genérico, entende-se que é um Motor de Jogos não especializado em produzir nenhum gênero de jogo, podendo ser ajustado para funcionar com vários gêneros. A necessidade da criação de um motor de jogos dá-se pela complexidade de desenvolvimento de jogos. Os mesmos trabalham paralelamente com mídias visuais e auditivas, ao mesmo tempo em que interagem com o telespectador. Para facilitar a reprodução dessas mídias, e prover uma interface de alto nível para produção de jogos, este motor foi desenvolvido. O desenvolvimento do trabalho se deu no desenvolvimento de uma arquitetura de software para dar suporte às necessidades comuns em jogos, nomeadamente controle de gráficos, sons, entrada de dados do usuário e controle de atualizações. A partir disso, foi desenvolvida uma API que implementasse essa arquitetura, provendo as funcionalidades especificadas e tantas outras úteis para o desenvolvimento de jogos, terminando com um curto exemplo básico de uso da mesma em conjunto com as marcações NCL. Foi feito um estudo de caso para testar as capacidades do motor, uma implementação do jogo Tetris para TV Digital. Por fim, o jogo foi implementado com sucesso, apropriando-se de várias funcionalidades comuns ao tipo do jogo desenvolvido, mostrando a capacidade do motor de jogos.

Palavras-chave: Desenvolvimento de Jogos, Motor de Jogos, TV digital, Lua.

Abstract

This thesis aims to develop a game engine, flexible, extensible and of generic purpose for the Brazilian Digital Television System (SBTVD), through Ginga middleware, using NCLua, which is the integration of programming language NCL and Lua. By flexible is meant a software that, despite having restrictions of use can be adapted to solve specific problems. By extensible is meant that the software can be evolved in its internal features to meet user needs unmet by the original specification. And for generic purpose, it is understood that it is a game engine not specialized in any game genre, and can be adjusted to work with various genres. The need to create a game engine comes from the complexity of game development. Games work in parallel with both visual and sound media, while interacting with the viewer. To facilitate the reproduction of these media, and provide a high-level interface for game development, this engine was built. A architecture was developed to support common needs games including graphic control, sounds, user data IO and loop updates. From this, an API that implemented this architecture was developed, providing the specified functionality and many other useful functions for developing games, finishing with a short basic example of its use with with NCL tags. A case study was done to test the engine, an implementation of Tetris game for Digital TV. The game was successfully implemented, appropriating many common features to its kind, showing the capacity of the game engine.

Keywords: Game Develpment, Game Engine, Digital TV, Lua.

Lista de ilustrações

Lista de códigos

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface
BGM	Background Music
DFR	Dependente do Frame Rate
FPS	Frames Por Segundo / Frames Per Second
GBF	Gamework's Brazilian Framework
GW	Ginga Wings
IFR	Independente do Frame Rate
OO	Orientação a Objetos
PC	Computador Pessoal (Personal Computer)
SBTVD	Sistema Brasileiro de Televisão Digital
SE	Sound Effect
SO	Sistema Operacional
SRS	Super Rotation System
STB	Setop-Box
TVDi	TV Digital Interativa

Sumário

1 Introdução

O surgimento do Sistema Brasileiro de Televisão Digital favoreceu a criação de uma nova plataforma de desenvolvimento de software. A TV digital permite maior qualidade de imagem e som, além de interatividade para o telespectador. Entre todas as aplicações possíveis, para esse novo ambiente, destacam-se os jogos digitais que atraem pessoas de todas as idades em todo o mundo (CORRÊA et al., 2013). Jogos digitais reúnem todos os tipos de mídias - vídeo e áudio - e reagem à interação do jogador, tendo essa distinção de outros tipos de provedores de entretenimento, como filmes. A interatividade associada às mídias permite jogos receberem a classificação de aplicações de hipermídia (FERRARI, 2011).

Como agregadores de diversos tipos de mídia e interação, jogos podem ser aplicações de complexo desenvolvimento. Enquanto existem eventos em que o desafio é produzir um jogo completo em 48 horas (DIáRIO CATARINENSE, 2013), outros podem levar anos para serem completados (HUGHES, 2003), notavelmente mais tempo do que a produção de filmes campeões de bilheteria. Normalmente esta dificuldade emergia da necessidade de otimizar o jogo o máximo possível para se ter uma experiência agradável, mas com o avanço nesse campo de desenvolvimento, novos problemas surgiram, ao ponto dos times de desenvolvimento se focarem em tentar chegar perto de produzir as funcionalidades desejadas, pois as funcionalidades em si já se tornavam custosas demais para se alcançar (BLOW, 2004).

Para diminuir a complexidade da produção de jogos, foram criados, nos anos 90, os motores de jogos (*game engines*), softwares que tem o propósito de interfacear o acesso ao hardware da máquina, permitindo ao grupo de desenvolvedores trabalhar somente na parte que interessa, ou seja, o jogo em si (GREGORY, 2009). Ao longo do tempo, os motores foram evoluindo e agregando mais funções, e sendo capazes de compilar jogos para mais plataformas, dentre elas a TV Digital, foco de desenvolvimento desse trabalho. Existem diversos trabalhos interessantes na área, por exemplo, motores de jogos como o Unity Engine, que conseguiu obter um alto grau de abstração para o desenvolvimento de jogos, e outros motores menos poderosos, como o Game Maker. Infelizmente, nenhum desses motores suporta produção de jogos para TV Digital com Ginga. Outros motores já foram desenvolvidos para trabalhar com Ginga, como por exemplo o motor de jogos ATHUS.

Este trabalho tem como objetivo principal criar um motor de jogos, de propósito genérico em jogos 2D, para SBTVD, utilizando a linguagem Lua, em conjunto com código NCL. O motor tem como foco flexibilidade e extensibilidade, além de um rápido processamento. Por flexível, entende-se um software que, apesar de apresentar restrições

de uso pode ser adaptada para resolver problemas específicos. Por extensível, entende-se que o software pode ser evoluído em suas funcionalidades internas pelo usuário para atender necessidades não supridas pelo funcionamento original. E por propósito genérico, entende-se que é um Motor de Jogos não especializado em produzir nenhum gênero de jogo, podendo ser ajustado para funcionar com vários gêneros.

Objetivos específicos deste trabalho seguem:

- Desenvolver uma arquitetura básica para o motor, capacitada para rodar jogos em TV Digital com Ginga.
- Prover uma API que implemente essa arquitetura, com foco na flexibilidade e extensibilidade e um exemplo básico de uso.
- Testar o motor com um estudo de caso, para verificar sua funcionalidade e flexibilidade.

A organização do trabalho segue como descrito: o capítulo 2, Trabalhos Relacionados, descreve alguns motores de jogos e suas funcionalidades, tendo foco em sua arquitetura, quando possível. O detalhamento da arquitetura do Ginga Wings descrita neste trabalho pode ser verificado no capítulo 3, assim com da API disponível, com um jogo de exemplo mostrando, passo a passo, como configurar o ambiente NCL para utilizar o motor. Seguido a isto, no capítulo 4, um estudo de caso, temos a descrição da criação de um jogo, Tétris, utilizando o motor. Além de uma contextualização sobre Tetris e suas regras, há a demonstração do resultado final da implementação. O Apêndice A mostra um trabalho interessante utilizando este mesmo motor, porém em uma versão antiga e não tratada neste trabalho, enquanto que os Anexos A e B disponibilizam informações sobre as Linguagens NCL(RATAMERO, 2007) e Lua(OOKI, 2013), utilizadas neste trabalho.

2 Trabalhos Relacionados

Os motores de jogos começaram a surgir a partir dos anos 90, numa necessidade de acelerar o trabalho de produção de jogos, utilizando-se o conceito de componentes reutilizáveis (ou seja, partes da aplicação eram reaproveitadas em jogos diferentes, acelerando a produção). Várias implementações de motores de jogos já foram criadas, utilizando diversas técnicas para facilitar o trabalho de autoria de jogos. Muitos motores avançaram além da implementação de interfaces para acessar funções básicas do hardware alvo e propuseram interfaces de alto nível, facilitando a produção de imagens, eventos do jogo, entre outros aspectos (GREGORY, 2009). A princípio, os motores eram extremamente especializados. Por exemplo, a série de jogos de RPG intitulada Baldur's Gate, da BioWare, se utilizou do Infinity Engine, especializado em jogos de RPG que seguiam as regras AD&T (BIOWARE, 2013). A popularidade desse tipo de *software* aumentou nos anos 90, com o aparecimento de jogos *mods*[1] do famoso jogo Doom, criado pela Id Software (GREGORY, 2009). Isso foi possível graças à separação que a equipe fez entre os componentes do jogo – deixando bem definidos os códigos que controlavam o renderizador gráfico, a colisão de objetos, o sistema de áudio, os recursos de mídia e imagem, as regras de jogo, etc. -, o que permitiu que outros grupos de desenvolvedores modificassem apenas as partes que lhes interessava (basicamente os recursos de áudio e imagens, arquivos de armas, inimigos, fases, etc), utilizando-se de *Toolkits*[2] liberados pela empresa. Hoje temos motores bem mais genéricos do que os dos anos 90. Motores como Unreal ou Unity podem ser utilizados e modificados para criar uma vasta variedade de jogos para várias plataformas, 3D e 2D.

O propósito deste capítulo é apresentar algumas dessas ferramentas, destacando o que há de vantagens e desvantagens de cada uma, a fim de se obter uma visão geral do que se pode esperar de um motor de jogos. Primeiramente é apresentado o motor Unity Engine, por ter bastante relevância no mercado. O motor é considerada completo, disponibilizando várias ferramentas para auxiliar no desenvolvimento de jogos. No entanto, não foi possível descrever sua arquitetura pela mesma não ser de domínio público, e portanto são descritas suas características, mostrando o que oferecem ao usuário para desenvolvimento de um jogo. A seguir, o motor Game Maker é descrito, tendo acompanhado o desenvolvimento de jogos por vários anos e ser uma ferramenta de preferência para muitos. Ele está sujeito à mesma restrição do Unity Engine, e portanto apenas é feito um apanhado de suas características. Por fim, são apresentados alguns motores voltados para o desenvolvimento de jogos para TVDi.

[1] Nome dado a modificações de partes de um jogo, com o propósito de adicionar funcionalidades e conteúdo.

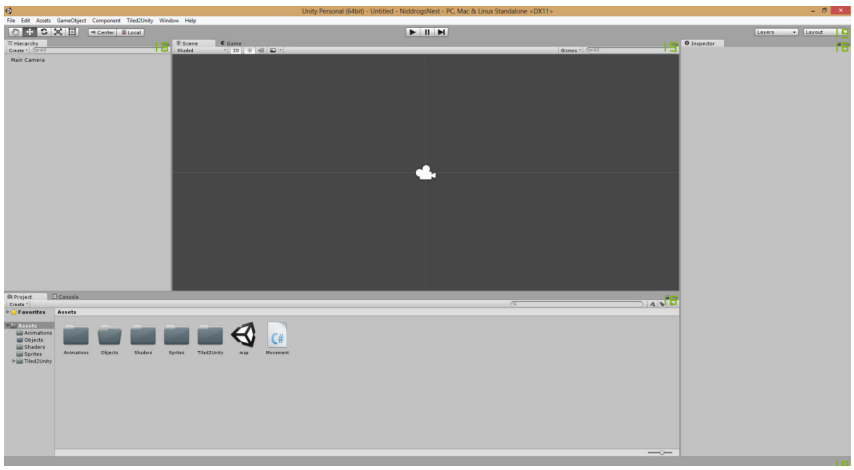
[2] Toolkit é um conjunto de elementos básicos de uma GUI. Normalmente são implementados como uma biblioteca de rotinas ou uma plataforma para aplicativos que auxiliam numa tarefa.

2.1 Unity Engine

A Unity Engine é um motor de jogos de propósito geral, desenvolvido pela Unity Technologies. Ele é utilizado para desenvolvimento de jogos para PC, consoles, dispositivos móveis e sites web. Por ser de propósito geral, ele geralmente vem sem nenhuma estrutura de jogo pronta, obrigando o desenvolvedor a montar seu próprio ambiente propício para o desenvolvimento do jogo. É capaz, ainda, de produzir games em 3D e 2D, tendo dois modos de trabalho para cada tipo de plano. Entre os aspectos do motor, o que mais se destaca é a sua portabilidade: ele é capaz de compilar um mesmo código de jogo para as diversas diferentes plataformas disponíveis, considerando-se ajustes e cuidados para cada caso. Naturalmente, um jogo graficamente otimizado para PC não vai rodar bem em um celular devido a sua menor potência de hardware, então nem sempre a transição de uma plataforma para outra será trivial.

Para atingir este grau de portabilidade, o motor utiliza diversas ferramentas para manipulação de gráficos, dependendo da plataforma alvo. Direct3D para Windows e XBox, OpenGL para Mac e também Windows, OpenGL ES para dispositivos móveis (Android, iOS, Windows Phone, etc.), WebGL para browsers, e APIs proprietárias nos casos de consoles.(UNITY TECHNOLOGIES, 2015b) É possível especificar modos de compressão de texturas e resolução para cada plataforma, e o motor dá suporte a várias técnicas de iluminação como *Bump Mapping*, *Reflection Mapping*, *Ambient Occlusion*, etc (UNITY TECHNOLOGIES, 2015c).

Figura 1 – Interface Gráfica de Usuário da Unity Engine em seu formato padrão



Fonte: Produzido pelo Autor

O motor trabalha com Mono para compilação de seus scripts, uma implementação em código aberto da .NET Framework. (MONO PROJECT, 2015) Aceita implementações tanto em C# ou UnityScript (chamado simplesmente de JavaScript nos manuais). O Unity

Engine vem com um grande acervo de exemplos, tutoriais e documentação tanto para scripts como uso do motor em si, melhorando a produtividade do usuário.

Para controle da física, Unity integra a funcionalidade da *physics engine* PhysX, desenvolvida pela nVidia, em ambientes 3D. Para ambientes 2D, utiliza Box2D. (AGUIAR, 2014) O motor expõe diversos componentes que determinam um comportamento físico, como Caixas (3D e 2D, separadamente), Esferas, etc. A tarefa de criar colisões, efeitos mais complexos como molas, massa, reação a força, roupas, é toda interfaceada por esses componentes e poucas linhas de código, simplificando conhecimentos complexos de matemática com simples chamadas à API.

Como Unity é um motor de propósito genérico, ele vem sem estruturas prontas para jogos específicos (por exemplo, as raquetes e a bola em um jogo de Pong), obrigando o desenvolvedor a criar toda a estrutura do zero. Unity disponibiliza a possibilidade de extensão, que é utilizada pela comunidade para trazer essas estruturas já prontas para jogos(UNITY TECHNOLOGIES, 2015a), apesar da maioria não ser gratuita. O time de desenvolvedores está disposto a dar ouvidos a sugestões de melhorias na ferramenta, disponibilizando toda uma área somente para feedback.

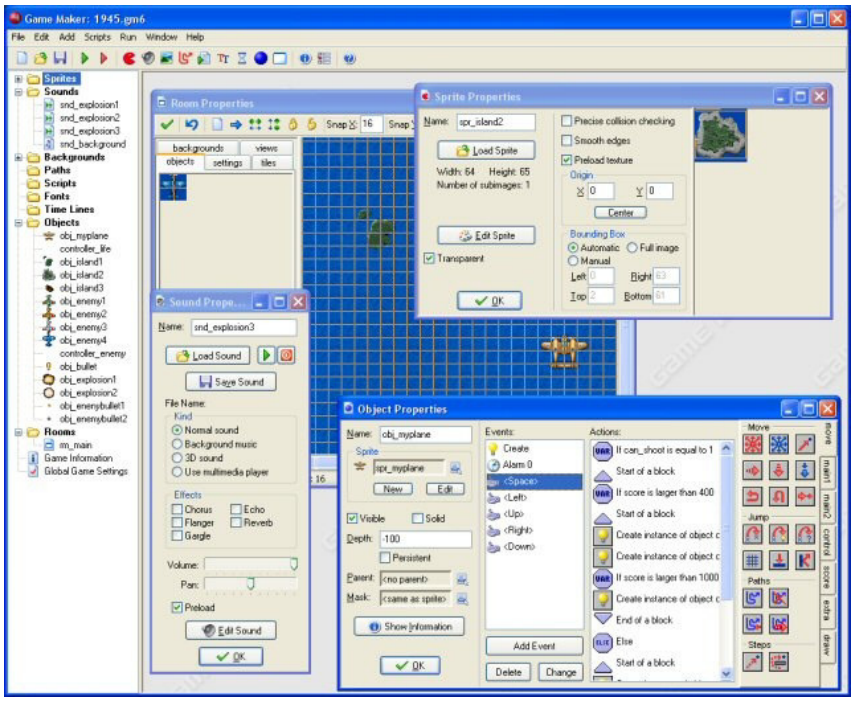
O editor possui uma interface simples, separando as diversas categorias de desenvolvimento em diferentes visões, todas ajustáveis de acordo com as necessidades do usuário. Os dados são guardados em estruturas de árvores, permitindo associação entre objetos sob a forma pai-filho. A interface é toda planejada para atender ao modelo *drag-n-drop* (clicar e arrastar), facilitando a interação com o mouse. Para cada objeto na cena, é possível adicionar vários componentes, como scripts, sensibilidade à física, métodos de renderização, entre outros. A API Mono expõe uma completude de métodos para manipular esses objetos através de scripts, de forma que possam assumir diversas funções no jogo, como interfaces de usuário, personagens na cena, textos, etc.

2.2 Game Maker

Game Maker(YOYO GAMES, 2015) foi desenvolvido em Delphi pela YoYo Games. O motor foi criado com o propósito de facilitar o desenvolvimento de jogos para iniciantes na área. Para este fim, os desenvolvedores criaram um modelo de *actions*, onde é possível determinar o comportamento dos objetos do jogo através de simples cliques, sem necessidade de profundos conhecimentos em programação. O motor não é focado em um único gênero de jogo, apesar de ter um foco 2D, e é capaz de compilar seus jogos para diversas plataformas. Possui, ainda, uma linguagem própria para programações mais avançadas, chamada de *Game Maker Language*.

Uma das características consideradas marcantes pelos usuários é a rapidez do desenvolvimento. O motor dá diversas ferramentas para manipular os objetos do jogo,

Figura 2 – Interface Gráfica do Game Maker



Fonte: (YOYO GAMES, 2015)

acelerando a produção, ao ponto de ser capaz de entregar jogos completos em questão de horas(GROCHOWIAK, 2012). Apesar de não ter um gênero como foco de desenvolvimento, a ferramenta é considerada flexível o suficiente para desenvolver qualquer tipo de jogo, considerando-se a dificuldade inerente a cada gênero.

Em termos de processamento gráfico, o motor não é muito robusto, apesar de ser possível desenvolver belos efeitos 2D utilizando a *Game Maker Language*. Apesar de existirem reclamações com relação à velocidade de processamento dos jogos, há exemplos de jogos carregados de efeitos gráficos que rodam na velocidade padrão(60fps) (GROCHOWIAK, 2012).

Sendo um motor voltado para usuários menos técnicos, ou em aprendizado (COUPRIE, 2015), ele cumpre bem o seu papel. Seus comandos são fáceis de aprender, a sintaxe do código não é complicada, existe muita documentação e tutoriais, tornando-se acessível para usuários com pouco conhecimento técnico, como por exemplo artistas. Por outro lado, os editores embutidos na ferramenta, como o editor gráfico e o editor de cenas são extremamente simplórios, o que em muitos casos pode incentivar o usuário a realizar essas tarefas em programas dedicados.

2.3 TUGA

O *middleware* TUGA (FERREIRA; SOUZA, 2009) propõe dar suporte ao desenvolvimento de jogos em TVD, utilizando Java. Esse *middleware*, além da estrutura de execução dos jogos, disponibiliza também uma API de desenvolvimento de aplicações para a TV digital. O *middleware* oferece suporte aos seguintes sistemas: Sistema Gráfico, Sistema Sonoro e Sistema de Entrada.

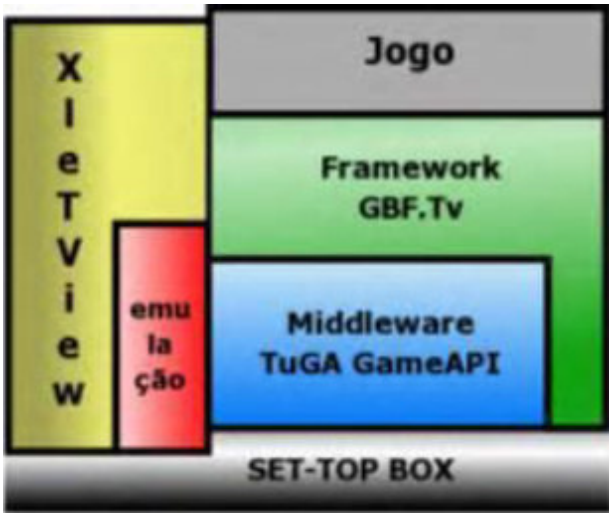
O sistema gráfico é o responsável por permitir a integração com o vídeo, ou seja, por permitir que as imagens e as primitivas gráficas sejam apresentadas no monitor de TV/PC, dando um *feedback* visual ao jogador.

O sistema sonoro é o responsável por permitir o efeito de imersão, com o suporte a uma boa ambientação sonora durante a atividade interativa imersiva.

O sistema de entrada é o responsável por permitir a interação do jogador com o dispositivo, dando assim inicio ao processo de interação.

Além das funcionalidades iniciais apresentadas, o *middleware* possui uma camada auxiliar, o *framework* GBF Tv, que fornece os seguintes recursos essenciais para auxiliar em alto nível a criação de jogos: Gerenciador de Sprites; Gerenciador de Fontes; Gerenciador de Personagens; Gerenciador de Sons; Gerenciador de Interface Gráfica com Usuário; Gerenciador de Tempo. A arquitetura de todos componentes do TUGA pode ser vista na Figura 3.

Figura 3 – Arquitetura do Motor de Jogos TUGA



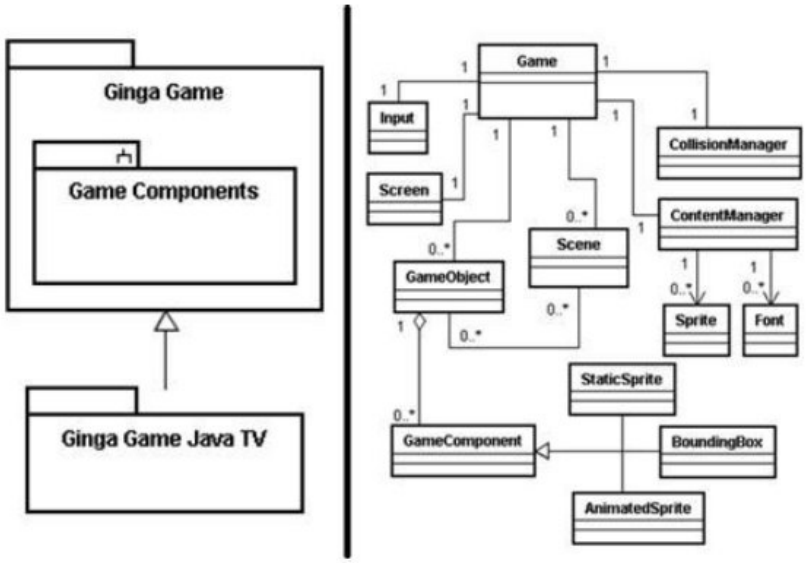
Fonte: (FERREIRA; SOUZA, 2009)

2.4 Ginga Game

GingaGame (BARBOZA; CLUA, 2009) é um *framework* de desenvolvimento de jogos para a TV Digital. Seu objetivo é fornecer uma estrutura que torna mais fácil o desenvolvimento de jogos para a TV Digital e torna essa tarefa mais semelhante ao desenvolvimento de jogos para PC.

Este *framework* é subdividido em três diferentes pacotes Java (vide Fig. 4), de forma que as classes que precisam de recursos específicos da plataforma estão em um pacote separado das classes que não têm este tipo de dependência. Assim, a migração do GingaGame para outra plataforma pode ser feita apenas fazendo as modificações necessárias em pacotes específicos, enquanto os outros permanecem inalterados.

Figura 4 – Arquitetura do Motor de Jogos Ginga Game



Fonte: Adaptado de (BARBOZA; CLUA, 2009)

O pacote `GingaGame` fornece algumas interfaces abstratas que devem ser implementados em uma plataforma específica. Neste pacote são definidos os conceitos básicos do *framework*, como objetos e componentes do jogo, cenas e o modelo de aplicativo que gerencia esses objetos.

O `GingaGame.GameComponent` possui um conjunto de componentes prontos para uso. Esses componentes devem ser adicionados a objetos em uma cena do jogo. Entre os componentes desenvolvidos estão: `AnimatedSprite` (que permite o desenho de imagens animadas), `StaticSprite` (para desenhar imagens estáticas), e `BoundingBox` (para controle de colisão usando retângulos).

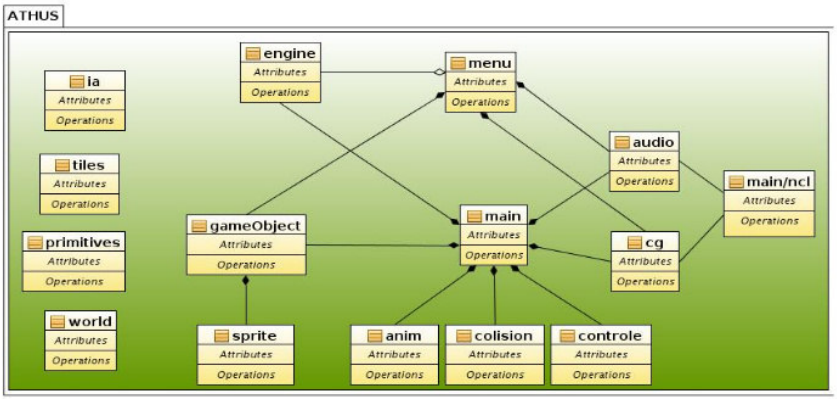
O último pacote, `GingaGameJavaTV`, inclui todas as classes específicas da plataforma.

Um exemplo de recurso específico da plataforma é o gerenciador de janelas. JavaTV usa a classe `HScene` (do pacote HAVi) para acessar a janela do aplicativo. Essas classes são separadas das demais classes do *framework* para tornar fácil a mudança de plataforma de execução do jogo quando necessário. Dessa forma, somente o código específico da plataforma deve ser modificado, mas as interfaces permanecem as mesmas.

2.5 ATHUS

ATHUS (SEGUNDO; SILVA; TAVARES, 2010) é uma *framework* genérica que oferece suporte para desenvolvimento de jogos para TVDi. Ela age como uma API de alto nível para desenvolvedores de Ginga-J e como módulos para os desenvolvedores de Ginga-NCL. Como se pode ver na Figura 5, há duas versões de implementações da ATHUS,uma baseada em Java e outra baseada em Lua para o uso com Ginga-NCL. Os módulos desenvolvidos proveem suporte básico para autoria de jogos.

Figura 5 – Arquitetura do Motor de Jogos ATHUS



Fonte: (SEGUNDO; SILVA; TAVARES, 2010)

O módulo do Engine atua como interface para o acesso da TV, ou seja, ela simplifica toda a chamada necessária para imprimir uma imagem na TV, ou para obter informações a partir desta, como a resolução da TV. O módulo Menu facilita o trabalho de criação de menus selecionáveis, e consegue criar submenus. O módulo Main implementa a lógica de jogo, além de conectar todos os outros módulos. O módulo Game Object representa os objetos ativos na cena, como jogadores, NPCs, baús e itens. Objetos estáticos são representados por outros módulos. O módulo Sprite controla animações em imagens segmentadas. O módulo Audio se responsabiliza por gerar código NCL para tocar sons e vídeos, devido a estas funcionalidades não estarem disponíveis para acesso com Java ou Lua. O módulo Anim controla a velocidade de atualização dos *frames* do jogo, ou seja, FPS. O módulo Colisão detecta, como o nome diz, colisões entre os objetos da cena.

O módulo Control é responsável por receber e processar a entrada de dados do usuário, apesar de ser o módulo Main que reage a essas entradas.

2.6 Quadro Geral

Ao analisar esses 5 motores de jogos, percebemos que implementam funcionalidades em comum, e outras únicas a cada um. O Unity Engine deve ser considerado um modelo a ser seguido por sua capacidade de simplificar múltiplas tarefas e ainda prover uma interface de usuário eficaz, enquanto que o Game Maker se destaca pela facilidade de desenvolvimento de jogos 2D, sendo utilizada para ensinar pessoas novas na área. Ambos TuGA e Ginga Game apresentam uma framework para desenvolvimento de jogos para TV Digital com Ginga, porém utilizando-se Java. Apenas ATHUS apresenta um modelo para desenvolvimento utilizando NCL, ao mesmo tempo que tem também suporte a Java, apesar de não ser possível fazer um desenvolvimento paralelo dos dois modelos.

Os pontos em comum nesses motores são a forma de tratamento de imagem, onde, em geral, utilizam todos o conceito de sprite para gerenciar os gráficos, diferenciando-se o Unity Engine que modifica seu plano 3D para um plano ortogonal para mostrar sprites em um modelo 2D; tratamento de som, onde a maioria, de acordo com sua plataforma alvo, dá a possibilidade de tocar sons com o hardware, tendo-se um destaque para os motores da plataforma Ginga que ficam à merce do middleware para poderem executar sons no dispositivo; tratamento de entrada de dados, onde cada um dispõe de métodos para tratar o apertar de botões, de acordo com a plataforma. Controle de lógica de jogo é feito por todas, dando liberdade ao desenvolvedor de se focar no relacionamento dos objetos do jogo em si, e vez da estrutura básica que faz tudo funcionar.

Estes trabalhos relacionados mostram duas coisas: nos casos dos motores que não compilam para Ginga, as funcionalidades que foram bem recebidas pela comunidade de desenvolvedores, enquanto que as que compilam para Ginga demonstram o que já é possível realizar utilizando o *middleware*. O Ginga Wings deverá seguir estas informações para entregar um motor de jogos que atenda aos requisitos estabelecidos nos objetivos.

3 Ginga Wings

3.1 Arquitetura do Software

O Ginga Wings (GW) é um motor especializado em jogos com gráficos 2D, com o propósito de ser facilmente adaptado - para os casos em que sua estrutura atrapalhar, de alguma forma, o desenvolvimento do jogo -, e também extensível por dentro de suas funcionalidades, ou seja, o usuário poderá melhorar seu funcionamento interno para adicionar novas funcionalidades.

A plataforma escolhida para o desenvolvimento do motor foi Ginga-NCL. NCL (Anexo A) é uma linguagem de marcação assim como HTML, e de extremo alto nível, que por si só permite a criação de aplicações de hipermídia para TV Digital. Por ser simples de usar, como HTML, é uma ferramente apropriada para a autoria de aplicações de hipermídia, já que não é necessário um treinamento para lógica de programação para poder manipulá-la, permitindo que usuário sem muito conhecimento na área, como designers visuais, criem suas próprias aplicações. A linguagem de scripting embarcada com Ginga-NCL é Lua (Anexo B), que é por si só uma linguagem extremamente flexível de uso, o que faz dela uma candidata perfeita para conter o código base do motor.

Ele se utiliza de boa parte da API de NCLua (ITU, 2009) para prover funcionalidades através do *Set-Top Box*[1]. Ele roda[2], inteiramente, com código Lua e se comunica com o Documento NCL nativamente, sendo essa uma de suas principais características. Apesar do desenvolvimento ser voltado para *Set-Top Boxes*, como o Ginga também roda em dispositivos móveis, o motor pode ser utilizada para jogos nestes dispositivos.

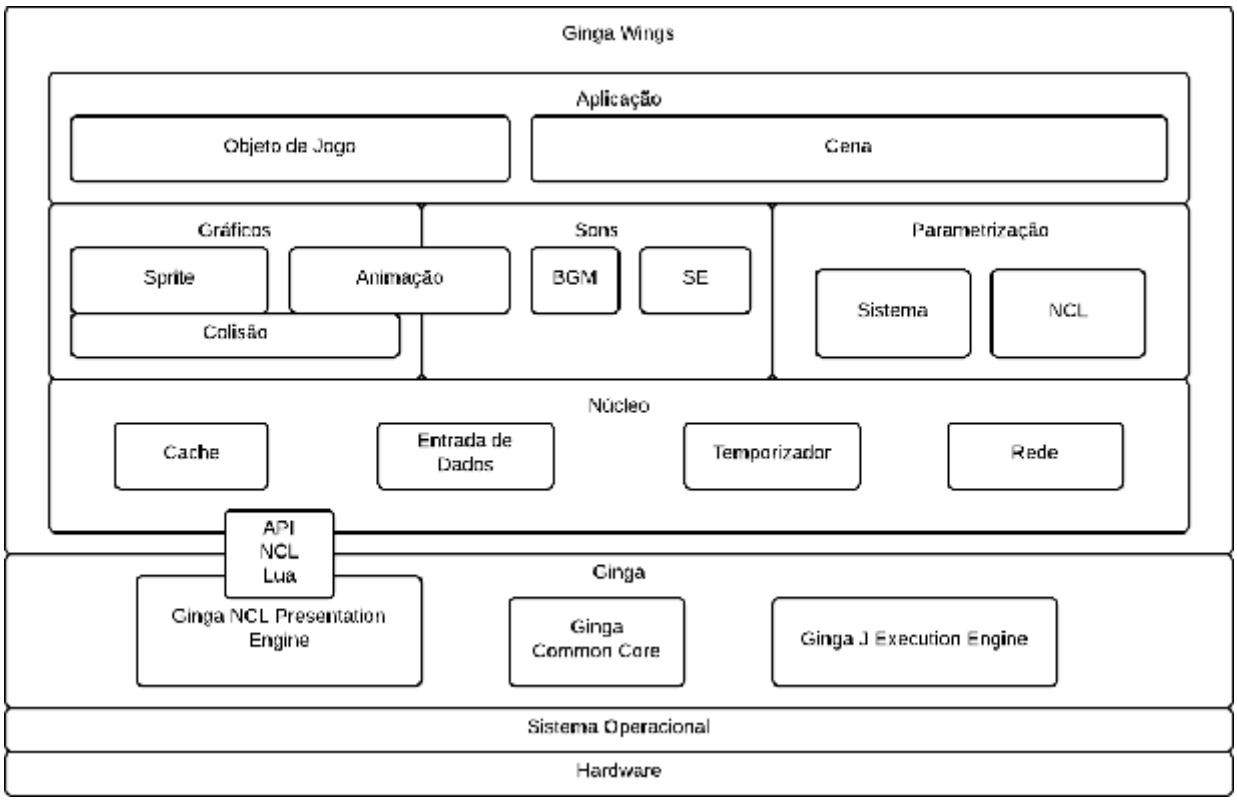
É importante destacar a comunicação com o código NCL. Através dessa funcionalidade, é possível passar qualquer tipo de parametrização para o código Lua. Dentro das inúmeras possibilidades de configuração pelo NCL, é possível ajustar o adaptar o motor a ponto de permitir a configuração completa de um jogo a partir de NCL somente. No Apêndice A, temos o exemplo de um jogo criado usando a peso esta comunicação, a ponto de ser possível configurar uma fase do jogo implementado somente com parâmetros NCL. Pelo fato do mesmo ter sido desenvolvido em uma versão antiga, não correspondente à atual, do motor Ginga Wings, foi colocado em um tópico a parte.

A visão geral da arquitetura do motor pode ser observada na Figura 6, com detalhamentos nos próximos tópicos.

[1] Equipamento que se conecta a um televisor e a uma fonte externa de sinal, convertendo o conteúdo do sinal para a saída de video, em alguns casos permitindo interação.

[2] Termo técnico para executar um software em uma plataforma

Figura 6 – Arquitetura Ginga Wings



Fonte: Produzido pelo Autor.

3.1.1 Hardware

Hardware é o componente físico no escopo de uma aplicação, responsável por executar o código gerado por ela. Geralmente será um Set-Top Box, mas pode ser um Dispositivo Móvel. Não é necessário detalhamento uma vez que a função do S.O. e do Middleware Ginga é deixar o tratamento do Hardware transparente.

3.1.2 Sistema Operacional

O Sistema Operacional é o principal sistema executado no Hardware, na maioria dos dispositivos. Normalmente em PCs, o SO se ocupa de gerenciar os recursos de Hardware, distribuindo-os para as aplicações. Por isso, em um PC, é de se esperar que nenhuma aplicação tenha total posse dos recursos de Hardware. Dispositivos Móveis e Set-Top Boxes, por outro lado, por terem uma função definida (contraposta ao SO de PC, que tem propósito geral), geralmente alocam praticamente todos os recursos à aplicação. A implementação de referência utiliza o Linux como Sistema Operacional padrão, pelo fato do mesmo ser um Software Livre.

3.1.3 Ginga

Figura 7 – Visão Geral do Middleware Ginga

Fonte: (ITU, 2009)

O middleware é a camada de software que executa entre o sistema operacional e as aplicações. No caso específico do Sistema Brasileiro de TV Digital (SBTVD), o middleware Ginga foi desenvolvido. A Figura 7 apresenta uma visão geral da organização do Ginga.

Como pode ser visto, ele é formado por diversos subsistemas, os quais foram implementados usando a linguagem C++. O núcleo comum do Ginga (CommonCore) implementa as funcionalidades principais do middleware, e acima dele temos uma camada de serviços específicos que consiste na implementação das APIs utilizadas pelas aplicações que executam sobre o middleware. Dois tipos de aplicações são suportadas pelo middleware brasileiro, as aplicações NCL (Ginga-NCL) e as aplicações Java (Ginga-J). Neste cenário, as aplicações acessam suas APIs específicas e estas APIs utilizam o núcleo comum para efetuarem suas operações.

Em especial, ignoramos a Máquina de Execução em Ginga-J, e consideramos somente a de Apresentação, Ginga-NCL. Esta, em especial, possui o que chamamos de Objetos Imperativos NCLua, o que nada mais é do que a comunicação nativa existente entre o NCL e a Linguagem de scripting Lua. A criação, a ativação, pausa e encerramento do objeto NCLua são todos gerenciados a partir do NCL.

O Middleware disponibiliza uma API que expõe métodos úteis para que o código Lua possa se utilizar dos recursos do sistema, tais como métodos para acesso aos pixels da tela reservada ao script, envio e recebimento de eventos e nós NCL, envio e recebimento de pacotes de rede pela internet, captura de entrada de dados da parte do usuário, entre

outros. O motor se baseia fortemente nestes métodos expostos para a implementação de suas funções.

3.1.4 GingaWings

Por fim, chegamos ao componente Ginga Wings. O primeiro nível da arquitetura é o grupo Núcleo, que possui os componentes Cache, Entrada de Dados, Temporizador e Rede. O componente Cache é responsável pelo gerenciamento básico de memória para sons, músicas e imagens, garantindo que um recurso não será várias vezes carregado na memória, e que recursos não utilizados pelo jogo sejam devidamente descartados da memória. O componente Entrada de Dados fornece várias formas de detecção de entrada do usuário, enriquecendo a API originada do Ginga. O componente Temporizador se utiliza da API de eventos do Ginga para garantir que o sistema será executado sempre em tempo constante. E por fim o componente de Rede oferece uma forma amigável de se enviar e receber pacotes via web.

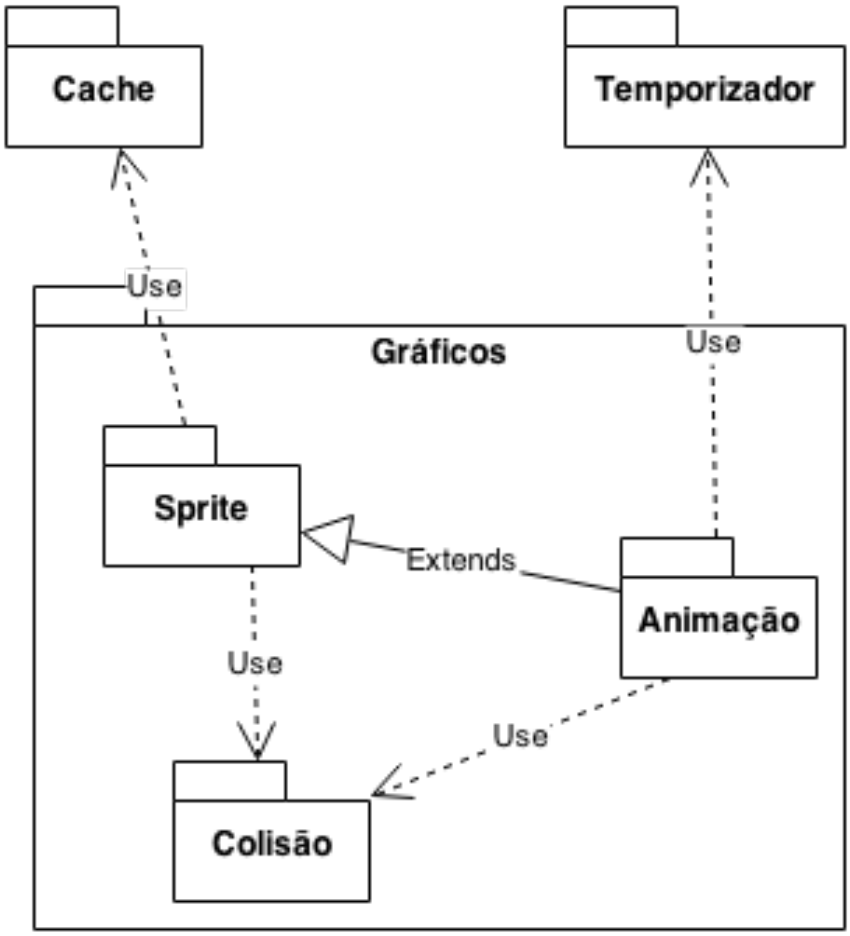
Gráficos

O componente Gráficos pode ser detalhado pela Figura 8, na forma de pacotes. A estrutura básica do pacote é o `Sprite`. O `Sprite` é uma representação da posição, alargamento (zoom), e rotação de uma imagem, mas não é uma imagem em si. A imagem fica armazenada no `Cache`, de quem o `Sprite` depende. Um `Sprite` pode ter um objeto de `Colisão` associado a ele. O GW automaticamente detecta esta colisão e dispara eventos de caso este `Sprite` colida com outros. O pacote `Animação` é uma sequência de `Sprites`, gerenciada pelo `Temporizador`. Ela pode ter seu próprio objeto de colisão à parte do `Sprite`. A animação geralmente apenas muda qual `Sprite` está sendo exibido no momento, sendo a mudança gerenciada por um tempo t, o qual o `Temporizador` pode oferecer. A `Animação` também possui sua própria localização, alargamento e rotação à parte dos seus sprites.

Audio

O componente Audio, conforme ilustrado na Figura 9, também possui dependência com o `Cache`. Diferentemente do que ocorre com imagens, não há uma exposição de métodos para manipulação de sons diretamente. Por isso é preciso utilizar-se de eventos NCLua para que algum som seja tocado através do Documento NCL, não do código Lua. O objetivo do `Cache` neste caso é gerar estes eventos. Mais detalhes na seção de API. Existem dois tipos de Áudio, BGM e SE. BGM signifca Backgroung Music, ou música de fundo. Basicamente são músicas que tocam em loop infinito enquanto o jogador manipula o jogo. SE significa Sound Effects, ou Efeitos Sonoros, que são podem ser descritos como onomatopeias na gramática. As SEs são utilizadas pela `Animação` para tocar sons sincronizados.

Figura 8 – Diagrama de Pacotes. Ginga Wings – Gráficos

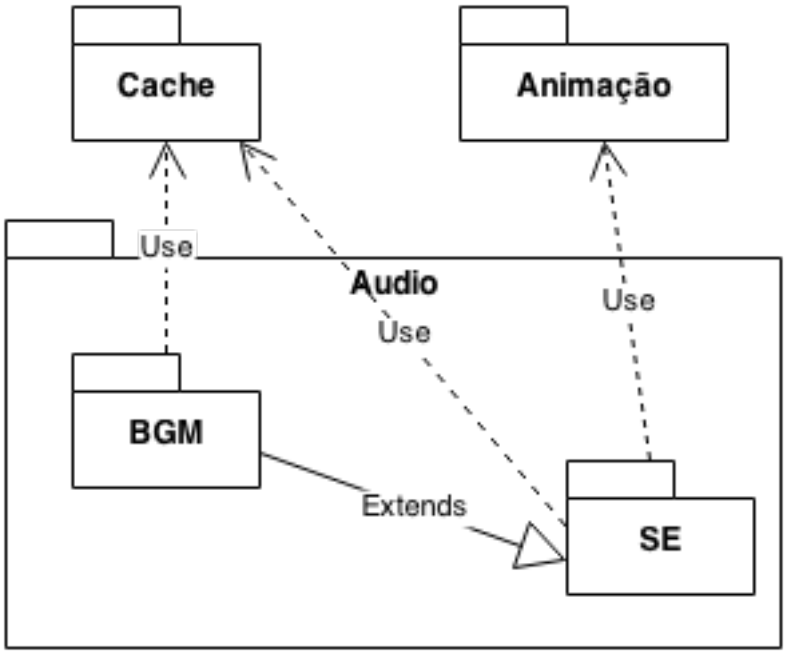


Fonte: Produzido pelo Autor

Parametrização

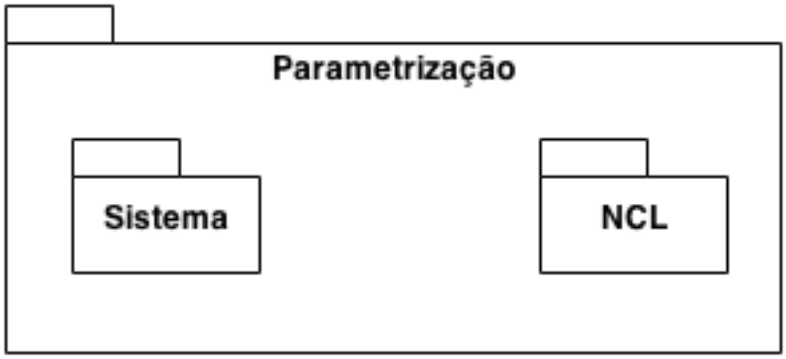
A Parametrização é uma das principais características do sistema: permite que o usuário, a partir de um documento NCL, envie parâmetros para o código Lua, configurando e controlando o jogo a partir do próprio NCL. Utilizando-se de estruturas como Âncoras e Links, é possível criar um motor básico de jogos e orientá-lo totalmente pelo seu código NCL: por exemplo, é possível definir uma fase inteira de um jogo como Super Mario World utilizando NCL e NCLua. O código Lua teria o motor básico, com os eventos, cenários, comportamentos de inimigos e itens, etc, enquanto que o código NCL seria o responsável por dizer como a fase seria desenhada, quais inimigos aparecem nela, e onde aparecem. Ele pode utilizar âncoras para determinar o fim de jogo caso o tempo termine, por exemplo. É preciso, no entanto, que o código Lua esteja pronto para receber e interpretar estes parâmetros. A parametrização NCL captura esses parâmetros e os envia para dentro do código Lua, onde classes especializadas devem estar prontas para recebê-los e processá-los. A parametrização de Sistema possui toda e qualquer configuração geral do jogo: qual a

Figura 9 – Diagrama de Pacotes. Ginga Wings – Audio



Fonte: Produzido pelo Autor

Figura 10 – Diagrama de Pacotes. Ginga Wings – Parametrização



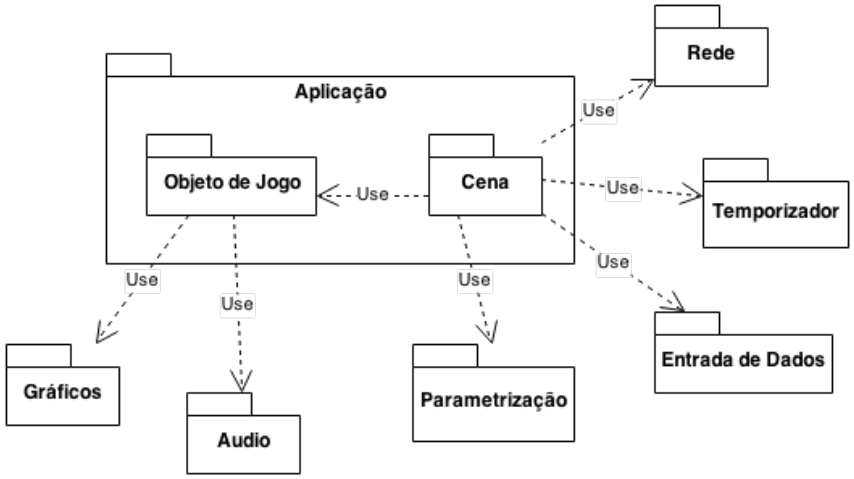
Fonte: Produzido pelo Autor

resolução base que ele executa, FPS aceitável, entre outras configurações, que também podem ser adaptadas para as necessidades do jogo.

Aplicação

Por fim temos o componente de Aplicação, que utiliza todos os outros componentes. Seu principal pacote é a Cena, que junta os componentes de Entrada de Dados, Temporizador, Rede e Parametrização em um só lugar. O Temporizador determina, neste pacote, quantas vezes ele será atualizado por segundo. A Cena não interage diretamente com

Figura 11 – Diagrama de Pacotes. Ginga Wings – Aplicação



Fonte: Produzido pelo Autor

Músicas e Imagens, agindo através de Objetos de Jogo, estes sim, que encapsulam toda a estrutura de Mídia.

3.2 Detalhamento da API

A criação de jogos geralmente envolve códigos grandes e complexos. Alguns jogos mais simples podem ter apenas algumas dezenas de linhas de código os definindo completamente, desde a aplicação gráfica, a entrada de dados e as respostas do jogo. Porém, para a maioria dos casos um jogo é uma série de códigos grande e complexa. Com o crescimento de qualquer tipo de código, sua manutenção tende a ficar cada vez mais complicada. Uma forma de amenizar essa complexidade é o emprego de orientação a objetos (OO), que permite uma maior organização do código ao agrupar o funcionamento do sistema em objetos, suas possíveis ações e seus relacionamentos com outros objetos (LEWIS; LOFTUS, 2008). Com isto em vista, tornou-se uma necessidade a implementação de tal funcionalidade, já que Lua não dá suporte a OO, sendo na verdade uma linguagem de protótipos. O uso desta linguagem, mesmo com este revés, ainda se justifica por facilitar a flexibilização do código, por se tratar de uma linguagem de scripting.

Código 3.1 – Exemplo de Criação de Classe

```
1.      GW_Codes.classes.Block = {}
2.      GW_Codes.classes.Block.superclass = "OBJECT"
3.      function GW_Codes.classes.Block:init()
4.              GWClasses.Block().__superclass().init(self)
5.      end
```

O GW disponibilizar uma estrutura para emular a funcionalidade de classes de OO, permitindo definir uma classe, criar instâncias dela, e também ter herança não múltipla. É extremamente fácil criar uma classe utilizando este módulo, como mostra o exemplo no Código 3.1.

Lua funciona utilizando tabelas e metatabelas, que são os objetos nativos responsáveis por emular o funcionamento de classes para a GW. A linha 1 cria uma tabela que armazenará a estrutura da classe `Block`. A segunda linha define qual é a classe pai da classe `Block`. No caso, a classe `OBJECT`, que é a classe padrão, já implementada na GW, que disponibiliza todas as funcionalidades básicas da GW. Todas as classes devem herdar de `OBJECT`. É possível trabalhar sem ela, porém será mais difícil tirar proveito do que a GW tem a oferecer. A linha 3 define uma função que é inicializada toda vez que um objeto dela é instanciado. A linha 4 faz o equivalente a super em linguagens orientadas a objeto, permitindo que a classe pai inicialize seus atributos e métodos junto com a inicialização da classe filha. A passagem do parâmetro self é obrigatória para que a classe pai possa alterar a metatabela da filha. A linha 5 termina a função.

Para criar uma instância da classe `Block`, basta chamar o método new.

Código 3.2 – Instanciando uma Classe

```
1.      GW_Classes['Block'].new()
```

O método criará uma instância da classe `Block` e passará a referência para a variável block. A tabela `GW_Classes` guarda uma referência para todas as classes instanciáveis – a exemplo, a classe `OBJECT`. O nome da classe será igual ao passado para a variável `GW_Codes` no Código 3.1.

Class:__class() : table

Retorna a estrutura de classe do objeto em questão.

Class:__classname() : String

Retorna o nome da classe do objeto.

Class:__superClass() : metatable

Retorna a estrutura de classe da classe pai do objeto.

Class:__is_a(string: nomedaclasse) : boolean

Verifica se o objeto é da mesma classe que a informada no parâmetro. Neste caso, verifica se os nomes são iguais.

Class:__is_a(table: classe) : boolean

Verifica se o objeto é da mesma classe que a informada no parâmetro. Neste caso, verifica se a metatable é igual à metatabela de classe do objeto.

Class:__respond-to(string: método) : boolean

Verifica se o método passado existe na classe em questão. Por ser uma tabela simulando uma classe, é possível que um objeto tenha mais funções visíveis do que a classe disponibiliza. Estas funções não são inspecionados por esta função.

Com a definição de classes já feita, é possível prosseguir para a descrição dos componentes da API. Ao fim do capítulo, é apresentado um projeto básico que visa mostrar como utilizar o GW em conjunto com NCL.

3.2.1 Game_Object

Como mencionado, existe uma classe base, chamada simplesmente de OBJECT. Ela disponibiliza as funções básicas do GW, por isso é recomendado que qualquer objeto de jogo herde desta classe. Um objeto que herde de OBJECT automaticamente terá acesso a disparadores de eventos (por exemplo, disparar um evento quando um objeto Block tocar em outro), terá acesso a um comportamento, invocado automaticamente, desenho na tela, automático, funções de acesso à rede e de destruição do objeto. A classe objeto dá a seus filhos acesso a um sprite, uma posição na tela, e uma tabela de variáveis locais.

Object:update() : boolean

É a função principal da classe, chamando sua função de comportamento, gerenciando seus eventos e também o desenhando na tela. Caso o objeto tenha sido desposado, não realiza nenhuma dessas ações. Retorna true caso consiga realizar todas as suas operações.

Object:dispose() : boolean

Desposa um objeto, ou seja, este objeto não será mais considerado como ativo pela GW. Retorna true caso consiga desposar o objeto.

Object:draw()

Desenha o objeto na tela.

Object:__create_event(string: event)

Cria um evento como o nome event. O evento criado é uma tabela, que possui a entrada command, uma função que é executada quando o evento é disparado, e outra

entrada result, que guarda o resultado do evento gerado para que possa ser devidamente tratado.

Object:___consume_event(string: event)

Faz uma chamada à entrada command do event parâmetro caso ele tenha sido disparado. Retorna true caso tenha sucesso. Deve ser invocado por funções interessadas em obter uma funcionalidade do evento, por exemplo, um objeto Gato iria chamar o evento Rato:___consume_event('saiu_da_toca') e ler, em seguida, Rato.events['saiu_da_toca'] para verificar se o rato saiu da toca para que ele possa então caçá-lo. Um evento pode ser consumido até que reinicie o loop. O método command é invocado todas as vezes, portanto é preciso ser cauteloso ao codifica-lo.

Object:_____fire_event(string: event)

Dispara o evento event. Todos os objetos que invocarem a função ___consume_event() poderão obter os dados do evento gerado. Deve ser chamado pelo próprio objeto. Aproveitando o exemplo do gato e do rato, é o objeto rato que define Rato.___fire_event('saiu_da_toca'), para que enfim o gato possa invocar o método ___consume_event() e correr atrás do rato fujão.

3.2.2 Timer

Este módulo, Game_Timer, é o responsável por controlar os eventos temporais na aplicação, implementado o Temporizador. Ela armazena o tempo geral da aplicação, a taxa de frame rates (que determina quantas vezes por segundo a tela é renderizada, e portanto, o sistema é atualizado) esperada pela aplicação, e o tempo entre um loop e outro. Por padrão, a taxa de frame rates é 20, tendo em vista que o Set-Top Box não foi projetado para rodar aplicações pesadas, e o olho humano conseguir distinguir imagens fluídas em até 18 frames por segundo. Por causa disso não é possível criar animações muito fluídas sob a configuração padrão. Ela pode ser alterada de acordo com as necessidades do usuário, porém a performance do sistema como um todo deve ser levada em consideração.

Game_Timer:sync(number: total_time) : number

Existem duas maneiras de se programar eventos temporais com frame rate: utilizando o frame rate como unidade de tempo; e utilizando o frame rate como medida de tempo. O primeiro cria uma estrutura temporal Dependente do Frame Rate (DFR). Se a taxa de frames for normal (usemos nosso exemplo de 20 frames por segundo), a aplicação irá executar em velocidade normal. Se, por algum motivo, a aplicação contiver um processo pesado e o frame rate cair para 10, a aplicação irá rodar duas vezes mais lento. Por outro lado, caso a aplicação receba atenção total do processador e conseguir

rodar a 30 frames, o jogo irá rodar mais rápido. A segunda maneira, mais conhecida como Jogo em Tempo Real ou Independente de Frame Rate (IFR), se utiliza do intervalo entre cada loop do temporizador para calcular quanto tempo realmente passou, atualizando todos os objetos de jogo de acordo com o tempo passado. Desta forma, o jogo mantém sua velocidade de execução mesmo com a variação de frames por segundo – que por outro lado gera "paralizações" no jogo, geralmente chamadas de lag. Por outro lado, algumas aplicações ficam mais complexas, por exemplo, a detecção de colisão – é necessário realizar previsões dos movimentos dos objetos da cena para verificar se eles colidiram durante o intervalo de tempo. Com o devido cuidado, ambas as formas de manipular o tempo são viáveis, e o GW dá suporte a ambas.

Código 3.3 – Exemplo de uso da função sync

```
1.      pixels = Game_Timer:sync(20) --pixels = 20
```

A DFR é como a GW funciona normalmente. Para se trabalhar com a IFR, existe esta função no módulo Timer. Ele recebe um tempo e infere o quanto passou dele no intervalo entre os frames. O parâmetro deve indicar o movimento, através do tempo, por 1 segundo, sob o qual o método vai trabalhar. Por exemplo, se o Herói do jogo se movimenta a 20 pixels por segundo, em um jogo rodando a 20 FPS (frames por segundo), com a aplicação desta função, ele andará 1 pixel por segundo, a cada frame. Se por algum motivo a velocidade do jogo cair para 14 FPS, a velocidade do Herói será de 1.42 pixels por segundo.

3.2.3 Graphics

Implementando o componente Gráficos da arquitetura, é responsável pela renderização dos gráficos na tela. É possível informar uma ordem para que a renderização seja realizada. A GW possui uma resolução, nativa, que deve ser informada para o módulo. O jogo é renderizado na resolução nativa e após a renderização, se ajusta ao tamanho da tela da TV.

Game_Graphics:insertLayer(number: n, table: obj)

Caso o objeto `obj` possua o método `draw`, que é chamado para realizar a renderização do mesmo, ele é incluído na lista de objetos a serem renderizados pelo módulo. Objetos com um `n` menor são desenhados primeiro, portanto os que tiverem os maiores valores ficarão por cima no resultado final. Não há garantia de ordenação para objetos com valores `n` iguais.

Código 3.4 – Definindo a ordem de renderização para a classe Board

```
1.      GW_Codes.classes.Board.drawLayer = 1
2.      Game_Graphics:insertLayer(4, board)
```

Para escolher um valor `n` padrão para todos os objetos de uma classe, basta seguir o exemplo no Código 3.4, linha 1. Valores individuais podem ser aplicados, também, como mostrado na linha 2.

3.2.4 Sprite

Módulo básico, que armazena os gráficos da aplicação, implementado os componentes Sprite e Animação da arquitetura. Dá suporte tanto a gráficos estáticos quanto animados. Possui também suporte para física básica, na forma de colisões.

Sprite.new(string: path) : table

Cria uma instância da classe Sprite com a imagem referenciada pelo caminho path. Caso não encontre a imagem cria uma Sprite sem imagem. O Sprite criado não é subdividido, tem animação ativado (mas não executa nenhuma), e tem uma área de colisão de acordo com o tamanho da imagem.

Sprite:set_crop(number: w, number: h)

Determina em quantos pedaços a imagem será repartida. Esta partição não altera o arquivo de imagem utilizado pelo sprite, apenas determina pedaços de tamanho igual para serem desenhados com a função `draw()`. Qual parte será renderizada deve ser determinada pelo atributo `anim_index`. O `index` é definido a partir do número de linhas e colunas. Por exemplo, se $w = 4$ e $h = 4$, o $index = 9$ representará a parte da imagem localizada na linha 3, coluna 1.

Código 3.5 – Exemplo de uso da função set_crop

```
1.      board = Sprite.new("board.png");
2.      board:set_crop(4,4)
3.      board.anim_index = 6
```

Sprite:setup_frames(number: index, number: steady, string: sound, table: offset)

Permite configurar um frame de animação para o Sprite. Index informa qual o índice da animação (os frames são organizados de acordo com esse índice); steady informa quantos milissegundos o frame permanecerá ativo (valores muito baixos podem tornar o frame quase indistinguível, valores muito altos pode "paralisar" a animação); sound informa que o frame deve tocar o som informado pelo campo no início do frame. Um valor vazio ou nulo terão o mesmo efeito, sem som. Offset é uma tabela, com valores x e y, indicando que o Sprite deve mover sua imagem mas sem mover a matriz. Ou seja, a imagem renderizada vai mudar sua posição de pintura na tela, mas o objeto Sprite manterá sua posição original.

Código 3.6 – Exemplo de uso da função setup_frames

```
1.      board = Sprite.new("board.png");
2.      board:setup_frames(0, 4, null, {x=0, y=0})
```

3.2.5 Cache

Módulo responsável por gerenciar os recursos de mídia da aplicação. Gerencia tanto recursos de áudio como de imagem. O gerenciamento de imagens é automático, porém o de áudio, por depender do NCL para ser feito, precisa de configurações mais cuidadosas.

Game_Resources.load_image(string: path)

Carrega a imagem referenciada no caminho path na memória e retorna uma referência para ela. Chamadas posteriores a este mesmo path apenas retornarão uma referência. Alterações no arquivo de imagem, portanto, afetarão todos os objetos que chamarem a referenciarem.

Game_Resources.load_background(string: path)

Invoca um recurso de imagem do caminho padrão "media\backgrounds\". É um método auxiliar com a única finalidade de facilitar a obtenção de recursos de imagem.

Game_Resources.load_character(string: path)

Invoca um recurso de imagem do caminho padrão "media\characters\". É um método auxiliar com a única finalidade de facilitar a obtenção de recursos de imagem.

Game_Resources.free_image(string: path)

Informa ao Cache que a imagem em path deve ser liberada da memória. O Cache vai registrar o pedido, porém a imagem só será removida se todos os recursos que pediram a imagem o liberarem.

Game_Resources.play_SE(string: label)

Envia uma requisição ao NCL para tocar o áudio label. SE é a sigla para Sound Effects (Efeitos Especiais). SEs são geralmente utilizadas para representar sons ambientes e de HUDs / sistema (Head Up Display, como geralmente são chamadas as telas de interface do jogo, como status, vida, dinheiro, etc). Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

Game_Resources.stop_SE(string: label)

Envia uma requisição ao NCL para parar o áudio label. Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

Game_Resources.pause_SE(string: label)

Envia uma requisição ao NCL para pausar o áudio label. Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

Game_Resources.play_BG(string: label)

Envia uma requisição ao NCL para tocar o áudio label. BG é a sigla para Background Sound (Sons de fundo), ou seja, canções que ficam tocando, em loop, enquanto a aplicação executa. Quando a canção termina sua execução ela é iniciada novamente, criando um loop. Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

Game_Resources.stop_BG(string: label)

Envia uma requisição ao NCL para parar o áudio label. Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

Game_Resources.pause_BG(string: label)

Envia uma requisição ao NCL para pausar o áudio label. Uma configuração especial para este tipo de som deve ser provida pelo arquivo .ncl que chamada o script do GW.

3.2.6 Scene

Módulo que centraliza e controla todos os outros. Os Game_Objects devem ser manipulados junto com a lógica do jogo, utilizando este módulo. Uma instância do GW só tem 1 objeto Scene, que deve gerenciar os vários recursos.

GWScene:begin()

Esta função deve ser utilizada para inicializar todos os objetos da aplicação que rodarão junto com a scene.

Código 3.7 – Exemplo de uso da função begin

```
1.      function GWScene:begin()
2.              self.vars.color = 'red'
3.              self.vars.board = GWClasses['Board'].new()
4.              self.vars.score = GWClasses['Score'].new()
5.              table.insert(self.objects, self.vars.board)
```

```
6.              Game_Graphics:insertLayer(self.vars.board.
                        drawLayer, self.vars.board)
7.              table.insert(self.objects, self.vars.score)
8.              Game_Graphics:insertLayer(self.vars.score.
                        drawLayer, self.vars.score)
9.              GW_Codes.classes.Block.createBlocks()
10.             self:createToy()
11.             Game_Resources.play_SE('playback')
12.     end
```

GWScene:behavior()

Esta função determina o comportamento da Scene, e as interações entre os objetos. A lógica de comportamento interna do objeto não deve ser implementada aqui. O consumo de eventos, do Game Object, deve ser aplicado aqui.

Código 3.8 – Exemplo de uso da função behavior

```
1.      function GWScene:behavior()
2.              if self.vars.toy:__consume_event('endGame') then
3.                      self:End()
4.              end
5.              if self.vars.toy:__consume_event('gotFired') then
6.                      self:deleteObject(self.vars.toy)
7.                      self:createToy()
8.              end
9.              if self.vars.board:
                __consume_event('fieldChanged') then
10.                     local ls = self.vars.board.
                events['fieldChanged'].result
11.                     self.vars.score:
                addScore(ls * 100 + (ls - 1) * 50)
12.             end
13.     end
```

GWScene:End()

Finaliza a Scene e a aplicação. Caso algum dado deva ser escrito na finalização, este método deve ser sobrescrito. O Quadro 8 mostra um exemplo da aplicação desta função.

GWScene:getObjects(string: className) : table

Obtém todos os objetos registrados na Scene com o nome de classe className. Útil para gerenciar os objetos na Scene durante as chamadas ao behavior.

Código 3.9 – Exemplo de uso da função getObjects

```
1.      blocks = GWScene:getObject('Block')
```

GWScene:deleteObject(table: obj)

Deleta o objeto da Scene.

Código 3.10 – Exemplo de uso da função deleteObject

```
1.      blocks = GWScene:getObject('Block')
2.      for k,v in pairs(blocks) do
3.              GWScene:deleteObject(v)
4.      end
```

3.2.7 Parameters

Tem como objetivo servir como a ponte configurável entre o NCL e o código Lua. Através destes objetos se passa argumentos para a aplicação Lua, que determinarão como a aplicação irá funcionar. Deve-se levar em conta que esta funcionalidade é opcional para a execução da aplicação Lua – um jogo pode rodar, tranquilamente, sem depender dos parâmetros vindos do NCL. Ela deve ser utilizada pelo desenvolvedor para permitir que o desenvolvedor NCL possa personalizar o jogo de acordo com suas necessidades.

GWInitializer:parse(id: string,par: string)

Método invocado no código principal, recebe todos os parâmetros enviados pelo NCL. Sua funcionalidade deve ser reescrita para atender às necessidades da aplicação. Deve-se levar em conta que, apesar de ser possível enviar parâmetros durante a vida útil da aplicação (Scene), os primeiros parâmetros são enviados antes da Scene iniciar, não existindo, portanto, objetos inerentes à Scene neste momento.

3.2.8 Collision

Tem a função de gerenciar as colisões entre objetos na cena. Um objeto colide com outro caso sua área toque na área do outro objeto. Existem várias abordagens para se obter este efeito. A primeira, mais simples, é a colisão de quadrados. Define-se um quadrado mínimo que cerca o objeto, e verifica-se se este colidiu com o outro.

Este tipo colisão é eficaz em jogos que não necessitam de muita precisão na colisão. Objetos com áreas muito pequenas, e similares à área do retângulo também podem utilizar este modo de colisão. Seu custo de processamento é baixo.

Para casos em que há uma necessidade de maior precisão, mas não o suficiente para gerar um custo de computação, é possível se utilizar de um segundo método de colisão, a colisão por círculos, que simplesmente detecta se o círculo que contém o objeto está

Figura 12 – Exemplo de Colisão por Área de Quadrado. Não há colisão entre os dois primeiros, mas há colisão nos subsequentes

Fonte: Produzido pelo Autor

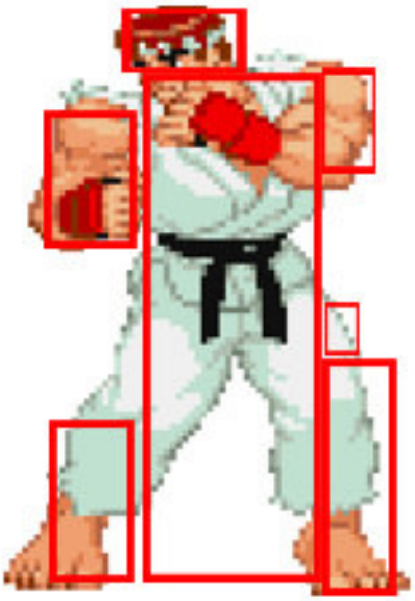
na área de um outro. Em jogos 3D, esse tipo de colisão é usada para objetos que não requerem muito controle de colisão, utilizando esferas no lugar de círculos.

Figura 13 – Exemplo de Colisão por Área de Círculo. Não há colisão entre os dois primeiros, mas há colisão nos subsequentes

Fonte: Produzido pelo Autor

Estes dois primeiros casos são suportados pela GW. São simples de implementar e possuem pouco custo. Existe uma terceira maneira, extremamente precisa, utilizada em jogos 2D (que geralmente tem seus gráficos desenhados com pixel-art), que é a Colisão de Pixels Perfeita. Este tipo de colisão lê a matriz de pixels do gráfico de ambos os objetos (não sendo necessário definir a área de colisão), e verifica, de acordo com a posição absoluta do objeto na tela, se algum dos pixels sobrescreve ao outro. Caso sim, existe uma colisão. Este tipo de colisão é custoso e não é oferecido por padrão pela GW. É possível, no entanto, estabelecer um algoritmo que simule a precisão da Colisão de Pixels Perfeita. A GW permite que se aplique várias áreas de colisão por objeto. Utilizando-se a Colisão por Quadrado, pode-se fragmentá-los para que cubram apenas a área desejada no gráfico, criando uma ilusão de Colisão de Pixels Perfeita. A desvantagem dessa abordagem é a configuração, geralmente custosa para o usuário que desenvolverá as animações, por ter que definir os quadros de colisão a cada quadro da animação.

Figura 14 – Exemplo de Configuração de Colisão com Múltiplos Quadrados



Fonte: Produzido pelo Autor

___collision_setup(offx: number, offy: number, wd: number, hg: number, frame: number, ct: number)

Cria uma configuração de colisão para objeto no frame especificado, com duração ct (em milissegundos), com tamanho wd x hg na posição offx vs offy. É possível criar várias caixas num mesmo frame para dar suporte ao modelo explicado acima.

left() : number

Retorna o limite mais à esquerda de todas as caixas de colisão no frame atual.

right() : number

Retorna o limite mais à dereita de todas as caixas de colisão no frame atual.

top() : number

Retorna o limite mais acima de todas as caixas de colisão no frame atual.

bottom() : number

Retorna o limite mais abaixo de todas as caixas de colisão no frame atual.

colide(obj: table) : boolean

Determina se o parâmetro obj possui caixas em colisão com as do objeto do qual a chamada foi realizada.

3.2.9 Input

Módulo responsável por controlar a entrada de eventos do controle, a fim de interagir com o jogo. Possui basicamente as entradas vermelho, verde, azul, amarelo, cima, baixo, esquerda, direita, menu, sair, sendo esses botões um padrão em controles de TVs Digitais. Todos os métodos retornam um boolean indicando se um botão foi pressionado ou não, diferenciando-se a quantidade de vezes quando o evento é disparado.

GW_Input.pressed(key: string) : boolean

Retorna true enquanto o botão key estiver pressionado. Útil para eventos repetitivos em jogos, como por exemplo aceleração em jogos de corrida.

GW_Input.trigger(key: string) : boolean

Retorna true apenas no frame em que o botão key é pressionado. Se o botão continuar pressionado, retorna false continuamente até que ele seja solto. Útil para eventos não repetitivos em jogos, por exemplo, o salto de um personagem de um jogo de side-scrooling.

GW_Input.repeat(key: string) : boolean

Retorna true repetidamente enquanto o botão key estiver pressionado, em pequenos intervalos.

3.3 Aplicação de Exemplo

Como exemplo de uso da ferramenta, será mostrado o passo-a-passo da criação de um jogo básico, um Hello World[3]. O jogo deverá apresentar o uso das principais funcionalidades da ferramenta. Como objetivos nesse exemplo, teremos:

- Apresentação de cenário de fundo e de fase;
- Um personagem controlado pelo controle remoto;
- Funcionamento de física (gravidade e colisões);

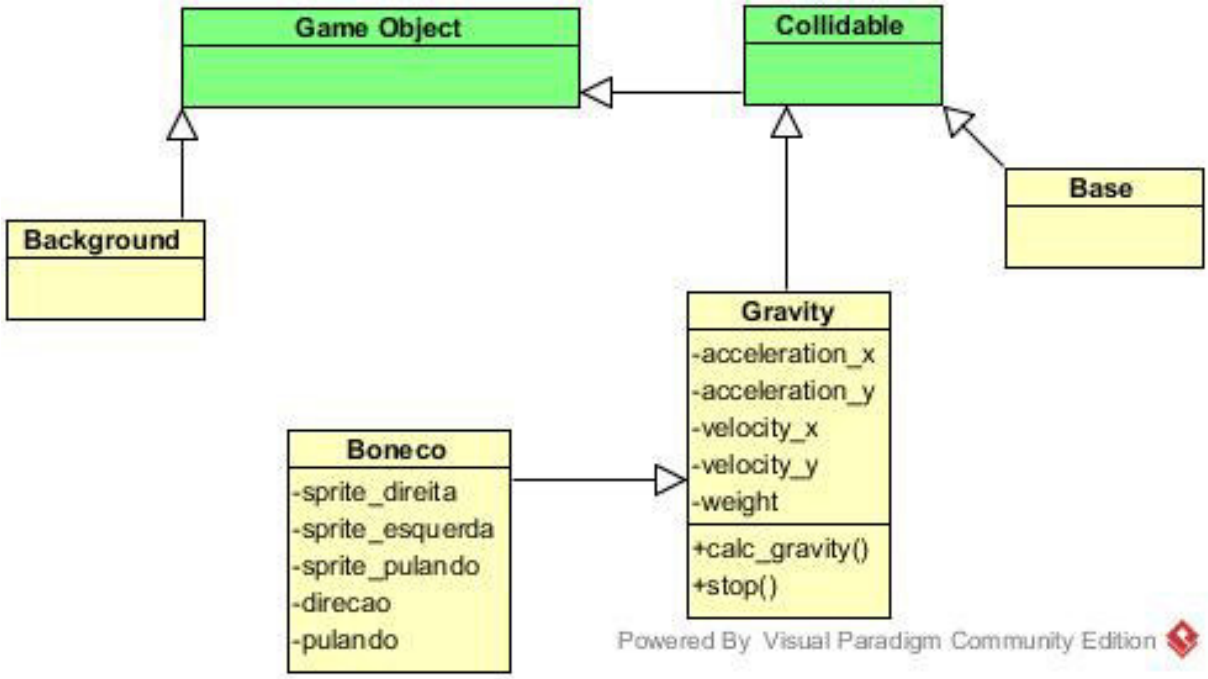
[3] Demonstração básica de uso de um software para primeiras experiências.

- Reprodução de BGM em loop.

Para programar o cenário de fundo e de fase, são necessárias duas classes diferentes. O cenário de fundo (Background) não possui interatividade com o jogador (apesar de em alguns tipos de jogos reagir a ele, como em cenários de fundo que utilizam a técnica de parallax scrolling), e, portanto é mais fácil de implementar. O cenário de fase (Base) interage com o jogador através de colisões, sendo o chão da fase. O jogador, se colidir com a face superior do cenário de fase, deve permanecer nela, não a atravessando.

O personagem de fundo deve responder ao botar vermelho do controle (ação de pular) e às setas esquerda e direita (movimento). Para programar o pulo é necessário implementar gravidade. Para a implementação de gravidade, o personagem deve ter peso, que será considerado 1. Enquanto o personagem não estiver em contato com a face superior de nenhum cenário de fase, deve receber uma aceleração negativa de -9.8m/s. Se ele pular, deve receber uma aceleração superior a 9,8 m/s a fim de alcançar uma altura desejada. A Figura 15 deve apresentar o diagrama de classes para o jogo de exemplo.

Figura 15 – Diagrama Entidade Relacionamento para a Aplicação de Exemplo



Fonte: Produzido pelo Autor

Todos os objetos da cena (Background, Boneco e Base) herdam de Game_Object, que possui os atributos básicos de um objeto de jogo. As classes Boneco e Base precisam reagir a colisões, portanto também herdam de Collidable (que por sua vez herda de Game_Object). A classe boneco precisa reagir à gravidade, portanto herda de Gravity, que gerencia informações quanto à gravidade.

A implementação da classe Background no Código 3.11, seguido da implementação da classe Base no Código 3.12, da classe Gravity no Código 3.13 e da classe Boneco no Código 3.14.

Código 3.11 – Classe Background

```
1.      GW_Codes.classes.Background = {}
2.      GW_Codes.classes.Background.superclass = "OBJECT"
3.      function GW_Codes.classes.Background:init()
4.              GWClasses.Background.__superClass().init(self)
5.              self.sprite =
                Sprite.new('/backgrounds/back1.png')
6.      end
```

Código 3.12 – Classe Base

```
1.      GW_Codes.classes.Base = {}
2.      GW_Codes.classes.Base.superclass = "COLLIDABLE"
3.      function GW_Codes.classes.Base:init()
4.              GWClasses.Base.__superClass().init(self)
5.              self.sprite =
                Sprite.new('/characters/base.png')
6.              self.pos.x = 0
7.              self.pos.y = 600-33
8.              self.vars.__collision_data.loop = true
9.              self:__collision_setup(0,0,800,33,100)
10.     end
```

Código 3.13 – Classe Gravity

```
1.      GW_Codes.classes.Gravity = {}
2.      GW_Codes.classes.Gravity.superclass = "COLLIDABLE"
3.      function GW_Codes.classes.Gravity:init()
4.              GWClasses.Gravity.__superClass().init(self)
5.              self.vars.accy = 0.0
6.              self.vars.accx = 0.0
7.              self.vars.velx = 0.0
8.              self.vars.vely = 0.0
9.              self.vars.weight = 1.0
10.     end
11.
12.     function GW_Codes.classes.Gravity:calc_gravity()
13.             local add_accel = {
                        x = Game_Timer:sync(self.vars.accx),
                        y = Game_Timer:sync(self.vars.accy) }
14.             self.vars.velx = self.vars.velx + add_accel.x
15.             self.vars.vely = self.vars.vely + add_accel.y
16.             self.pos.x = self.pos.x + self.vars.velx
17.             self.pos.y = self.pos.y + self.vars.vely
```

```
18.     end
```

Código 3.14 – Classe Boneco

```
1.      GW_Codes.classes.Boneco = {}
2.      GW_Codes.classes.Boneco.superclass = "Gravity"
3.      function GW_Codes.classes.Boneco:init()
4.              GWClasses.Boneco.__superClass().init(self)
5.              self.vars.sprites = {}
6.              self.vars.sprites.d =
                Sprite.new('/characters/mario_parado.png')
7.              self.vars.sprites.e =
                Sprite.new('/characters/mario_parado_m.png')
8.              self.sprite = self.vars.sprites.d
9.              self.pos.x = 800/2-16
10.             self.pos.y = 16
11.             self.vars.weight = 2.5
12.             self.vars.lado = 'd'
13.             self.vars.__collision_data.loop = true
14.             self:__collision_setup(9,7,15,19,100)
15.     end
16.
17.     function GW_Codes.classes.Boneco:behavior()
18.             GWClasses.Boneco.__superClass().behavior(self)
19.             local vel = 5.0
20.             self.vars.velx = 0.0
21.             if GW_Input.pressed('CURSOR_RIGHT') then
22.                     self.sprite = self.vars.sprites.d
23.                     self.vars.velx = vel
24.             end
25.             if GW_Input.pressed('CURSOR_LEFT') then
26.                     self.sprite = self.vars.sprites.e
27.                     self.vars.velx = -vel
28.             end
29.             local tempx = Game_Timer:sync(self.vars.velx)
30.             self.pos.x = self.pos.x + self.vars.velx
31.     end
```

As classes Background e Base são estáticas na tela, portanto é seguro aplicar-lhes um posicionamento estático por toda a vida da aplicação.

Tendo sido implementadas as classes de funcionamento básico do jogo, podemos passar para a criação da Scene que irá gerenciar a lógica e recursos. Será dever da Scene instanciar e destruir objetos, invocar BGMs, determinar o fluxo da lógica (muitas vezes chamada de Máquina Finita de Estados), etc. A GW provê uma Scene já pronta para uso, ela não deve ser instanciada. Em casos da necessidade do uso de mais de uma Scene, o

ideal é a implementação de uma nova classe que desempenhe a função, sendo controlada pela Scene provida pela GW.

No caso da aplicação de exemplo, a Scene vai instanciar o Backgroun, a Base e o Boneco. Vai tocar uma BGM em modo infinito (loop) e vai gerenciar os eventos disparados pela interação dos objetos. O código segue no Código 3.15.

Código 3.15 – Código da Cena

```
1.      function GWScene:begin()
2.              print("cena inicializada")
3.              self.vars.background =
                GWClasses['Background'].new()
4.              self.vars.mario = GWClasses['Boneco'].new()
5.              self.vars.base = GWClasses['Base'].new()
6.              table.insert(self.objects, self.vars.background)
7.              Game_Graphics:
                insertLayer(1, self.vars.background)
8.              table.insert(self.objects, self.vars.base)
9.              Game_Graphics:insertLayer(1, self.vars.base)
10.             table.insert(self.objects, self.vars.mario)
11.             Game_Graphics:insertLayer(2, self.vars.mario)
12.             Game_Resources.play_BG('playback', true)
13.             self.vars.gravity = 9.8
14.     end
15.
16.     function GWScene:behavior()
17.             local mario = self.vars.mario
18.             if mario:right() >= Game_Graphics.width then
19.                     mario.pos.x = Game_Graphics.width -
                (mario:frame().offset.x + mario:frame().width)
20.             end
21.             if mario:left() <= 0 then
22.                     mario.pos.x = -mario:frame().offset.x
23.             end
24.             mario:calc_gravity()
25.             if mario:collide(self.vars.base) then
26.                     mario.vars.accy = 0.0
27.                     mario:stop()
28.                     mario.pos.y = self.vars.base:top() -
                        (32-7)
29.             else
30.                     if mario.vars.accy < 0 then
31.                             mario.vars.accy =
                        mario.vars.accy + self.vars.gravity
32.                     else
33.                             mario.vars.accy =
                                self.vars.gravity
```

```
34.                     end
35.             end
36.             if GW_Input.pressed('CURSOR_UP') and
                        mario:collide(self.vars.base) then
37.                     mario.vars.accy = mario.vars.accy -
                        self.vars.gravity * 5
38.             end
39.     end
```

As linhas 2-11 criam os objetos na cena e os registram junto à Scene e ao módulo de gráficos (Game_Graphics). O registro com o Game_Graphics permite que o módulo reconheça os objetos como desenháveis e vai invocar o método draw() de cada objeto registrado para que apareça na tela. A linha 12 faz com que a BGM playback seja tocada. Na linha 13, setamos a gravidade global para o valor 9,8 (m/s).

Na função behavior o objeto mario (Boneco) tem sua gravidade setada para a global. Da linha 18-23 é feito o tratamento para quando o objeto tentar sair da tela pelos lados. Nas linhas 25-35 é feito o tratamento para quando o objeto colide com a base. As linhas 36-38 tratam o pulo do boneco, adiciona aceleração negativa (assim ele irá para cima).

Com a Scene pronta, é necessário agora configurar os sons. A configuração do som exige que se utilize o módulo Parameters. A GW não tem como tocar sons nativamente, nem saber sua duração. Serão utilizados parâmetros para se informar esses dados através do NCL. Apenas uma sobrescrita da função parse é necessária, como mostrado no Código 3.16.

Código 3.16 – Código dos Parâmetros

```
1.      function GWInitializer:parse(id,par)
2.              if string.match(par, "BGM%d+") ~= nil then
3.                      print("Registering BGM "..id)
4.                      local duracao =
                        string.match(par, "BGM(%d%d%d%d)")
5.                      local minutos =
                        tonumber(string.sub(duracao, 0, 2))
6.                      local segundos =
                        tonumber(string.sub(duracao, 3))
7.                      Game_Resources.
                        register_sound(id,
                                minutos * 60 + segundos)
8.              end
9.              if par == "End" then
10.                     print("BGM "..id.." ended")
11.                     Game_Resources.end_BG(id)
12.             end
13.     end
```

Nesta implementação, a função recebe um parâmetro informando o id da mídia no NCL e sua duração no parâmetro par. A GW reconhece que é um som caso contenha a palavra BGM seguido de 4 inteiros, que representam a duração do som em minutos e segundos. Essa duração é utilizada para saber quando o som terá terminado e mandar tocar de novo. Para que funcione, é preciso que o NCL disponibilize links para inicializar e finalizar uma mídia de som. Todo o trabalho do lado do código Lua está pronto. Agora é necessário criar o código NCL que irá invocar e parametrizar a aplicação Lua. O código segue nos Códigos 3.17 e 3.18.

Código 3.17 – Regiões e Descritores de Exemplo

```
<!-- Regioes -->
<regionBase>
    <region id="screen" width="100%" height="100%"  zIndex="5"/>
    <region id="music" width="100%" height="100%" zIndex="0"/>
</regionBase>
<!-- Descritores -->
<descriptorBase>
    <descriptor id="dsScreen" region="screen" focusIndex="240"/>
    <descriptor id="dsMusic" region="music">
        <descriptorParam name="soundLevel" value="1"/>
    </descriptor>
</descriptorBase>
```

Código 3.18 – Portas, Medias e Links

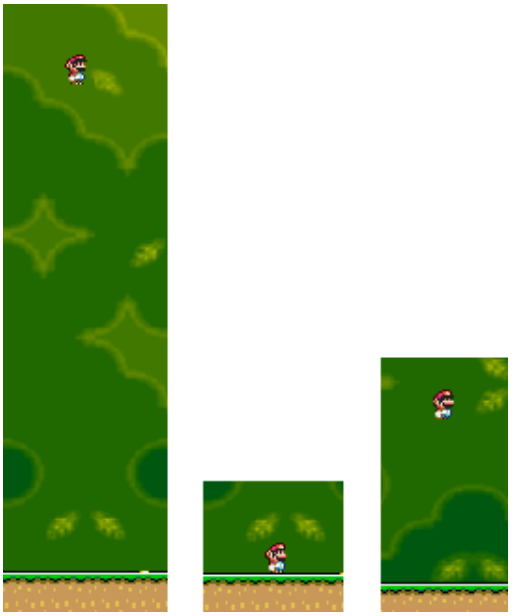
```
<!-- Porta -->
<port id="entryPoint" component="lua"/>
<media id="lua" src="codes/main.lua" descriptor="dsScreen">
    <property name="start1"/>
    <property name="playback"/>
</media>
<media id="playback" type="audio/mp3"
        src="media/backtracks/teste.mp3"/>
<media id="programSettings" type="application/x-ginga-settings">
    <property name="service.currentKeyMaster" value="240"/>
</media>
<!-- DEFINICAO DOS LINKS AQUI!!!!!!!! -->
<link xconnector="onBeginSetN">
    <bind role="onBegin" component="lua" />
    <bind role="set" component="lua" interface="start1"/>
    <bind role="set" component="lua" interface="playback">
        <bindParam name="var" value="BGM0036"/>
    </bind>
</link>
<link xconnector="onEndAttributionStart">
```

```
    <bind role="onEndAttribution" component="lua"
        interface="playback" />
    <bind role="start" component="playback" />
</link>
```

É criada a mídia playback com o som a ser reproduzido. São inseridas propriedades na mídia Lua, que têm finalidades específicas: a propriedade start1 informa à GW que ela deve esperar por 1 parâmetro pelo menos (além do próprio start1). Isso ocorre porque a passagem de parâmetros entre o NCL e o Lua não segue a ordem estabelecida no código XML e em alguns casos, quando há muitos links, alguns deixam de ser enviados. O parâmetro playback será o id do som na GW. Este parâmetro deve ser setado com o valor BGM0036. Como dito acima, BGM diz à GW (neste exemplo) que o parâmetro é um som, enquanto que 0036 informa que o som tem a duração de 0 minutos e 36 segundos. Com esta configuração, está pronta o Hello World da GW, apresentado na Figura 16.

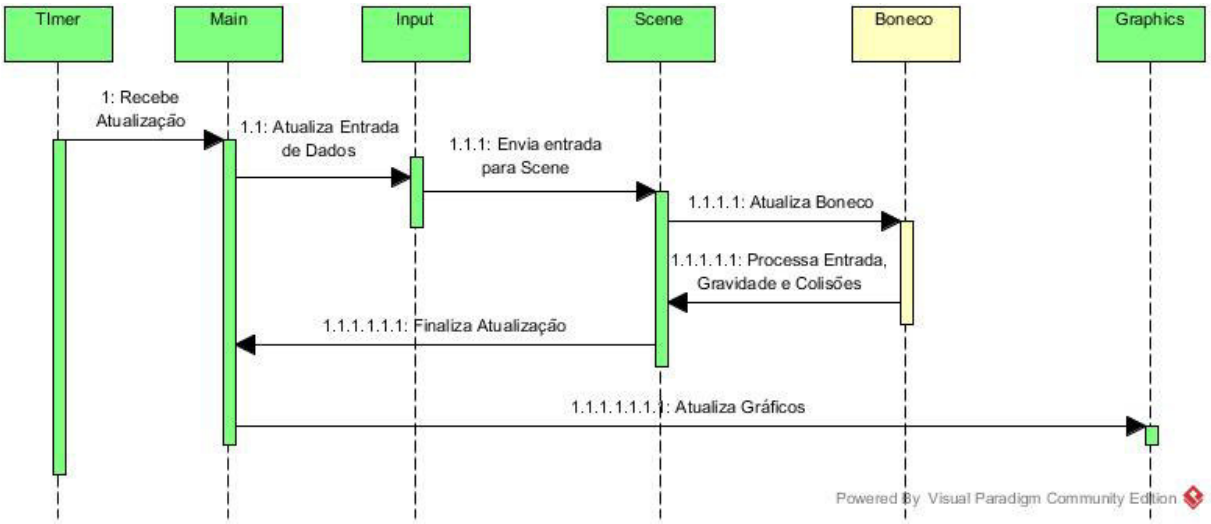
O diagrama de sequencia na Figura 17 demonstra como ocorre todo o processo interno principal do jogo. Primeiramente, o Timer determina quantos frames por segundo o jogo terá. A partir desta informação, ele invoca o método de atualizaçao da Main. A Main então se responsabiliza por atualizar todos os componentes dependentes de frames, como o Input, Scene e Graphics. Primeiramente, o Input é processado, pois a Scene depende da entrada de dados do usuário para poder criar a reação em seus objetos. Após o Input ser atualizado, é a vez da Scene, que por sua vez atualiza cada um de seus Game Objects. Temos por exemplo, na Figura 17, o Game Object Boneco, que faz uma leitura da entrada de dados para se mover, além de aplicar a gravidade em si mesmo, para efeito de simulação de pulo, e também detecta se colidiu com algum outro Game Object da cena. Finalizados todos esses passos, a Main pode finalmente atualizar o Graphics com toda a informação processada. Este processo se repete indefinidamente ou até que o jogo alcance seu objetivo.

Figura 16 – Boneco em Queda Livre, em repouso, e pulando



Fonte: Produzido pelo Autor

Figura 17 – Diagrama de Sequencia para o Jogo de Exemplo



Fonte: Produzido pelo Autor

4 Estudo de Caso: Tetris

A fim de testar o GW, este capítulo apresenta um estudo de caso de um jogo, Tetris, implementada para TVDi. Tetris é uma escolha interessante para implementação: sua forma de tratar os objetos na cena obrigam o desenvolvedor a adaptar as funcionalidades existentes no motor para que possam trabalhar de forma otimizada. Como veremos mais adiante, Tetris ainda possui um conjunto de regras consideradas globais, impostas pela Tetris Company, que determinam parte do funcionamento de cada implementação do jogo, afim de que tenham uma experiência similar em todos os casos. Com esse tipo de afirmativa, um jogador que jogue o jogo em sua versão de Game Boy não terá dificuldades para jogá-lo em sua versão para Nintendo DS, uma vez que os controles e a maioria das reações do jogo se mantém padronizadas (METTS, 2006). Essas regras inserem todo um contexto mais complexo ao jogo, tornando sua implementação interessante para testar o poder de fogo do motor.

Tetris é um jogo fácil de adaptar ao controle remoto tradicional, possuindo poucos comandos e estes geralmente sendo não-entrelaçados, ou seja, o jogador geralmente irá executar um comando de cada vez (rotacionar, mover, derrubar, trocar) em vez de tentar vários comandos simultâneos (como ocorreria em jogos de luta). A imagem de alguns tipos de controle, destacados seus botões principais reconhecidos pelo Ginga, pode ser visto na Figura 18.

O propósito deste capítulo é apresentar o jogo Tetris e suas regras, e a partir disso mostrar uma implementação, utilizando as bases da GW, que se adeque à maioria das regras, adaptadas para melhor funcionar com o uso do controle remoto. Uma implementação aproveitando-se mais da comunicação NCL também foi desenvolvida, mas em uma versão ainda muito primária do motor GW, sendo incluída em apêndice para apreciação.

4.1 Tetris

4.1.1 História

Em 1984, enquanto trabalhava para o Centro de Computação da Academia de Ciências da URSS, Alexey Pajitnov anteviu um jogo eletrônico em que os jogadores poderiam arranjar peças de um quebra-cabeças em tempo real, enquanto eles caíram da parte superior do campo de jogo em velocidades cada vez maiores(TETRIS, 2014). Usando um computador 60 Electronika, ele projetou um jogo que apresentava sete distintas peças geométricas, cada um composto por quatro quadrados. Alexey chamou o jogo de "Tetris", uma combinação da palavra "tetra"(palavra grega que significa "quatro") e "tênis"(seu

Figura 18 – Controles remotos para TV, destacadas suas categorias de botões.

Fonte: (GALABO, 2014)

esporte favorito). O jogo foi portado para o PC IBM e tornou-se um sucesso imediato com seus colegas, se espalhando como fogo em toda a União Soviética.

Em pouco tempo, Tetris começou sua expansão global, lançando em PCs na América do Norte e Europa. O jogo foi apresentado em 1988 na Consumer Electronics Show, em Las Vegas, onde Henk Rogers, designer de jogos e editor, se deparou com Tetris. Ele ficou imediatamente viciado e achou que havia algo especial sobre o jogo. Sua empresa, a Bullet-Proof Software, obteve os direitos para lançamentos de Tetris para PC e NES no Japão, e mais de 2 milhões de cópias foram vendidas. Já em 1989, Henk se encontrou com Alexey, autor do jogo. Henk manteve posse dos direitos de Tetris para portáteis e os licenciou para a Nintendo, que colocou Tetris para lançamento junto com seu novo console portátil Game Boy. Esta associação resultou na venda de 35 milhões de cópias, alavancando ainda mais os nomes das duas empresas.

Em 1997, Henk estabeleceu a Blue Planet Software, Inc. como agente exclusivo para a marca Tetris. Pouco depois, The Tetris Company foi formada, tornando-se a fonte de todas as licenças para Tetris. Muitos aspectos do jogo tornaram-se padronizados. Através de empresas como G-mode, Blue Lava Wireless e EA, Tetris é embarcado para sistemas mobile, fazendo com que o jogo fosse reconhecido como pioneiro na indústria de games casuais.

4.1.2 Regras de Tetris

As regras no jogo tetris se definem como especificado a seguir. O site oficial de Tetris não especifica as regras do jogo, e portanto essas regras citadas a seguir foram extraídas através da experiência os jogadores (FAHEY, 2012, cap. 5).

O jogo acontece dentro de uma máquina, chamada Tétrion. O Tétrion é formado de dez espaços horizontais e 20 verticais, onde se pode mover os tetraminós. Tetraminós escolhidos aleatoriamente caem do topo do Tétrion, um de cada vez. Cada tetraminó entra no campo com uma orientação e cor dependente de seu formato. Parte do Tétrion, chamada *piece preview*, mostra as próximas peças que irão entrar em campo.

O jogador pode rotacionar o tetraminó que está caindo em noventa graus, considerando que haja espaço para que a peça rotacione. Algumas versões do jogo ajustam a posição do tetraminó para que ele consiga realizar a rotação.

O jogador pode movimentar o tetraminó para os lados (um espaço por vez), considerando que haja espaço para que a peça se mova. As peças não podem ultrapassar paredes ou outros blocos.

No topo-esquerda, ou em outros casos, no fundo-direita do Tétrion, pode existir uma área chamada *hold box*, onde o jogador pode armazenar um tetraminó para uso no futuro. A qualquer momento, enquanto um tetraminó cai, o jogador pode movê-lo para a *hold box*, fazendo com que o tetraminó armazenado seja levado ao topo do Tétrion.

Cada tetraminó se move para baixo, devagar. Geralmente, o jogador pode usar algum método para "derrubar"o tetraminó, ou fazê-lo se mover mais rápido para baixo. Quando a peça cai no chão ou em outras peças, ela irá aguardar um pouco antes de se fixar ao campo. Durante este tempo o jogador poderá ainda movê-lo para tentar encaixá-lo em algum lugar. Depois de fixo, o jogador já não pode mover mais o tetraminó.

Quando o tetraminó se fixa e preenche todo o espaço de uma linha, esta linha irá ser limpa (as peças na linha, removidas). Os blocos acima da linha irá se mover para baixo, de acordo com a quantidade de linhas preenchidas.

Se o campo não estiver preenchido com blocos, a próxima peça entra. Se o tetraminó não puder ser alocado no campo, pelo menos parcialmente, devido às regras acima, é considerado fim de jogo.

4.1.3 Diretrizes

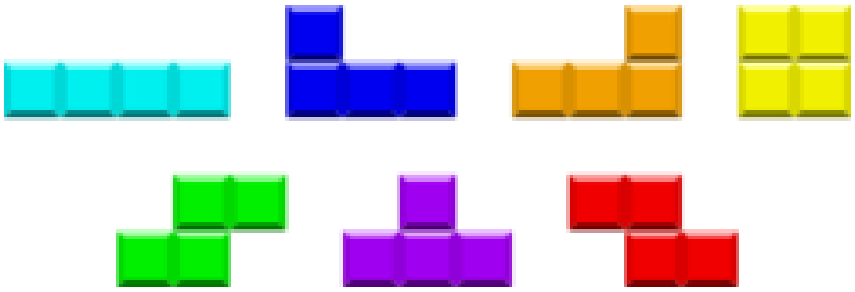
A partir da criação da empresa Tetris Company, foi criada uma diretriz para que todos os jogos lançados posteriormente mantivessem um padrão de jogabilidade (METTS, 2006). Essas diretrizes, assim como as regras, não estão disponíveis oficialmente e foram inferidas através da experimentação dos jogadores.

- Campo de jogo do Tétrion tem tamanho de 10x22 células, sendo as linhas acima da 20ª ocultas (servem para inserir tetraminós parcialmente no campo).
- Cores dos tetraminós

  - I: ciano
  - O: amarelo
  - T: roxo
  - S: verde
  - Z: vermelho
  - J: azul
  - L: laranja

- As peças I e O aparecem nas colunas centrais
- As outras aparecem guidas mais à esquerda
- Os tetraminós aparecem em formato horizontal, com a parte plana apontada para baixo
- O *Super Rotation System* (SRS) especifica a rotação do tetrominó
- Mapeamento padrão para os controles:
  - As setas para cima, para baixo, para direita e para esquerda. Para cima derruba imediatamente a peça (*Hard Drop*), para baixo a derruba mais rapidamente (*Soft Drop*), para os lados, move a peça.
  - O botão de ação da esquerda rotaciona a peça 90º em sentido anti-horário, enquanto que o botão de ação da direita rotaciona em sentido horário, também em 90º.
- Sistema gerador aleatório de peças (*Random Generator*)
- "Armazenar a peça": O jogador pode pressionar um botão para enviar o tetraminó que está caindo para a *hold box*, e qualquer tetraminó que já estiver na *hold box* se move para o topo da tela. Este comando não pode ser usado novamente até que a peça trocada se fixe no Tétrion. Jogos com pouca disponibilidade de botões podem ignorar esta funcionalidade.
- O jogo precisa ter a função de peça fantasma.
- O jogador só pode subir o nível de dificuldade limpando linhas ou relizando o movimento *T-Spin*.
- O jogo precisa utilizar a logo desenvolvida por Roger Dean ou uma variação.
- O jogo precisa incluir a canção clássica de Tétris, Korobeiniki.

Figura 19 – Tipos de Tetraminós



Fonte: (TETRIS WIKIA, 2015)

- O jogador perde o jogo quando a peça aparecer sobre outra peça já fixa no Tétrion, ou quando ela se fixa completamente fora do escopo de visão do campo.

Pelo fato dessas regras não serem de domínio público, não é possível estabelecer com precisão algumas características, como a velocidade de queda, o sistema de pontuação, entre outros.

Super Rotation System

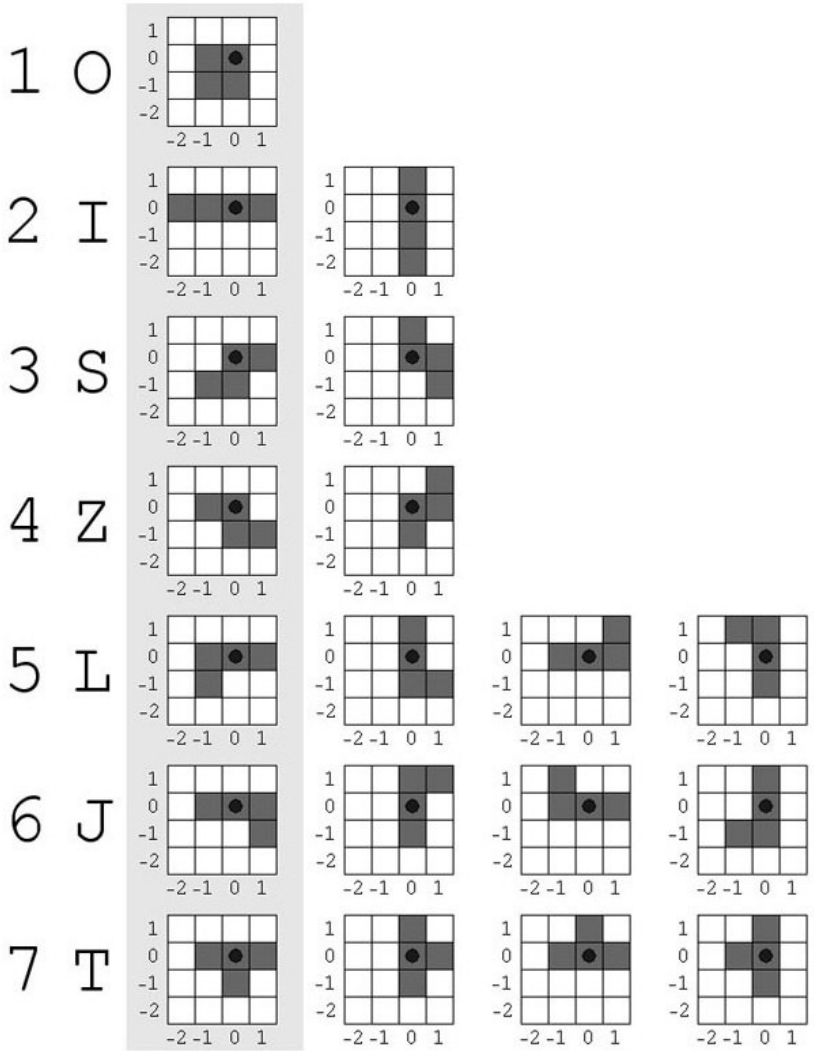
Super Rotation System, ou SRS, é o atual padrão de comportamento dos tetrominós. SRS determina onde e como tetrominós aparecem, como eles rotacionam, e quais *wall kicks* eles podem executar. Todos os tetraminós são definidos dentro de um quadrado, e rotacionam pelo seu eixo, a não ser que sejam impedidos. As peças com largura 3 (J, L, S, T, Z) são posicionadas nas duas linhas superiores do quadrado e (para J, L e T) com o lado plano para baixo. I é colocado na linha superior. Todos os tetraminós aparecem nas duas linhas ocultas no topo do campo de jogo. São posicionados no meio dessas linhas, com preferência à esquerda caso não seja possível colocá-los exatamente no centro. Uma vez que o tetraminó pouse (no chão ou em outro tetraminó), ele não se fixa até que se passe o tempo de espera para fixação (*delay*). Este comportamento, chamado de *Infinity* pela *Tetris Company*, reseta o *delay* quando o tetraminó é movido ou rotacionado. O Infinity não é implementação padrão por permitir que uma peça fique indefinidamente em jogo utilizando esses movimentos. O movimento *Hard Drop* não sofre *delay*, fixando-se automaticamente, na maioria dos casos (já que nem todas as implementações de Tetris seguem à risca as diretrizes) (JARAGOCHI, 2008).

Um *wall kick* ocorre quando o jogador rotaciona a peça e não há espaço para que tal movimento seja realizado. O jogo então move a peça para uma posição em que seja possível realizar a rotação sem colidir com os tetraminós já fixados no campo. O algoritmo mais simples desse movimento é tentar movimentar a peça um espaço para a direita, e então um para a esquerda, e falhar caso nenhuma das duas opções seja possível.

The Random Generator é o nome dado ao algoritmo utilizado para gerar a sequência de tetraminós. Ele gera uma sequencia de todos as peças, permutadas randomicamente. Essas peças são servidas ao Tétrion antes de ser gerado uma nova sequência. Nos jogos que implementam este gerador, acredita-se que haja probabilidade igual de aparecimento para cada peça, tornando menos provável casos onde se passa muito tempo sem receber uma peça específica. O algoritmo produz uma diferença entre duas peças iguais de, no máximo, 12 e no mínimo, 4 peças. Algumas implementações quebram essa regra, fazendo sempre uma peça na primeira posição (I, J, L ou T).

A *Ghost Piece* é uma representação de onde o tetraminó irá eventualmente cair se não for movido. Tem uma cor geralmente mais transparente, e é desenhado por trás da peça original. Movimentos no tetraminó são refletidos na *Ghost Piece*.

Figura 20 – Rotações do Tetraminó, utilizadas pelo SRS.



Fonte: (FAHEY, 2012, cap. 5.3)

4.2 Documentação da Aplicação

A aplicação tem como objetivo criar um jogo de Tetris para TVDi, seguindo as diretrizes já expostas (com algumas ressalvas, mais detalhadas à frente), para um jogador, capaz de gerar e manter um *hi-score* para fins auto-comparativos. O jogo terá diversos níveis de dificuldade, sendo que a diferença entre os níveis está na velocidade de queda das peças. Das diretrizes descritas em 4.1.3, este jogo não irá ter a logo, nem utilizar a canção requerida, nem a função de T-Spin como bônus de pontuação. A rotação dos tetraminós, para conforto do uso com o controle, será restrita somente a um botão, que é a seta para cima no controle.

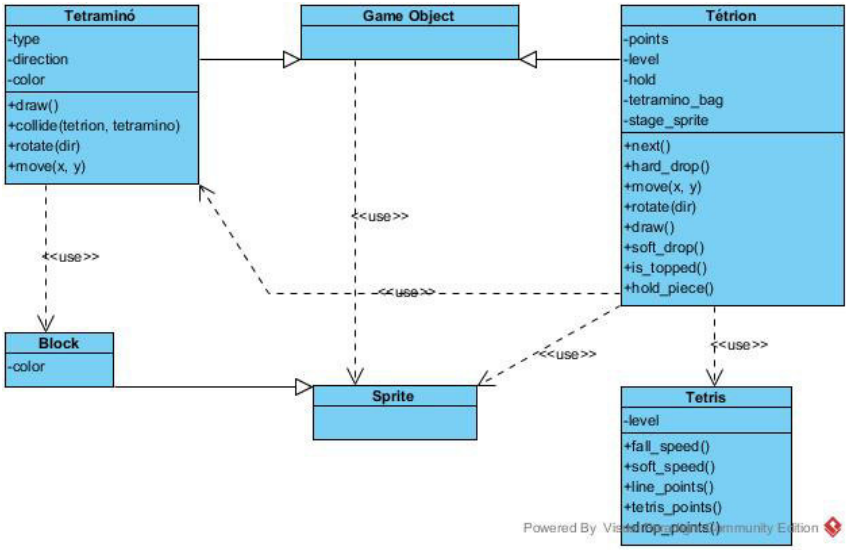
A aplicação consiste de algumas classes, aproveitando as funcionalidades fornecidas pela GW. Há uma classe representando o Tetraminó, que herda de Game_Object, e que é responsável por gerenciar o comportamento isolado do tetraminó (como rotação, tipo, e cor da peça). Há uma classe representando os blocos que compõe o tetraminó. Para fins de otimização de desempenho, esta classe apenas herda de Sprite, de onde tira as informações de imagem. Como ela não tem funcionalidades intrísecas a ela que sejam servidas pela classe Game_Object, instanciar a classe Block como um Game_Object seria uma perda desnecessária de desempenho. Há uma classe Tétrion, onde se passa todo o jogo, e que deve implementar as diretrizes especificadas em 4.1.3. Esta classe consome as configurações especificadas na "classe"Tétris, que é responsável por armazenar o modo de funcionamento do jogo. Como não herda de nenhuma classe disponibilizada pela GW, e Lua não disponibiliza a funcionalidade de classes apriori, Tétris é apenas um arquivo .lua que armazena configurações.

É importante frisar que mesmo numa aplicação de exemplo utilizando a GW, nem todos os comportamentos devem ser criados com base nas funcionalidades disponibilizadas pelo motor. Em jogos, a performance (nomeadamente, os frames por segundo) é tão importante quanto a jogabilidade e gráficos[1]. Um jogo que tenha todas as características em desenvolvidas (gráficos, jogabilidade, história, replay, sons e músicas, etc), mas que entrega performance ruim ou instável pode ser tão mal recebido quanto um jogo cheio de erros (bugs)[2]. Por isso, otimização é geralmente tratado como um sobrenome para jogos. Em jogos 3D, a otimização é geralmente toda concentrada no aspecto gráfico do jogo, devido a estes rodarem em, geralmente, processadores poderosos, com múltiplos núcleos, e uma placa de vídeo dedicada, configuração supostamente capaz de rodar qualquer aspecto lógico do jogo. As partes gráficas, no entanto, devido à alta complexidade para tentar reproduzir imagens realistas, requerem alto investimento em otimização. Em Set-Top Boxes, por seu baixo poder de processamento, temos que levar em consideração otimizações tanto a nível gráfico quanto a nível de processamento e memória. Recomenda-se evitar o uso

[1] <http://www.eurogamer.net/articles/digitalfoundry-2014-frame-rate-vs-frame-pacing>
[2] <http://www.dsogaming.com/editorial/top-5-worst-optimized-pc-games-of-2014/>

desnecessário das classes fornecidas pela GW, pelo fato de adicionarem peso computacional ao objeto. Elas somente devem ser usadas se suas funcionalidades forem essenciais ao funcionamento do objeto, como é o caso do Tetraminó e do Tétrion. A classe Bloco apenas precisa do funcionamento do Sprite, enquanto que o arquivo Tétris não precisa de nenhum dos dois, não herdando portanto nenhuma funcionalidade desnecessária.

Figura 21 – Diagrama de Classes. Tetris



Fonte: Produzido pelo Autor

Na Figura 21 podemos ver um Diagrama de Classes simples que resolve o problema lógico do jogo Tétris. A seguir, explicamos o funcionamento de cada função, levando em consideração as diretrizes descritas em 4.1.3.

A classe mais simples, *Block*, é apenas uma extensão de Sprite. Não adiciona nenhum funcionamento extra, a não ser obter o gráfico dos blocos utilizado no jogo. A classe bloco armazena o sprite de todas as cores de blocos, distribuídos numa única imagem, e apenas escolhe qual parte da imagem será desenhada (o bloco em si). Este método de desenho é mais eficiente do que ter uma imagem para cada cor de bloco, uma vez que é mais rápido apenas escolher uma parte de uma imagem para ser desenhada do que mudar a imagem a ser desenhada (Figura 22).

Figura 22 – Sprite com Blocos do Tetraminó

Fonte: Produzido pelo Autor

Se utilizando da classe *Block*, temos a classe Tetraminó, representada como uma matriz 4x4 assim como podemos ver na Figura 20. Esta classe é responsável por determinar o formato do tetraminó (atributo *type*) e sua direção (atributo *direction*). O formato da peça determina sua cor, e vice-versa. Os formatos são determinados pelas letras, descritas nas diretrizes. As direções são baixo, esquerda, acima, direita, numeradas de 0 a 3, respectivamente, tendo como direção inicial o valor 0, para satisfazer as diretrizes. O método *draw()* desenha blocos na tela, obedecendo o tipo e a direção. O método *collide(tetrion)* determina se este tetraminó colidiu com o tétrion de alguma forma. O método *rotate(dir)*, que recebe um valor positivo ou negativo, rotaciona a peça de acordo com a direção informada. E por fim, o método *move(x, y)*, que também recebe números positivos ou negativos, movimenta o tetraminó levando em conta as direções informadas. As movimentações sempre acontecem em 1 unidade, independente do valor. O que varia é a direção em que essa unidade é avançada. Nota-se que o método não se responsabiliza por validar se o movimento é válido. A estrutura de dados em que o tetraminó é armazenado é uma matriz 4x4, formada por números, 0 e 1, onde 0 significa não haver nenhum bloco ali, e 1 o contrário. Ela é preenchida levando em consideração o tipo e a direção.

A "classe"Tetris tem o propósito de armazenar as configurações gerais do jogo. Possui o atributo *level (nível)* que determina a dificuldade do jogo. O método *fall_speed()* determina o intervalo que a peça espera para cair 1 unidade automaticamente, levando em consideração o nível. O método *soft_speed()* determina a velocidade de queda da peça quando o jogador está utilizando o movimento *Soft Drop*, levando em consideração o nível. O método *line_points()* determina quantos o jogador recebe ao limpar uma linha. O método *tetris_points()* determina quantos pontos o jogador recebe ao realizar um tetris (limpar 4 linhas de uma só vez). O método *drop_points()* determina uma pontuação para quando o jogador fixar um tetraminó no campo.

Por fim, a classe Tétrion tem como responsabilidade gerir todas essas outras classes de acordo com as diretrizes de jogo. O atributo *points* armazena os pontos obtidos pelo jogador, O atributo *lines* informa quantas linhas o jogador já limpou. O atributo *hold* é utilizado para a função *Hold* em jogos de Tetris, onde é possível guardar um tetraminó para uso tardio. O *tetramino_bag* armazena todas peças geradas randomicamente para serem colocadas em jogo. *stage_sprite* apenas guarda os sprites utilizados para desenhar o fundo. A função *next()* tira o próximo tetraminó de *tetramino_bag* e o coloca em campo. A função *hard_drop()* faz a peça cair automaticamente na mesma posição que está a *Ghost Piece*. A peça se fixa imediatamente após esse comando. O método *move(x, y)* requisita movimento do tetraminó, ao mesmo tempo em que valida sua nova posição, realizando algoritmos descritos no SRS para ajuste de posição. O método *rotate(dir)* requisita um movimento de rotação ao tetraminó, validando sua posição após e ajustando caso haja necessidade, utilizando o SRS. O método *draw()* é sobrescrito para desenhar todos os objetos na tela (o tetraminó, os próximos tetraminós, o *hold*, os pontos, linhas limpas, nível

e imagens de fundo). O método *soft_drop()* acelera a velocidade de queda do tetraminó. Em níveis altos, usar esta função pode ter o mesmo efeito de *hard_drop()* por causa de uma velocidade muito alta. O método *is_topped()* verifica se a peça está presa no topo do campo, que é a condição de fim de jogo. O método *hold_piece()* requisita que o tetraminó atual seja colocado no campo *hold*. Somente uma peça que não saiu do *hold* pode ir para lá, para evitar trocas infinitas enquanto o jogador decide onde colocar a peça.

4.3 Resultados

O desenvolvimento da aplicação se deu utilizando-se 8 arquivos lua com propósitos específicos, seguindo o proposto na Figura 21, e suas funções podem ser vistas na tabela 1. Grande parte do desenvolvimento se deu na classe `board`, que ficou responsável por gerenciar a lógica interna de rotações e movimentação, além da função de *hold*, e aplicação de pontuação. A classe `manager` ficou responsável por sincronizar as entrada de dados com a lógica do `board`.

Com o código provido pelo GW, evitou-se a implementação de outras diversas necessidades do jogo, como controle de gráficos, loop interno do jogo, controle de sons ou de entrada de dados. Os códigos 4.1 e 4.2 mostram o cerne do arquivo `manager.lua`, responsável por integrar essas capacidades. A `Scene` se encarrega de chamar as atualizações e métodos de desenho de cada objeto registrado a ela, quando necessário. O desenvolvimento foi focado em resolver as complexidades envolvidas no algoritmo específico do jogo, através da classe `Board`. Regras de atualização e entrada de dados ficaram triviais, uma vez que o motor faz as chamadas a esses métodos de forma transparente, uma vez devidamente configurados. Por causa da estrutura de dados adotada, não foi possível utilizar a função padrão de desenho disponibilizada pelo `Game_Object`. As classes Bloco, Tetraminó e Board tiveram que reescrever a função `draw()`.

Código 4.1 – Inicialização dos Objetos da Cena - Tetris

```
...
self.vars.board = GWClasses['Board'].new()
self.vars.background1 = GWClasses['Background'].new()
...
table.insert(self.objects, self.vars.board)
table.insert(self.objects, self.vars.background1)
Game_Graphics:insertLayer(2, self.vars.board)
Game_Graphics:insertLayer(1, self.vars.background1)
...
Game_Resources.play_BG('playback', true)
```

Como o GW tem o propósito de ser estendida, em vez de oferecer várias funcionalidades de alto nível, tivemos que implementar uma funcionalidade para desenhar textos

Tabela 1 – Lista de Arquivos para Projeto Tetris

Arquivo	**Classe**	**Linhas**	**Propósito**
background.lua	Background	6	Representar, em forma de objeto, o fundo da cena. Este objeto poderia ser simplesmente criado como uma instância de `Game_Object` da API, alterando-se apenas o `sprite`, porém foi feito assim para preservar o conceito de objetos.
block.lua	Background	20	Alterar a funcionalidade de desenho padrão do GW. Esta classe carrega o gráfico dos tetraminós em um único sprite, e, dependendo de como é configurado, se limita a mostrar uma parte desse sprite na tela.
board.lua	Board	412	Conter a lógica de orientação do jogo. Movimenta e rotaciona o tetraminó ativo (controlado pelo jogador), permite armazenar a peça (*hold*), e derrubá-la de vez (*hard drop*). Também tem um tratamento diferenciado do modo de desenho para dar suporte aos Blocos. Porém, outros objetos com os quais interage, como Background, são desenhados no modo padrão.
ex_parameter.lua	GWInitializer	13	Fazer tratamento de parâmetros NCL para tornar possível registrar e tocar sons através de chamadas Lua.
font.lua	Nenhuma	22	Criar a funcionalidade de desenho de textos arbitrários na tela.
manager.lua	GWScene	96	Tratar a entrada de dados do usuário, enviando como comandos próprios para a Board. Gerenciar a velocidade de queda dos tetraminós. Enviar comando de finalização de mídia para o NCL, uma vez o jogador receba um *Game Over*.
tetramino.lua	Tetramino	242	Armazenar a estrutura de dados do tetraminó. Também modifica a forma de desenho padrão para dar suporte a Blocos. Quase 80 % do total de linhas foram consumidas em configurações das posições dos blocos em cada tipo de tetraminó e suas rotações.
tetris.lua	Nenhuma	64	Prover tabela para gerenciar informações de sistema do jogo, como dificuldade ou quantos pontos se ganha por linha eliminada.

Fonte: Produzido pelo Autor

na tela a partir de uma fonte em formato de imagem, que foi utilizada para desenhar os números que aparecem na tela. Esta funcionalidade, escrita no `font.lua`, pode ser distribuída e embutida em qualquer outra aplicação que utilize o motor.

Código 4.2 – Tratamento de Entrada de Dados - Tetris

```
local move = {x = 0, y = 0}
if GW_Input.pressed('CURSOR_RIGHT') then
        move.x = 1
```

```
elseif GW_Input.pressed('CURSOR_LEFT') then
        move.x = -1
end
if GW_Input.pressed('CURSOR_DOWN') then
        move.y = 1
end
if GW_Input.trigger('CURSOR_UP') then
        self.vars.board:order_rotate()
end
...
```

Enfim, a implementação do jogo foi focada apenas em seus problemas: resolução de colisões, rotação do Tetraminó, e outras ações especificadas, além de que um código mais apropriado para o tratamento do desenho dos Tetraminós e também das peças já encaixadas no tabuleiro, através da classe Block. Todo o resto estrutural, controle de FPS, entrada de dados, desenho de outros objetos, toque de sons foi facilmente desenvolvido com chamadas à API. O resultado final pode ser observado na Figura 23. Um outro trabalho, desenvolvido em uma versão antiga deste motor, porém com mais foco em configurações pelo NCL, pode ser visto no Apêndice A.

Figura 23 – Tela do jogo Tetris desenvolvido utilizando-se o motor Ginga Wings

Fonte: Produzido pelo Autor

5 Conclusões

De acordo com a definição de motor de jogos, feita por Gregory (GREGORY, 2009), o Ginga Wings atingiu seu propósito. Ela separa os componentes de desenvolvimento de um jogo - renderizador, controlador de som, assets (cache), entrada de dados - de forma independente, juntando tudo para trazer um ambiente quase automático de desenvolvimento de jogos para STBVD, considerando-se as partes essenciais de desenvolvimento de jogos para a plataforma. O motor, por não forçar um design de componente voltado para um determinado gênero de jogo, é flexível o suficiente para produzir jogos para a maioria dos gêneros 2D. Por ser codificado em Lua e utilizar conceitos de OO, é extremamente fácil estender suas funções ou criar novas funcionalidades para seus componentes.

A criação de um jogo no motor mostrou que tais componentes aceleram muito o desenvolvimento. Ele abstrai a maior parte da dificuldade de se desenvolver para a plataforma, permitindo um desenvolvimento mais focado no produto e não na infraestrutura. O desenvolvimento do jogo Tetris, utilizando o GW, se limitou a configurar o ambiente NCL e criar a lógica do jogo, e atividades mais complexas como sincronizar o jogo com a velocidade de entrega dos frames (FPS) se torna trivial com apenas uma chamada ao pacote Timer. A capacidade de receber continuamente parâmetros do NCL pode criar um incentivo ao uso desta ferramenta, bastando o usuário com domínio de código Lua criar uma estrutura para tratar os parâmetros e deixar algum usuário de domínio de código NCL configurá-lo para executar, de fato, a autoria do jogo.

Dificuldades neste trabalho incluíram a pouca quantidade de artigos científicos que pudessem ser utilizados como fontes - artigos na internet e sites de desenvolvedores foram utilizados em seu lugar. A implementação do jogo Tetris também se mostrou um pouco problemática devido as suas regras ocultas - a maior parte inferida por jogadores ou pesquisadores que analisaram o código fonte do jogo original.

Para trabalhos futuros, o motor por si mesma já direciona o caminho do trabalho. Se por um lado o motor consegue abstrair parte da complexidade do desenvolvimento do jogo, por outro lado sua simplicidade pode ser um revés em um primeiro momento. A falta de extensões, no atual momento, pode incentivar um usuário a procurar uma ferramenta mais robusta. Torna-se então prioridade para futuros trabalhos a implementação de extensões úteis para desenvolvimento de jogos 2D em geral, como gerenciadores de Tiles, fontes para escrita de textos, melhor tratamento de animações, desenvolvimento completo da funcionalidade de rede. Existem vários caminhos para se tomar, e há muito potencial para se desenvolver com o motor.

Referências

AGUIAR, R. *Unity3D v4.3: 2D vs 3D Physics*. 2014. Acesso: 19 jun 2015. Disponível em: <http://x-team.com/2013/11/unity3d-v4-3-2d-vs-3d-physics/>. Citado na página 17.

BARBOZA, D. C.; CLUA, E. W. G. Ginga game: a framework for game development for the interactive digital television. In: IEEE. *Games and Digital Entertainment (SBGAMES), 2009 VIII Brazilian Symposium on*. [S.l.], 2009. p. 162–167. Citado na página 20.

BIOWARE. *IESPD*. 2013. Acesso: 19 jun 2015. Disponível em: <http://gibberlings3.net/iesdp/index.htm>. Citado na página 15.

BLOW, J. Game development: Harder than you think. *Queue*, ACM, v. 1, n. 10, p. 28, 2004. Citado na página 13.

CORRÊA, A. G. D. et al. Jogos educacionais para tv digital interativa. *Revista Trilha Digital*, v. 1, n. 1, 2013. Citado na página 13.

COUPRIE, S. *Teach With... Game Maker*. 2015. Acesso: 19 jun 2015. Disponível em: <https://sites.google.com/a/share.epsb.ca/teachcs/game-maker>. Citado na página 18.

DIáRIO CATARINENSE. *SP Jam reúne desenvolvedores de jogos digitais e analógicos em competição*. 2013. Acesso: 19 jun 2015. Disponível em: <http://diariocatarinense.clicrbs.com.br/sc/economia/noticia/2013/07/sp-jam-reune-desenvolvedores-de-jogos-digitais-e-analogicos-em-competicao-4200941.html>. Citado na página 13.

FAHEY, C. *Tetris*. 2012. Acesso: 19 jun 2015. Disponível em: <http://colinfahey.com/tetris/tetris.html>. Citado 2 vezes nas páginas 51 e 55.

FERRARI, P. *Hipertexto, hipermídia: as novas ferramentas da comunicação digital*. [S.l.]: Editora Contexto, 2011. Citado na página 13.

FERREIRA, D.; SOUZA, C. Tuga: Um middleware para o suporte ao desenvolvimento de jogos em tv digital interativa. *Centro Federal de Educação Tecnológica do Ceará.*, 2009. Disponível em: <http://code.google.com/p/tugasdk/downloads/detail>. Citado na página 19.

GALABO, R. J. F. Padrões de design de interação para aplicativos de comércio televisivo com foco na experiência do usuário. 2014. Citado na página 51.

GREGORY, J. *Game engine architecture*. [S.l.]: CRC Press, 2009. Citado 4 vezes nas páginas 13, 15, 62 e 66.

GROCHOWIAK, T. *Professional developer's look at GameMaker*. 2012. Acesso: 19 jun 2015. Disponível em: <http://moacube.com/blog/professional-developers-look-at-gamemaker/>. Citado na página 18.

HUGHES, D. *Tales from Development Hell: Hollywood Film-Making the Hard Way*. [S.l.]: Titan, 2003. Citado na página 13.

ITU. *Nested context language (NCL) and Ginga-NCL for IPTV services*. [S.l.], 2009. H. 761. Citado 2 vezes nas páginas 23 e 25.

JARAGOCHI. *TGM-ACE SRS Study*. 2008. Acesso: 19 jun 2015. Disponível em: <http://web.archive.org/web/20081216145551/http://www.the-shell.net/img/srs_study.html>. Citado na página 54.

LEWIS, J.; LOFTUS, W. Java software solutions foundations of programming design. *Pearson Education Inc*, 2008. Citado na página 29.

METTS, J. *Tetris from the Top: An Interview with Henk Rogers*. 2006. Acesso: 19 jun 2015. Disponível em: <http://www.nintendoworldreport.com/interview/11267/tetris-from-the-top-an-interview-with-henk-rogers>. Citado 2 vezes nas páginas 50 e 52.

MONO PROJECT. *Companies using Mono*. 2015. Acesso: 19 jun 2015. Disponível em: <http://www.mono-project.com/docs/about-mono/showcase/companies-using-mono/>. Citado na página 16.

OOKI, R. H. *Linguagem Lua: nova estrutura e aplicações*. [S.l.], 2013. Citado na página 14.

PUC RIO. *What is Lua?* 2015. Acesso: 19 jun 2015. Disponível em: <http://www.lua.org/about.html>. Citado na página 78.

RATAMERO, E. M. *Tutorial sobre a linguagem de programação NCL (Nested Context Language)*. [S.l.], 2007. Citado 3 vezes nas páginas 14, 70 e 71.

SEGUNDO, R.; SILVA, J. C. F. da; TAVARES, T. A. Athus: A generic framework for game development on ginga middleware. In: IEEE. *Games and Digital Entertainment (SBGAMES), 2010 Brazilian Symposium on*. [S.l.], 2010. p. 89–96. Citado na página 21.

TETRIS. *Tetris History*. 2014. Acesso: 19 jun 2015. Disponível em: <http://tetris.com/about-tetris/timeline/>. Citado na página 50.

TETRIS WIKIA. *Tetromino*. 2015. Acesso: 19 jun 2015. Disponível em: <http://tetris.wikia.com/wiki/Tetromino>. Citado na página 54.

UNITY TECHNOLOGIES. *Asset Store*. 2015. Acesso: 19 jun 2015. Disponível em: <https://www.assetstore.unity3d.com/en/>. Citado na página 17.

UNITY TECHNOLOGIES. *Build Once Deploy Anywhere*. 2015. Acesso: 19 jun 2015. Disponível em: <http://unity3d.com/unity/multiplatform/>. Citado na página 16.

UNITY TECHNOLOGIES. *Using DirectX 11 in Unity*. 2015. Acesso: 19 jun 2015. Disponível em: <http://docs.unity3d.com/Manual/DirectX11>. Citado na página 16.

YOYO GAMES. *GameMaker: Studio*. 2015. Acesso: 19 jun 2015. Disponível em: <http://www.yoyogames.com/studio>. Citado 2 vezes nas páginas 17 e 18.

Apêndices

APÊNDICE A – GingaFighters

Este foi o trabalho que deu origem ao desenvolvimento do GW. O propósito do trabalho era criar um clone do jogo AeroFighters[1] para TV Digital, sob a disciplina de Hipermídia lecionada pelo Prof. Carlos de Salles Soares Neto. Todas as estruturas do jogo foram criadas seguindo, adaptando de acordo com as necessidades, o modelo de motor de jogos discutido por (GREGORY, 2009). Ao fim do trabalho, percebemos a possibilidade de, em vez de criar o jogo todo utilizando Lua, o que obrigaria a qualquer time a ter um especialista nesta linguagem de código, poderíamos abstrair essa necessidade utilizando Links de NCL para enviar comandos para o Lua, que por fim os interpretaria e executaria a ação requisitada, permitindo assim que um time de desenvolvedores NCL criasse uma fase do jogo sem necessidade de intervenção de um desenvolvedor Lua.

O jogo GingaFighters é, na verdade, um motor de jogos especializado no estilo do jogo homônimo, construído em cima da arquitetura do GW. Os jogos são totalmente configurados a partir de código NCL seguindo uma linguagem simples desenvolvida com esse propósito. O motor oferece 4 tipos diferentes de inimigos que podem ser escolhidos a partir do NCL, além de 4 tipos de tiro. A nave controlada pelo jogador tem 5 pontos de escudo que o protegem de tiros de naves inimigas, além de algumas vidas para casos de derrota. Durante a fase, é possível configurar para que bônus aleatórios apareçam. Esses bonus podem restaurar escudo e mudar o tiro da nave. A forma de configuração é mostrada na Tabela 2. Uma imagem do jogo pode ser visto na Figura 24.

Nos Códigos A.1 e A.2, mostramos como seria feita a configuração de um jogo a partir de manipulação do NCL em chamadas simples.

Código A.1 – Exemplo de Código NCL pra Inicializar a Configuração do Motor GingaFighters

```
<media id="lua" src="main.lua" descriptor="dsLua">
        ...
        <property name="heroi"/>
        <property name="inimigo1"/>
        <property name="inimigo1_x200"/>
        <property name="inimigo1_t40"/>
        <property name="inimigo2"/>
        <property name="inimigo2_x86"/>
        <property name="inimigo2_t68"/>
        ...
        <property name="bonus1"/>
        ...
```

[1] Mais informações em: http://jogosonline.uol.com.br/aero-fighters_4921.html

```
        <property name="bonus1_t15"/>
        ...
        <property name="ground1"/>
        <property name="ground2"/>
        ...
        <property name="boss"/>
        <property name="boss_t400"/>
        ...
</media>
```

Código A.2 – Exemplo de Código NCL pra Configurar e Enviar os Parâmetros do Motor GingaFighters

```
<link xconnector="con#onBeginSetN">
<bind role="onBegin" component="lua" />
<bind role="set" component="lua" interface="heroi">
<bindParam name="var" value="media/plane1stand.png"/>
</bind>
<bind role="set" component="lua" interface="inimigo1">
<bindParam name="var" value="media/gingaEnemy.png"/>
</bind>
<bind role="set" component="lua" interface="inimigo1_x200"/>
<bind role="set" component="lua" interface="bonus1">
<bindParam name="var" value="media/bonus_shield_1.png"/>
</bind>
...
<bind role="set" component="lua" interface="ground1">
<bindParam name="var" value="media/beta_back1.jpg"/>
</bind>
...
<bind role="set" component="lua" interface="boss">
<bindParam name="var" value="media/boss1.png"/>
</bind>
...
<bind role="set" component="lua" interface="bonus1_t15"/>
...
<bind role="set" component="lua" interface="boss_t400"/>
</link>
```

Tabela 2 – Lista de Configurações reconhecidas pelo GingaFighters

Parâmetro	**Efeito**
heroi	Quando enviado, deve conter um arquivo de mídia. São possíveis 2 tipo de configuração da nave, determinados pelo arquivo de mídia enviado.
inimigoID	Configura uma instância de um inimigo para o jogo. ID é um identificador único que identificará o inimigo. O valor enviado no parâmetro definirá o tipo e comportamento do inimigo. São possíveis 4 tipos de inimigo.
inimigoID_xPOS	Configura o inimigo com identificador ID a aparecer na posição POS no eixo das coordenadas. Ou seja, onde na tela, a partir do topo, o inimigo vai aparecer.
inimigoID_tTEMPO	Configura o inimigo com identificador ID a aparecer após TEMPO frames. Ou seja, determina quando o inimigo aparece.
bonusID	Configura um bônus com identificador ID. São possíveis 4 tipos de bônus.
bonusID_tTEMPO	Determina quando o bônus de identificador ID irá aparecer no jogo, após TEMPO frames.
groundID	Configura o cenário de fundo. As imagens do cenário de fundo devem possuir o tamanho de 800x600 (resolução padrão do jogo) e são ordenadas pelo ID, que deve ser sequencial.
boss	Configura qual será o chefe enfrentado ao fim da fase. O arquivo de mídia enviado com o parâmetro determina o tipo do chefe.
boss_tTEMPO	Configura após quantos frames o chefe da fase irá aparecer.

Fonte: Produzido pelo Autor

Figura 24 – Imagem do jogo GingaFighters em execução

Fonte: Produzido pelo Autor

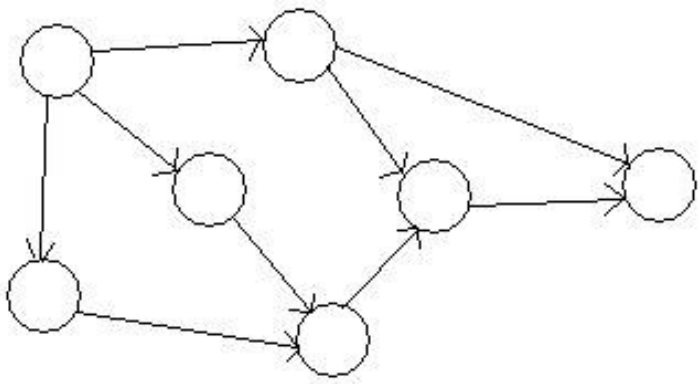
Anexos

ANEXO A – Linguagem NCL

A linguagem NCL é uma linguagem do tipo declarativa, isto é, ela especifica de maneira imperativa o que deve ser feito, em oposiçãao às linguagens procedurais, que descrevem como fazer alguma coisa. Como exemplos destes dois tipos de linguagens, temos HTML representando a abordagem declarativa, usada para descrever o conteúdo de uma página web, e Java ou C representando a abordagem procedural, usadas para especificar um algoritmo para executar determinada tarefa.

A NCL é baseado no modelo NCM (Nested Context Model, ou Modelo de Contextos Aninhados). Este modelo usa os conceitos de nós (nodes) e elos (links) para descrever documentos hipermídia (documentos que contém diversos tipos de mídia, além de interação com o usuário). Podemos exemplificar este modelo pela Figura 25.

Figura 25 – Estrutura de um documento hipermídia



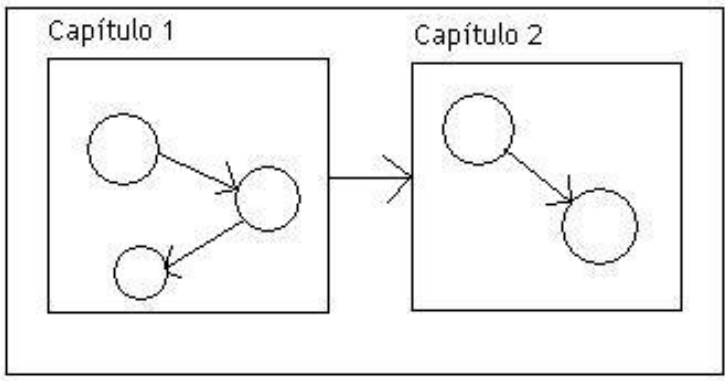
Fonte: (RATAMERO, 2007)

Neste modelo, estes grafos podem ser aninhados, ou seja, cada nó pode ser, na verdade, um conjunto de nós e elos. Isto permite tornar a estrutura de um documento mais enxuta e organizada. Este aninhamento pode ser feito através de um tipo especial de nó, chamado de nó de composiçãao (composite node) ou de contexto (context node). Assim, fica claro que podemos trabalhar com dois tipos de nós:

- nós de mídia: representam figuras, textos, vídeos e demais tipos de mídia;
- nós de contexto: representam um conjunto de nós e elos.

Podemos ver como estes nós se relacionam entre si na Figura 26.

Figura 26 – Estrutura de um documento NCM com composições



Fonte: (RATAMERO, 2007)

Para a autoria de documentos usando o modelo NCM, foi criada a linguagem NCL para a elaboração de documentos hipermédia. Um programa formatador é utilizado para interpretar um documento NCL e apresentar o conteúdo audiovisual interativo representado por ele.

A.1 Estrutura Básica de um Documento NCL

Um documento NCL deve possuir um cabeçalho de arquivo NCL, um cabeçalho do programa, o corpo do programa e o encerramento do documento. Os nós (sejam de mídia ou de contexto), elos e outros elementos que definem a estrutura e o conteúdo do programa são definidos na parte do corpo do programa.

Para definirmos por onde a apresentação do programa será iniciada, devemos criar portas. Estas portas servem como ponto de entrada em um documento ou nó de contexto. No momento da exibição do documento, devemos informar por que porta deseja-se iniciar a apresentação, propiciando assim que um único documento NCL tenha diversos pontos de entrada. Caso a porta de inécio não seja informada, o formatador usará uma porta-padrão que depende da implementação do mesmo.

A seguir, temos um exemplo de documento NCL:

```
1: <?xml version="1.0" encoding="ISO-8859-1"?>
2:
3: <ncl id="exemplo01"
xmlns="http://www.telemidia.puc-rio.br/specs/xml/NCL23/profiles"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.telemidia.pucrio.
br/specs/xml/NCL23/profiles/NCL23.xsd">
4: <head>
5: <regionBase>

7: </regionBase>
8: <descriptorBase>

10: </descriptorBase>
11: <connectorBase>

eles disparam -->
13: </connectorBase>
14: </head>
15: <body>
16: <port id="pInicio" component="ncPrincipal" interface="iInicio"/>

18: </body>
19: </ncl>
```

As linhas de 1 a 3 definem o cabeçalho de arquivo NCL. Da linha 4 à linha 14 temos o cabeçalho do programa. As linhas 15 a 18 contém o corpo do programa, e a linha 19 o encerra.

A.1.1 Regiões

Uma região nada mais é do que uma área na tela (ou outro dispositivo de saída) onde será exibido um determinado nó de mídia. Estas regiões podem ser aninhadas (regiões dentro de regiões), tornando a estrutura mais organizada. Todas as regiões devem ser definidas no cabeçalho do programa (`regionBase`).Um exemplo de definição de regiões é o seguinte:

```
<region id="rgTV" width="1920" height="1080">
        <region id="rgVideo1" left="448" top="156" width="1024"
height="768" />
</region>
```

Os atributos `height`, `width`, `left` e `top` definem a altura, largura, coordenada esquerda e coordenada superior da região. O atributo `id` dá um nome único a esta região, nome este que será referenciado, por exemplo, nos descritores das mídias associadas a esta região. Podemos definir ainda os atributos `background`, que atribui uma cor de fundo, e `zIndex`, que indica quais regiões aparecerão sobre quais no caso de regiões sobrepostas.

A.1.2 Descritores

Um descritor define como será apresentado um nó de mídia, incluindo em que região ele aparecerá. Os descritores devem ser definidos no cabeçalho do programa (`descriptorBase`). Um exemplo de descritor é o seguinte:

```
<descriptor id="dVideo1" region="rgVideo1" />
```

O atributo `id`, como nas regiões, dá um nome único a este descritor, que será referenciado quando da criação de um nó de mídia relacionado a este descritor. O atributo `region` associa uma região a este descritor. Além destes atributos, pode-se definir os atributos `player`, que diz qual a ferramenta de apresentação que será utilizada para mostrar nós de mídia associados a este descritor, e `explicitDur`, que diz qual será a duração temporal da apresentação dos nós de mídia relacionados a este descritor.

A.1.3 Portas

Portas servem para garantir acesso externo ao conteúdo de um contexto. Assim sendo, para que um elo aponte para um nó interno a um contexto, este nó deve apontar para uma porta que leve ao nó interno desejado. Um exemplo de definição de porta segue abaixo:

```
<port id="pInicio" component="video1" />
```

Podemos ver todo o *body* do documento como um grande contexto. Assim, precisamos de uma porta de entrada que aponte para o primeiro nó de mídia ou contexto a ser apresentado quando da execução do documento.

O atributo `id` atribui um nome único, pelo qual esta porta será referenciada sempre que necessário. O atributo `component` diz a qual nó de mídia ou contexto esta porta está associada. Há ainda o atributo `interface`. Este atributo indica a qual porta esta porta deve ser relacionada, no caso de `component` ser um nó de contexto, ou a qual âncora ela deve ser relacionada, caso `component` seja um nó de mídia.

A.1.4 Contextos

Os contextos servem para estruturar o documento NCL. Desta forma, eles podem ser aninhados, procurando sempre refletir a estrutura do documento e tornar a organização do programa mais intuitiva.

Define-se contexto da seguinte forma:

```
<context id="ctxNome">
...
</context>
```

O atributo `id`, como nos demais itens, define um nome único pelo qual o contexto será referenciado. Além deste, temos os atributos `descriptor`, que identifica qual descritor definirá a apresentação do contexto, e `refer`, que faz referência a outro contexto já definido, do qual este contexto herdará tudo menos o atributo `id`.

A.1.5 Nós de Mídia

Um nó de mídia caracteriza o objeto de conteúdo propriamente dito, seja ele um vídeo, um texto, um áudio, etc. O nó de mídia deve identificar o arquivo com o conteúdo da mídia, além do descritor usado para regular a apresentação deste objeto de mídia. Um exemplo de nó de mídia é o seguinte:

```
<media type="video" id="video1" src="media/video1.mpg" descriptor="
   dVideo1"/>
```

O atributo `type` define de que tipo de mídia se trata: vídeo, áudio, texto, etc. O atributo `id` dá um nome único ao nó de mídia. O atributo `src` indica onde está o arquivo fonte daquele nó de mídia, e o atributo `descriptor` indica qual descritor será usado para a execução daquele objeto. Outro atributo que pode ser utilizado é o `refer`, que referencia um outro nó de mídia do qual este nó usará todos os atributos, menos o id.

A.2 Elos e Conectores

Os nós de mídia ou contexto possuem relações entre si, caracterizando um documento hipermídia. Estas relações podem ser de diversos tipos. Desta forma, surge o conceito de conectores, objetivando definir a relação semântica contida nos elos em NCL.

Assim sendo, o conector (do tipo causal) serve para definir a relação entre um ou mais nós de origem (chamados assim por ativar o elo) e um ou mais nós de destino (que serão afetados pela ativação do elo). Um conjunto de conectores já foi bem-definido pela grupo de trabalho da PUC-RJ que desenvolveu a linguagem NCL.

Geralmente, um arquivo externo contendo todos os conectores (chamado de base de conectores) pode ser importado pelo documento NCL. Assim, o autor de um documento hipermídia deste tipo não precisa se preocupar com a criação e desenvolvimento de conectores se ele não tiver esta necessidade. Para importar uma base de conectores, podemos incluir o seguinte código no cabeçalho de programa:

```
<connectorBase>
        <importBase alias="connectors" baseURI="connectorBase.ncl" />
</connectorBase>
```

O parâmetro `alias` dá um "apelido"à base de conectores. Este `alias` será referenciado quando da criação de elos. O parâmetro `baseURI` diz ao documento NCL onde ele deve procurar pela base de conectores a ser importada.

Um elo é criado utilizando-se os conectores e aplicando nós de mídia ou contexto a papéis (ou *roles*) estabelecidos pelo conector. Vejamos um exemplo:

```
<link id="lVideo1Titulo1Start"
        xconnector="connectors#onBegin1StartN">
        <bind component="video1" role="onBegin" />
        <bind component="titulo1" role="start" />
</link>
```

O atributo `id` dá um nome único ao elo estabelecido. O parâmetro `xconnector` indica qual será o conector utilizado. Ele pode ser usado de duas formas: no caso de uma base de conectores importada, dá-se o `alias` do conector e o nome do conector após o sharp (#). Neste caso, a base de conectores tem como `alias` "connectors" e o conector usado é o "onBegin1StartN". Como o próprio nome indica, este conector liga vários nós de forma que, quando um determinado nó começa, diversos outros são iniciados simultaneamente.

Para indicar quais nós desempenharão os "papéis" determinados pelo conector, usa-se estruturas do tipo `bind`. Nestas estruturas, o parâmetro `component` deve indicar a `id` de um nó (no caso de um nó de contexto, deve-se ainda especificar uma porta usando o atributo interface), enquanto o parâmetro `role` deve indicar qual será o papel que este nó terá na execução deste elo. No exemplo dado, o nó `video1` tem o papel `onBegin` e o nó `titulo1` tem o papel `start`. Assim sendo, quando a execução do nó `video1` for iniciada, este nó será disparado, iniciando assim a execução do nó `titulo1`.

Dois tipos de conectores podem ser definidos: conectores causais e conectores de restrição.

Os conectores causais definem dois ou mais papéis. Estes papéis indicarão quais as condições sob as quais o elo será ativado e que ações serão efetuadas quando da ativação deste elo. A seguir, é dado um exemplo de conector causal:

```
<causalConnector id="onBegin1Start1">
        <conditionRole id="onBegin" eventType="presentation">
                <eventStateTransitionCondition transition="starts"/>
        </conditionRole>
        <actionRole id="start" eventType="presentation">
                <presentationAction actionType="start"/>
        </actionRole>
        <causalGlue>
                <simpleTriggerExpression conditionRole="onBegin"/>
                <simpleActionExpression actionRole="start"/>
        </causalGlue>
</causalConnector>
```

O atributo `id` dá um nome ao conector. Em seguida, criam-se os papéis: um `conditionRole` estabelece uma `condição` sob a qual o elo que utilize este conector será ativado. Vê-se que foi criado um papel "onBegin", equivalente à transição "starts" em um evento do tipo "presentation". Ou seja, isto significa que o elo será ativado quando a apresentação do nó que ocupa este papel for iniciada.

Logo depois, cria-se um papel de nome "start". Ele é um papel do tipo `actionRole`, o que significa que ele corresponde a uma ação que será efetuada por um elo que utilize este conector. Este papel corresponde a uma ação "start" no evento do tipo "presentation". Isto é, quando o elo for ativado, a apresentação do nó que ocupa este papel será iniciada.

Por fim, é preciso mostrar claramente qual é a lógica que liga os papéis: para isso, criamos o campo `causalGlue`. Este campo indicará qual é a "cola" que liga os papéis criados anteriormente. Associa-se o `conditionRole` ao "Trigger" do evento, e o `actionRole` ao "Action". Assim, fica claro que o `conditionRole` será o gatilho do elo, enquanto o `actionRole` será a ação a ser executada pelo elo.

A.3 Âncoras

Uma âncora é um ponto de entrada em um nó, seja ele de mídia ou de contexto. Quando se usa âncoras, objetiva-se usar um segmento de um nó como origem ou destino de um elo. Há dois tipos de âncoras: âncora de conteúdo e âncora de atributo.

Uma âncora de conteúdo delimita uma parte da mídia a ser utilizada como ativadora de um elo. Assim sendo, ela consiste em uma seleção de um determinado "pedaço" de uma mídia. Por exemplo, podemos usar uma certa área de uma imagem, um certo trecho de um áudio ou vídeo, etc. A âncora é definida como um elemento area dentro de um nó de mídia. Um exemplo é dado a seguir:

```
<media type="video" id="video1" src="media/video1.mpg" descriptor="
   dVideo1">
        <area id="aVideoLegenda01" begin="5s" end="9s"/>
        <area id="aVideoLegenda02" begin="10s" end="14s"/>
        <area id="aVideoLegenda03" begin="15s" end="19s"/>
</media>
```

O atributo `id` dá um identificador à âncora. Além do `id`, a âncora deve delimitar a qual parte da mídia ela se refere. Assim, há atributos como `coords`, `begin`, `end`, `dur`, `first`, `last`, `text` e `position`, todos eles servindo ao mesmo objetivo: delimitar uma parte de uma imagem (no caso de `coords`), de um áudio ou vídeo (no caso de `begin`, `end`,`dur`, `first` e `last`) ou de um texto (`text` e `position`).

Já as âncoras de atributo referem-se a determinados atributos de nós que serão manipulados por elos. Estes atributos podem ser, por exemplo, o volume do som em um

áudio ou vídeo, dimensões de exibição de uma imagem, etc. Estas âncoras devem ser definidas como elementos do tipo `attribute` dentro de um nó de mídia ou contexto.

ANEXO B – Lua

Lua é uma linguagem de programação poderosa, rápida e leve, projetada para estender aplicações. Lua combina sintaxe simples para programação procedural com poderosas construções para descrição de dados baseadas em tabelas associativas e semântica extensível. Lua é tipada dinamicamente, é interpretada a partir de bytecodes para uma máquina virtual baseada em registradores, e tem gerenciamento automático de memória com coleta de lixo incremental. Essas características fazem de Lua uma linguagem ideal para configuração, automação (scripting) e prototipagem rápida (PUC RIO, 2015). Embora seja mais utilizado para viabilizar scripts que trabalhem como uma extensão do programa principal, Lua pode ser usada inteiramente para a construção de um sistema mais complexo com milhares de linhas de código e sendo este segundo mais difícil de ser visto. A linguagem Lua está na versão 5.2, possui a licença do MIT, ou seja, é uma linguagem de programação que é livre e de código aberta para que as pessoas utilizem como bem entenderem, sendo que estas podem utilizar para uso comercial sem ter a necessidade de arcar com custos ou burocracia sobre a utilização da Linguagem Lua em seu projeto.

B.1 MetaTabelas

Metatabelas são tabelas comuns do Lua que consegue manipular o valor recebido com certas operações especiais, sendo possível alterar vários aspectos comportamentais das operações sobre um valor especificando campos específicos na metatabela do valor, ou seja, a partir de um campo de uma metatabela pode-se alterar os valores armazenados nelas utilizando-se de operações especificas, configuradas ou não, das metatabelas.

Uma metatabela controla como um objeto se comporta em vários aspectos como operações aritméticas, comparações com relação à ordem, concatenação, operação de comprimento e indexação, assim uma metatabela também pode definir uma função a ser chamada quando um objeto userdata é coletado pelo coletor de lixo. Para cada uma destas operações Lua associa uma chave específica denominada de “evento”. Quando é realizado uma destas operações sobre um valor, Lua verifica se este valor possui uma metatabela com o evento correspondente, caso seja encontrado o valor associado àquela chave (o metamétodo) controla como Lua irá se comportar referente a operação que a chamou.

B.2 Valores e Tipos

Lua é uma linguagem dinamicamente tipada, ou seja, não é necessário declarar o tipo da variável e suas variáveis são definidas em tempo de execução dependendo do

que estiver sendo armazenado no momento. Lua possui oito tipos básicos de variáveis que são: nil, boolean, number, string, function, userdata, thread e table, cada qual com sua característica do tipo de variável, nil como sendo um valor único, geralmente é utilizado para indicar valor nulo.

Boolean indica valor booleano true ou false, com uma diferença que uma expressão será considerada falsa caso o valor seja igual à nil ou a false e verdadeira em todas as outras situações, mesmo quando o valor for zero. Diferente das demais linguagens, Lua possui somente um tipo numérico que é ponto flutuante por padrão, que é o number, o tipo number pode representar um qualquer número inteiro de 32 bits.

O tipo string representa uma cadeia de caracteres, que pode ser delimitado por aspas simples ou duplas, nesse caso a regra é igual à de outras linguagens como o Javascript, se uma cadeia de caracteres começar com aspas simples deve terminar com aspas simples, assim como com aspas duplas. Cadeias de caracteres são únicas, então toda vez que uma cadeia de caracteres do tipo string é alterada, na verdade a pessoa que está programando está criando uma nova cadeia e cada cadeia de caracteres pode armazenar qualquer caractere de tamanho de 8 bits.

Function é o tipo que representa funções em Lua e também pode representar as funções escritas em C. Funções em Lua são consideradas valores de primeira classe o que significa que funções podem ser armazenadas em variáveis, passadas como parâmetros, ou retornadas como resultados. O tipo userdata permite que dados quaisquer em C possam ser armazenados em variáveis Lua, este tipo corresponde a um bloco de memória e não possui operações pré-definidas em Lua exceto atribuição e teste de identidade, além do que somente é possível alterar esses valores utilizando-se da API C, garantindo assim a integridade dos dados do programa principal.

O tipo thread representa fluxos de rotinas independentes usados para implementar co-rotinas, com isso Lua dá suporte a todas as co-rotinas dos sistemas.

O tipo table implementa arrays associativos, em outras palavras são arrays que podem ser indexados não somente por números, mas sim por qualquer valor diferente de nill. As tabelas dentro da linguagem Lua podem ser heterogêneas podendo conter todo e qualquer tipo de valor. As tabelas são os únicos mecanismos de estruturar os dados, podendo, dependendo dos valores passados representar arrays comuns, tabelas de símbolos, conjuntos, registros, grafos, árvores, etc. Para representar registros Lua utiliza o nome do campo como um índice.

Valores dos tipos table, function, thread e userdata são objetos, as variáveis não contêm realmente os valores, apenas referencia para eles. Atribuição, passagem de parâmetro, e retorno de funções sempre lidam com referências para tais valores; estas operações não efetuam nenhum tipo de cópia de dados do valor referenciado.

B.3 Variáveis

Variáveis em qualquer linguagem tem a função de armazenar um valor determinado pelo programa ou usuário para que este seja utilizado em um operação especificada pelo programa em tempo de execução. Em Lua existem três tipos de variáveis: variáveis globais, variáveis locais e campos de tabela. Um nome qualquer pode denotar uma variável global ou uma variável local (ou um parâmetro formal de uma função, que é um caso particular de variável local). Em Lua a menos que a variável seja declarada como variável local todas as variáveis são globais, variáveis locais possuem escopo léxico e podem ser acessadas por funções definidas dentro do seu escopo.

```
local x = 33
local y
print (x,y) -- 33 nil
if x > 10 then
        local x = 5 -- alterand um "x" local
        y = 9
        print(x,y) -- 5 9
else
        x = 2 -- alterando "x" mais externo
        print(x,y) -- 2 nil
end
print(x,y) -- 33 9
```

Qualquer variável antes de receber um valor tem como valor padrão o tipo nil; e acesso as variáveis globais e a campos de tabelas podem ser mudadas através do uso de metatabelas. Todas as variáveis globais são mantidas como campos em tabelas Lua comuns, chamadas de tabelas de ambiente ou simplesmente de ambientes e cada função tem faz sua própria referência a um ambiente, sendo que todas as variáveis globais dentro de uma função irão se referir para esta tabela de ambiente. Quando uma função é criada, ela herda o ambiente da função que a criou.

Em Lua é possível atribuir valores em diversas variáveis em um mesmo comando, seguindo uma ordem de atribuição o valor vai corresponder a variável que se encontra na mesma posição, como se fosse um "array", sendo que quando o número de variáveis declaradas é diferente dos valores atribuídos a elas a linguagem ajusta automaticamente as listas, preenchendo com valores do tipo nil ou desprezando os valores a mais.

Exemplo:

```
a, b = 2
c, d = 2, 4, 6
```

No exemplo acima "a" recebe o valor 2, enquanto "b" recebe o valor nil por não ter sido atribuído nenhum valor a ele. Enquanto na segunda linha as variáveis "c" recebe 2,

"d" recebe 4 e o valor "6" é desprezado por não haver nenhuma variável que possa receber esse valor.

Uma das vantagens sobre a atribuição múltipla de valores para as variáveis é que pode-se trocar os valores entre as variáveis sem a necessidade de uma variável auxiliar.

```
A = 3
B = 7
A,B = B,A
```

Nesse exemplo as saídas das variáveis serão A = 7 e B = 3. Isso ocorre porque ao executar o a atribuição de valores múltipla o valor ao ser passado para variável ainda está sendo refenciado, então a variável só irá assumir o novo valor ao término da linha de comando que está executando a ação de troca de valores. Outra característica das variáveis em Lua é que ao ela podem receber resultados de formulas com operadores relacionais. Segue o exemplo abaixo

```
a = 4 < 3
b = 4 > 3
```

Nesse caso os operadores relacionais ao serem atribuídos a uma variável têm dois tipos de retorno nil quando o resultado é false e 1 quando for verdadeiro. No caso do exemplo passado a variável "a" recebe o valor nil, enquanto b recebe o valor 1.