UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM ESTUDO SOBRE O DESENVOLVIMENTO DE JOGOS ORIENTADO A MODELOS COM CODIFICAÇÃO MANUAL

ELY FERNANDO DO PRADO

ORIENTADOR: PROF. DR. DANIEL LUCRÉDIO

São Carlos – SP

Agosto/2014

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM ESTUDO SOBRE O DESENVOLVIMENTO DE JOGOS ORIENTADO A MODELOS COM CODIFICAÇÃO MANUAL

ELY FERNANDO DO PRADO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software

Orientador: Prof. Dr. Daniel Lucrédio

São Carlos – SP

Agosto/2014

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

P896ed

Prado, Ely Fernando do.
Um estudo sobre o desenvolvimento de jogos orientado a modelos com codificação manual / Ely Fernando do Prado. -- São Carlos : UFSCar, 2015.
133 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2014.

1. Engenharia de software. 2. Desenvolvimento orientado por modelos. 3. Jogos eletrônicos. 4. Flexibilidade. 5. Eficiência. I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Um Estudo sobre o Desenvolvimento de Jogos Orientado a Modelos com Codificação Manual"

Ely Fernando do Prado

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Prof. Dr. Daniel Lucrédio
(Orientador - DC/UFSCar)

Profa. Dra. Vânia Paula de Almeida Neris
(DC/UFSCar)

Prof. Dr. Esteban Clua
(UFF)

São Carlos
Setembro/2014

De maneira geral, dedico este trabalho a todos aqueles que duvidam de sua capacidade, para que saibam que é possível chegar a qualquer lugar que os sonhos têm alcance.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter iluminado meus caminhos e principalmente por ter colocado diante de mim pessoas espetaculares que contribuíram positivamente no meu trabalho e na minha vida.

Tenho eterna gratidão pelos meus pais que me ensinaram com palavras e com exemplos os valores que carrego. Agradeço também meus irmãos, pois cada um do seu jeito colaborou para que eu me tornassem quem eu sou hoje: Eliana pelo seu espírito inovador, Elias pelos pés no chão e Elisângela pelo coração de amor. Me considero o resultado da mistura de vocês três.

Não poderia deixar de agradecer também a pessoa que me fez enxergar o mundo por outra perspectiva, me fazendo ver que eu poderia ir muito mais longe. Kenia, mais que uma namorada, você tem sido uma verdadeira parceira que me acompanhou e também me instigou a encarar os mais diversos desafios, inclusive o de me aventurar a entrar no mestrado. Seu amor, carinho, sinceridade e cumplicidade foram fundamentais para que eu chegasse até aqui.

Meus sinceros agradecimentos ao meu orientador, Daniel, por ter conseguido lapidar uma pedra bruta que eu era. Obrigado por ter compreendido minhas dificuldades de conhecimento, tempo e distância. Sem suas anotações, conversas gravadas, conexões remotas, reuniões e incontáveis e-mails seria impossível ter concluído este trabalho. Aos poucos você me mostrou que fazer ciência e fazer programas de computador são duas coisas completamente diferentes.

Agradeço os membros da minha banca de qualificação Dr. Fabiano Ferrari e Dra. Vânia Neris pelas importantes contribuições ao trabalho. Também agradeço todos os professores do PPG-CC pelos valiosos ensinamentos a mim oferecidos.

Obrigado aos meus companheiros de mestrado pela receptividade e apoio, especialmente aos colegas do LaBDES. Obrigado ao Elias baiano que como um irmão mais velho me serviu de inspiração e me socorreu diversas vezes. Agradeço por fim meus colegas de trabalho Alysson, Fernando, Davidson e toda equipe de professores da Libertas e Unifran pela cooperação nas minhas pesquisas.

Os que se encantam com a prática sem a ciência são como os timoneiros que entram no navio sem timão nem bússola, nunca tendo certeza do seu destino.

Leonardo da Vinci

RESUMO

Nos últimos anos, a indústria dos jogos eletrônicos tem assistido a um crescimento exponencial em termos de novos títulos e também na complexidade dos jogos. Este crescimento é responsável por muitos avanços nas tecnologias computacionais. Mas também revelou problemas relacionados com o processo de desenvolvimento. É muito comum que projetos de jogos excedam o orçamento e o cronograma, para não mencionar o elevado número de jogos com defeitos entregues ao consumidor final. As ferramentas atuais que apoiam o desenvolvimento de jogos, tais como motores de jogo, permitem grande flexibilidade e liberdade artística, porém elas necessitam de profissionais com alto grau de conhecimento tecnológico, o que significa que muitas vezes é necessário um alto investimento financeiro. Para ajudar a resolver esses problemas, a literatura sugere o uso de *Model-Driven Development* (MDD). O aumento do nível de abstração oferecido pelo MDD pode conduzir a maiores níveis de reutilização, e da automação conseguida na geração de código ajudando a reduzir o tempo de desenvolvimento. Mas para ocorrer a automação, o desenvolvedor muitas vezes tem que pagar o preço da redução da flexibilidade já que no MDD existe a falta de flexibilidade nos projetos, porém com código manual isso pode ser superado. Diante desses desafios é proposta neste trabalho uma abordagem de desenvolvimento de jogos dirigida a modelos capaz de oferecer eficiência sem perder a flexibilidade nos projetos. Esse objetivo pode ser alcançado por meio da integração de múltiplas DSLs com códigos escritos à mão através da definição de padrões de projetos específicos. Para avaliar essa abordagem, foi desenvolvida uma ferramenta capaz de auxiliar o desenvolvimento de jogos seguindo os conceitos do MDD, a qual posteriormente foi submetida a experimentos para validar sua eficácia. De acordo com a análise experimental foi possível observar que a abordagem proposta pode oferecer a eficiência e flexibilidade desejada no desenvolvimento de jogos. Também foi notado que devido as automatizações oferecidas pelos geradores de código, desenvolvedores inexperientes puderam criar seus projetos com maior facilidade. Contudo o estudo indica que a abstração oferecida pelo MDD dificulta o aprendizado da tecnologia utilizada.

Palavras-chave: Desenvolvimento Orientado a Modelos, MDD, Jogos, Flexibilidade, Eficiência

ABSTRACT

The electronic gaming industry has recently witnessed an exponential growth in terms of titles and overall complexity of the games. This growth is responsible for many advances in computational technologies. But it also revealed problems related to the development process. It is very common that game projects exceed budget and schedule, not to mention the high number of bugs delivered to the final consumer. Current tools that support the development of games, such as game engines, allow great flexibility and artistic freedom, but require professionals with a high degree of technological knowledge, which means that a high financial investment is often needed. To help solving these problems, the literature suggests the use of Model-Driven Development (MDD). The increased abstraction level made possible through MDD can lead to higher reuse levels, and the automation achieved by code generation helps to reduce development time. But for automation, the developer often has to pay the price of reduced flexibility since the MDD exists a lack of flexibility in designs, but with manual code that can be overcome. Given these challenges, it is proposed an approach for game development that targets models to offer efficiency but without losing the flexibility in projects. This goal can be achieved through the integration of multiple DSLs with handwritten code, with the help of design patterns. To evaluate this approach, it was developed a tool capable of assisting the development of games following the concepts of MDD, which was subsequently subjected to experiments to validate its effectiveness. According to the experimental analysis, it was possible to observe that the proposed approach can provide the desired efficiency and flexibility in game development. It was also noted that due to the automation offered by code generators, inexperienced developers could create their projects with greater ease. However the study indicates that abstraction offered by MDD hinders the learning of the technology used.

Keywords: Model Driven Development, MDD, Games, Flexibility, Efficiency

LISTA DE FIGURAS

GLOSSÁRIO

API – *Application Programming Interface*

DSL – *Domain-specific language*

GCM – *Game Content Model*

GPL – *General-Purpose Language*

GUI – *Graphical User Interface*

MDD – *Model-Driven Development*

MDGD – *Model-Driven Game Development*

NPC – *Non-player characters*

SLGML – *SharpLudus Game Modelling Language*

UML – *Unified Modeling Language*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

Os jogos eletrônicos são uma das indústrias do entretenimento mais rentáveis do mundo hoje em dia. Petrillo et al. (2009) constataram que a renda dos jogos no ano de 2009 esteve na casa dos bilhões de dólares, superando até mesmo a receita da indústria do cinema. Segundo Brightman (2014), em 2014 espera-se arrecadar em torno de 64 bilhões de dólares com jogos eletrônicos, com expectativa de chegar em poucos anos à marca de 100 bilhões de faturamento anual. Este crescimento exponencial da demanda por novos e melhores títulos tem feito com que a indústria dos jogos eletrônicos tenha que confrontar diversos desafios.

Dentre os problemas enfrentados pela indústria dos jogos eletrônicos, vale destacar aqueles relacionados à tecnologia, pois normalmente os jogos, principalmente os jogos comerciais, exigem que sejam utilizadas tecnologias de ponta, incluindo plataformas ainda não consolidadas, as quais exigem uma grande curva de aprendizado. Alinhada a este problema, está a dificuldade em se encontrar profissionais capacitados no mercado de trabalho. Com esses problemas o planejamento do tempo de desenvolvimento também fica prejudicado, já que é difícil prever as dificuldades que serão encontradas durante o desenvolvimento do projeto (PETRILLO et al., 2009).

A abordagem de desenvolvimento orientado a modelos (*Model-Driven Development* ou MDD) se oferece como boa alternativa para elevar o nível de abstração no desenvolvimento de jogos e resolver alguns dos problemas enfrentados pela indústria. Ela se baseia na utilização de modelos que representam formalmente aspectos do software, como a estrutura, o comportamento e as necessidades de um determinado domínio. Desta forma é oferecido um aumento na produtividade, promoção da interoperabilidade e portabilidade entre diferentes plataformas, além de tornar a manutenção de software mais fácil produzindo códigos com uma melhor qualidade (TANG; HANNEGHAN, 2011a).

No MDD os modelos são os artefatos primários do desenvolvimento e os desenvolvedores contam com transformadores capazes de automatizar o processo de codificação dos modelos em sistemas executáveis. Um dos principais objetivos do MDD é proteger os desenvolvedores das complexidades da plataforma de implementação subjacente (FRANCE; RUMPE, 2007).

Para possibilitar a criação de modelos é necessária uma ferramenta de modelagem, onde através dela o engenheiro de software possa produzir os modelos que descrevam os conceitos do domínio. Normalmente uma linguagem específica de domínio, ou DSL (*Domain-Specific Language*) é utilizada para implementar estas características (DEURSEN; KLINT; VISSER, 2000). As DSLs normalmente são intuitivas e fáceis de se utilizar, ao mesmo tempo em que os modelos criados por elas são semanticamente completos e corretos, permitindo sua compreensão pelo computador sem que haja enganos (LUCREDIO, 2009). As DSLs consistem em construções que captam fenômenos no domínio que elas descrevem, podendo cobrir uma vasta gama de formas. Elas podem fazer comunicação entre componentes de software ou podem ser incorporadas em um assistente que pede ao usuário de forma iterativa para obter informações de configuração, por exemplo (FRANCE; RUMPE, 2007).

Existem também trabalhos que demonstram a utilização da abordagem MDD com GPLs (*General-Purpose Languages*), como a UML, mas as DSLs são mais bem aplicáveis quando se tem domínios mais específicos, como é o caso dos jogos computacionais (WALTER; MASUCH, 2011).

Os modelos servem de entrada para transformações. Os motores de transformação analisam alguns aspectos dos modelos e, em seguida, geram os artefatos resultantes, como o código-fonte, as entradas de simulação, as descrições de implantação, ou representações de modelos alternativos. Esse processo de desenvolvimento reduz tarefas tediosas e propensas a erros (SCHMIDT, 2006).

Apesar de ser uma abordagem atraente, o MDD não tem sido utilizado pela indústria de jogos eletrônicos e poucas das DSLs foram utilizadas fora das universidades (DORMANS, 2012).

De acordo com Dormans (2012), atualmente não é possível planejar e projetar um jogo no papel e esperar bons resultados simplesmente construindo o jogo exatamente conforme sua especificação, pois muitas vezes o que parece ser bom no papel não funciona como esperado. Dessa forma, o ideal é que os projetos de jogos tenham grande flexibilidade, ou seja, devem ser capazes de aceitar alterações em suas funcionalidades fazendo com que o jogo possa ser alterado durante o desenvolvimento e possa ficar diferente daquilo que foi estipulado no seu escopo inicial. Neste ponto, o MDD é tido como prejudicial à criatividade dos desenvolvedores, já que torna o projeto mais rígido (FURTADO; SANTOS; RAMALHO, 2011). Assim há uma necessidade

do projeto ter suporte a variabilidade a fim de manter um grande número de customizações no projeto.

Uma forma de se permitir flexibilidade em projetos de jogos com MDD é fazendo com que se utilize múltiplas DSLs, conforme proposto por Dormans (2012), juntamente com a possibilidade de haver integração entre código gerado e o código inserido manualmente. Esta abordagem é também sugerida por Furtado, Santos e Ramalho (2011) e Jovanovic et al. (2011). No entanto, Dormans (2012) relata que o uso de vários modelos também apresenta seus problemas, pois nem sempre é clara a forma como as transformações podem ser definidas entre os vários modelos. Além disso, ainda há poucos estudos que evidenciam o sucesso dessa abordagem na área dos jogos computacionais, sendo necessário mais pesquisas para aumentar o conhecimento sobre o assunto.

1.1 Objetivo

Este trabalho tem por objetivo propor uma abordagem de desenvolvimento de jogos que permita unir os benefícios do MDD, porém sem perder a flexibilidade necessária para projetos de jogos. Dentre esses benefícios, o principal deles é o ganho de produtividade, fazendo com que o desenvolvimento seja mais eficiente. Pesquisas mostradas por Furtado, Santos e Ramalho (2011) e (DORMANS, 2011) por exemplo, já comprovaram que o MDD traz resultados mais rápidos no desenvolvimento de jogos eletrônicos, porém eles tiveram problemas quanto à flexibilidade dos projetos.

Portanto este trabalho visa buscar mecanismos que permitam definir uma abordagem de desenvolvimento de jogos capaz de agilizar os atuais processos aumentando o nível de abstração das tecnologias utilizadas, como os motores de jogos, porém permitindo que sejam inseridos códigos manuais a fim de deixar o projeto o mais flexível possível. Assim poderão ser automatizados processos no desenvolvimento com MDD e, ao mesmo tempo, manter o projeto flexível para receber personalizações.

Para alcançar este objetivo foram exploradas as técnicas de integração entre DSLs, bem como os mecanismos para incorporar código manual juntamente com o código gerado pelos transformadores do MDD, seguindo as sugestões de Dormans (2012), Furtado, Santos e Ramalho (2011) e Jovanovic et al. (2011). Com isso, espera-se contribuir para a resolução dos problemas enfrentados pela indústria dos jogos eletrônicos.

De forma complementar esta pesquisa procurou alcançar também contribuições relacionadas ao aprendizado da tecnologia oferecido pela abordagem proposta. De acordo com Deur-

sen, Klint e Visser (2000), o aumento no nível de abstração dos códigos quando se programa utilizando DSLs, ao invés de código nativo, traz uma maior facilidade no desenvolvimento. Além disso, as DSLs normalmente têm uma sintaxe mais simplificada devido seu nível maior de abstração, o que faz com que a curva de aprendizado nesse tipo de abordagem seja menor. Dessa forma é permitido que profissionais com pouco conhecimento e experiência possam produzir seu próprios projetos. Porém ao abstrair detalhes técnicos do desenvolvimento, pode haver uma maior dificuldade no aprendizado da tecnologia, dificultando assim que sejam realizadas alterações manuais no projeto através da inserção de código nativo. Nesta dissertação procurou-se avaliar se essa dificuldade realmente existe em projetos de jogos eletrônicos com MDD e se ela pode ser amenizada quando os geradores criam código que seguem padrões de projeto específicos para permitir a inserção de código manual.

Para alcançar esses objetivos, após uma pesquisa exploratória sobre os temas relacionados, foram desenvolvidos protótipos de jogos sem o uso da abordagem MDD. Tais protótipos utilizaram um motor de jogo, semelhante ao que ocorre na indústria atualmente. Essa análise teve como finalidade levantar os artefatos que podem ser automatizados pelos transformadores do MDD. Também foram definidos quais artefatos deveriam ser codificados manualmente a fim de permitir maior flexibilidade no desenvolvimento.

Com base nessa análise inicial, foi desenvolvida uma ferramenta MDD que engloba a definição dos modelos, *templates*, transformadores de código e editores visuais. Através dessa ferramenta procurou-se avaliar a abordagem de desenvolvimento proposta através de análise experimental. Em suma, essa análise tem por objetivo **avaliar a eficiência, flexibilidade e aprendizado proporcionado pela abordagem MDD em projetos de jogos eletrônicos com a inserção de códigos manuais e utilização de padrões de projeto**.

1.2 Estrutura da Dissertação

Esta proposta está organizada em outros cinco capítulos, além deste capítulo introdutório, com os seguintes temas:

- O Capítulo 2 apresenta o levantamento bibliográfico acerca dos conceitos de jogos eletrônicos e desenvolvimento orientado a modelos. Também são levantadas algumas abordagens que apoiam a utilização de múltiplas DSLs e que permitem a inserção de código manual.
- O Capítulo 3 reúne algumas abordagens encontradas na literatura que descrevem a aplicação do desenvolvimento orientado a modelos para criação de jogos eletrônicos.

- O Capítulo 4 descreve o desenvolvimento de protótipos de jogos utilizando a abordagem tradicional com um motor de jogo e logo em seguida apresenta estudos dos subdomínios desses protótipos. A partir desse estudo são descritos os processos de desenvolvimento da ferramenta MDD para o desenvolvimento de jogos.

- No Capítulo 5 são apresentados os detalhes da experimentação da abordagem de desenvolvimento proposta neste trabalho. Também são discutidos os resultados dos experimentos com base nas métricas definidas neste mesmo capítulo.

- O Capítulo 6, por fim, apresenta algumas conclusões incluindo as contribuições e síntese dos principais resultados, além de possibilidades de trabalhos futuros.

Capítulo 2

LEVANTAMENTO BIBLIOGRÁFICO

2.1 Jogos Eletrônicos

Jogos eletrônicos são aplicações de software interativas criadas principalmente para fins de entretenimento. Porém hoje em dia, pode-se estender os objetivos dos jogos para as áreas da saúde, publicidade, treinamento, educação, ciência, pesquisa, produção e trabalho (TANG; HANNEGHAN, 2011b). Os primeiros jogos eletrônicos foram desenvolvidos em departamentos de pesquisa de universidades e instalações militares, quando estudantes, programadores, professores e pesquisadores destas instituições transformavam seus computadores *mainframe* em máquinas de jogos. Neste cenário, o primeiro jogo eletrônico memorável surgiu em 1958 quando Willy Higinbotham, dos Laboratórios Nacionais Brookhaven em Nova York criou o jogo *Tennis for Two*, capaz de simular uma partida de tênis em um computador analógico (NOVAK, 2011).

O desenvolvimento de jogos é uma tarefa que exige muita dedicação tanto no lado artístico como técnico. As pessoas que têm boa criatividade e conhecimento de computação são os melhores candidatos para formar uma equipe de desenvolvimento de jogos (TANG; HANNEGHAN, 2011b).

Um jogo eletrônico é uma aplicação de software. Deste modo, no seu desenvolvimento são realizadas tarefas semelhantes às encontradas no desenvolvimento de outros tipos de software, como análise de requisitos, projeto e codificação. Em se tratando da tarefa de codificação, Blow (2004) afirma que para a programação em PCs, normalmente utiliza-se um ambiente de desenvolvimento do próprio compilador, como o Microsoft Visual Studio por exemplo.

2.1.1 Ferramentas para desenvolvimento de jogos

Visando facilitar o desenvolvimento de jogos eletrônicos, foram criadas ao longo dos anos diversas ferramentas. Furtado e Santos (2006) afirmam que ocorreu uma grande evolução nessas ferramentas de desenvolvimento de jogos, e classificam-nas em três categorias: APIs multimídia, ferramentas visuais e motores de jogos.

APIs multimídia são bibliotecas que podem ser utilizadas para ter acesso direto ao hardware, lidando com placas gráficas, placas de som e dispositivos de entrada. Elas permitem que o desenvolvedor não tenha que se preocupar com as peculiaridades de baixo nível dos dispositivos. Furtado e Santos (2006) relatam que as APIs multimídia são muito utilizadas e certamente irão permanecer em uso por muito tempo, diretamente ou indiretamente. As APIs multimídia amplamente utilizadas são a OpenGL e Microsoft DirectX (FURTADO; SANTOS, 2006).

Enquanto essas bibliotecas lidam com as funções de baixo nível, o jogo em si ainda tem que ser programado manualmente. As APIs Multimídia tratam de funcionalidades mais genéricas, não oferecendo o nível de abstração desejado pelos programadores de jogos. Recursos como tratamento de transição entre estados do jogo, modelagem de comportamento ou inteligencia artificial não são apoiadas por estas APIs (FURTADO; SANTOS, 2006). O desenvolvimento de jogos utilizando as APIs é realizado através de códigos, e não visualmente, impedindo automações e consequentemente diminuindo a produtividade do desenvolvimento. Outro ponto negativo é que todo o código da lógica do jogo é escrito junto às chamadas das funções da API, diminuindo a legibilidade do código e dificultando sua modularidade (CALIXTO; BITTAR, 2010).

Ferramentas visuais para a criação de jogos visam a criação de jogos completos sem que seja necessário nenhuma programação, normalmente interagindo apenas com o mouse. Nestas ferramentas, o usuário final é auxiliado com interfaces gráficas de fácil uso para a criação do jogo. As ferramentas podem gerar jogos de forma genérica ou podem ser focadas na criação de jogos pertencentes a um gênero específico de jogo, como é o caso do RPG Maker, ferramenta especializada na criação de jogos de RPG (FURTADO; SANTOS, 2006).

Embora as ferramentas visuais possam parecer uma ótima solução, suas possibilidades são bastante limitadas. Alguns tipos de jogos podem ser produzidos assim, porém esta abordagem não é adequada para jogos comerciais, onde há uma exigência de maior complexidade para criação de jogos sofisticados. Mesmo com a possibilidade de utilizar linguagens de *script* para incluir alguma funcionalidade mais elaborada dentro destas ferramentas, elas continuam sendo mais indicadas para amadores e iniciantes no desenvolvimento de jogos. Isso se deve ao fato que essas linguagens exigem um conhecimento avançado, porém uma vez que os desenvolve-

dores ganham experiência em programação, eles preferem ter os benefícios de linguagens de programação orientadas a objeto, com o apoio de ambientes de desenvolvimento robustos e suporte a depuração, em vez de trabalhar com linguagens de script dentro de um ambiente que não foi originalmente concebido para codificação. (FURTADO; SANTOS, 2006) e (CALIXTO; BITTAR, 2010).

Motores de jogos, também conhecidos como *Game Engines* ou *Game Software Frameworks* contemplam o atual estado da arte das ferramentas de desenvolvimento de jogos. Eles podem ser considerados como um conjunto de APIs reutilizáveis, que reúnem funcionalidades comuns no desenvolvimento de jogos, como animações, gestão de fases, eventos, interação com o jogador, física, etc. Eles fornecem interfaces de programação onde os desenvolvedores não precisam se preocupar com detalhes de implementação de baixo nível, e eles não estão restritos às limitações impostas pelas ferramentas visuais (FURTADO; SANTOS, 2006). Os motores de jogos são construídos para gerenciar e realizar a comunicação entre os vários componentes do software sem falhas, facilitando o uso das APIs e outros componentes específicos. (TANG; HANNEGHAN, 2011b)

Calixto e Bittar (2010) citam ainda que os motores de jogos se diferem das APIs multimídia, pois com as APIs as chamadas às funções são feitas diretamente pelo código do jogo, enquanto que ao se utilizar um motor de jogo essas chamadas são encapsuladas e tratadas automaticamente. Desta forma, o desenvolvedor fica livre dos detalhes técnicos de baixo nível e das particularidades de implementação de uma API (CALIXTO; BITTAR, 2010).

Procurando estreitar seu domínio, os motores de jogos abordam apenas um subconjunto dos jogos. Por exemplo, um motor de jogo 3D tem muitas questões específicas diferentes de um motor de jogo para plataforma 2D (FURTADO; SANTOS, 2006). Os motores de jogos estão atrelados a estruturas específicas de jogos, tem componentes de tecnologia bem definidos, como gráficos, som, interface de usuário, física básica do jogo, etc. Por exemplo, *Unreal Engine 4* [1] é adequado para desenvolvimento de jogos de ação ou jogos de aventura com visão em primeira ou em terceira pessoa, requerendo uma reestruturação para apoiar jogos no gênero de esportes (TANG; HANNEGHAN, 2011b).

Devido à grande quantidade de motores de jogos disponíveis no mercado, deve ser feita uma análise para decidir qual o mais adequado para cada projeto. Nesta análise devem ser levados em consideração o orçamento, suporte, recursos oferecidos, facilidade de uso, adequação, preço e plataforma de hardware apoiados pelo motor de jogo (TANG; HANNEGHAN, 2011b). Devido à complexidade inerente dos motores de jogos, deve ser notado que há uma alta curva de

[1] http://www.unrealengine.com/unreal_engine_4

aprendizado para dominar essas ferramentas, podendo envolver custos consideráveis, tais como custos de aquisição, custos de treinamento, custos de personalizações e os custos de integração (FURTADO; SANTOS, 2006). Calixto e Bittar (2010) reforçam sobre a alta curva de aprendizado necessária para a utilização de motores de jogos. O uso requer conhecimentos sobre sua arquitetura, sintaxe e peculiaridades da programação, tornando o uso da ferramenta pouco intuitiva ao primeiro contato.

Tang e Hanneghan (2011b) classificam os motores de jogos de acordo com sua licença. Daqueles que possuem licenças comerciais, o *Unreal Engine*[2], *Source Engine*[3], *CryEngine*[4] e *Id Tech*[5] (anteriormente conhecido como *Quake Engine*) são alguns exemplos de motores de jogos altamente sofisticados que podem ajudar os desenvolvedores a produzir jogos de boa qualidade. Estas tecnologias estão disponíveis para o licenciamento e exigem um pagamento para cada titulo de jogo produzido ou para cada cópia do jogo vendido. Outro exemplo de ferramenta comercial é o *Torque Game Engine* (TGE)[6] que necessita de licenciamento, porém não cobra *royalties* como as citadas anteriormente. Também existem motores de jogos comerciais que estão disponíveis gratuitamente para fins educacionais, como é o caso do *3DSTATE 3D Engine*[7], *XNA*[8], *CryEngine 3*[9], *Unreal Development Kit*[10] (UDK) e *Unity 3D*[11] (TANG; HANNEGHAN, 2011b).

Outra categoria de motores de jogos são os de código aberto, que permitem que os desenvolvedores possam produzir seus jogos tanto para fins não comerciais como comerciais, sem nenhum custo. Dentre essas ferramentas, alguns exemplos são o *Apocalyx Engine*[12], *Baja Engine*[13], *Blender Game Engine* [14], *Irrlicht Engine*[15], *jMonkeyEngine*[16], *jPCT*[17], *Lilith3D*[18], *Object Oriented Graphic Rendering Engine* (OGRE)[19], *Panda3D*[20] e *Nebula Device 3*[21] (TANG;

[2]http://www.unrealengine.com/
[3]http://source.valvesoftware.com/
[4]http://www.crytek.com/cryengine
[5]http://www.idsoftware.com/
[6]http://www.garagegames.com/products/torque-3d
[7]http://www.3dstate.com/
[8]http://msdn.microsoft.com/xna/
[9]http://mycryengine.com/
[10]http://www.unrealengine.com/udk/
[11]http://unity3d.com/
[12]http://apocalyx.sourceforge.net/
[13]http://www.bajaengine.com/
[14]http://www.blender.org/
[15]http://irrlicht.sourceforge.net/
[16]http://jmonkeyengine.com/
[17]http://www.jpct.net/
[18]http://sourceforge.net/projects/lilith3d/
[19]http://www.ogre3d.org/
[20]http://www.panda3d.org/
[21]https://code.google.com/p/nebuladevice/

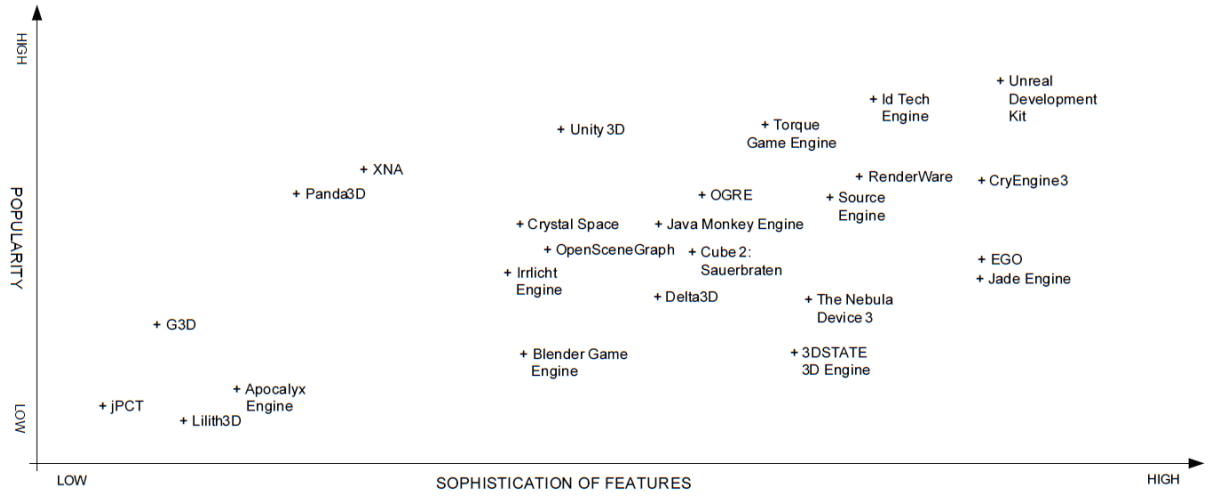

Figura 2.1: Motores de jogos. Tang e Hanneghan (2011b)

HANNEGHAN, 2011b). Esses são alguns exemplos de motores de jogos de código aberto que possibilitam a criação de jogos tridimensionais, porém existem muitos outros específicos para jogos bidimensionais ou para plataformas específicas como dispositivos móveis e consoles de videogame.

Segundo Blow (2004), a decisão de utilizar componentes e motores de jogos de terceiros deve ser sempre precedida por uma análise de custo / benefício detalhada. Não há nenhuma garantia de que a utilização destes componentes vai realmente acelerar o desenvolvimento, principalmente se eles falharem, o que é um risco a se considerar. Devido a este fato, muitas empresas desenvolvedoras de jogos eletrônicos optam por criar o seu próprio motor de jogo.

Os motores de jogos proprietários são criados pela própria empresa de desenvolvimento para seu uso interno. Geralmente eles têm uma qualidade superior ao motores de jogos comerciais ou de código aberto, mas dificilmente são aplicáveis a outros desenvolvedores fora da empresa que o criou. Como exemplo, pode ser citado o *RenderWare*[22], desenvolvido pela Criterion Games e adquirida pela Electronic Arts (EA). *EGO*[23] foi desenvolvido internamente pela Codemasters e é usado na produção de uma série de jogos como *Colin McRae: Dirt*, *Race Driver: Grid* e *Operation Flashpoint: Dragon Rising*. *Jade Engine*[24] da Ubisoft é usado em jogos como *Rayman Raving Rabbids I* e *II* e *Teenage Mutant Ninja Turtles*. A titulo de ilustração, Tang e Hanneghan (2011b) apresentam na Figura 2.1 informações sobre os motores de jogos mais conhecidos, onde é feita uma comparação sobre a sofisticação e a popularidade destes motores de jogos (TANG; HANNEGHAN, 2011b).

[22] http://www.renderware.com/
[23] http://www.codemasters.com/
[24] http://www.ubi.com/

2.1.2 Problemas no desenvolvimento de jogos

Apesar de todo sucesso, a indústria de jogos eletrônicos tem tido problemas. Vários desafios computacionais são descobertos no ambiente dos jogos, pois normalmente eles utilizam tecnologias sofisticadas e o hardware é frequentemente levado ao seu limite. Além dos desafios computacionais existem os problemas ligados ao processo de criação dos jogos, pois os prazos estão cada vez mais curtos, a complexidade dos jogos é cada vez maior e há uma crescente demanda por novo títulos (FURTADO; SANTOS, 2006).

A literatura especializada sobre o desenvolvimento dos jogos eletrônicos tem descrito problemas neste setor causados pela adoção errada ou parcial de metodologias no desenvolvimento, ou mesmo pela ausência de uma metodologia, fazendo com que os projetos extrapolem o tempo e o orçamento, podendo trazer diversos defeitos de construção (PETRILLO et al., 2009).

De acordo com Blow (2004), a parte mais difícil no desenvolvimento de um jogo sempre foi a da engenharia e com o aumento da complexidade dos jogos, o principal desafio técnico tem sido simplesmente obter o código que produza um resultado final semelhante a funcionalidade desejada.

Petrillo et al. (2009) relatam os seguintes problemas relacionados ao desenvolvimento de jogos:

- Definição de escopo: a incapacidade de estabelecer claramente o escopo do projeto faz com que os requisitos descobertos durante o desenvolvimento causem mudanças estruturais significativas na arquitetura do software gerando sérios problemas. Tal problema pode desencadear outro problema, que é o de corte de recursos durante o desenvolvimento, já que os recursos necessários extrapolaram o previsto.

- Planejamento de tempo: falta uma visão realista para estimar o plano inicial de tempo do projeto. O fato do desenvolvimento de jogos eletrônicos necessitar de uma equipe multidisciplinar pode provocar espera de trabalho de outros profissionais, gerando mais atrasos.

- Tempo crítico: Este problema se refere ao excesso de trabalho acumulado próximo ao fim do projeto, fazendo com que a carga de trabalho se concentre nesta fase, muitas vezes levando a equipe à exaustão, degradando a capacidade das pessoas em produzir um trabalho de qualidade.

- Problemas tecnológicos: todo jogo depende da tecnologia, o que representa um grande esforço e investimento de tempo. Este problema se agrava ao se trabalhar com uma nova

tecnologia ou plataformas ainda não consolidadas. Outros problemas comuns são gerados devido a falhas de programação de aplicativos de terceiros, como APIs ou motores de jogos utilizados no desenvolvimento.

- Equipe de profissionais: a dificuldade em encontrar profissionais experientes na área de desenvolvimento de jogos é algo recorrente. Também há relatos de problemas quando um desses profissionais deixa a equipe, o que aliado à falta de documentação do projeto, gera um grande problema no processo de desenvolvimento.

Inovações no processo de desenvolvimento têm marcado os jogos nos últimos tempos, gerando novas ferramentas, processos e estruturas de equipes. Seguindo estas tendências, pode-se imaginar quais serão as próximas inovações acerca do processo de desenvolvimento de jogos. Procurando fazer esta previsão, Novak (2011) apresenta uma entrevista com profissionais experientes e pertencentes a empresas significativas no mercado de jogos eletrônicos. As empresas em questão são: Gas Powered Games, Rainbow Studios e Ahlquist Software.

A empresa Gas Powered Games [25] foi fundada em maio de 1998. Já em seu primeiro projeto, *Dungeon Siege* de 2002 juntamente com sua continuação, *Dungeon Siege II* de 2005 conseguiram alcançar uma marca de 1,7 milhões de cópias vendidas. Esses jogos obtiveram grande sucesso de crítica e público. Outros projetos importantes são os jogos *Space Siege*, *Demigod*, *Supreme Commander* e *Supreme Commander 2*. A Gas Powered Games emprega mais de 60 designers, programadores, escritores e artistas.

Rainbow Studios [26] foi uma das maiores empresas desenvolvedoras de jogos do sudoeste norte-americano, tendo lançado jogos premiados para computadores e também para os principais consoles do mercado. Alguns exemplos são os jogos *Cars*, *Cars Mater-National Championship*, *Pixar getting a Kinect game* e *MX vs. ATV Alive*. Em 07 de novembro de 2001, a Rainbow Studios foi adquirida pela empresa THQ, grande nome no mercado de jogos eletrônicos. Apesar de ter produzido uma série de jogos de grande sucesso, a THQ declarou falência em 23/01/2013.

Ahlquist Software [27] é uma empresa com mais de 20 anos de experiência em desenvolvimento de software, incluindo ferramentas GUI (*Graphical User Interface*) de suporte para circuitos integrados, ferramentas de desenvolvimento web e programação de motores de jogos. Sua principal atividade atualmente se concentra no desenvolvimento de otimizações no motor

[25] Gas Powered Games, disponível em http://www.gaspowered.com/team.php. Acesso em: 24/05/2013

[26] Rainbow Studios, disponível em http://www.ign.com/companies/rainbow-studios. Acesso em: 24/05/2013

[27] Ahlquist Software, disponível em http://www.ahlquistsoftware.com/Pages/Resume.html. Acesso em: 24/05/2013

de jogo *Unreal 3* para computadores e para os consoles XBox 360 e PS3.

Em suas entrevistas, Novak (2011) obteve as seguintes informações:

- John Comes (Designer Chefe da Gas Powered Games) acredita que no futuro será gasto menos tempo com a tecnologia e mais tempo com os aspectos artísticos do jogo.
- Jay Gawronek (Diretor Técnico da Rainbow Studios) afirma que com o processo correto e as ferramentas adequadas será possível fazer com que os artistas, designers e programadores trabalhem de maneira mais consistente, tornando cada um desses profissionais menos dependentes um do outro, ao contrário do que ocorre normalmente onde o processo de desenvolvimento dos jogos segue um ciclo de vida onde os designers fazem um trabalho inicial, depois os artistas começam a criar os materiais para finalmente os programadores conduzirem o restante do projeto, levando muitas horas de trabalho.
- John Ahlquist (Fundador da Ahlquist Software) indica que o processo de programação de jogos usará mais práticas de engenharia de software, pois o crescimento das ferramentas de desenvolvimento torna os métodos improvisados ineficazes, e os processos de engenharia de software podem acelerar a produção.

Petrillo et al. (2009) afirmam que na verdade, todos os principais problemas que ocorrem na indústria tradicional de software também são encontrados na indústria dos jogos eletrônicos. Nessa linha de pensamento, pode-se imaginar que as mesmas soluções aplicadas à indústria de software, como o MDD por exemplo, podem ser aplicadas também na industria dos jogos eletrônicos. Petrillo et al. (2009) reforçam ainda problemas relacionados à análise de requisitos, onde são definidos objetivos irrealistas e requisitos indefinidos. O otimismo demasiado e a ingenuidade na definição do escopo são os fatores que elevam a ocorrência da maioria dos problemas desses dois setores. Porém jogos eletrônicos enfrentam um agravante, já que eles seguem uma tendência de ter uma complexidade maior na elaboração dos requisitos. Não existem métodos eficientes para determinar elementos subjetivos como diversão. Portanto faz-se necessário estender as técnicas tradicionais de engenharia de requisitos para apoiar o processo criativo no desenvolvimento de jogos eletrônicos (PETRILLO et al., 2009).

Outro problema recorrente na indústria dos jogos é o da comunicação entre as equipes. No desenvolvimento de jogos são necessárias tarefas de equipes multidisciplinares, incluindo artistas plásticos, músicos, roteiristas e engenheiros de software. Essa mistura de pessoas fornece um ambiente de trabalho mais criativo, mas acaba dividindo as equipes em ”artistas” e ”programadores” o que traz consigo sérios problemas de comunicação, uma vez que cada equipe utiliza vocabulários específicos para expressar suas ideias (PETRILLO et al., 2009).

Através dos estudos citados anteriormente, percebe-se que a indústria de jogos eletrônicos segue uma tendência de buscar jogos criativos, com escopo aberto, deixando os programadores e artistas livres para qualquer mudança no projeto. Ao mesmo tempo há uma cobrança cada vez maior por novos títulos em curtos espaços de tempo. Somado a isso existem as complicações tecnológicas, pois as atuais ferramentas de desenvolvimento exigem uma alta curva de aprendizado. De acordo com Petrillo et al. (2009), normalmente no desenvolvimento de jogos eletrônicos são utilizadas tecnologias avançadas que envolvem um grande esforço e investimento para seu aprendizado. Encontrar profissionais que dominam tais tecnologias ou capacitar desenvolvedores para que se adequem a elas envolve um grande esforço e investimento. Essa combinação de desafios muitas vezes torna a tarefa de desenvolver jogos algo muito dispendioso. Petrillo et al. (2009), Blow (2004), Furtado e Santos (2006) e Dormans (2012) concordam que existe a necessidade no mercado de jogos eletrônicos de se criar um novo paradigma de desenvolvimento que solucionem ou ao menos amenizem os problemas enfrentados por esse setor.

2.2 Model-Driven Development

O desenvolvimento de software orientado a modelos (MDD - *Model-Driven Development*) propõe a reutilização de código por meio de programação generativa, linguagens específicas de domínio e transformações de software. Desta forma são poupados esforços em tarefas repetitivas, as quais podem ser realizadas automaticamente com o auxílio de modelos (LUCREDIO; ALMEIDA; FORTES, 2012). Como forma de promover a reutilização, o MDD reduz a diferença semântica entre o domínio do problema e o domínio da implementação, por meio de modelos de alto nível, permitindo que os desenvolvedores não tenham que lidar com as complexidades da plataforma de implementação (FRANCE; RUMPE, 2007).

A proposta do MDD já foi pesquisada sob outras nomenclaturas, como MDE (*Model-Driven Enginneering*) (SCHMIDT, 2006) e MDSD (*Model-Driven Software Development*) (VOELTER; GROHER, 2007).

O MDD tem como ideia principal dar importância aos modelos no processo de software, de forma que esses modelos tornem-se parte integrante do software. Assim, o engenheiro de software não precisa interagir manualmente com todo o código-fonte, concentrando-se nos modelos de mais alto nível. A partir desses modelos, os transformadores de software conseguem gerar automaticamente o código-fonte, fazendo com que esses modelos deixem de ser apenas um guia e passem a fazer parte do software (VOELTER; GROHER, 2007).

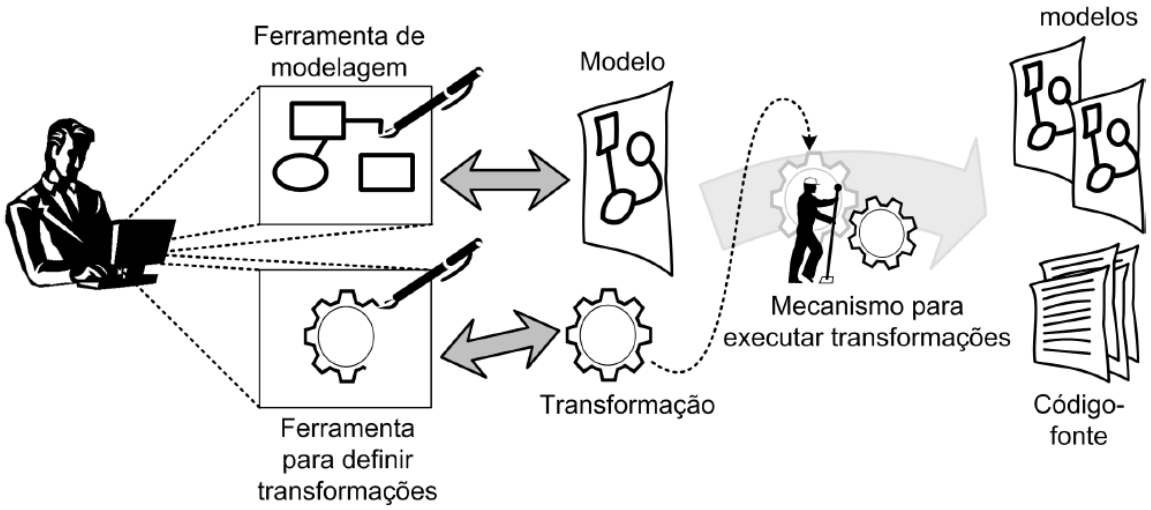

Figura 2.2: Principais elementos do MDD. (LUCREDIO, 2009)

Para seguir a abordagem MDD, fazendo as transformações automatizadas, são necessários alguns elementos, dentre os quais se destacam-se: (LUCREDIO, 2009)

- Ferramenta de Modelagem: ferramenta utilizada para produzir modelos que descrevem conceitos do domínio. Tal ferramenta deve ser de fácil utilização e deve ser capaz de criar modelos que sigam corretamente regras semânticas, já que eles serão interpretados por um computador. Normalmente essas características são implementadas utilizando uma linguagem específica de domínio (*Domain Specific Language* ou DSL).

- Ferramenta para definir transformações: permite que sejam construídas as regras de mapeamento de modelo para modelo ou de modelo para código, idealmente da forma mais natural possível.

- Modelos: servem de entrada para transformações a fim de gerar outros modelos ou o código-fonte. Eles representam o conhecimento do domínio da aplicação.

- Mecanismos para executar transformações: aplicam as transformações definidas pelo engenheiro de software mantendo informações de rastreabilidade que permitam saber a origem de cada elemento gerado.

- Novos modelos e código fonte: são o resultado esperado após a execução das transformações.

A Figura 2.2 representa os principais elementos do MDD, e como eles estão interligados entre si.

Linguagens são uma parte essencial no desenvolvimento de sistemas, sendo utilizadas para modelagem que abstrai a implementação ou baseada numa tecnologia de implementação específica. Muitas linguagens são de uso geral, por exemplo Java ou UML (*Unified Modeling Language*) que são aplicáveis a uma grande variedade de domínios de problemas. Em outras situações, pode-se utilizar linguagens específicas de domínio (DSL) que fornecem um conjunto altamente especializado de conceitos para atingir um domínio de problema mais restrito (HAHN, 2008). Normalmente as DSLs são pequenas, baseadas num conjunto de abstrações e notações restritos ao domínio considerado, assim pode-se trabalhar mais facilmente para resolver problemas ligados àquele domínio especificamente (DEURSEN; KLINT; VISSER, 2000).

Uma DSL pode ser do tipo textual (permitindo especificar programas) ou visual (permitindo especificar modelos ou diagramas). Normalmente uma DSL possui uma sintaxe abstrata utilizada para definir os conceitos do domínio, uma sintaxe concreta que permite representar os conceitos do domínio na forma textual ou através de notação gráfica utilizando diagramas, e uma sintaxe de serialização que permite persistir os modelos de forma que possam ser interpretados, processados e intercambiados entre diversas ferramentas de modelagem (GRONBACK, 2009).

Dentre as principais vantagens da abordagem MDD destaca-se o aumento da produtividade no desenvolvimento, já que as tarefas repetitivas podem ser implementadas nas transformações. Desta forma, um único modelo pode gerar uma grande quantidade e diversidade de código. Também pode-se citar vantagens com questões relacionadas à portabilidade, onde um mesmo modelo pode ser transformado em código compatível com diferentes plataformas, e interoperabilidade, onde cada parte do modelo pode ser transformada em código para uma plataforma diferente. Fatores como o aumento na facilidade de manutenção, documentação e comunicação entre as equipes são também apontados como vantagens ao se adotar o MDD, pois a documentação mantém-se atualizada com os modelos mais facilmente que na maneira tradicional. As verificações e otimizações de código também podem ser feitas com menor esforço diretamente nos modelos, podendo utilizar-se de ferramentas de otimizações automáticas, além de ferramentas verificadoras da semântica que auxiliam a garantir a corretude do software. Por fim, uma vantagem inerente à abordagem MDD é a facilidade em reutilizar códigos que podem ser facilmente regerados para outro contexto (KLEPPE; WARMER; BAST, 2003), (DEURSEN; KLINT, 1998), (BHANOT et al., 2005) e (MERNIK; HEERING; SLOANE, 2005).

Apesar de todas as vantagens citadas na utilização da abordagem MDD, existem algumas desvantagens que devem ser levadas em consideração. Dentre elas vale citar o problema de rigidez no software produzido, já que grande parte do código é gerada automaticamente, fi-

cando fora do alcance do desenvolvedor (THOMAS, 2004). Este problema tem sido o maior alvo de crítica para adoção do MDD no desenvolvimento de jogos (DORMANS, 2012) e (FURTADO; SANTOS; RAMALHO, 2011). Além disso, o código gerado pode incluir código desnecessário, prejudicando o desempenho do produto final. Outras desvantagens estão ligadas ao aumento da complexidade no início do desenvolvimento, pois artefatos do MDD, como ferramentas de modelagem, transformações e geradores de código, são mais difíceis de construir. Tal complexidade torna necessário que os profissionais sejam capacitados a criarem estes artefatos do MDD, exigindo um maior investimento inicial. Porém, há relatos de que os ganhos posteriores são bastante relevantes, podendo levar ao retorno deste investimento em poucas iterações (AMBLER, 2003) e (THOMAS, 2004).

Adotar a abordagem MDD ainda é um desafio para os desenvolvedores. France e Rumpe (2007) classificam estes desafios da seguinte forma:

- **Desafios de modelagem de idioma:** refere-se às dificuldades em se criar as abstrações do problema e formalizar a semântica da linguagem. Isto exige que o desenvolvedor compreenda os conceitos, terminologias e as relações que representam o domínio do problema para criar uma notação para a linguagem.
- **Desafios de separação de interesses:** surge especialmente em sistemas complexos onde são envolvidas múltiplas linguagens de forma heterogênea. Algumas funcionalidades podem ter conflitos com outras o que torna o desafio ainda maior. É necessária a separação dos interesses através de pontos de vista bem definidos para evitar falhas de integração.
- **Desafios de manipulação e gerenciamento de modelos:** são problemas relacionados com a definição, análise e transformações dos modelos. Estes desafios incluem a manutenção dos vínculos de rastreabilidade entre os elementos do modelo para apoiar a evolução do modelo e a integração do código gerado com o código não gerado, ou seja, inserido manualmente pelo desenvolvedor.

Esses desafios, em especial os dois últimos, dizem respeito à necessidade de coexistência de múltiplas DSLs com código feito à mão, tópico da próxima seção.

2.3 Integração de DSLs com código manual

Um desafio recorrente ao se aplicar MDD está no fato de que esta abordagem reduz a flexibilidade do projeto. Furtado, Santos e Ramalho (2011) e Dormans (2011) apontam este problema e tentam solucioná-lo através da divisão do projeto em múltiplos subdomínios e também com múltiplas DSLs. Subdomínios são um conjunto de funcionalidade similares, que atendem

a uma determinada área de um projeto e podem ser reutilizados em outros projetos. Contudo combinar vários subdomínios não é algo trivial, sendo necessário um estudo mais aprofundado sobre o tema. A arquitetura deve estar preparada para lidar com os múltiplos subdomínios e possivelmente com as múltiplas DSLs, além de ter que definir como se dá a interação entre eles.

Uma maneira de se integrar múltiplos subdomínios é desenvolver um único metamodelo, onde os subdomínios já estejam integrados. Assim, as interdependências entre os subdomínios já estarão resolvidas de forma explícita. Para que ocorra essa abordagem, cada elemento dos subdomínios deve ser inspecionado e cada elemento que depende de um elemento de outro subdomínio deve ser documentado corretamente. Como algumas dessas dependências tornam-se aparentes apenas após a implementação estar avançada, a cada interação do desenvolvimento os subdomínios devem evoluir procurando por novas relações entre os subdomínios (LUCREDIO, 2009).

Outra maneira é fazer a integração entre diferentes DSLs. Esta integração deve ser gerenciada de forma que o desenvolvedor possa especificar modelos em quaisquer DSLs utilizando conceitos que se relacionam. Esta integração pode ser baseada em nome, onde um atributo texto na DSL que contém a referência aponta para o nome de um elemento na DSL sendo referenciada. Outra forma de se fazer a integração entre diferentes DSLs é através de pontes entre modelos que consiste na criação de um elemento na DSL que contém a referência, sendo uma cópia do elemento da DSL referenciada. Ambas técnicas exigem que seja feita checagem de integridade referencial manualmente e no caso da referência baseada em nome ainda é necessário fazer a checagem de tipos (WARMER; KLEPPE, 2006).

Mesmo após vencer as dificuldades de integração entre vários subdomínios e também entre várias DSLs, Furtado, Santos e Ramalho (2011) apontam que no caso do desenvolvimento de jogos, o uso exclusivo de DSLs não é suficiente. Para aumentar a flexibilidade e tornar o projeto extensível, Furtado, Santos e Ramalho (2011) sugerem o uso de padrões de projeto a fim de permitir adição de código manual no jogo.

Os padrões de projeto normalmente são utilizados para resolver problemas de variabilidade baseada em DSLs. Eles podem oferecer uma boa integração entre código gerado e o código inserido manualmente. De acordo com o tipo de dependência entre o código gerado e o não-gerado pode-se definir o padrão de projeto mais adequado conforme com as seguintes situações (LUCREDIO, 2009):

- Código gerado depende de código não-gerado: consiste em fazer o gerador produzir código que usa código existente não-gerado, como um framework ou biblioteca. O padrão

Facade (GAMMA et al., 1995) pode ser utilizado para simplificar este tipo de interação, onde uma única classe agrupa todas as classes e métodos em um único ponto tornando as dependências mais explícitas e protegendo contra mudanças no código não-gerado. Assim, pequenas adaptações podem ser feitas diretamente nessa classe. O padrão *Adapter* (GAMMA et al., 1995) pode ser utilizado para coletar, filtrar e/ou preparar a informação necessária para o código gerado protegendo o gerador de mudanças maiores neste código.

- Código não-gerado depende de código gerado: acontece quando o código não-gerado espera que um comportamento ou estrutura seja gerado. Padrões como o *Template method* ou *Factory* (GAMMA et al., 1995) podem ser utilizados, assim o código não-gerado não precisa saber detalhes sobre como as classes e métodos que ele utiliza são implementados. Também é possível remover a dependência entre código não-gerado e gerado por meio dos padrões de Injeção de dependência ou Localizador de serviço (FOWLER, 2004), onde as dependências são colocadas em agentes externos, responsáveis pela injeção das mesmas através de configuração programática ou textual (normalmente XML). Estes padrões assumem que há um elemento fixo entre os dois lados da dependência: uma interface, uma classe abstrata ou uma assinatura de método. Porém, através da reflexão é possível fazer com que o método chamado fique desconhecido pelo compilador fazendo chamadas reflexivas.

- Código gerado depende de código gerado: quando um subdomínio depende de outro deve-se garantir a integridade deste relacionamento. Uma forma de garantir essa integridade é utilizar os nomes dos elementos como referências, o que simplifica o processo de implementar referências entre modelos (HESSELLUND; CZARNECKI; WASOWSKI, 2007). Um verificador separado pode ajudar a garantir a integridade referencial entre os modelos (WARMER; KLEPPE, 2006).

- Código gerado precisa ser estendido: nesses casos o ideal é incluir a geração de classes abstratas ou interfaces, e assim utilizar subclasses para implementar as partes faltantes. Técnicas como receitas que exibem avisos sobre os passos a serem seguidos para completar o código podem ajudar a garantir a implementação manual correta. Técnicas de mesclagem entre código manual e código gerado não são uma boa prática, pois o código manual pode ser perdido após uma regeneração (WARMER; KLEPPE, 2006). Um padrão útil nesses casos é o *"merging"* de geradores, que consiste em criar um modelo separado para especificar as partes faltantes e então combinar estes modelos utilizando um gerador específico (*"merger"*).

2.4 Considerações Finais

Conforme apresentado por Petrillo et al. (2009), os jogos eletrônicos compõem um importante setor tanto na área artística como na área computacional, tendo grande destaque quanto a sua abrangência e lucratividade em ambos setores.

Apesar de ser uma área promissora, os jogos eletrônicos têm sofrido alguns problemas, conforme descrito por Petrillo et al. (2009) e Blow (2004). Dentre esses problemas destacam-se aqueles relacionado à definição de escopo, planejamento de tempo, tecnologia e na dificuldade em encontrar profissionais capacitados.

Paralelamente aos problemas enfrentados pela indústria dos jogos eletrônicos, Kleppe, Warmer e Bast (2003), Deursen e Klint (1998), Bhanot et al. (2005) e Mernik, Heering e Sloane (2005) destacaram que no desenvolvimento de software há grandes ganhos em termos de produtividade e facilidade no desenvolvimento com a adoção da abordagem MDD. Isso deve-se ao fato de que no MDD há uma maior reutilização de artefatos, além do desenvolvimento se dar em uma linguagem com maior nível de abstração. Tais vantagens podem ser adequadas para solução dos problemas enfrentados pela industria dos jogos eletrônicos, especialmente quanto aos problemas de tempo de desenvolvimento, já que no MDD há uma maior produtividade, e aos problemas tecnológicos, pois o MDD propõe o uso de DSLs que normalmente têm uma curva de aprendizado menos íngreme do que as linguagens de uso geral.

Kosar et al. (2010) demonstraram que as DSLs são mais eficientes no que se diz respeito ao aprendizado, sendo mais facilmente aprendidas em todas as dimensões cognitivas do que as linguagens de propósito geral (GPLs). As DSLs também são menos propensas a erros, especialmente para desenvolvedores iniciantes. No estudo feito por Kosar et al. (2010) foi notada uma taxa de sucesso no aprendizado com DSLs em torno de 15% melhor do que com GPLs.

Porém, jogos eletrônicos são projetos flexíveis, que estão sujeitos a sofrerem alterações durante seu desenvolvimento. Uma maneira de se conseguir flexibilidade nos projetos com MDD é utilizar múltiplas DSLs e também permitir que se inclua códigos manuais. Contudo é necessário fazer um levantamento dos projetos de jogos que utilizaram a abordagem MDD a fim de averiguar os reais benefícios obtidos. Este levantamento é apresentado no próximo capítulo.

Capítulo 3

TRABALHOS CORRELATOS

3.1 Jogos Eletrônicos e MDD

Desenvolvimento de jogos é uma tarefa que exige grande conhecimento técnico e artístico. Os desenvolvedores de jogos normalmente contam com algumas ferramentas, como motores de jogos para auxiliar o desenvolvimento, porém a maioria dessas ferramentas tem uma grande curva de aprendizado e exige alto conhecimento técnico sobre desenvolvimento de jogos. O uso de motores de jogos é prática comum entre os desenvolvedores de jogos comerciais, pois permitem maior controle e flexibilidade para criar artisticamente o software do jogo, mas fazem com que a linha de produção seja muito dependente de artistas especializados e programadores experientes, necessitando de um alto investimento financeiro (TANG; HANNEGHAN, 2011b). Unido a este fato, existe uma tendência no aumento da complexidade no desenvolvimento de jogos, o que torna necessário a criação de ferramentas para melhorar a produtividade em termos de tempo, custo e qualidade (REYNO; CUBEL, 2009b). Reyno e Cubel (2008) relatam que pode-se perceber isto ao observar que nos anos 80 um jogo poderia ser desenvolvido em 3 meses por um programador que fazia o design e também a arte, já em 2005, um jogo podia ser desenvolvido por uma equipe de 20 a 100 especialistas multidisciplinares, incluindo programadores, designers de jogos, artistas, escritores, atores, músicos, etc, com um orçamento de mais de 10 milhões de dólares para 4 anos de desenvolvimento.

Blow (2004) também faz uma comparação entre como era o desenvolvimento de jogos em 1994 e como ficou em 2004. Nestes 10 anos notou-se um crescimento na dificuldade ao se desenvolver jogos, especialmente devido a problemas ligados ao aumento do tamanho e da complexidade dos projetos e dos problemas ligados ao crescimento dos requisitos específicos. Outro fator relevante, levantado por Karamanos e Sgouros (2012) é que nos últimos anos tem surgido múltiplas plataformas emergentes para jogos, como por exemplo os dispositivos móveis,

consoles e PCs, fazendo com que seja necessária a criação de ferramentas que automatizem a implementação multiplataforma dos jogos.

Percebe-se portanto uma disputa de forças opostas. De um lado, a criação de jogos exige flexibilidade no desenvolvimento, para que os programadores consigam implementar as diferentes nuances do desejo artístico por trás do projeto do jogo. Do outro lado, as pressões tecnológicas e de mercado sugerem o uso de automação e ferramentas, que melhoram a produtividade, mas acabam engessando a liberdade e flexibilidade do desenvolvimento.

Neste cenário, surgiu o MDGD (*Model-Driven Game Development* ou Desenvolvimento de Jogos Orientado a Modelos) como alternativa para solucionar este problema. Ao concentrar-se na especificação dos modelos ao invés do código que implementa os jogos, pode ser alcançado um nível maior de abstração e automatização do processo de desenvolvimento. Ao mesmo tempo, modelos podem oferecer um grau de liberdade maior do que é possível com uma ferramenta pré-projetada, flexibilizando um pouco o desenvolvimento (REYNO; CUBEL, 2009b).

Em uma abordagem MDGD, pode-se fazer uma especificação formal de diferentes partes do projeto representado-as em modelos. Assim, o jogo é especificado através de sua modelagem, sem ter necessidade prévia de profundo conhecimento técnico no desenvolvimento de jogos, pois esses modelos podem ser transformados em código automaticamente (TANG; HANNEGHAN, 2011b).

3.2 Exemplos de aplicação do MDGD

Uma tarefa adequada para o uso do MDGD é a prototipação de jogos. Reyno e Cubel (2009b) implementam uma ferramenta para prototipagem de jogos em plataforma 2D. Como exemplo de protótipo é utilizado um jogo clássico de plataforma 2D, o Bubble Bobble, como pode ser visto na Figura 3.1.

Para representar a estrutura do jogo de plataforma 2D, foi utilizado um diagrama de classes com estereótipos, para descrever as entidades do jogo. O diagrama de estado foi usado para representar o comportamento básico do jogo, considerando o comportamento do jogador e dos inimigos, bem como suas colisões. Também foi definido um diagrama de mapa de controles, que definem as entradas que o jogador pode enviar para interagir com o jogo. Com a aplicação do MDGD, é feita a geração do protótipo do jogo de plataforma 2D em C++. Neste protótipo está implementado o Game Loop, caixas de gravidade e detecção de colisão, além dos controles descritos no diagrama de mapa de controles para detectar a ação do jogador. A implementação manual deste protótipo demorou uma semana, enquanto com implementação usando a ferra-

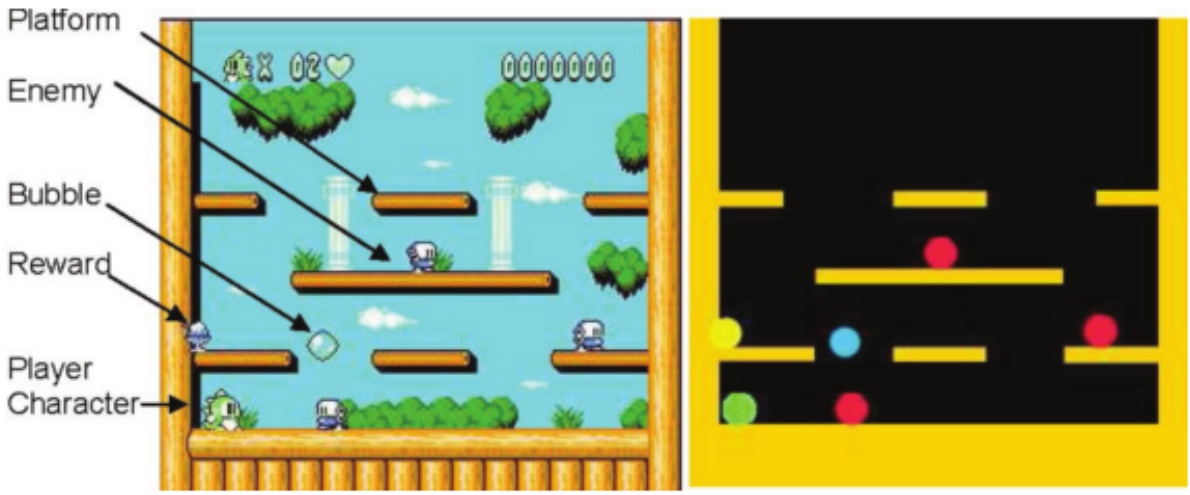

Figura 3.1: Protótipo do jogo Bubble Bobble. Reyno e Cubel (2009b)

menta MDGD demorou apenas algumas horas, gerando automaticamente 93% do código. Um segundo experimento foi realizado, desenvolvendo um protótipo do jogo Super Mario Bros. Neste protótipo, o código gerado automaticamente representa 94% do total de linhas de código, mostrando resultados muito satisfatórios no aumento de produtividade e redução no tempo de desenvolvimento (REYNO; CUBEL, 2009b).

Um dos primeiros passos no desenvolvimento de um jogo baseado em modelos está no trabalho de engenharia de requisitos, onde pode ser desenvolvido o documento chamado de Especificação de Requisitos do Software do Jogo. Tal documento representa a visão dos requisitos por parte dos desenvolvedores. Para tanto, pode-se utilizar uma versão adaptada do diagrama de casos de uso, utilizando estereótipos para representar a estrutura do jogo, como cenas, atos, telas e desafios (COOPER; LONGSTREET, 2012).

Em se tratando de modelagem de jogos, Tang e Hanneghan (2011b) sugere utilizar uma combinação das várias técnicas de modelagem de software para os diferentes aspectos do jogo, uma vez que nem sempre é possível representar todos os aspectos dos jogos com uma única linguagem. Dentre essas diversas técnicas, vale destacar a modelagem de jogos usando UML. Por exemplo, o diagrama de casos de uso pode representar a interação do jogador com o jogo, identificando todas as possíveis interações que o jogo pode sofrer. O diagrama de classes pode representar os objetos do jogo e sua estrutura estática, além de poder representar a associação entre os objetos do jogo. Uma coleção de classes pode ser visualizada usando um diagrama de componentes, enquanto um diagrama de implantação pode ser usado para representar a disposição física do jogo. O diagrama de atividades pode ser utilizado para modelar o comportamento dos objetos do jogo, podendo-se estender essa modelagem com o diagrama de sequência. Dessa forma, a UML oferece uma gama de abordagens úteis para modelagem e documentação de

jogos e sua escolha pode facilitar a tradução da especificação em código fonte (TANG; HANNEGHAN, 2011b).

Uma alternativa à UML é uso de DSLs, porém Reyno e Cubel (2009a) afirmam que os jogos carecem de uma linguagem de especificação precisa para definir sua jogabilidade[1], fazendo com que vários *game designers* definam a jogabilidade em linguagem natural. Isso é um problema grave para o MDGD, pois a linguagem natural é ambígua e imprecisa. Procurando solucionar este problema, Reyno e Cubel (2009a) propõem a definição de uma linguagem de especificação da jogabilidade mais precisa através de modelos, onde o jogo é representado em um nível alto de abstração técnica. Esta metodologia de desenvolvimento faz com que o modelo seja tratado como o principal artefato do software. Assim é permitido maior reutilização e otimização de tarefas repetitivas e demoradas, reduzindo o tempo de implementação e também a quantidade de erros, construindo jogos de maior qualidade técnica (REYNO; CUBEL, 2009a).

Um metamodelo é um modelo criado para o desenvolvimento das DSLs, elaborando regras, métodos e conceitos que serão utilizados, com nível de detalhamento que englobe todas as características genéricas de um jogo. Calixto e Bittar (2010) apresentam na Figura 3.2 a representação de um metamodelo a fim de demonstrar essas características genéricas de um jogo. Para elaboração deste metamodelo foi utilizada a ferramenta *Eclipse Modeling Framework*, na qual é possível gerar um arquivo .xmi que tem como função fornecer fácil acesso às informações entre modelos. Este metamodelo apresentado tem a seguinte estrutura de classes:

- Game: agrega as principais características do jogo, como o nome e seus objetivos. Contém também valores para os enumeradores que definem a quantidade de jogadores, o tipo de licença do jogo e o seu estilo de câmera.
- Platform: define detalhes técnicos e limitações das plataformas em que o jogo será compatível.
- Ranking: classe opcional para armazenar os dados das maiores pontuações dos jogadores.
- PlayerAccount: classe opcional para definir se o jogo necessita de cadastro por parte do jogador.
- GameCharacter: agrupa todos os personagens do jogo, definindo seus papéis como personagem do jogador ou inimigo.

[1] Ações que o jogador pode exercer nos jogos, descrevendo sua experiência em relação aos controles e desafios de um jogo.

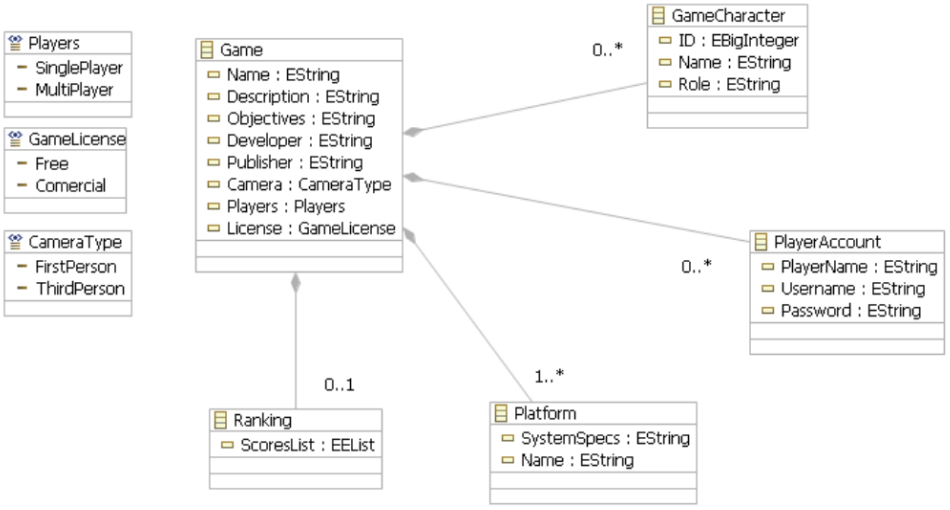

Figura 3.2: Metamodelo para o desenvolvimento de jogos. (CALIXTO; BITTAR, 2010)

DSLs têm um grande potencial de uso na área de desenvolvimento de jogos, mas é importante que elas sejam adaptadas às exigências do seu público alvo específico. Por isso, deve-se utilizar uma abordagem que permita a criação de DSLs como parte do processo de design do jogo. Uma DSL pode permitir que desenvolvedores escrevam programas que são fáceis de entender para designers com nenhum conhecimento de programação. Tal abordagem foi demonstrada por Walter e Masuch (2011) com o desenvolvimento de uma DSL para geração de jogos de Aventura 2D Point&Click. Este tipo de jogo se caracteriza por mostrar histórias de aventura e permitir a interação do jogador através de cliques de mouse em determinadas áreas do jogo, fazendo com que o personagem vá caminhando por todo o cenário. A criação de linguagens específicas de domínio pode reduzir a distância entre o projeto do jogo e sua implementação. Além disso, os geradores de código garantem um nível de qualidade de código consistente, já que as operações repetitivas são realizadas sempre da mesma forma (WALTER; MASUCH, 2011).

As técnicas de MDGD também podem ser aplicadas em fábricas de software. Neste contexto o desenvolvedor precisa definir produtos com características semelhantes para especificar uma infraestrutura MDD de apoio ao desenvolvimento. Tal infraestrutura depende primeiramente da definição da DSL. Em seguida deve ser disponibilizado um editor capaz de utilizar a linguagem visual. Como estudo de caso do MDGD aplicado em uma fábrica de software, Maier e Volk (2008) propõem o desenvolvimento de uma série de jogos clássicos, iniciando pelo Pacman. Assim foi utilizada uma DSL como seu editor de fases e uma DSL adicional textual para descrever a lógica do jogo. Neste estudo de caso foi notado um grande ganho de produtividade, onde grupos de estudantes conseguiram implementar protótipos de jogos executáveis dentro de poucas horas (MAIER; VOLK, 2008).

Outro exemplo de aplicação de MDGD está na fábrica de software SharpLudus apresentada

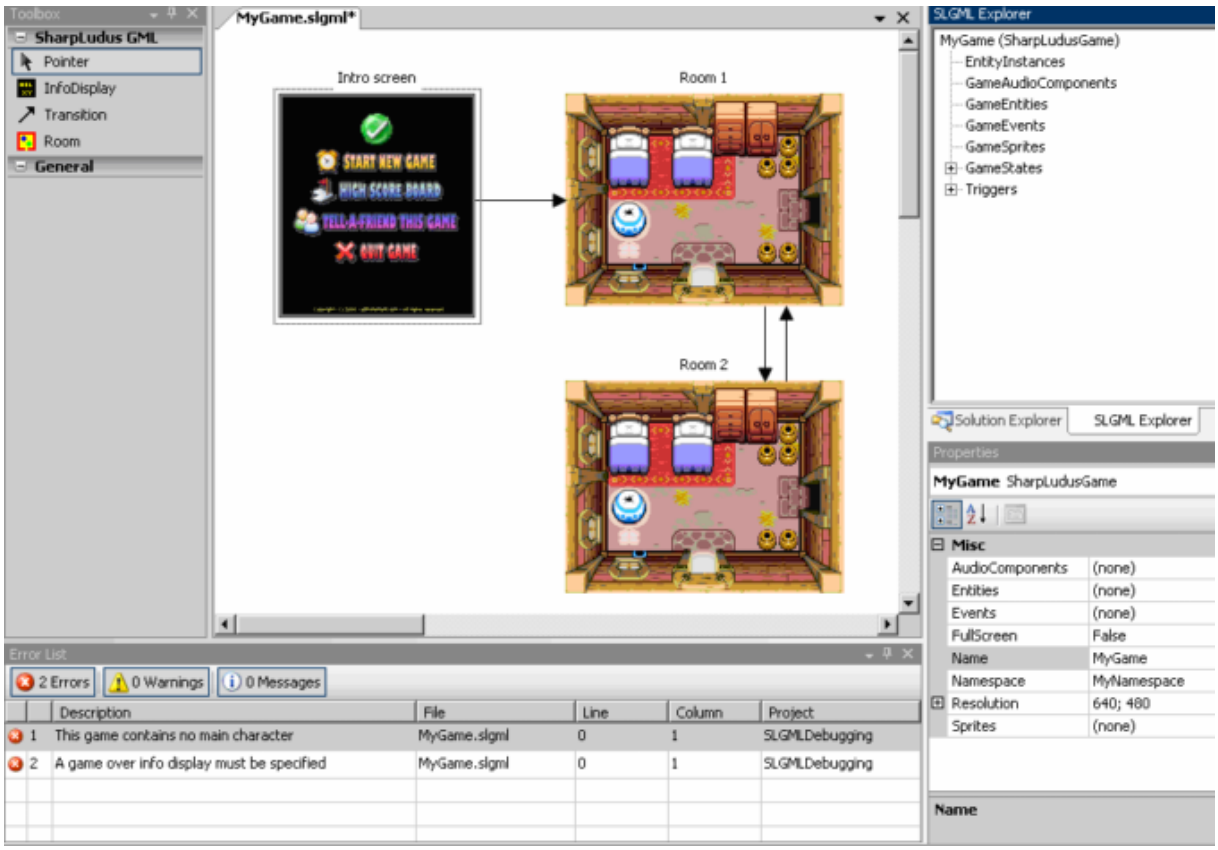

Figura 3.3: Modelagem utilizando SLGML. (FURTADO; SANTOS, 2006)

por Furtado e Santos (2006), a qual se concentra em produtos no gênero de jogos de aventura em duas dimensões. Como solução para produtividade no desenvolvimento de seus jogos, são fornecidas aos desenvolvedores duas DSLs. A primeira refere-se à modelagem do jogo, permitindo a especificação dos fluxos de estados do jogo, como telas de informações, cenários e condições de saída. A segunda permite aos desenvolvedores especificarem as informações úteis do jogo, como pontuação, vidas restantes, etc. Um gerador de código foi associado a essas DSLs, fazendo com que ao receber um diagrama nomeado como SLGML (*SharpLudus Game Modeling Language*), o gerador crie os códigos de programação automaticamente na linguagem C#. A Figura 3.3 mostra um exemplo de modelagem utilizando o diagrama SLGML.

A produtividade adquirida com uso do SLGML nesse estudo relatado é inegável, pois através de seu gerador de código, em menos de uma hora de desenvolvimento conseguiu-se gerar 16 classes com 3900 linhas de código. Outro item importante é que as linhas de código geradas automaticamente representam principalmente as rotinas mais entediantes e propensas a erros no processo de desenvolvimento do jogo (FURTADO; SANTOS, 2006).

De acordo com Smith, Cooper e Longstreet (2011), uma metodologia de desenvolvimento orientado por modelos baseia-se num bom planejamento, com informações detalhadas sobre

especificações de requisitos, documentos de projeto e modelagem. Tal metodologia pode não ser a ideal quando o objetivo é ensinar a alunos os conhecimentos ligados a gráficos, inteligência artificial, redes de computadores, sons, etc. Porém o uso do MDGD traz a vantagem de tornar o desenvolvimento muito simples e gerar software consistente (SMITH; COOPER; LONGSTREET, 2011).

Em resumo, conforme discutido nos trabalhos que aplicaram o MDGD, pode-se perceber que há potencial para ganhos consideráveis na produtividade, reduzindo o tempo de desenvolvimento e exigindo menor conhecimento técnico das ferramentas, como APIs ou motores de jogos. Apesar do MDGD resolver alguns dos problemas enfrentados pela indústria dos jogos eletrônicos, nesta pesquisa não foi encontrado nenhum relato de sua adoção por parte das indústrias. Porém existem vários estudos que descrevem bons resultados na aplicação do MDGD no desenvolvimento de jogos educacionais (TANG; HANNEGHAN, 2010), como apresentado na próxima seção.

3.3 MDGD em jogos educacionais

Um domínio bastante recorrente na literatura acadêmica na área de jogos é o de educação. Segundo Tang e Hanneghan (2010), os jogos educacionais têm se mostrado como ótima ferramenta para o auxílio do aprendizado, especialmente para o público jovem que é aficionado ao mundo dos jogos. O ensino baseado em jogos tem sido defendido por muitos especialistas que afirmam que os jogos podem proporcionar uma experiência de aprendizado melhor em comparação aos métodos didáticos tradicionais. Tal fato tem feito com que professores especialistas em algum domínio específico do ensino, porém com pouco conhecimento computacional na área de desenvolvimento, queiram planejar, desenvolver e atualizar seu material didático em formato de jogos eletrônicos. Neste âmbito, o MDGD pode oferecer uma alternativa para estes profissionais, para que eles não tenham que passar por processos complexos de desenvolvimento de software.

Embora as ferramentas atuais para o desenvolvimento de jogos, como os motores de jogos, proporcionem ótima flexibilidade e controle para a criação de jogos, elas dependem de artistas e programadores altamente especializados. Dessa forma ao se tratar de profissionais especialistas no ensino com pouco domínio das técnicas de desenvolvimento de jogos, a tarefa de desenvolver um jogo torna-se muito custosa. O MDGD se apresenta como solução para estes profissionais, dando-os capacidades para o desenvolvimento de jogos educacionais, sem a exigência de grande domínio no uso das ferramentas de desenvolvimento de software do jogo (TANG; HANNEGHAN,

2010).

De acordo com Tang e Hanneghan (2011b) os esforços atuais em MDGD tem focado principalmente no público amador, porém há uma crescente adoção do MDGD para a produção de jogos educacionais.

Os benefícios do MDGD para o desenvolvimento de jogos educacionais são certamente atraentes, mas sua realização ainda é um desafio para os desenvolvedores. Procurando solucionar este problema Tang e Hanneghan (2011a) propõem um Modelo de Conteúdo de Jogo (*Game Content Model* ou GCM) baseado em estudos do *game design* e desenvolvimento. Este GCM é usado para documentar a especificação do projeto de um jogo de computador e será o modelo utilizado para a construção de outros modelos para o framework MDGD voltado para jogos educacionais. O GCM proposto é dividido nas seguintes partes:

- Game Structure: descreve a estrutura, arquitetura e o fluxo do jogo;
- Game Presentation: contém componentes de mídia como textos, gráficos, sons e vídeos;
- Game Simulation: define a dimensão do jogo (2d ou 3d), além da física e tempo do jogo;
- Game Rules: são as regras de relação entre os objetos e o mundo do jogo;
- Game Event: é um disparador de eventos que está associado ao cenário do jogo;
- Game Objective: descreve os objetivos do jogo;
- Game Objects: são os objetos que populam o mundo do jogo;
- Game Scenario: representa o ambiente do jogo;
- Game Player: definições do usuário da aplicação do jogo; e
- Game Theme: descreve de forma textual os requisitos de arte relacionados ao jogo.

O objetivo do GCM é encapsular a complexidade do desenvolvimento de jogos educativos, proporcionando uma plataforma para o desenvolvimento através da transformação de modelos usando ferramentas de MDGD (TANG; HANNEGHAN, 2011a).

Jovanovic et al. (2011) concordam que os jogos educacionais têm sido um tema relevante nos últimos anos, e Bransford, Brown e Cocking (apud JOVANOVIC et al., 2011, p. 1) relatam que jogar videogames estimula a liberação de dopamina no cérebro, que é um precursor químico para a memória, promovendo assim um melhor aprendizado. Jovanovic et al. (2011) propõem

o uso do MDGD para o desenvolvimento deste tipo de jogo. Com o MDGD é oferecido um conjunto de transformações de modelo que facilitam a atividade de design, mas que continua necessitando de algumas definições dos designers. Deste modo, é possível focar nos conceitos que não estão tão ligados à tecnologia e sim ao domínio do problema que são os jogos educacionais (JOVANOVIC et al., 2011) .

Segundo Minović et al. (2009), quando se trata de jogos educacionais, deve-se tomar cuidado com o design e a jogabilidade, para que o jogo seja atraente e consiga prender a atenção do aluno. Desta forma, o aluno se sente motivado em aprender, pois está também se divertindo. No entanto, desenvolver um jogo com boa interface de usuário, unindo qualidade no design e na jogabilidade, não é algo trivial.

Os trabalhos apresentados nesta seção que abordaram o uso do MDGD em jogos educacionais obtiveram bons resultados, já que elevar o nível de abstração no desenvolvimento permitiu que pessoas com pouco conhecimento das tecnologias e ferramentas de desenvolvimento de jogos conseguissem criar seus próprios jogos educacionais. Tais benefícios podem ser aproveitados também para outros tipos de jogos, inclusive jogos comerciais.

3.4 Múltiplos Modelos com MDGD

Nas seções anteriores, onde foram apresentados trabalhos sobre MDGD, é possível notar que a maior parte das abordagens utiliza múltiplos modelos de entrada. Por exemplo, Tang e Hanneghan (2011b) consideram dez aspectos de um jogo em sua linguagem de modelagem chamada de SLGML (*SharpLudus Game Modelling Language*). A SLGML inclui duas linguagens, cada uma concentrando-se em um aspecto diferente do jogo. O objetivo é prover maior eficiência. Através da utilização de múltiplos modelos no MDGD consegue-se uma flexibilidade suficiente para lidar com a agilidade no desenvolvimento de jogos, ao mesmo tempo em que os processos são automatizados. Dessa forma cria-se um ambiente de trabalho mais flexível, permitindo que os designers possam adotar técnicas de MDD um passo de cada vez.

Dentre esses passos tomados para cada ciclo do desenvolvimento, deve-se começar pelos passos mais genéricos e que podem ser facilmente reutilizados em projetos posteriores. Assim, o MDGD fará com que o designer não tenha que fazer tantas tarefas triviais ou repetitivas no desenvolvimento do jogo, permitindo que o mesmo se concentre nas tarefas criativas (DORMANS, 2012).

Dormans (2011) propôs a utilização de vários modelos para a geração de estágios, ou fases, de um jogo através do MDGD. Tal metodologia permite que um designer de estágios de jogo

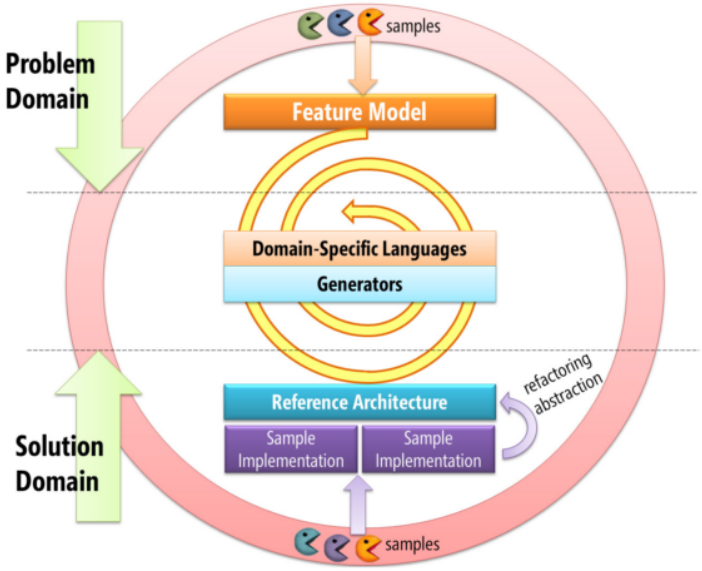

Figura 3.4: Processo iterativo de desenvolvimento dirigido por modelos. Furtado, Santos e Ramalho (2011)

gere uma série de modelos diferentes, trabalhando lentamente para completar o estágio total. O MDGD pode trabalhar com muitos modelos diferentes, alguns dos quais são específicos para um determinado domínio, enquanto outros são mais genéricos. Através da engenharia dirigida por modelo, a qualidade e a eficiência da produção do software são melhorados, melhorando assim a qualidade dos estágios criados por esta metodologia. (DORMANS, 2011).

Furtado, Santos e Ramalho (2011) discutem a utilização de vários modelos em um processo iterativo de desenvolvimento. Neste processo de desenvolvimento, a cada ciclo da espiral devem ser criados ou melhorados os artefatos de domínio, como DSLs e geradores para automatizar sub-domínios dos jogos. Em uma iteração subsequente, outro sub-domínio do domínio de destino do jogo é priorizado e elaborada através da mesma forma buscando a solução do domínio, como está ilustrado na Figura 3.4.

A fim de avaliar sua proposta, Furtado, Santos e Ramalho (2011) fizeram um estudo de caso com o desenvolvimento do domínio de jogos de arcade 2d, nomeado de ArcadEx. Nesses jogos, os jogadores controlam personagens principais e seus projéteis que colidem com outras entidades, como os personagens não jogáveis (NPCs, *non-player characters*) ou outros itens. A condição de vitória é especificada pelo designer de jogos através de eventos do jogo como, por exemplo, quando os inimigos são derrotados ou um objeto é coletado. Na primeira iteração, foi criada a GameDefinitionDSL capaz de controlar a transição entre telas, tal como por uma ação de entrada ou de um temporizador. Na segunda iteração, foram adicionadas à DSL definições pelas quais os desenvolvedores podem especificar as propriedades de alto nível de um jogo (como o seu modo de janela, resolução, etc.). Na terceira iteração, foi refinada a Ga-

meDefinitionDSL, permitindo que os desenvolvedores de jogos e designers pudessem atribuir imagens de fundo para as telas em vez de fazer isso manualmente. Na quarta iteração, foram exploradas as propriedades de música de fundo em cada tela, bem como o comportamento da música. As iterações subsequentes definiram a InputMappingDSL capaz de tratar os eventos de entrada e a EntityDSL definida para modelar os estados de cada entidade e animações. Após várias iterações para refinar as DSLs já criadas, foi definida a ScreenDSL responsável por mostrar informações de tela como ícones, textos ou barras de progresso. A cada iteração além da criação e do refinamento das DSLs eram desenvolvidos os geradores de código para essas DSLs.

Os jogos ArcadEx conseguiram ser desenvolvidos pelo usuário final com tempo entre um quinto a um quarto do tempo necessário para desenvolvê-los usando apenas *game engines*, comprovando sua eficiência. Como efeito colateral, existe uma redução nos níveis de flexibilidade do desenvolvimento. Outro desafio a esse modelo incremental está na incompatibilidade das DSLs com versões anteriores (FURTADO; SANTOS; RAMALHO, 2011).

3.5 Considerações Finais

O desenvolvimento de jogos eletrônicos pode beneficiar-se das vantagens já conhecidas pela abordagem MDD em projetos de software. Com essa abordagem pode-se conseguir uma melhor integração dos especialistas do domínio (por exemplo, *game designers*, escritores, artistas de conceito de jogo) no processo de desenvolvimento (WALTER; MASUCH; FUNK, 2012).

Nos estudos apresentados nesse capítulo, percebe-se que há ganhos consideráveis em termos de agilidade no desenvolvimento de jogos eletrônicos com o uso da abordagem MDGD. Também é notado que o MDGD propicia que os jogos possam ser desenvolvidos por profissionais que não possuem tanto conhecimento técnico. Isso se deve ao fato de que o MDGD eleva o nível de abstração no desenvolvimento ocultando detalhes técnicos e permitindo que o desenvolvedor interaja com linguagens mais compreensíveis para amadores e iniciantes. Normalmente as linguagens utilizadas no MDGD possuem uma curva de aprendizado menos íngreme, já que utilizam de uma sintaxe mais direta para resolver o domínio dos problemas a qual elas foram especificadas. Além de facilitar o desenvolvimento e o aprendizado, o MDGD reduz a quantidade de tarefas repetitivas e aumenta o reuso de artefatos de software.

Contudo, o MDGD ainda não teve uma adoção satisfatória na indústria dos jogos eletrônicos. Recentes pesquisas estão tentando viabilizar esta adoção através da definição de múltiplas DSLs, o que aumenta a flexibilidade no desenvolvimento, mas ainda não chega ao ideal desejado pela indústria pois limita a criatividade dos desenvolvedores. Furtado, Santos e Ramalho

(2011) e Jovanovic et al. (2011) comentam que a inserção de códigos manuais pode aumentar a flexibilidade dos jogos com MDGD, porém não foram encontrados estudos nessa pesquisa que demonstram esta abordagem no desenvolvimento de jogos.

Portanto, no próximo capítulo é apresentado o desenvolvimento de uma ferramenta que permite a criação de jogos com MDGD utilizando múltiplas DSLs. Em seguida são feitos estudos experimentais para avaliar o comportamento dessa abordagem ao se inserir códigos manuais para aumentar a flexibilidade no projeto.

Capítulo 4

JMEGENERATOR

4.1 Introdução ao Desenvolvimento

Conforme apresentado por Furtado e Santos (2006) e Petrillo et al. (2009), a indústria dos jogos eletrônicos necessita que seus processos de desenvolvimento sejam revistos e organizados para atender a demanda de produtos cada vez mais complexos. O MDGD pode tornar-se uma solução para esse problema, mas há uma barreira que impede sua adoção pela indústria dos games, já que esta abordagem é tida como prejudicial ao que se refere à criatividade dos desenvolvedores, conforme discutido por Furtado, Santos e Ramalho (2011).

De fato é sabido que o MDD tem como característica maior rigidez no desenvolvimento do software, conforme citado por Thomas (2004), fazendo com que sua aplicação no âmbito do desenvolvimento de jogos se restrinja a jogos menos flexíveis, como por exemplo os jogos educacionais. Tal abordagem é demostrada nos trabalhos de Tang e Hanneghan (2011b), Tang e Hanneghan (2010), Jovanovic et al. (2011) e Minović et al. (2009).

Uma forma de se aumentar a flexibilidade nos projetos de jogos eletrônicos com MDGD está em utilizar vários modelos em um processo iterativo de desenvolvimento, fazendo com que as DSLs sejam específicas para partes pontuais dos jogos. Estudos neste sentido foram discutidos por Furtado, Santos e Ramalho (2011) e Jovanovic et al. (2011), porém eles constataram que ainda não foi alcançado um nível de flexibilidade desejado, sugerindo a necessidade de se integrar códigos manuais aos códigos gerados pelos transformadores.

Porém, integrar várias DSLs e mesclar códigos gerados com códigos não gerados não é uma tarefa trivial. Estudos apontam que padrões de projeto solucionam este tipo de situação, conforme apresentado por Lucredio (2009), Gamma et al. (1995), Fowler (2004), Hessellund, Czarnecki e Wasowski (2007) e Warmer e Kleppe (2006). Porém não há estudos que compro-

vam a eficiência destas abordagens no processo de desenvolvimento de jogos eletrônicos.

Portanto, o objetivo deste trabalho foi propor uma abordagem MDGD que permita unir os benefícios já conhecidos do MDD, porém sem perder a flexibilidade desejada em projetos de jogos eletrônicos. Para isso, a abordagem proposta deve permitir a integração de múltiplas DSLs e também deve possibilitar a inserção de códigos manualmente.

A metodologia adotada foi inspirada no trabalho de Furtado, Santos e Ramalho (2011), no qual o desenvolvimento do jogo, ferramentas de modelagem e geradores são feitas em um processo iterativo, e logo depois é feita uma experimentação para avaliar o protótipo. Dessa forma, a metodologia foi divida nos seguintes tópicos:

1. Desenvolvimento de protótipos de jogos.
2. Estudo dos subdomínios.
3. Desenvolvimento do protótipo MDGD.
    3.1. Refatoração do código fonte.
    3.2. Desenvolvimento das ferramentas de modelagem.
    3.3. Desenvolvimento dos geradores de código.
4. Experimentação do MDGD.

4.2 Desenvolvimento de protótipos de jogos

Para explorar de forma prática os recursos normalmente utilizados no desenvolvimento de jogos eletrônicos, foram desenvolvidos alguns protótipos de jogos que permitiram conhecer as diversas funcionalidades de um motor de jogo. Os protótipos em questão deveriam utilizar tecnologias e conceitos que explorassem ferramentas que vêm sendo utilizadas pelo mercado atualmente, a fim de viabilizar este estudo não somente no meio acadêmico como também no meio comercial. Dessa forma foi definido como requisito mínimo para o escopo dos protótipos que utilizassem algum motor de jogo que seja atual, gratuito, que tenha recursos para exibição de gráficos em terceira dimensão e física que simule o mundo real. Tais protótipos foram de fundamental importância para adquirir conhecimento sobre o motor de jogo escolhido.

Um motor de jogo que mostrou-se capaz de atender os requisitos mínimos foi o jMonkeyEngine, que é um motor de jogo 3D para desenvolvedores Java. Como mostrado na Figura 2.1

da seção 2.1.2, o jMonkeyEngine apresenta uma das melhores relações entre popularidade e sofisticação dos motores de jogos gratuitos.

jMonkeyEngine é um projeto de código aberto e está sob a licença BSD(*Berkeley Software Distribution*), garantindo sua utilização de forma gratuita tanto para o desenvolvimento de jogos não comerciais, como também para jogos comerciais. Com o jMonkeyEngine o jogo é programado inteiramente na linguagem de programação Java, oferecendo assim uma ampla acessibilidade por desenvolvedores que já conhecem essa linguagem. Por utilizar a linguagem de programação Java para codificar seus jogos, permite que seja utilizado o paradigma de programação orientado a objetos a fim de organizar melhor o código. A linguagem Java traz também a característica de ser multiplataforma, fazendo com que seus jogos possam ser desenvolvidos e executados em diversos sistemas operacionais.

A versão 3.0 estável do jMonkeyEngine, também conhecida como jME3, foi lançada em 15/02/2014 e vem apresentando um forte crescimento em sua comunidade de desenvolvedores (JMONKEYENGINE, 2014).

Kusterer (2013) afirma que o jMonkeyEngine possui numerosas funcionalidades que permitem o desenvolvimento de jogos profissionais, sendo que muitos desses recursos somente são encontrados em motores de jogos comerciais. Dentre os diversos recursos oferecidos pelo jMonkeyEngine destacam-se:

- Transformação, projeção, e renderização de objetos 3d.
- Efeitos de iluminação e sombra.
- Estrutura de dados otimizada para cenários tridimensionais.
- Componente modular que controla a mecânica do jogo, suas interações e eventos.
- Suporte para o carregamento de recursos multimídia.
- Suporte para entradas do usuário como teclado, *joystick*, mouse, telas de toque e outros.
- Recursos gráficos para interface com usuário.
- Objeto de câmera intuitiva .
- Simulação de física.
- Recursos de efeitos especiais.
- Recursos de rede para jogos *multiplayer*.

Figura 4.1: Jogo Hostile Sector desenvolvido com jME3 (KUSTERER, 2013)

jMonkeyEngine não é uma ferramenta do tipo de arrastar e soltar que produz jogos com poucos cliques. Para criar um jogo verdadeiramente original com esse motor gráfico é necessário escrever códigos Java. Um exemplo da capacidade do jMonkeyEngine pode ser visto no jogo *Hostile Sector* [1] mostrado na Figura 4.1, que é um jogo de estratégia *multiplayer* baseado na conquista de territórios (KUSTERER, 2013).

Após definir os requisitos mínimos do projeto, iniciou-se o desenvolvimento dos protótipos. No primeiro protótipo foi desenvolvido um jogo com um personagem que podia caminhar por um cenário tridimensional parecido com uma cidade. Foi configurada uma câmera em terceira pessoa para o jogador, onde conforme o personagem vai se movendo a câmera o segue. Para codificar essa câmera foi utilizado o modo de perseguição (*chase*) disponibilizado pelas bibliotecas do jMonkeyEngine. O personagem jogável possui as ações de andar para frente e para trás, além de poder girar para a direita e esquerda e também saltar. Como o protótipo tem o objetivo de explorar os recursos do jMonkeyEngine, não foi estipulado um objetivo específico para o jogador, como encontrar algum item especial ou atirar em algum inimigo. A princípio o jogador deveria interagir com o cenário testando os recursos disponíveis, especialmente a simulação da física e a renderização de modelos 3d.

O ponto central do estudo proposto foi na codificação do jogo com o motor jMonkeyEngine. Portanto a modelagem de objetos tridimensionais não foi o foco desta pesquisa. Assim foram utilizados alguns modelos tridimensionais que estavam disponíveis na biblioteca de recursos do próprio jMonkeyEngine e em seus projetos de exemplo. Aqueles modelos 3d que não foram

[1] http://mindemia.com/hostilesector/

encontrados na API do jMonkeyEngine, como casas e estradas, foram modelados utilizando o software *Google Sketchup* [2].

O segundo protótipo foi desenvolvido com intensão de explorar a interação do personagem em primeira pessoa. Esse tipo de interação é muito comum em jogos do gênero FPS (*First Person Shooter*), onde o jogador explora um ambiente tridimensional como se ele estivesse exatamente na mesma posição do personagem jogável. Além de experimentar outras configurações de câmera, neste protótipo foi adicionado a funcionalidade de textura espacial, que dá uma melhor experiência gráfica mostrando imagens que simulam um céu ao fundo em todas as direções do cenário. Assim como o primeiro protótipo, o cenário foi formado por casa e arvores.

No terceiro protótipo foi explorado o gênero de jogo de corrida, onde o personagem jogável é um veículo e o cenário simula ruas e estradas. Nesse protótipo houve um esforço considerável para compreender as funcionalidades do personagem veículo. As ações que o jogador pode realizar foram de acelerar, frear, virar para esquerda e direita. Esse jogo permitiu realizar uma série de testes quanto ao controle de física disponibilizado pelo jMonkeyEngine, como gravidade, inércia e colisão. Foi implementada também uma câmera que segue o veículo como em terceira pessoa, mas que recebe uma inclinação para o lado sempre que o veículo está virando para prover uma melhor sensação nas curvas.

No quarto protótipo desenvolvido para o estudo do motor gráfico foi implementado um jogo que permitia a troca do personagem jogável durante a execução do jogo. O jogador inicia o jogo controlando um personagem humanoide, que no caso era uma espécie de ogro. No cenário pode ser encontrado um veículo que quando o personagem está próximo dele e o jogador envia o comando de troca de personagem ele entra dentro desse veículo. Dessa forma o jogador passa a controlar o veículo como personagem jogável. A partir do momento que o jogador está controlando o veículo ele pode enviar o comando para sair desse veículo a qualquer momento, voltando a controlar o personagem humanoide. Essa troca de personagem envolve além da complexidade em tratar as entradas do usuário para os diferentes personagens (humanoide e veículo), também a reconfiguração da câmera que passa a seguir um novo modelo tridimensional. O cenário foi desenvolvido mesclando os modelo utilizados nos três protótipos anteriores, dando mais riqueza ao cenário.

O quarto protótipo foi inspirado em gêneros de jogos onde o personagem pode caminhar pelo cenário andando à pé, ou pode também conduzir um veículo a sua escolha. A troca de personagem jogável acontece durante o jogo, podendo fazer tal troca quantas vezes desejar. Muitos jogos de sucesso exploram esse recurso de troca de personagem jogável, por exemplo

[2] http://www.sketchup.com/pt-BR

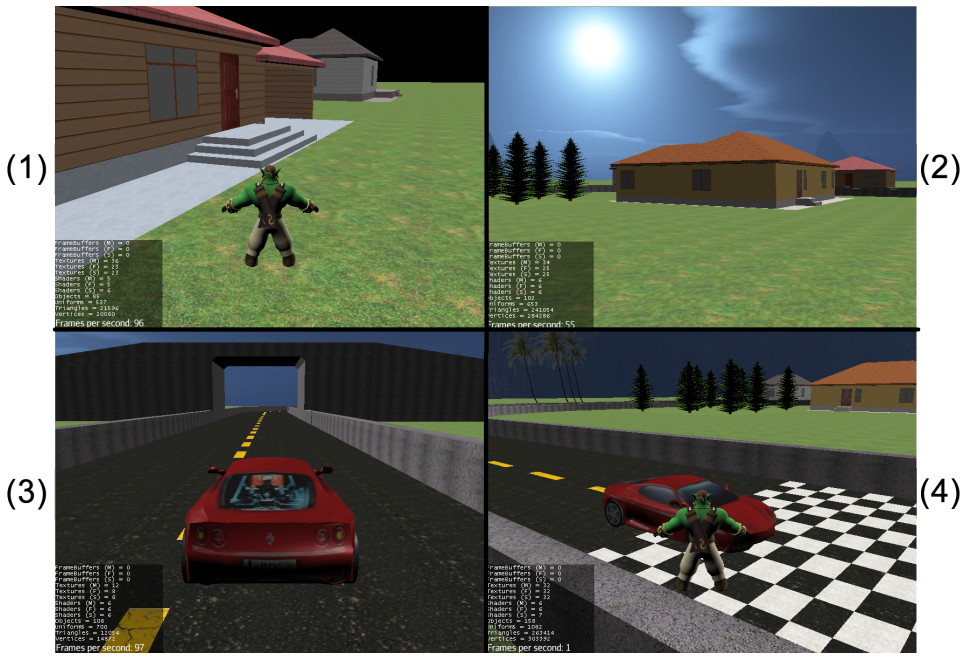

Figura 4.2: Protótipos com jMonkeyEngine3

os jogos *Watch Dogs* [3], *Battlefield* [4] e *Grand Theft Auto* [5], sendo que este último apresentou-se como um fenômeno de sucesso batendo recordes de vendas. Sua versão intitulada de *GTA V* é considerada como o produto de entretenimento que conseguiu, em menor tempo até então, atingir a marca de 1 bilhão de dólares de faturamento [6].

Apesar dos protótipos não chegarem a ser jogos completos, eles exploraram as principais funcionalidades disponibilizadas pelo jMonkeyEngine e se mostraram objeto de entretenimento funcional. Imagens dos protótipos descritos nessa seção podem ser vistos na Figura 4.2.

Essa primeira parte da metodologia foi de fundamental importância para a compreensão dos recursos disponíveis no motor de jogo. Através do desenvolvimento dos protótipos foi possível elaborar os tutoriais que serão mostrados mais a frente nessa pesquisa. Além disso, os protótipos permitiram o estudo dos subdomínios necessários para o desenvolvimento das ferramentas MDGD, conforme descrito as próximas seções.

[3] http://watchdogs.ubi.com/watchdogs/
[4] http://www.battlefield.com/
[5] http://www.rockstargames.com/grandtheftauto/
[6] http://www.guinnessworldrecords.com/news/2013/10/confirmed-grand-theft-auto-breaks-six-sales-world-records-51900/

4.3 Estudo dos subdomínios

Essa segunda parte da metodologia tem por objetivo estudar os protótipos desenvolvidos na seção anterior identificando seus subdomínios. Assim, deve-se encontrar pontos que podem ser automatizados com o MDGD e pontos onde será necessário inserir códigos manuais.

Analisando os códigos dos protótipos, percebe-se que muitas partes desses códigos compartilham de uma base comum, a qual possivelmente não sofrerão alterações, como a estrutura do projeto, o *game loop*, configurações de iluminação e definições das forças físicas. As chamadas às bibliotecas do jMonkeyEngine também serão utilizadas da mesma forma, mesmo alterando o escopo do projeto.

Porém, existem outras partes do projeto que poderão sofrer alterações, pois são diferentes em cada um dos protótipos. Também deve-se levar em consideração a característica de projetos de jogos normalmente terem escopo aberto, fazendo com que o desenvolvedor consiga identificar mudanças no projeto apenas após testar o protótipo funcional. Dentre os artefatos dos protótipos, foram identificados três importantes elementos que possuem grande variabilidade: a câmera, o personagem e o cenário.

- **Subdomínio Câmera:**

  A câmera pode levar o projeto a uma série de variações, onde por exemplo, nos protótipos foram utilizadas as câmeras em terceira pessoa, em primeira pessoa e também uma câmera especial para jogos de corrida que se inclina nas curvas. Em outros projetos percebe-se essa mudança também, como pode-se observar que nas duas primeiras versões do jogo *Grand Theft Auto* a câmera era posicionada acima do personagem jogável, já nas versões seguintes desse mesmo jogo a câmera é posicionada atrás do personagem, como em terceira pessoa. Com essa análise foram identificadas as seguintes variações de câmera:

  - Câmera de perseguição.
  - Câmera em primeira pessoa.
  - Câmera em terceira pessoa.
  - Câmera em terceira pessoa sensível às curvas.
  - Câmera acima do personagem.

- **Subdomínio Personagem:**

  O personagem jogável é outro subdomínio que poderá ser alterado. Nos protótipos foram desenvolvidos os personagens do tipo humanoide e veículo. Além da variação do tipo do

personagem existem as variações sobre a velocidade em que cada um dos personagens irá andar, qual a sua aceleração, peso e demais características que normalmente só poderão ser refinadas ao se executar o jogo. Por ser um artefato que sofrerá constantes mudanças até calibrar suas devidas características, também é interessante existir uma ferramenta que permita a geração automática de código. Dentre os recursos disponibilizados pelo jMonkeyEngine e a análise dos protótipos foram identificadas as seguintes variações de personagem:

  - Humanoide:
    - Velocidade em que o personagem gira para os lados.
    - Velocidade em que o personagem anda para frente.
    - Velocidade em que o personagem anda para trás.
  - Veículo:
    - Características do amortecedor.
    - Peso do veículo.
    - Fator de atrito dos pneus.
    - Aceleração máxima.
    - Velocidade máxima.
    - Velocidade do giro de direção.

- **Subdomínio Cenário:**

  O cenário pode possuir grandes alterações já que nos protótipos não foi definido qual seria a disposição dos elementos que o compõe. Apenas foi definido que o cenário deve possuir elementos como estradas, casas e arvores, mas a disposição desses elementos poderá mudar bastante. Portanto esse subdomínio convém ser automatizado, já que provavelmente sofrerá constantes mudanças. Dentre os elementos identificados, o cenário pode ser composto pelos seguintes:

  - Casas
  - Árvores
  - Estradas
    - Retas
    - Curvas
    - Rampas

* Pontes

Contudo, existem artefatos do projeto que não pertencem às partes estruturais do jogo e também não foram definidos como sendo partes automatizáveis. Por exemplo, o evento de troca de personagens, onde o jogador poderá mudar de um personagem humanoide para um veículo durante o jogo apresentado no quarto protótipo. Esse evento envolve não só a mudança de perspectiva da câmera, como também mudanças no tratamento das entradas do usuário. Esse é o tipo de alteração no projeto que depende dos códigos gerados e portanto deverão ser desenvolvidos manualmente.

4.4 Desenvolvimento do protótipo MDGD

A fim de permitir maior flexibilidade na criação do projeto, podem ser desenvolvidos geradores independentes para cada um dos artefatos definidos na seção anterior. Esta abordagem é demonstrada por Furtado, Santos e Ramalho (2011) e Jovanovic et al. (2011), onde são criados diversos geradores e diferentes DSLs aumentando a flexibilidade de projetos de jogos eletrônicos com MDD.

Para o desenvolvimento das ferramentas MDGD, primeiramente deve ser feita uma refatoração dos códigos desenvolvidos nos protótipos a fim de possibilitar a integração entre os subdomínios. Depois deve-se desenvolver as ferramentas de modelagem e em seguida o desenvolvimento dos geradores de código.

4.4.1 Refatoração do código fonte

Após efetuar o estudo dos protótipos elencando os subdomínios que deverão ser automatizados e aqueles que deverão ser codificados manualmente, é necessário refatorar os códigos que foram desenvolvidos nos protótipos. Essa refatoração é necessária para permitir melhor integração entre as partes. Com base nesse estudo foi feita uma modelagem do jogo eletrônico.

Para representar os objetos de um jogo eletrônico e sua estrutura estática Tang e Hanneghan (2011b) recomendam a utilização da UML através do diagrama de classes. Esse diagrama também se mostra útil para organizar os artefatos que possuem variabilidade e também a integração destes com o código que será inserido manualmente.

O diagrama de classe mostra um conjunto de classes, interfaces, colaborações e seus relacionamentos. Geralmente esse diagrama é utilizado em modelagem de sistemas orientados a objetos (BOOCH; JACOBSON; RUMBAUGH, 2006).

Para os artefatos que possuem variabilidade foram definidas *interfaces* para que sua implementação pudesse ser feita de diferentes formas, seguindo assim um padrão de projeto que permite integração com outras partes do código.

Deseja-se criar um projeto que tenha a capacidade de receber alterações manuais futuramente, visando a flexibilidade do projeto. Este mesmo projeto mostra-se portador de múltiplos subdomínios, os quais devem estar integrados entre si. Conforme descrito na seção 2.3 desta dissertação, uma maneira de integrar múltiplos subdomínios é fazer a integração entre diferentes DSLs através de padrões de projeto. No caso quando o código não-gerado depende de código gerado pode-se utilizar o padrão *template method*, fazendo com que o código não-gerado não precise saber detalhes da implementação do código gerado.

Padrões de projeto são uma descrição do conhecimento e experiência acumulados para solucionar um problema comum. Dessa forma a mesma solução pode ser reutilizada em diferentes aplicações. Os padrões de projeto permitem que sejam feitas abstrações genéricas através de objetos abstratos e concretos e sua interações (SOMMERVILLE, 2007).

No padrão de projeto *template method* é definido um esqueleto de um algoritmo em uma operação, postergando a implementação de algumas funcionalidades para subclasses. Assim é possível que as subclasses redefinam certos passos de um método sem mudar sua estrutura, permitindo a utilização do método por outras partes do projeto, sem a necessidade de conhecer sua implementação (GAMMA; JOHNSON; HELM RICHARD;VLISSIDES, 2006).

Conforme análise do código e seus subdomínios, juntamente com a necessidade de seguir o padrão de projeto *template method* para garantir a flexibilidade do projeto, foi desenvolvido o diagrama de classes mostrado na Figura 4.3. É importante notar que as interfaces PlayerInterface, CameraInterface e TrackInterface podem ser implementadas de diferentes formas, fazendo com que o projeto seja tão flexível quanto os requisitos esperados. As classes dos pacotes Camera, Player e Track tiveram seus atributos e métodos suprimidos para melhorar a visualização do diagrama, porém cada um desses pacotes são apresentados e discutidos a seguir.

Na classe Main são implementados os métodos para inicializar a aplicação e controlar o *game loop* do jogo. Seus atributos player, camera e track fazem referência às interfaces abstratas, fazendo com que não haja dependência quanto a forma de implementação que essas classes sofrerão. A classe Main possui os seguintes métodos:

- main: método estático responsável por inicializar a aplicação Java.
- simpleInitApp: método com a funcionalidade de instanciar objetos do jogo e configurar as características iniciais do jogo como gravidade e iluminação.

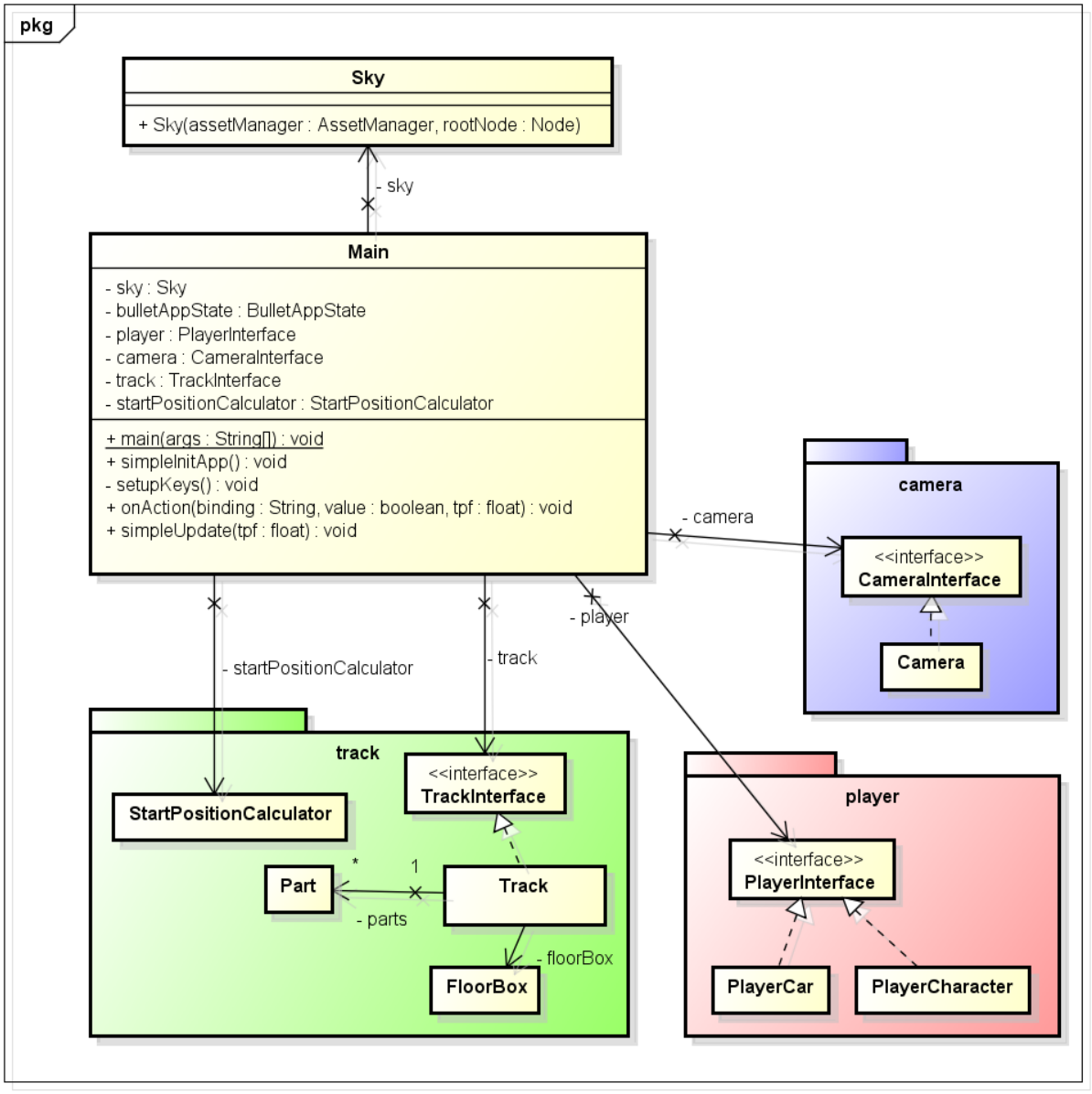

Figura 4.3: Diagrama de Classes Geral

- setupKeys: configura as teclas utilizadas durante o jogo.
- onAction: método chamado quando o usuário pressiona ou solta alguma tecla.
- simpleUpdate: método chamado a cada iteração do *game loop*, onde podem ser implementadas verificações durante a execução do jogo.

A classe Sky é responsável por implementar uma textura ao fundo do jogo em todas as direções simulando um céu ao horizonte. Esta classe não possui nenhum atributo ou método, ficando a cargo do seu construtor a definição de todas suas funcionalidades.

A interface PlayerInterface poderá sofrer implementação do personagem jogável, podendo ser um humanoide ou um veículo. Em ambas situações deverão ser implementados os mesmos métodos, porém com funcionalidades distintas, bem como seu modelo tridimensional. O mesmo ocorre com a CameraInterface que poderá implementar uma câmera disposta em diferentes posições. No caso da TrackInterface, poderão ser implementados diferentes cenários através de uma lista de objetos do tipo Part.

No pacote câmera é definida a interface abstrata CameraInterface e também a sua classe de implementação Camera. Seu diagrama de classes está apresentado na Figura 4.4. Seguindo as definições da interface, podem ser implementadas uma variedade de câmeras, conforme descrito na análise dos subdomínios. A implementação da câmera possui apenas o atributo camNode, que controla o nó da câmera. Seus métodos são os seguintes:

- ConfigureCam: método capaz de realizar as configurações iniciais da câmera. Ele recebe qualquer objeto que tenha implementado o PlayerInterface, dessa forma garante integração com uma grande variação de jogadores, independente da sua forma de implementação. Também recebe os objetos do tipo FlyByCamera e Camera que são fornecidos pela API do jMonkeyEngine.
- update: atualiza a câmera em relação à posição do jogador.
- detachCam: destrói o vinculo entre a câmera e o jogador, fazendo com que a câmera possa ser posteriormente configurada em outro jogador.

No pacote player são implementadas as funcionalidades do jogador. No diagrama da Figura 4.5 são mostradas duas formas de implementação do jogador. Na classe PlayerCar o jogador está implementado como um veículo, tendo assim atributos do tipo VehicleControl para controle do veículo e VehicleWheel para controle de cada roda do veículo. Também são declarados os atributos steeringValue para definição do valor de direção, accelerationValue para

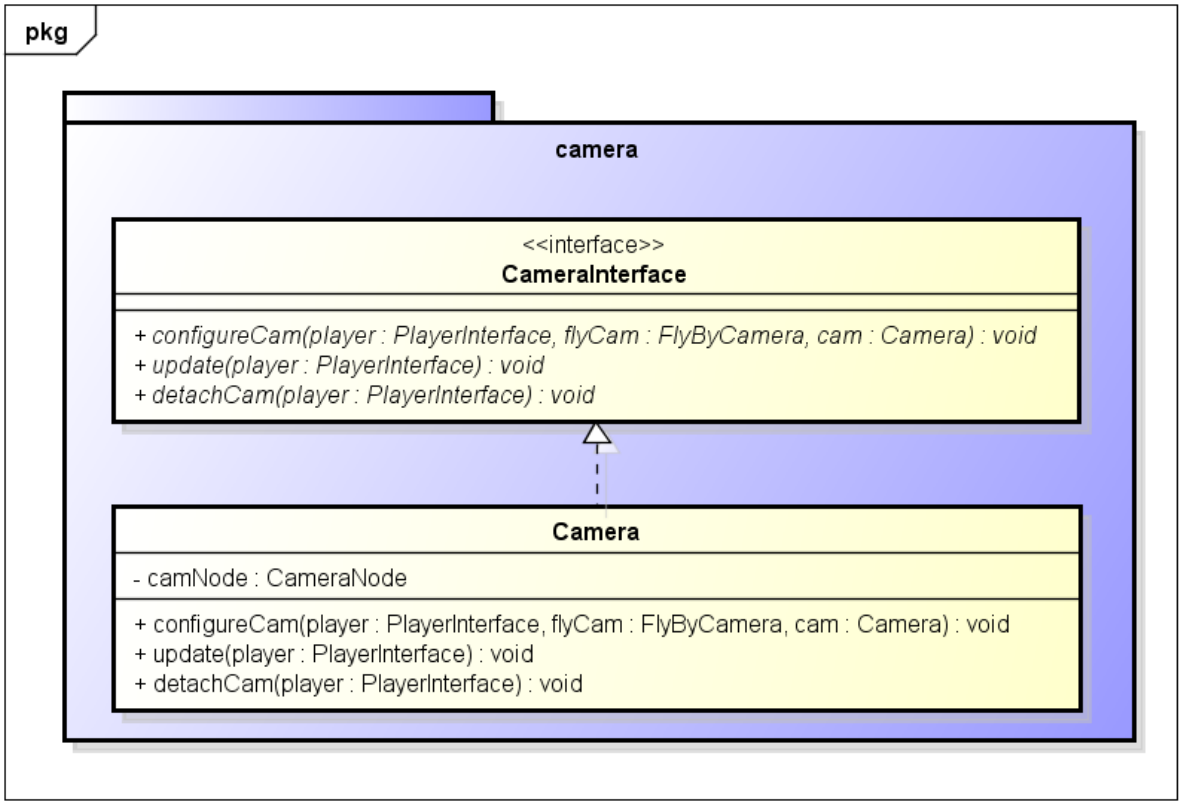

Figura 4.4: Diagrama de Classes do pacote Camera

definição do valor de aceleração, carNode que aponta para o nó do modelo 3d e as variáveis booleanas accelerate, turnLeft, turnRight e brake para definir os eventos de ação do veículo. Já a classe PlayerCharacter está implementando um jogador humanoide, portanto seus atributos são referentes a esse tipo de jogador. Os atributos da classe PlayerCharacter são physicsCharacter que define as características do personagem, characterNode que aponta para o nó do modelo 3d, walkDirection que define a direção que o personagem irá andar e viewDirection que define a direção que o personagem deverá estar virado. Também são definidas as variáveis booleanas forward, backward, leftRotate, rightRotate para controle dos eventos de ação do personagem. Ambas classes possuem os mesmo métodos, cada qual com sua implementação específica. Os métodos das classes que implementam a interface PlayerInterface são:

- buildPlayer: método que constrói o jogador e o insere no ambiente do jogo.
- unbuildPlayer: desconstrói o jogador retirando-o do jogo.
- update: método chamado a cada iteração do *game loop*, onde pode-se implementar ações personalizadas.
- getNode: retorna o nó do modelo 3d do personagem.

- turnRight: método que ativa ou desativa a ação de virar para direita.
- turnLeft: método que ativa ou desativa a ação de virar para esquerda.
- forward: método que ativa ou desativa a ação de ir para frente.
- backward: método que ativa ou desativa a ação de ir para traz.
- action: método que ativa ou desativa uma ação específica do jogador. No caso do personagem humanóide sua ação é pular.
- setPhysicsLocation: atribui uma nova posição do jogador.
- setPhysicisRotation: atribui uma nova rotação do jogador.

O pacote track tem seu diagrama de classes mostrado na Figura 4.6. Pode-se observar que nele há uma interface abstrata chamada TrackInterface. Nela são definidos o método createTrack, responsável por criar o cenário e o método getParts que retorna uma lista das partes que compõe o cenário. Em sua implementação foi instanciado apenas um atributo que compõe uma lista de objetos do tipo Part. Na classe Part são implementadas as partes do cenário, tendo como atributos as variáveis x, y e z que definem a posição desse elemento no jogo e o atributo *model* que define o nome do arquivo de modelo 3d que ele se refere. Além de seu construtor são descritos os métodos createPart que instancia o modelo 3d no cenário, *attach* que adiciona esta parte para ser visualizada no jogo e setPosition que define a posição do objeto no jogo.

A classe StartPositionCalculator tem a funcionalidade de calcular a posição inicial do personagem no cenário. Por isso, em seu construtor devem ser passados um objeto que tenham implementado o PlayerInterface e outro objeto que tenha implementado o TrackInterface. Assim, independente da forma como foi feita a implementação do jogador e do cenário, a classe StartPositionCalculator poderá fazer o cálculo da posição inicial no cenário e atribuir esta posição ao jogador.

A classe FloorBox implementa um chão com textura similar a um gramado. Seu tamanho é calculado automaticamente a partir de uma análise do tamanho do cenário.

Todas essas classes implementam um jogo funcional utilizando o motor de jogo jMonkeyEngine, o qual servirá como base para os geradores de código. Na próxima subseção será apresentado como se deu o desenvolvimento das ferramentas de modelagem neste desse projeto.

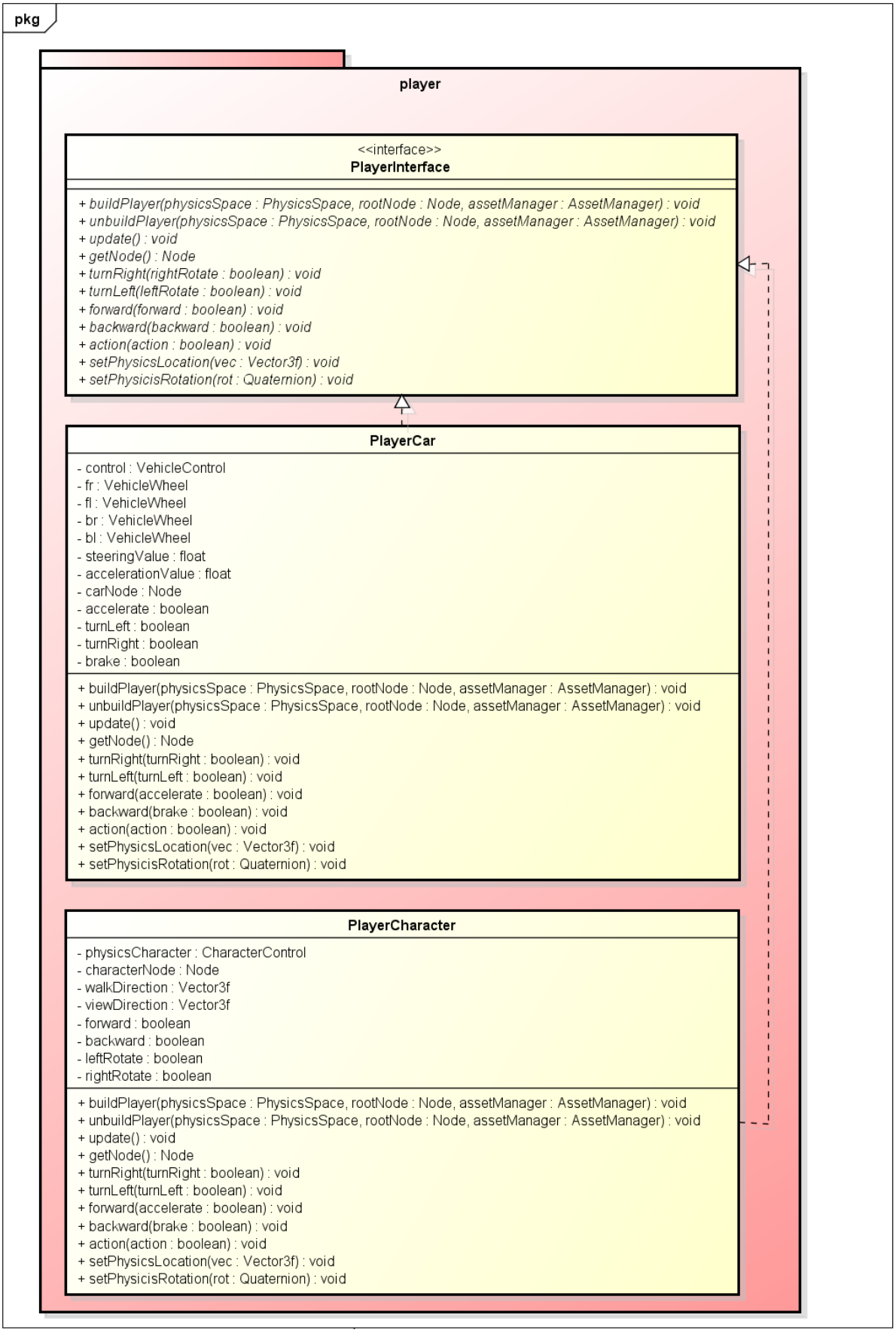

Figura 4.5: Diagrama de Classes do pacote Player

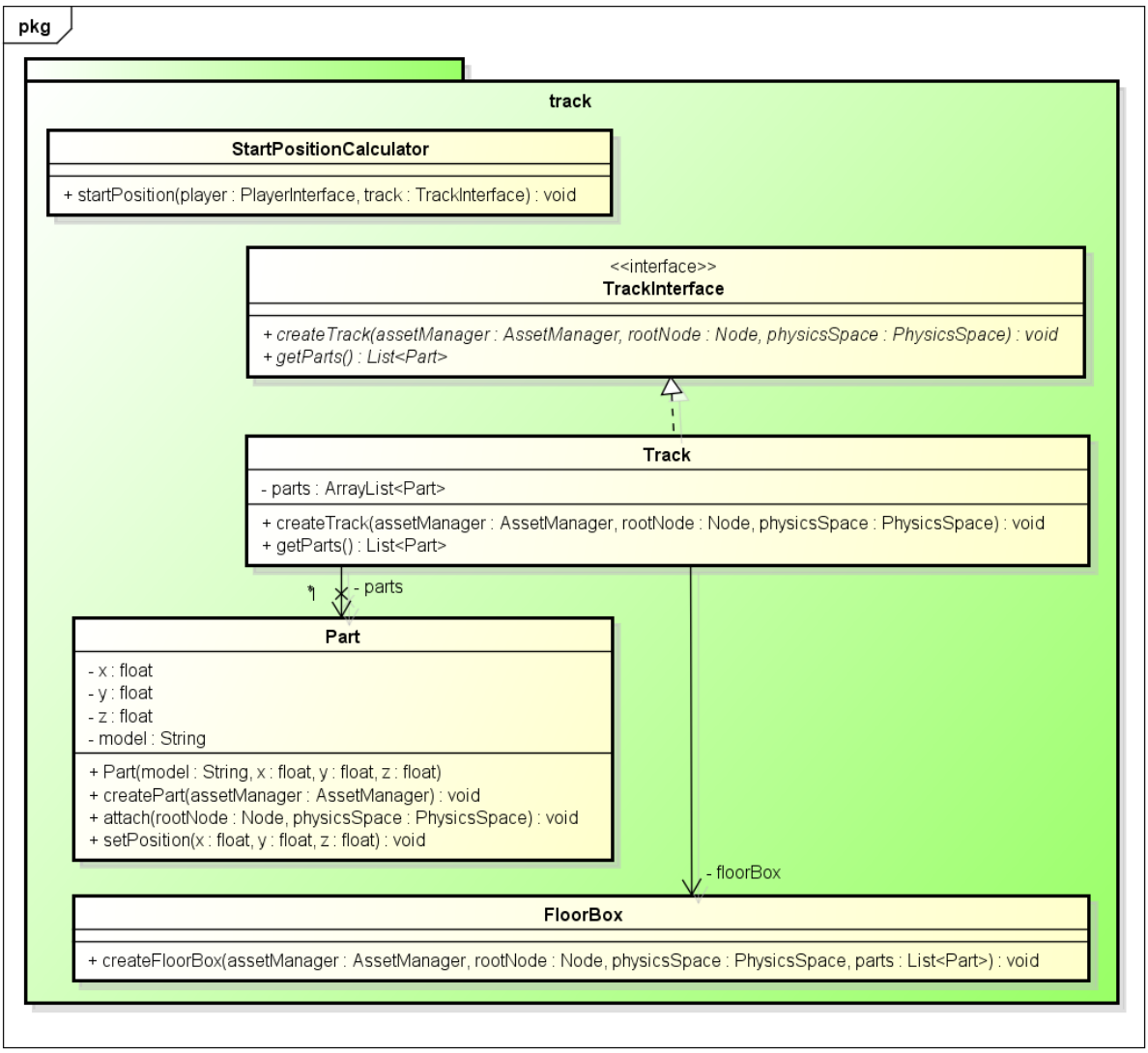

Figura 4.6: Diagrama de Classes do pacote Track

4.4.2 Desenvolvimento das ferramentas de modelagem

Para criação dos modelos foram utilizados arquivos no formato XML. Este formato é preferencial pela sua facilidade na escrita e leitura tanto pelos desenvolvedores quanto pelo computador.

Os arquivos XML servem como entrada para as transformações do código, porém editar arquivos XML manualmente não é uma tarefa desejável quando se pretende ter uma facilidade na edição. Uma boa alternativa para agilizar e garantir consistência na edição das entradas dos transformadores é desenvolver editores visuais. Assim o desenvolvedor especifica os atributos da entrada visualmente, e todo o código XML é gerado automaticamente. Caso precise realizar alguma modificação, bastar alterar no editor visual que o código XML é gerado novamente.

Juntamente com a API do jMonkeyEngine é disponibilizado uma IDE que facilita a criação e codificação de jogos. A IDE do jMonkeyEngine é baseada na plataforma NetBeans [7] , fornecendo além das amplas ferramentas de desenvolvimento da IDE NetBeans, plugins exclusivos para a criação de conteúdo do jogo. No NetBeans é possível criar projetos de plugins que são incorporados à IDE, dessa forma os editores visuais foram desenvolvidos como plugins para serem incluídos na IDE do jMonkeyEngine, fazendo com que sua utilização seja ainda mais fácil.

Foram definidos três modelos para o projeto: câmera, personagem e cenário. Cada modelo contém seus próprios atributos e valores pré-definidos.

No arquivo de modelo para câmera foi definido apenas a existência do atributo ”tipo”. Para esse atributo foram definidos os seguintes valores possíveis:

- Free: tipo de câmera livre, onde não há nenhuma interação entre a câmera e o personagem jogável.
- Back: a câmera é fixada atrás do personagem jogável, criando um visão de terceira pessoa. Neste caso há uma variável para definir a distancia que a câmera ficará do personagem.
- Over: a câmera é fixada acima do personagem jogável, podendo esta ser configurada de forma variável quanto a distância que ficará colocada.
- First Person: a câmera é coloca um pouco a frente do personagem, simulando uma visão de primeira pessoa.

[7] https://netbeans.org/

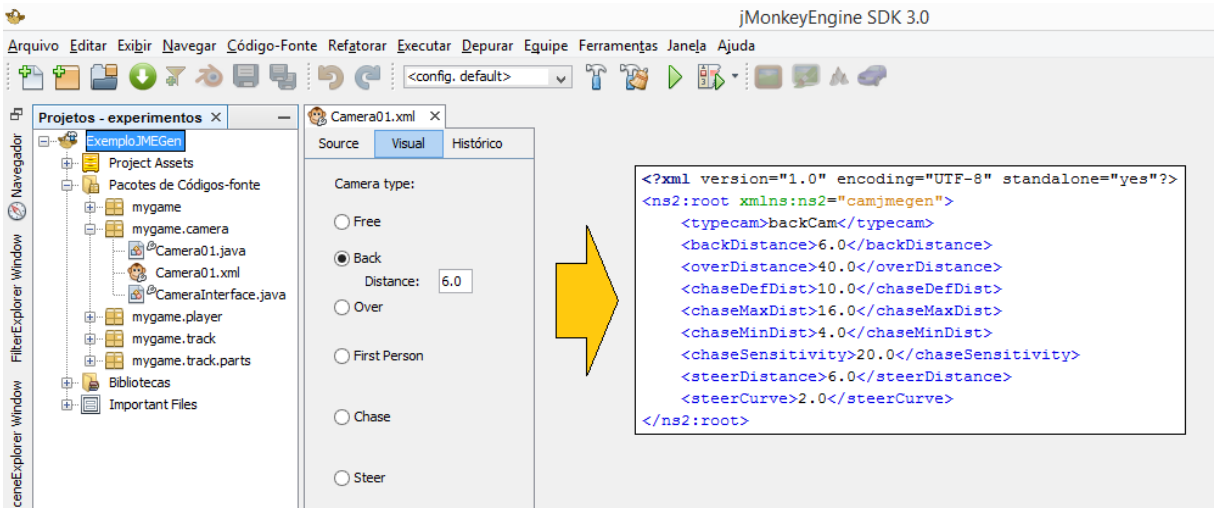

Figura 4.7: Editor visual da câmera integrado à IDE jMonkeyEngine

- Chase: a câmera persegue o personagem, acompanhando seu movimento. Poderão ser definidos valores para a distância máxima, mínima e também a distância padrão, além de uma variável para definir a sensibilidade da perseguição da câmera.

- Steer: tipo de câmera muito parecida com a câmera Back, porém ela tem a funcionalidade de inclinar um pouco para os lados dando uma melhor sensação de movimento em curvas, especialmente em jogos de corrida. São definidas variáveis para a distância da câmera e fração de movimentação nas curvas. Ao contrário dos outros tipos de câmera, não existe no jMonkeyEngine predefinições para esse tipo de interação da câmera com o veículo, portanto ela foi codificada manualmente e disponibilizada para o gerador de código.

Foi desenvolvido um editor visual para facilitar a edição do arquivo XML de modelo da câmera. Esse editor juntamente com um exemplo de arquivo de modelo de câmera podem ser visualizados na Figura 4.7.

Outro subdomínio do projeto de estudo é o personagem jogável. De acordo com as análises anteriores, o jogador pode ser um veículo ou um caracter humanoide. Dessa forma também foi criado um modelo juntamente com um editor visual. Nesse modelo foram definidos os seguintes atributos quando escolhido o tipo de personagem humanoide:

- Rotate Increase: número que corresponde ao incremento utilizado para rotacionar o jogador. Quanto maior este número, mais rápido será sua velocidade de rotação.

- Walk Forward: número correspondente ao incremento utilizado para movimentar o personagem para frente. Quando maior este número, maior será a velocidade em que o personagem andará.

- Walk Backward: valor utilizado para incrementar movimentos do personagem para traz.

Quando o desenvolvedor define o tipo de personagem como sendo um veículo, os atributos a serem definidos são outros, já que a interação do usuário com esse tipo de personagem é diferente. Os atributos do personagem do tipo veículo são os seguintes:

- Stiffness: rigidez do amortecedor
- Comp Value: compressão do amortecedor
- Damp Value: absorção do amortecedor
- Mass: peso do veículo em kg.
- Front Friction Slip: valor de atrito dos pneus dianteiros. Um valor baixo fará o carro derrapar mais facilmente.
- Back Friction Slip: valor de atrito dos pneus traseiros.
- Max Acceleration: aceleração máxima do veículo.
- Max Speed Km/Hour: velocidade máxima do veículo em km/hora
- Steering Increase: valor de incremento da direção. Quanto maior este número, mais rápida será a direção do veículo.

A partir dos atributos citados, foi definido o formato do arquivo de modelo de entrada e o editor visual que pode ser visto na figura 4.8. O editor visual é integrado à IDE de desenvolvimento do jogo, e permite a edição de todas as variáveis do modelo. O arquivo de modelo é um arquivo XML, e também pode ser editado manualmente. Sempre que é feita alguma alteração nos valores dos atributos através do editor visual, essa alteração é realizada automaticamente no arquivo de modelo e vice-versa.

Conforme analisado na subseção anterior o cenário também poderá ser personalizado, abrindo margem para uma grande variabilidade de cenários. Um cenário é composto por partes, sejam elas pedaços de estradas, curvas, pontes, casas, arvores, etc. Portanto o arquivo de modelo para a construção do cenário é composto por uma lista de objetos, os quais devem ter a descrição do modelo tridimensional que ele se refere e sua posição no espaço. No editor visual, basta escolher qual parte deseja-se adicionar ao cenário e clicar na posição desejada para que este seja então adicionado na lista de elementos. Na Figura 4.9 é mostrado o editor visual à esquerda e o arquivo XML de modelo à direita, sendo que ambos podem ser editados pelo desenvolvedor.

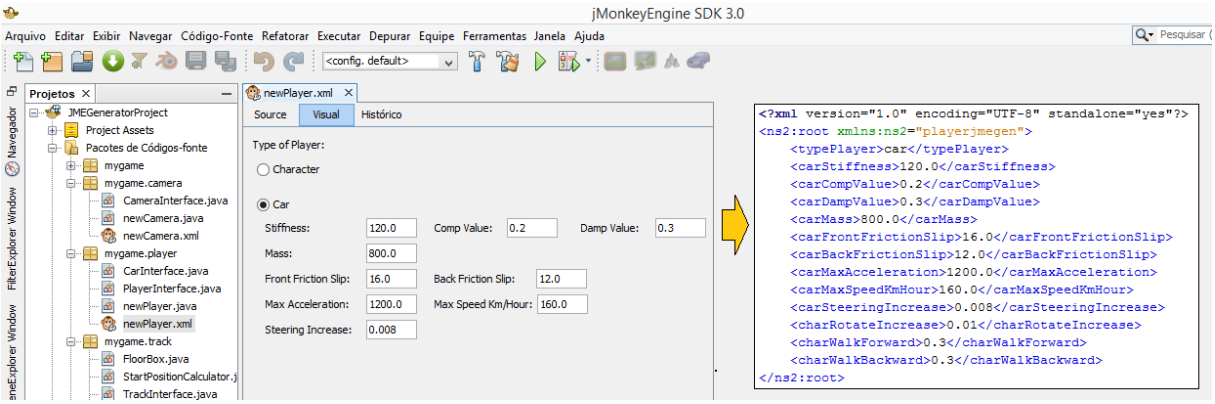

Figura 4.8: Editor visual do jogador integrado à IDE jMonkeyEngine

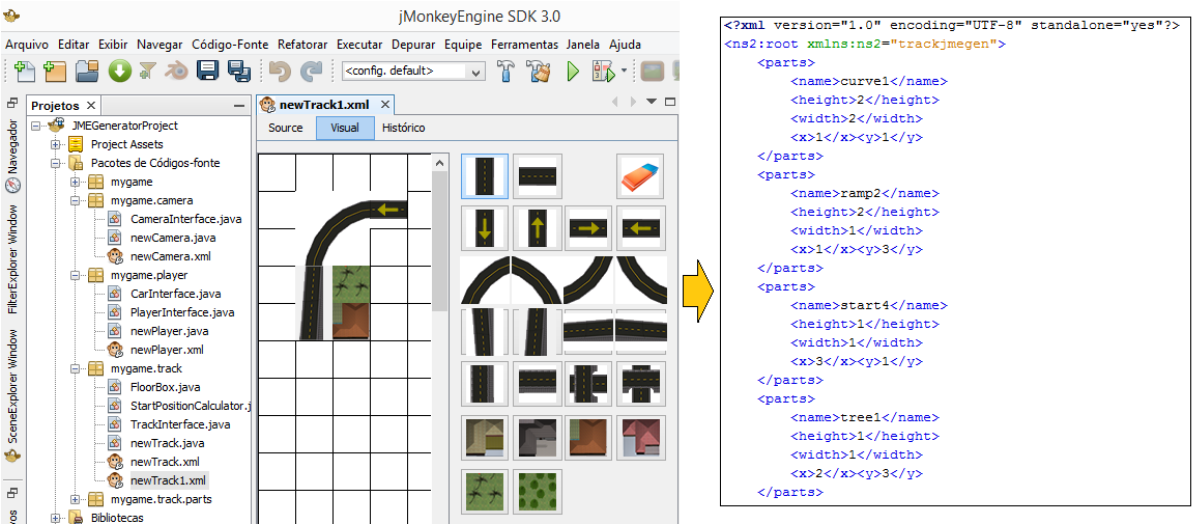

Figura 4.9: Editor visual do cenário integrado à IDE jMonkeyEngine

4.4.3 Desenvolvimento dos geradores de código

Normalmente, para construir geradores, o uso de *templates* é preferencial, já que permite a reutilização de código com versatilidade para parametrizações. Um *template* é formado por um arquivo instrumentado com construções de seleção e expansão de código que pode ser combinado com uma entrada resultando na geração de um código personalizado (CZARNECKI; EISENECKER, 2000). As entradas são os arquivos XMLs definidos na subseção anterior.

Uma ferramenta que pode ser utilizada para transformação de código através de *templates* é o FreeMarker. Ele é um sistema genérico para gerar saída de texto com base em modelos. O FreeMarker não é um aplicativo feito para usuários finais em si, mas algo que os programadores podem incorporar em seus produtos. Embora tenha algumas capacidades de programação, como estruturas condicionais e de repetição, ele não pode ser tido como uma linguagem de programação, ficando com a missão específica de gerar arquivos textuais que exibem os dados preparados em modelos (FREEMARKER, 2014).

FreeMarker é um pacote de software de código aberto projetado para geração de texto com base em modelos. Ele permite a geração de documentos textuais flexíveis como, por exemplo, páginas HTML ou arquivos de códigos fonte. FreeMarker opera em um modelo de dados específico com base em listas e mapas, onde as transformações são realizadas por um programa Java. Como consiste em um motor de *templates* genérico, o FreeMarker não está vinculado a uma tecnologia de apresentação em particular e pode ser usado para geração de diferentes tipos de arquivos. (RADJENOVIC; MILOSAVLJEVIC; SURLA, 2008)

A Figura 4.10 mostra uma parte do código do *template* para geração da câmera. Neste código pode-se ver como são feitas as notações condicionais para o FreeMarker, onde através de uma sintaxe parecida com o XML é possível definir qual código deverá ser gerado para cada tipo de câmera. Também são utilizadas as variáveis backDistance e overDistance para definir a distancia que câmera deverá ficar do personagem jogável.

Para o FreeMarker efetuar as transformações dos *templates* em código fonte é necessário passar os valores de entrada como parâmetro. Esses valores são passados através de um objeto do tipo *HashMap*, que é uma implementação de estrutura de dados capaz de armazenar um conjunto de chaves e valores. Como o arquivo de modelo foi armazenado no formato XML, é necessário fazer uma leitura do mesmo e convertê-lo para o objeto *HashMap*. Para isso, foi utilizada a API *Java Architecture for XML Binding*[8], também conhecida como JAXB. Essa API permite a transformação de arquivos XML em objetos Java e vice-versa. Após essa transformação,

[8]https://jaxb.java.net/

```
public void configureCam(PlayerInterface player, FlyByCamera flyCam, Camera cam) {
  <#if typecam == "backCam" || typecam == "overCam" || typecam == "firstPersonCam">
     flyCam.setEnabled(false);
     camNode = new CameraNode("CamNode", cam);
     camNode.setControlDir(CameraControl.ControlDirection.SpatialToCamera);
     player.getNode().attachChild(camNode);
  </#if>
  <#if typecam == "backCam">
     camNode.setLocalTranslation(new Vector3f(0, 3, 12));
     camNode.lookAt(player.getNode().getLocalTranslation(), Vector3f.UNIT_Y);
     camNode.move(0, 0, ${backDistance}f - 10);
  </#if>
  <#if typecam == "overCam">
    camNode.setLocalTranslation(new Vector3f(0, ${overDistance}f + 13, 12));
    camNode.lookAt(player.getNode().getLocalTranslation(), Vector3f.UNIT_Y);
    camNode.move(0, 0, (${overDistance}f + 13) * -13 / 55);
  </#if>
  <#if typecam == "firstPersonCam">
    camNode.setLocalTranslation(new Vector3f(0, 0, 12));
    camNode.lookAt(player.getNode().getLocalTranslation(), Vector3f.UNIT_Y);
    camNode.setLocalTranslation(new Vector3f(0, 3, 0));
  </#if>
  <#if typecam == "chaseCam">
    flyCam.setEnabled(false);
    chaseCam = new ChaseCamera(cam, player.getNode());
    chaseCam.setSmoothMotion(true);
    chaseCam.setRotationSensitivity(10f);
    chaseCam.setDefaultHorizontalRotation(FastMath.PI / 2);
```

Figura 4.10: Parte do Template para geração da câmera

basta transferir os valores dos atributos para o objeto *HashMap* que o FreeMarker se encarrega de gerar os códigos fontes definidos nos *templates*.

Após definir os arquivos de *template*, o formato dos arquivos de modelo, os editores visuais e os geradores de código, é possível desenvolver o jogo eletrônico pela abordagem MDGD. Para melhor compreensão do fluxo dos dados durante o desenvolvimento, na Figura 4.11 são exibidos os componentes desta abordagem, bem como a interação entre eles. Dentre esses componentes, há a representação dos editores visuais que servem como ferramentas para edição dos arquivos de modelo. A interação entre os arquivos de modelos e os editores visuais ocorre nos dois sentidos, já que o modelo pode ser editado tanto pelo editor visual, quanto diretamente no arquivo XML. Em seguida o arquivo de modelo, juntamente com os *templates* são utilizados como entrada pelo processador FreeMarker, o qual tem como saída o código fonte da implementação desejada.

Para facilitar ainda mais a criação dos projetos que utilizarão a abordagem MDGD, foi desenvolvido um assistente de criação de projetos como plugin do Netbeans, o qual também pode ser utilizado na IDE do jMonkeyEngine. Esse plugin tem como funcionalidade criar um projeto contendo as classes que não possuem transformadores, já que elas não apresentam

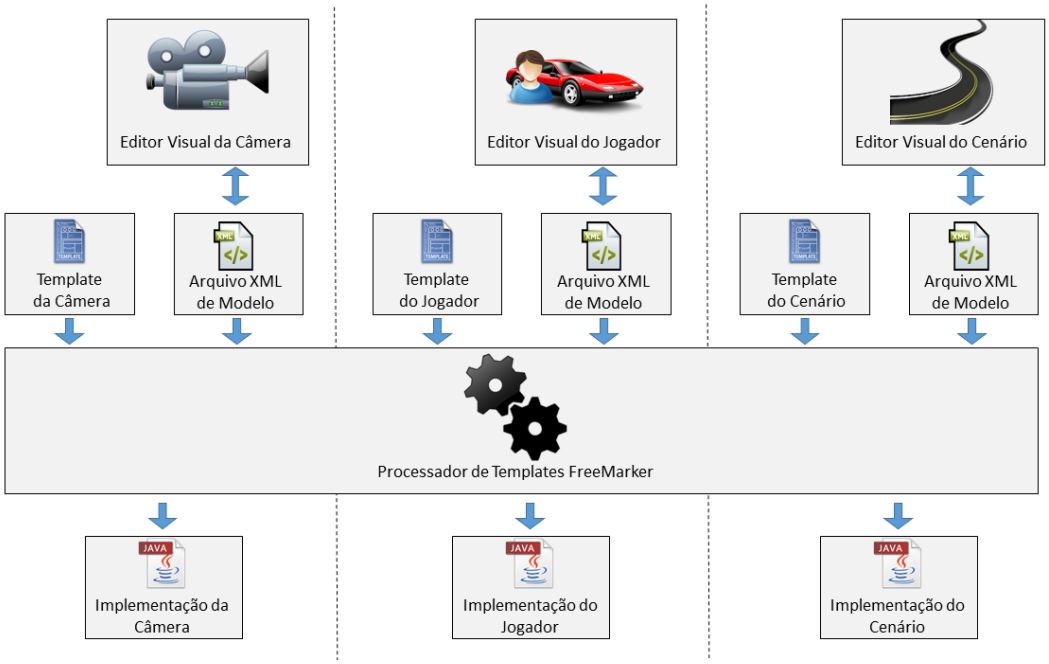

Figura 4.11: Componentes para Geração de Código porpostos neste trabalho

variabilidade no escopo do projeto. O assistente é mostrado na Figura 4.12. Percebe-se que é apresentado um defeito na classe Main devido a falta da implementação das interfaces Camera, Player e Track. Após realizar a implementação destas interfaces o defeito é corrigido.

Todo o conjunto de templates, editores visuais e geradores de códigos foram empacotados em um plugin que pode ser facilmente instalado na IDE NetBeans ou na IDE do jMonkeyEngine. Este plugin recebeu o nome de JMEGenerator.

As classes que serão geradas pelos transformadores seguiram o padrão de projeto *template method*, pois estão implementando interfaces abstratas genéricas. Dessa forma pode-se facilmente alterar o projeto, sem que seja necessário alterar as classes geradas, já que independente da forma que o gerador criar as classes, elas sempre obedecerão um padrão. Porém não é possível avaliar a eficiência das alterações manuais no projeto pelo próprio pesquisador autor do jMEGenerator, já que ele possui um conhecimento prévio sobre a tecnologia e as ferramentas. Portanto no próximo capítulo é apresentado um experimento com desenvolvedores voluntários que não possuem conhecimento prévio deste projeto.

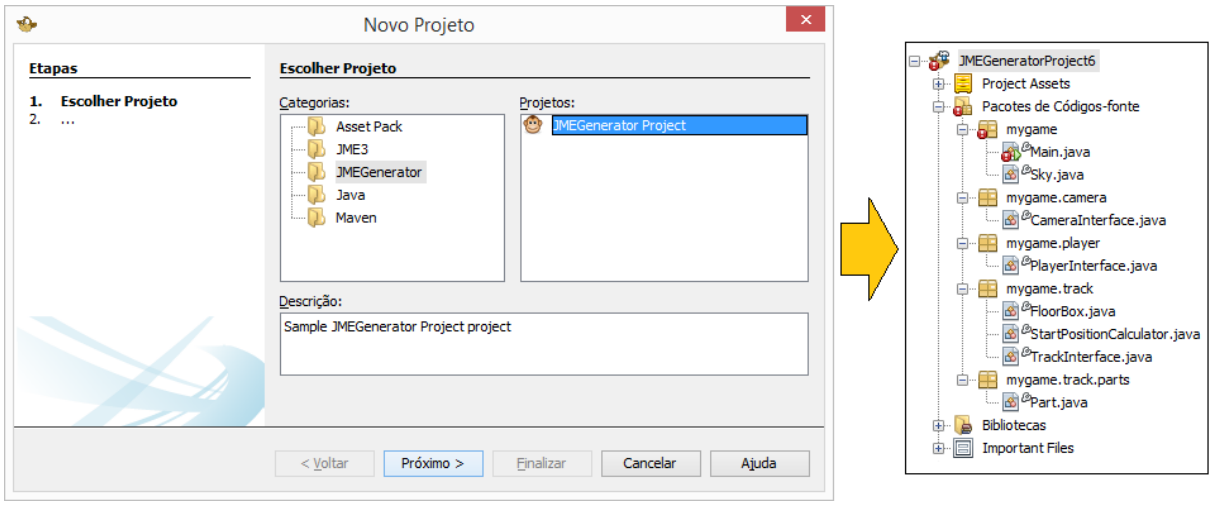

Figura 4.12: Assistente de novo projeto

Capítulo 5

EXPERIMENTAÇÃO DA ABORDAGEM MDGD

5.1 Considerações Iniciais do Experimento

Após a definição dos modelos e criação da ferramenta jMEGenerator, os jogos puderam ser desenvolvidos com bastante rapidez. O pesquisador alcançou, sem muito esforço, a marca de 2 minutos para a criação de um jogo completo. Esse número não significa muito, já que o pesquisador tem profundo conhecimento da ferramenta que ele mesmo criou, invalidando a afirmação de que o MDGD trouxe eficiência. Tampouco pode-se realizar estudo sobre flexibilidade, já que o desenvolvedor conhece o código que foi gerado e tem aprendizado prévio sobre o motor de jogo escolhido. Portanto fez-se necessário avaliar a abordagem de desenvolvimento MDGD com desenvolvedores que não conheçam a ferramenta desenvolvida nesta dissertação.

Travassos, Gurov e Amaral (2002) afirmam que a experimentação oferece uma forma sistemática, disciplinada e computável para avaliar atividades que envolvam a criação de novos métodos, técnicas, linguagens e ferramentas de desenvolvimento de software.

Dessa forma, com intuito de verificar se a abordagem de desenvolvimento de jogos MDGD proporciona flexibilidade, eficiência e aprendizado no desenvolvimento foram planejados e realizados quatro tarefas experimentais. Na seção 5.1 é feita a definição do experimento utilizando a abordagem GQM. Na seção 5.2 é descrito o planejamento deste experimento detalhando as métricas utilizadas. Na subseção 5.3 é apresentado como se deu a execução do experimento. Por fim na subseção 5.4 são apresentados os resultados e discussão do experimento.

5.2 Definição do Experimento

Para definição do experimento foi utilizada a GQM (*Goal Question Metric*), que é uma abordagem orientada a objetivos proposta por Victor R. Basili entre os anos 70 e 80, e vem sendo empregada em diversas medições de produtos e processos. Tem sido útil para diversas finalidades, dentre elas para aferir abordagens de engenharia de software (SHULL et al., 2006).

Segundo Solingen e Berghout (1999), com o GQM consegue-se estabelecer um sistema de medição direcionado a metas onde após definir estas metas, são levantadas as questões acerca delas e em seguida são identificadas as métricas necessárias para responder estas questões. Portanto o GQM está divido em três níveis:

- Nível Conceitual (*Goal*): quais são as metas/objetivos?
- Nível Operacional (*Question*): quais questões se deseja responder?
- Nível Quantitativo (*Metric*): quais métricas poderão responder as questões?

Tais níveis são descritos nas subseções a seguir.

5.2.1 Definição das Metas

Conforme previsto no GQM, foi feita uma listagem dos objetivos do processo de medição definindo suas metas através das seguintes dimensões:

- Objeto de Estudo: o que será analisado?
- Objetivo: Porque o objeto será analisado?
- Enfoque de Qualidade: Qual atributo do objeto será analisado?
- Ponto de Vista: Quem irá usar os dados coletados?
- Contexto: Em qual ambiente está localizado?

A experimentação da abordagem MDGD tem como principal objetivo avaliar a ferramenta jMEGenerator quanto a eficiência, flexibilidade e aprendizado que essa abordagem oferece em projetos de jogos eletrônicos. Portanto a meta (G1) conceitual do estudo é a seguinte:

- Objeto de Estudo: Analisar a abordagem MDGD para o desenvolvimento de jogos eletrônicos.

- Objetivo: Com o propósito de avaliação da ferramenta jMEGenerator quanto a sua aplicação no desenvolvimento de jogos eletrônicos;
- Enfoque de Qualidade: Com respeito à flexibilidade oferecida pela ferramenta jMEGenerator. Também diz respeito à eficiência em termos de tempo gasto e produtividade para o atendimento correto dos requisitos, além do aprendizado ofertado pela abordagem MDGD em projetos de jogos eletrônicos.
- Ponto de Vista: Do ponto de vista de desenvolvedores de software;
- Contexto: No contexto de estudantes de graduação em Sistemas de Informação e Ciência da Computação que já cursaram a disciplina de Programação Orientada a Objetos.

5.2.2 Questões

A partir da meta definida no primeiro nível do GQM, pode-se levantar quais são as questões acerca dos objetivos almejados. As questões que serão analisadas são as seguintes:

- Q1: Consegue-se ter flexibilidade em projetos de jogos com MDGD?
- Q2: Há maior eficiência no desenvolvimento de jogos utilizando a abordagem MDGD?
- Q3: A abordagem MDGD propicia o aprendizado da tecnologia?

5.2.3 Métricas

Métricas são medições utilizadas para calcular dados a fim de responder uma determinada questão. A primeira questão (Q1) avalia a flexibilidade em projetos de jogos com MDGD. Projetos podem ser considerados flexíveis quando permitem alterações não previstas no seu escopo inicial. Conforme descrito anteriormente, projetos de jogos têm uma forte tendência em fugir do escopo, devido à sua forma muitas vezes imprevisível de ser. Portanto para avaliar essa questão, uma métrica relevante é verificar se os participantes conseguem realizar corretamente alterações no jogo eletrônico, especialmente aquelas que não estavam previstas inicialmente no projeto. As alterações não previstas pelos geradores devem ser feitas a partir da inserção de códigos manuais ao projeto.

A segunda questão (Q2) trata sobre a eficiência no desenvolvimento de jogos utilizando a abordagem MDGD. Entende-se por eficiência a capacidade de atingir um resultado utilizando o mínimo de recursos possível. No caso dessa questão, o recurso gasto no desenvolvimento de

jogos é o tempo do desenvolvedor. Portanto a eficiência pode ser mensurada através do tempo despendido para realização das atividades do experimento.

Usar DSLs ao invés de programar manualmente com linguagens de propósito geral é mais simples, portanto mais fácil de aprender. Por outro lado, caso o desenvolvedor precise alterar o código diretamente, pode ser que ele, por ter usado geradores, não tenha aprendido o suficiente para conseguir efetuar tais alterações. Dessa forma surge a terceira questão (Q3), que procura responder se a abordagem MDGD prejudica ou não o aprendizado para o desenvolvimento de jogos. Portanto a métrica para essa questão será analisar se o participante do experimento consegue aprender a usar o jMEGenerator facilmente, mas também verificar se ele é capaz de entender o código produzido pelos geradores a ponto de ser de capaz de inserir códigos manualmente, ficando assim livre das limitações impostas pela abordagem MDGD. Nessa métrica deverá ser feita uma análise do tempo gasto tanto nas tarefas previstas pelos geradores, quanto nas tarefas não previstas. Por fim, para analisar o aprendizado deve-se levar em consideração o conhecimento prévio dos participantes. Essa análise pode ser realizada através dos formulários de caracterização dos participantes.

De acordo com a análise apresentada, pode-se levantar as seguintes métricas de avaliação:

- M1: Correto atendimento aos requisitos do jogo e também dos requisitos de alterações no projeto;
- M2: Tempo gasto em tarefas previstas pelos geradores do código;
- M3: Tempo gasto em tarefas não previstas pelos geradores do código;
- M4: Conhecimento prévio dos participantes;

Para melhor compreensão dos itens levantados através da abordagem GQM, pode-se organizá-los através de uma estrutura hierárquica, conforme visto na figura 5.1

5.3 Planejamento do Experimento

Num primeiro momento, foi feita a seleção do contexto no qual o experimento deveria ser conduzido. O experimento foi realizado com estudantes de graduação dos cursos de Sistemas de Informação e Ciência da Computação que já tivessem sido aprovados na disciplina de Programação Orientada a Objetos. Tal critério foi definido assim, pois deseja-se avaliar a abordagem MDGD sob a ótica de desenvolvedores que não tenham conhecimento do motor de

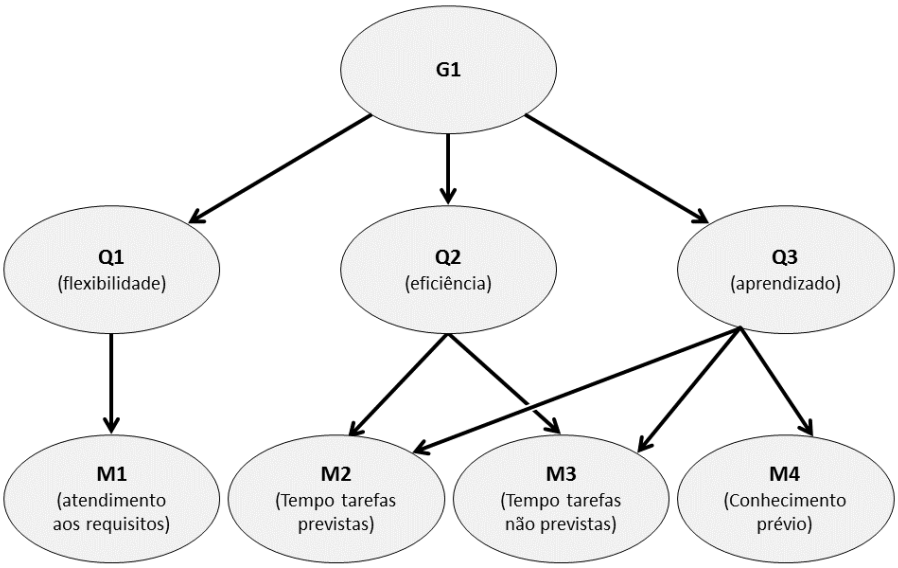

Figura 5.1: Estrutura Hierárquica do GQM

jogo proposto, porém que já posssuam conhecimento do paradigma de programação utilizado e da linguagem de programação Java. Foi criado então um formulário digital para caracterização dos participante através da ferramenta Google Docs [1] contendo as seguintes perguntas:

1. Nome.
2. Email.
3. Curso.
4. Já concluiu o curso de graduação? (Sim / Não)
5. Já cursou e foi aprovado na disciplina de Programação Orientada a Objetos? (Sim / Não)
6. Já desenvolveu algum jogo eletrônico antes? (Sim / Não)
7. Qual é seu nível de conhecimento na linguagem Java? (escala de 1 à 5)
8. Qual é seu nível de conhecimento sobre a abordagem MDD? (escala de 1 à 5)
9. Qual é seu nível de conhecimento sobre desenvolvimento de Jogos 3d? (escala de 1 à 5)

Após explicar sobre a proposta do experimento para diversos alunos, um total de 52 candidatos inscreveram-se através do formulário e prontificaram-se a colaborar com a pesquisa

[1] http://docs.google.com/forms/

realizando as tarefas do experimento. As perguntas 3, 4 e 5 foram utilizadas para selecionar os participantes, já que poderiam participar do experimento apenas estudantes dos cursos de Sistemas de Informação ou Ciência da Computação que já tivessem sido aprovados na disciplina de Programação Orientada a Objetos. Todos os 52 candidatos atenderam a esse requisito, mantendo o mesmo número de participantes. A participação do experimento foi totalmente voluntária, aproveitando da motivação natural que os estudantes normalmente tem por jogos.

Dentre esses participantes, foram separados dois grupos distintos. O primeiro grupo (Grupo A) deveria realizar todas as tarefas sem utilização da abordagem MDGD, codificando todo projeto diretamente no motor de jogo com a linguagem de programação Java. O segundo grupo (Grupo B) deveria realizar as mesmas tarefas utilizando a abordagem MDGD, através dos seus modelos, *templates* e geradores de código. A partir das questões 6, 7, 8 e 9 do formulário de inscrição, foi possível normalizar os grupos para que houvesse um balanceamento em relação ao conhecimento prévio dos participantes, formando assim dois grupos homogêneos com 26 participantes cada. A separação dos grupos e as respostas do formulário de caracterização do participante podem ser visualizadas no Apêndice A.

Com o objetivo de levantar as variáveis que atendam as métricas definidas na subseção anterior, foram definidas quatro tarefas para que cada participante do experimento pudesse realizar.

A primeira tarefa foi a de construir um jogo de corrida, onde o personagem jogável é um veículo. O jogador deve ter capacidade de efetuar as ações de acelerar, frear, virar para a direita e virar para a esquerda. O veículo deve ter como característica seu peso de 1.200 kg, aceleração máxima de 1400 e velocidade máxima de 200 km/h. A câmera deverá estar fixa no veículo, estando posicionado atrás deste. O cenário deve ter formato circular, composto por elementos conforme mostra na figura 5.2.

A primeira tarefa tem um cenário simples propositalmente para facilitar o aprendizado do desenvolvedor. Já a segunda tarefa envolve uma manutenção do jogo onde o participante deve alterar o cenário utilizando elementos mais complexos. O cenário desejado para a segunda tarefa está representado na figura 5.3 e é composto por elementos como casas, ponte e árvores.

A terceira tarefa também é uma manutenção no jogo, onde o desenvolvedor deverá trocar o personagem jogável que antes era um veículo para um personagem humanoide. As ações que o jogador poderá efetuar são andar para frente, andar para trás, girar para a direita, girar para esquerda e pular.

As três tarefas definidas anteriormente envolvem atividades que haviam sido previstas pelos geradores, fazendo com que o Grupo B, que desenvolveu as tarefas sob a abordagem MDGD,

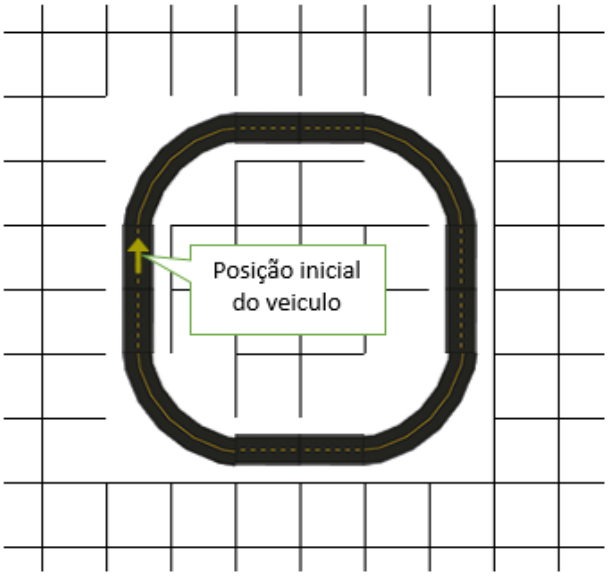

Figura 5.2: Cenário da Tarefa 1

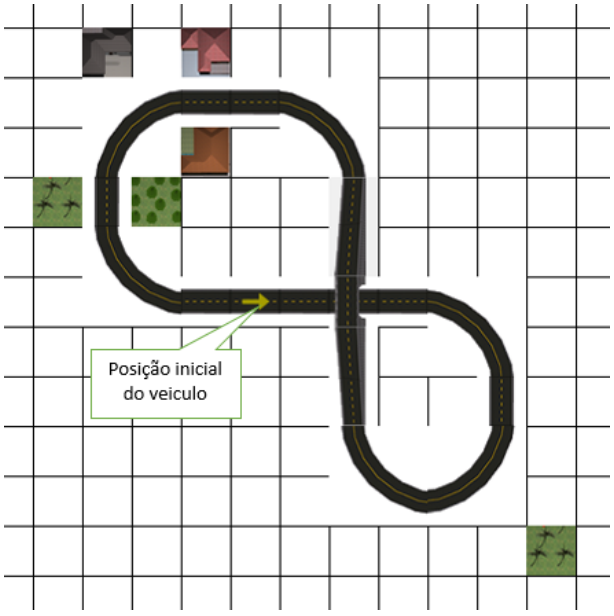

Figura 5.3: Cenário da Tarefa 2

obviamente encontraria maior facilidade. Portanto a quarta tarefa trata-se de uma manutenção no jogo que não foi prevista pelos geradores de código. A tarefa em questão pede para que o desenvolvedor permita a coexistência de dois personagens jogáveis. O jogo deveria iniciar com o personagem humanoide como sendo jogável, enquanto o veículo estaria posicionado ao lado sem sofrer nenhuma ação do jogador. Quando o jogador pressionar a tecla 'X' o personagem jogável deve ser trocado para o veículo, passando as ações de teclado e o controle da câmera para este novo personagem. Essa troca somente pode ocorrer se o personagem humanoide estiver a menos de 10 metros de distância do veículo. Nesse evento, além do personagem humanoide perder suas ações e seu vinculo com a câmera, ele deve desaparecer do cenário, como se estivesse entrado dentro do veículo. Quando o personagem corrente for o veículo, a troca para o personagem humanoide poderá ser feita em qualquer posição em que o veículo esteja, bastando o usuário pressionar novamente a tecla 'X'. Ao se efetuar essa troca, o personagem humanoide deve ser recolocado no cenário e reposicionado ao lado do veículo, além é claro, de passar a receber as ações do jogador e ter a câmera vinculada a ele.

Através dessas quatro tarefas executadas pelos dois grupos, onde cada grupo utiliza uma abordagem de desenvolvimento diferente, foram levantados os dados necessários para atingir as métricas e responder as questões do estudo.

5.4 Execução do experimento

Após a preenchimento do formulário de inscrição pelos candidatos e separação dos grupos, foram preparados e enviados os materiais de instrumentação necessários para a realização das tarefas. Todas as tarefas puderam ser feitas em casa pelos participantes.

Para a realização das tarefas foram elaborados dois tutoriais que servem de guia passo a passo para realizar todas as atividades do experimento. Um tutorial explica sobre o motor de jogo jMonkeyEngine e pode ser visto no Apêndice B. O outro tutorial trata sobre o JMEGenerator, que é a ferramenta descrita no capítulo 4 desta dissertação, e está disponível no Apêndice C. Os participantes do grupo A, que realizaram as tarefas manualmente receberam apenas o tutorial do jMonkeyEngine, enquanto os participantes do grupo B, que realizaram as tarefas com a abordagem MDD receberam ambos tutoriais. Todos os participantes receberam o tutorial do jMonkeyEngine pois a quarta tarefa teve que ser codificada manualmente independente da abordagem de desenvolvimento adotada. Outra característica que difere a quarta tarefa das demais é que seu desenvolvimento não está explicito no tutorial, necessitando que o desenvolvedor compreenda uma séria de conceitos do motor de jogo para sua codificação.

Além dos tutoriais, foram fornecidos um guia de orientações gerais do experimento e uma descrição detalhada sobre cada uma das tarefas a serem realizadas, que estão disponíveis respectivamente no Apêndice D e Apêndice E.

Ao final de cada tarefa do experimento, foi solicitado para os desenvolvedores que respondessem a um questionário com o intuito de avaliar a ferramenta e corrigir possíveis problemas nos tutoriais. Nesse questionário foram feitas as seguintes perguntas:

1. Número da Tarefa.
2. Os tutoriais fornecidos foram suficientes? (Sim / Não)
3. Qual nível de dificuldade você classifica esta tarefa? (escala de 1 à 5)
4. Relate como foi sua experiência nesta tarefa.
5. Indique sugestões ou melhorias referentes às ferramentas utilizadas nesta tarefa.

Uma informação importante para as métricas definidas é o tempo que o participante levou para realizar cada uma das tarefas. Como todas as tarefas foram realizadas em casa, o pesquisador criou e disponibilizou uma aplicação web para calcular e armazenar o tempo gasto em cada tarefa. Essa aplicação foi nomeada como TrialClock [2] e sua interface pode ser vista na Figura 5.4. Conforme pode ser visto na imagem, o participante seleciona a tarefa que ele deseja realizar para o experimento e indica a hora de inicio e a hora de fim, repetindo esse processo quantas vezes for necessário. Ao final da página é mostrado o valor total do tempo gasto naquela tarefa. Todas informações inseridas na aplicação TrialClock são armazenadas em um banco de dados para que o pesquisador possa consultá-las em tempo real.

Para evitar falhas na aplicação do experimento, foi solicitado a um participante que realizasse todas as tarefas antes dos outros participantes. Caso ele não conseguisse entender e realizar alguma tarefa, o material de instrumentação deveria ser revisto e as informações desse participante seriam descartadas. No entanto, esse primeiro experimento ocorreu com sucesso, onde o participante entregou o resultado das tarefas no decorrer de dez dias. Desse modo foi estipulado um prazo de três semanas para que os demais participantes realizem seus experimentos e enviassem os dados solicitados para análise e interpretação. Considerando o tempo gasto pelo primeiro participante, a margem de prazo de três semanas foi considerada suficiente para a realização das quatro tarefas.

[2] http://trialclock.appspot.com/

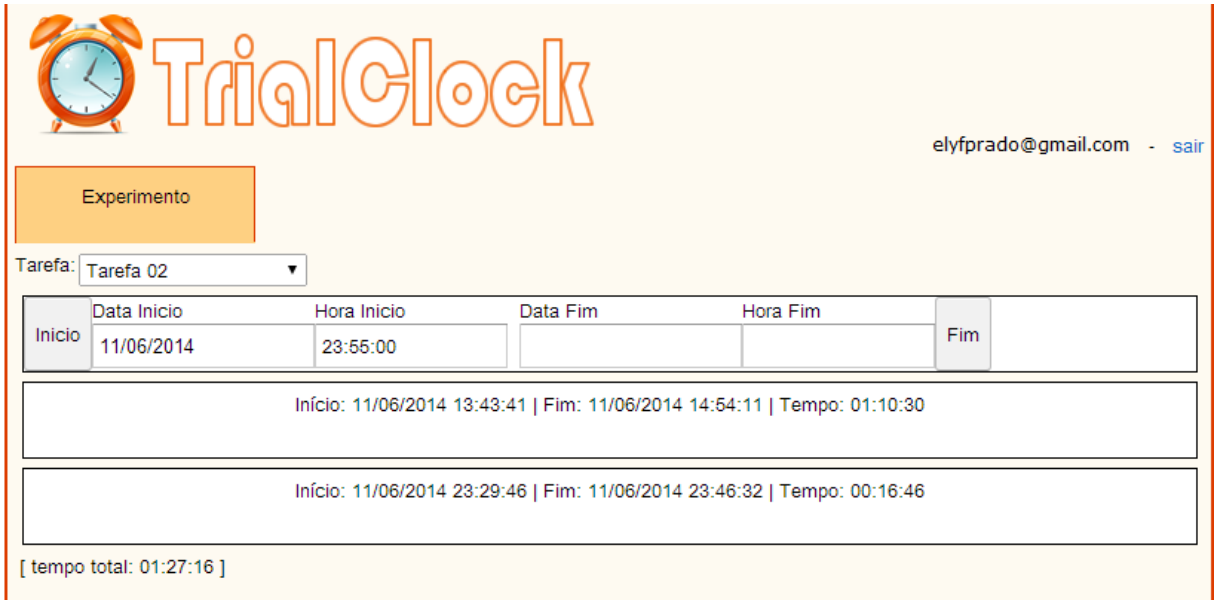

Figura 5.4: Aplicação para medição de tempo

5.5 Resultados e discussão

No prazo estipulado, os participantes enviaram os arquivos de código fonte das suas quatro tarefas, bem como informações sobre o tempo o qual gastaram em cada uma das tarefas. Porém nem todos os participantes realizaram o experimento e enviaram suas respostas. Dos 52 participantes inscritos, apenas 12 realizaram as atividades do experimento, sendo 6 do Grupo A e 6 do Grupo B. Esse baixo número de concluintes explica-se provavelmente pelo fato de que todos os participantes eram voluntários e não tinham nenhum outro incentivo além do ganho de conhecimento da tecnologia. Portanto, os resultados a serem interpretados serão em relação aos dados dos 12 participantes que entregaram as quatro tarefas do experimento.

A partir dos arquivos de código fonte e informações enviadas pelo aplicativo TrialClock, juntamente com as respostas do formulário de caracterização do participante foi possível levantar as informações necessárias para atender às métricas do estudo. O resultado do experimento seguido pela discussão dos mesmos são apresentados nas subseções a seguir.

5.5.1 Resultados

A primeira métrica levantada nas definições do GQM (M1) refere-se ao correto atendimento dos requisitos do jogo e também dos requisitos de alterações no projeto. Portanto foi feita uma análise nos projetos enviados pelos participantes, verificando se os mesmos atendiam corretamente as funcionalidades desejadas em cada uma das quatro tarefas do experimento. Essa análise foi feita pelo pesquisador através da execução do código fonte enviado pelos participan-

tes. A partir dessa análise foi possível levantar os dados mostrados na Tabela 5.1. Essa tabela mostra uma relação dos doze participantes separando-os pelo seu grupo e a informação do atendimento correto ou não dos requisitos em cada uma das quatro tarefas. O não atendimento dos requisitos referiu-se no grupo A em dificuldades no posicionamento dos elementos do cenário no ambiente tridimensional. Em ambos os grupos ocorreu também casos de não atendimento do requisito de troca de personagem, onde os participantes apresentaram dificuldades em tratar as entradas do usuário para os diferentes personagens e também de anexar a câmera ao novo personagem.

Tabela 5.1: Atendimento aos Requisitos

Grupo A - Abordagem Tradicional

Num	Tarefa 1	Tarefa 2	Tarefa 3	Tarefa 4
1	Sim	Sim	Sim	Sim
2	Sim	Não	Não	Não
3	Sim	Sim	Sim	Não
4	Sim	Sim	Sim	Não
5	Sim	Sim	Sim	Sim
6	Sim	Sim	Sim	Sim
Corretos	**6**	**5**	**5**	**3**

Grupo B - Abordagem MDGD

Num	Tarefa 1	Tarefa 2	Tarefa 3	Tarefa 4
7	Sim	Sim	Sim	Sim
8	Sim	Sim	Sim	Não
9	Sim	Sim	Sim	Sim
10	Sim	Sim	Sim	Não
11	Sim	Sim	Sim	Não
12	Sim	Sim	Sim	Não
Corretos	**6**	**6**	**6**	**2**

A segunda métrica (M2) refere-se o tempo gasto em tarefas previstas pelos geradores do código, que são as tarefas 1, 2 e 3. Uma relação detalhada sobre o tempo em que os participantes gastaram para executar cada tarefa do experimento encontra-se na Tabela 5.2.

A quarta tarefa não estava prevista nos geradores de código do jMEGenerator, sendo necessário ser codificada manualmente no projeto de jogo. O tempo de execução dessa tarefa atendeu à terceira métrica (M3) do estudo, que analisou o tempo gasto em tarefas não previstas pelos geradores do código. A lista do tempo gasto nessa tarefa pode ser visto na Tabela 5.3.

A quarta métrica (M4) do estudo pede informações sobre o conhecimento prévio dos participantes. Esse conhecimento pode ser mensurado através da análise das respostas do formulário de caracterização do participante. Vale ressaltar que foram considerados apenas os doze can-

Tabela 5.2: Tempo para execução das tarefas previstas

Grupo A - Abordagem Tradicional

Num	Tarefa 1	Tarefa 2	Tarefa 3	Tempo Total
1	2:32:10	0:53:44	0:30:47	3:56:41
2	3:39:00	1:47:00	0:30:00	5:56:00
3	1:42:11	1:27:16	0:17:18	3:26:45
4	5:18:00	2:26:00	3:51:00	11:35:00
5	13:25:50	8:06:05	1:22:15	22:54:10
6	4:45:41	2:03:13	0:23:16	7:12:10
Média	**5:13:49**	**2:47:13**	**1:09:06**	**9:10:08**

Grupo B - Abordagem MDGD

Num	Tarefa 1	Tarefa 2	Tarefa 3	Tempo Total
7	0:36:00	0:16:00	0:07:00	0:59:00
8	0:03:42	0:05:02	0:01:49	0:10:33
9	0:09:02	0:04:13	0:01:40	0:14:55
10	0:26:21	0:11:22	0:07:37	0:45:20
11	0:19:16	0:04:53	0:01:32	0:25:41
12	0:55:00	0:12:00	0:06:00	1:13:00
Média	**0:24:54**	**0:08:55**	**0:04:16**	**0:38:05**

Tabela 5.3: Tempo para execução das tarefas não previstas

Grupo A - Abordagem Tradicional

Num	Tarefa 4
1	0:52:01
2	0:46:00
3	1:37:20
4	1:32:00
5	3:33:19
6	2:47:34
Média	**1:51:22**

Grupo B - Abordagem MDGD

Num	Tarefa 4
7	3:39:00
8	2:35:04
9	6:00:00
10	1:26:06
11	2:45:00
12	2:28:00
Média	**3:08:52**

didatos que conseguiram entregar as tarefas do experimento. O perfil de caracterização desses candidatos pode ser visto na Tabela 5.4. O valor de média de conhecimento de cada grupo foi calculado a partir da média aritmética entre o nível de conhecimento em Java, MDD e Jogos 3d dos participantes.

Tabela 5.4: Caracterização dos Participantes Concluintes

Grupo A - Abordagem Tradicional

Num.	Curso	Graduado	Aprovado em POO	Já desenvolveu algum jogo?	Nível de conhecimento em Java	Nível de conhecimento em MDD	Nível de conhecimento em Jogos 3d
1	CC	Não	Sim	Sim	4	2	3
2	CC	Não	Sim	Sim	3	0	0
3	CC	Não	Sim	Sim	3	1	3
4	CC	Não	Sim	Sim	3	0	2
5	CC	Não	Sim	Sim	3	0	1
6	SI	Sim	Sim	Sim	3	0	2
						Média:	1,83

Grupo B - Abordagem MDGD

Num.	Curso	Graduado	Aprovado em POO	Já desenvolveu algum jogo?	Nível de Conhecimento em Java	Nível de Conhecimento em MDD	Nível de Conhecimento em Jogos 3d
7	SI	Não	Sim	Sim	3	0	1
8	CC	Não	Sim	Sim	3	0	0
9	CC	Não	Sim	Sim	3	0	1
10	CC	Não	Sim	Sim	4	0	0
11	CC	Não	Sim	Não	4	0	2
12	CC	Não	Sim	Não	3	0	1
						Média:	1,39

* SI = Sistemas de Informação
* CC = Ciência da Computação

5.5.2 Discussão

Com todas as métricas em mãos, foi possível responder as questões levantadas na abordagem GQM. A primeira questão (Q1) a ser respondida diz respeito a existência de flexibilidade em projetos de jogos com MDGD. Conforme apresentado na métrica M1, houve participantes pertencentes tanto do Grupo A como também do Grupo B que conseguiram atender corretamente todos os requisitos estipulados nas tarefas do experimento. No Grupo A, onde os participantes utilizaram a abordagem tradicional de desenvolvimento, 79% das tarefas foram feitas corretamente. No Grupo B, onde os participantes utilizaram a abordagem MDGD com o jMEGenerator, houve um índice de 83% de tarefas realizadas de forma correta. Isso indica que foi possível obter flexibilidade no desenvolvimento de jogos com MDGD, podendo alterar o escopo do jogo inclusive com características não esperadas pelos geradores.

A segunda questão a ser analisada é sobre a eficiência no desenvolvimento de jogos utilizando a abordagem MDGD. Analisando as métricas M2 e M3 que tratam respectivamente do tempo gasto para execução das tarefas previstas e das tarefas não previstas pelos geradores, percebe-se facilmente que houve maior eficiência no desenvolvimento do jogo eletrônico utilizando a abordagem MDGD. O grupo B conseguiu executar todas as quatro tarefas do experimento com média de 34,3% do tempo médio gasto pelo Grupo A. É notado uma eficiência da abordagem MDGD ainda maior quando se analisa apenas as tarefas previstas pelos geradores, onde o Grupo B conseguiu executar tais tarefas com 6,9% do tempo médio gasto pelos participantes do Grupo A, porém é importante considerar a tarefa não prevista para manter a flexibilidade do projeto.

Pode-se questionar a validade do estudo de eficiência da abordagem MDGD, pois encontramos participantes que gastaram um tempo muito diferente que a maioria do grupo. Por exemplo, o participante 5 do Grupo A demorou mais de 26 horas para execução do experimento das 4 tarefas, enquanto a média de seu grupo foi em torno de 11 horas. Portanto analisando os mesmos dados, porém descartando o participante que levou mais tempo e também o participante que levou menos tempo em cada grupo, temos a média de tempo total gasto em todas tarefas de 8:43:12 para o Grupo A e 3:33:50 para o Grupo B. Sendo assim, mesmo com o descarte dos valores extremos, os participantes que utilizaram a abordagem MDGD em média gastaram apenas 40,9% do tempo que os participantes que desenvolveram as atividades de maneira tradicional.

A terceira questão levantada (Q3) está relacionada ao fator de aprendizado que a abordagem MDGD pode oferecer a respeito da tecnologia utilizada. Essa questão utiliza como métrica as informações quanto ao tempo gasto nas atividades previstas (M2) e não previstas pelos geradores (M3) e também a avaliação do conhecimento prévio dos candidatos (M3). Na Figura 5.5 é mostrado um gráfico que demonstra a média de tempo gasto pelos participantes levando em consideração os dados as métricas M2 e M3. Nessa análise percebe-se que os participantes que utilizaram a abordagem tradicional demoraram maior tempo na execução da primeira tarefa, mas foram reduzindo esse tempo a cada tarefa executada. Já os participantes que utilizaram a abordagem MDGD gastaram um tempo bastante pequeno nas atividades previstas pelos geradores, mas demoraram mais para executar a tarefa não prevista pelo gerador. Essa demora maior na execução da última tarefa pelo Grupo B deve-se pelo fato de que os participantes não tinham total conhecimento do código que foi gerado, e no momento em que foi necessário inserir código manual houve um esforço maior para compreender o código. Esse esforço para compreensão do código já existente foi bastante reduzido para os participantes do Grupo A, já que eles tiveram que codificar manualmente todas as tarefas. Por isso, o Grupo A conseguir executar a Tarefa 4 com 59% do tempo gasto pelos participantes do Grupo B. Portanto, há indícios de que

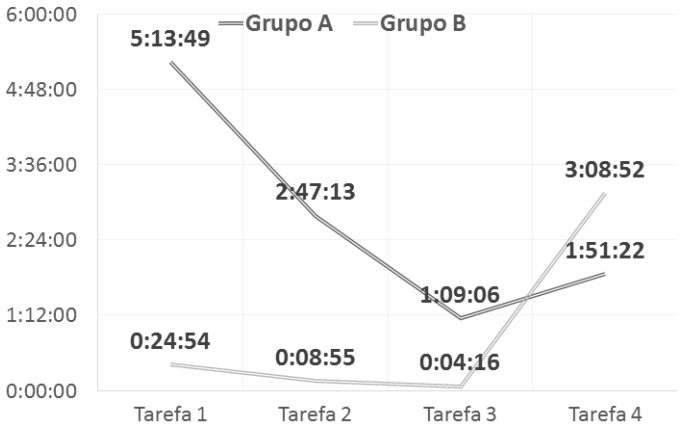

Figura 5.5: Tempo para execução das tarefas

a abordagem MDGD prejudica o aprendizado ao que se refere à codificação manual do projeto, já que os geradores de código automatizam boa parte da codificação.

Quanto a métrica do conhecimento prévio dos participantes (M4), é interessante fazer uma comparação entre as 52 pessoas inscritas para o experimento e as 12 pessoas que realizaram de fato o experimento. Acontece que o conhecimento prévio dos participantes dos grupos estavam balanceados no momento da divisão, tendo 50% dos candidatos com alguma experiência no desenvolvimento de jogos e 50% sem nenhuma experiência no desenvolvimento de jogos em ambos os grupos. A mesma proporção foi aplicada tanto no Grupo A quanto no Grupo B, no entanto dentre os participantes que realizaram o experimento a proporção ficou diferente. No Grupo A, onde os participantes teriam que desenvolver o código manualmente sem a ajuda dos geradores, apenas pessoas que possuíam previamente alguma experiência com desenvolvimento de jogos eletrônicos conseguiram concluir o experimento. No entanto, no Grupo B onde os participantes desenvolveram as atividades seguindo a abordagem MDGD, 33% dos que concluíram o experimento não possuíam nenhuma experiência prévia no desenvolvimento de jogos eletrônicos. Esses números indicam uma tendencia de que a abordagem MDGD permite que pessoas novatas na área de desenvolvimento de jogos eletrônicos consigam criar seus próprios jogos. Tal informação é reforçada ao analisarmos a média de conhecimento prévio dos participantes sobre a linguagem Java, MDD e jogos 3D. Conforme mostrado no Apêndice A, os dois grupos tinham inicialmente conhecimento médio equivalente, com valor de 1,44. No entando os participantes que conseguiram concluir o experimento do Grupo A apresentaram um conhecimento médio maior, no valor de 1,83. Já o Grupo B apresentou uma média de conhecimento menor, sendo o valor de 1,39. Isso indica que com o MDGD não há necessidade prévia de muito conhecimento em Java, MDD e jogos tridimensionais. Tal análise pode ser melhor visualizada na Figura 5.6.

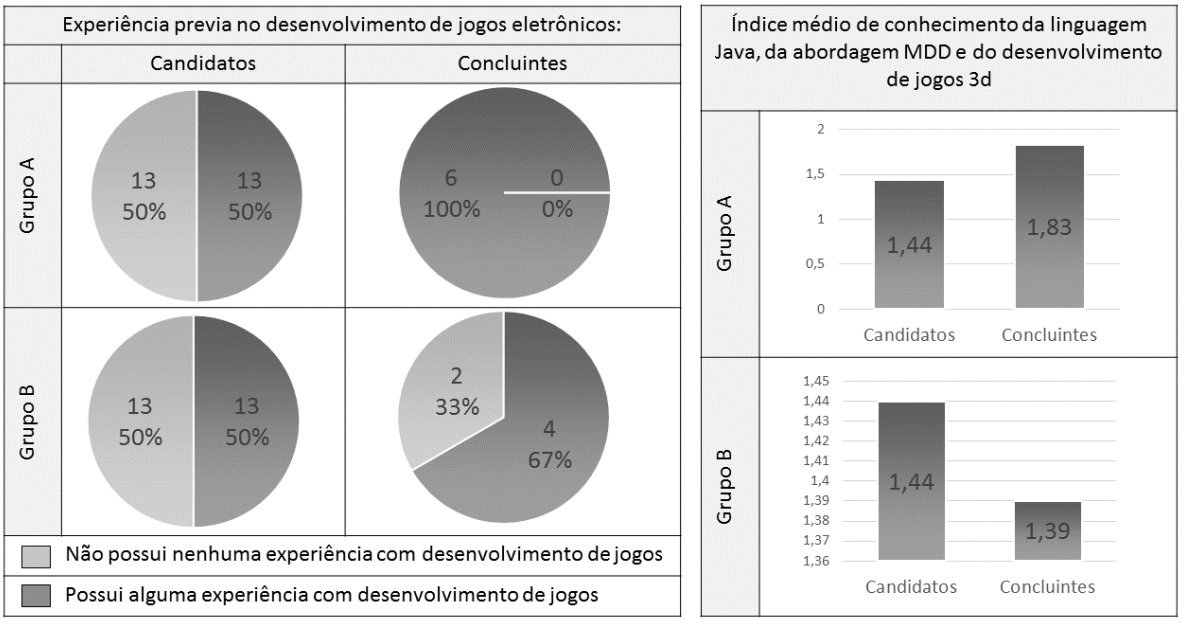

Figura 5.6: Análise de candidatos inscritos e concluintes

A partir das métricas levantadas pode-se afirmar que a abordagem MDGD traz uma menor curva de aprendizado para a criação de jogos, facilitando sua adesão até mesmo para desenvolvedores com pouco conhecimento na área. O MDGD abstrai do desenvolvedor detalhes técnicos do motor de jogo automatizando processos e agilizando o desenvolvimento, porém essa abstração inibe em parte o aprendizado da tecnologia, fazendo com que o desenvolvedor tenha dificuldades em entender e alterar o código manualmente.

5.6 Ameaças à Validade do Experimento

Determinar a validade de um experimento é um item que não deve ser ignorado quando é feita uma análise de seus resultados. Portanto deve ser feito um estudo sobre as ameaças que experimento pode estar exporto. Um experimento tem sua validade adequada quando os resultados são válidos para a população de interesse do estudo (WOHLIN et al., 2000).

Wohlin et al. (2000) classifica as ameaças que podem colocar em risco a validade de um quatro tipos: validade de conclusão, validade interna, validade de construção e a validade externa. Essas ameaças foram analisadas no experimento realizado, conforme mostrado a seguir:

- Validade de Conclusão: trata de questões estatísticas do experimento ou suposições erradas sobre os dados. Como a amostra de participantes que concluíram o experimento foi relativamente pequena, as conclusões foram analisadas facilmente. Porém essa mesma

característica pode trazer limitações quanto a validade de conclusão, já que a amostra menor leva a uma maior frequência de dados oriundos de simples coincidências. Uma amostra maior poderia proporcionar estudos estatísticos com dados mais seguros e livres dessas coincidências. Contudo, ao analisar os dados não ouve nenhuma discrepância que pudesse levar a conclusões distorcidas. No caso das métricas M2 e M3 que analisaram o tempo de desenvolvimento, foi feito um estudo comparativo descartando os valores extremos a fim de validar o experimento. De fato tal medida comprovou que as conclusões estavam corretas, já que os resultados mantiveram-se equivalentes mesmo com a mudança nos dados analisados.

- Validade Interna: refere-se à verificação de diferenças nos resultados por consequência de uma medição incorreta ou outros fatores relacionados aos participantes e seus comportamentos durante o experimento. Para amenizar tal ameaça foi solicitado que os participantes não comunicassem entre si sobre o experimento e muito menos trocassem experiências sobre as ferramentas utilizadas para evitar a geração de falsos resultados. Além disso, foi feita uma seleção dos participantes onde apenas estudantes que já tivessem concluído a disciplina de Programação Orientada a Objetos pudessem participar do experimento. Esse critério fez com que os candidatos apresentassem uma certa equivalência nos seus níveis de conhecimento e experiência. Também não foi oferecido nenhum tipo de recompensa aos participantes de modo a não criar expectativas para que eles se comportassem com empenho anormal durante o estudo. Por fim, a partir do formulário de caracterização dos participantes, foi feita a divisão dos grupo da forma mais homogênea possível para inibir as ameaças internas.

- Validade de Construção: está ligada às questões de má definição da base teórica, má definição do processo de experimentação ou problemas de fatores humanos e sociais. Para inibir essa ameaça foram selecionados participantes que tinham pouco ou nenhum conhecimento sobre as ferramentas que iriam trabalhar. Todos os participantes desenvolveram as tarefas utilizando os tutoriais fornecidos pelo pesquisador. Os materiais de instrumentação foram os mesmos, o perfil dos participantes era bastante similar e todos eles tiveram as mesmas condições para realização das tarefas. Sendo assim, os dados para coleta foram definidos de modo a garantir conformidade com os objetivos do estudo. Contudo pode-se considerar uma ameaça à validade de construção do experimento o fato dos grupos que enviaram suas respostas terem ficado desbalanceados quanto ao conhecimento prévio médio. Esse desbalanceamento veio de uma seleção natural e foi tradado como indicador de que o MDGD proporciona melhor adesão por participantes que têm menor grau de conhecimento da tecnologia e também por pessoas inexperientes no de-

senvolvimento de jogos. Porém não se pode negar que esse desbalanceamento pode ter trazido ameaças aos demais resultados.

- Validade Externa: refere-se às questões que afetam a habilidade de generalizar os resultados de um experimento para a prática industrial. Procurando reduzir essas ameaças foram utilizados para o experimento ferramentas que comumente são usadas pela indústria dos jogos eletrônicos, como por exemplo os motores de jogos. Também realizou-se um estudo para que as atividades realizadas no experimento fossem inspiradas em jogos reais que estão disponíveis no mercado. Contudo existe uma limitação nesse experimento que pode ser considerado como uma ameaça externa, que é o fato dele ter sido aplicado apenas no meio acadêmico, sendo que todos os participantes são estudantes de graduação. O perfil dos participantes foi escolhido assim para que todas as pessoas tivessem o grau de conhecimento semelhante. Dessa forma justifica-se, em projetos futuros, que seja replicado o mesmo experimento com profissionais experientes no ramo de jogos eletrônicos para validar se serão encontrados os mesmos resultados.

Capítulo 6

CONCLUSÃO

6.1 Contribuições e Síntese dos Principais Resultados

As pesquisas relacionadas ao processo de desenvolvimento de jogos eletrônicos têm mostrado que esse setor está em crescente expansão, porém há uma carência de métodos e abordagens que facilitem a criação de novos produtos. Um jogo eletrônico, apesar de possuir suas particularidades, ainda é um software e portanto na sua criação existem muitas características similares às encontradas no desenvolvimento de outros tipos de software. O MDD tem sido apresentado como boa solução para aumentar a eficiência e qualidade no desenvolvimento de software de forma geral. No entanto existem incertezas se a mesma abordagem pode ser vantajosa para o desenvolvimento de jogos eletrônicos, já que eles normalmente possuem um escopo bastante aberto e variável, fazendo com que projetos de jogos necessitem ter um bom nível de flexibilidade.

A principal contribuição deste trabalho de pesquisa foi a investigação quanto à flexibilidade alcançada em projetos de jogos eletrônicos utilizando a abordagem dirigida a modelos (MDGD) juntamente com a utilização de padrões de projetos e inserção de códigos manuais. Além disso foram analisadas questões sobre eficiência e aprendizado na utilização dessa abordagem de desenvolvimento em projetos de jogos.

Diante disso, neste trabalho foi desenvolvida uma ferramenta chamada jMEGenerator a fim de analisar o comportamento da abordagem MDGD sob a ótica de desenvolvedores que não possuíam experiência prévia no desenvolvimento de jogos com tal abordagem.

A partir de um experimento realizado com participantes voluntários, conseguiu-se analisar informações de que é possível obter flexibilidade em projetos de jogos eletrônicos utilizando a abordagem MDGD. Para tanto deve-se permitir a inserção de códigos manuais através de

padrões de projeto bem definidos em projetos organizados. Além disso, o MDGD provou ser capaz de trazer maior eficiência quando comparado à abordagem tradicional de desenvolvimento de jogos. Em contrapartida, o MDGD mostrou que por facilitar a codificação, ele prejudica o aprendizado dos códigos gerados, já que o desenvolvedor trabalha com um nível maior de abstração.

Pode-se identificar através dos estudos levantados que profissionais que já possuem conhecimento prévio sobre o motor de jogo utilizado terão muitas vantagens em utilizar a abordagem MDGD. Isso deve-se ao fato de que com o MDGD consegue-se maior produtividade nas partes do projeto já esperadas pelos transformadores e no caso de mudanças não esperadas é possível inserir códigos manualmente, mantendo assim a mesma flexibilidade que se tem em projetos de jogos que não utilizam a abordagem MDGD no desenvolvimento.

Tais resultados não podem ser considerados totalmente conclusivos, necessitando ainda novas e diferentes experimentações para que haja a certeza de que outros fatores desconhecimentos não influenciaram os resultados.

6.2 Trabalhos Futuros

Existem trabalhos que podem estender as pesquisas realizadas nesta dissertação. Como oportunidades de melhoria, podem ser realizados estudos quanto à qualidade alcançada em projetos de jogos com a abordagem MDGD. Tais estudos podem avaliar a qualidade utilizando métricas que levem em consideração a quantidade de problemas detectados através de inspeções no código. Também podem ser utilizadas métricas de análise dos níveis de complexidade, manutenibilidade, coesão e acoplamento do código.

No futuro também podem ser realizados estudos quanto a inclusão de outros subdomínios no projeto ou mesmo experimentos com outros gêneros de jogos. Outras linguagens de programação, outros motores de jogos e até mesmo outros padrões de projeto também podem ser explorados com a abordagem MDGD para validar se os benefícios encontrados neste trabalho se repetem em outros ambientes de desenvolvimento.

Outro estudo importante para o futuro é a avaliação da abordagem MDGD com especialistas experientes na área de desenvolvimento de jogos. Para tal, podem ser realizadas entrevistas semi-estruturadas com a finalidade de coletar as opiniões dos especialistas sobre a abordagem MDGD, e também sobre as ferramentas criadas nesta dissertação. Além do estudo com profissionais baseado em entrevistas, pode ser feito uma replicação do experimento com os especialistas. Dessa forma consegue-se validar se há diferenças entre a utilização da abordagem MDGD

por estudantes com pouca experiencia e a utilização da mesma abordagem por profissionais experientes na área.

REFERÊNCIAS

AMBLER, S. Agile model driven development is good enough. *Software, IEEE*, v. 20, n. October, p. 71–73, 2003. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1231156>.

BHANOT, V. et al. Using Domain-Specific Modeling to Develop Software Defined Radio Components and Applications. *The 5th OOPSLA Workshop on Domain-Specific Modeling, San Diego USA*, p. 33 – 42, 2005. Disponível em: <http://www.dsmforum.org/events/DSM05/bhanot.pdf>.

BLOW, J. Game development: Harder than you think. *Queue*, ACM, New York, NY, USA, v. 1, n. 10, p. 28–37, fev. 2004. ISSN 1542-7730. Disponível em: <http://doi.acm.org/10.1145/971564.971590>.

BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. *Uml - Guia do Usuário - Tradução da 2ª Edição*. [S.l.]: Campus Editora, 2006.

BRANSFORD, J.; BROWN, A.; COCKING, R. *How People Learn: Brain,Mind, Experience, and School*. [S.l.]: Washington, DC: Nat. Acad. Press, 2000.

BRIGHTMAN, J. *Game software market to hit 100 billion by 2018 - DFC*. 2014. Disponível em: <http://www.gamesindustry.biz/articles/2014-06-25-game-software-market-to-hit-usd100-billion-by-2018-dfc>.

CALIXTO, J. G. M.; BITTAR, T. J. Criação de Jogos Eletrônicos com Desenvolvimento de Jogos Orientado a Modelos. *Enacomp*, 2010. Disponível em: <http://www.enacomp.com.br/2010/cd/artigos-completos.php>.

COOPER, K. M. L.; LONGSTREET, C. S. Towards model-driven game engineering for serious educational games: Tailored use cases for game requirements. *2012 17th International Conference on Computer Games (CGAMES)*, IEEE, p. 208–212, jul. 2012. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6314577>.

CZARNECKI, K.; EISENECKER, U. W. *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 0-201-30977-7.

DEURSEN, A. van; KLINT, P. Little languages: little maintenance. *Journal of Software Maintenance*, John Wiley & Sons, Inc., New York, NY, USA, v. 10, n. 2, p. 75–92, mar. 1998. ISSN 1040-550X. Disponível em: <http://dx.doi.org/10.1002/(SICI)1096-908X(199803/04)10:2¡75::AID-SMR168¿3.0.CO;2-5>.

DEURSEN, A. van; KLINT, P.; VISSER, J. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, n. 6, p. 26–36, jun. 2000. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/352029.352035>.

DORMANS, J. Level design as model transformation: a strategy for automated content generation. ACM, New York, NY, USA, p. 1–8, 2011. Disponível em: <http://doi.acm.org/10.1145/2000919.2000921>.

DORMANS, J. The Effectiveness and Efficiency of model driven game design. Springer-Verlag, p. 542–548, 2012. Disponível em: <http://dx.doi.org/10.1007/978-3-642-33542-6-71>.

FOWLER, M. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. http://www.martinfowler.com/articles/injection.htm p.

FRANCE, R.; RUMPE, B. Model-driven Development of Complex Software : A Research Roadmap. *29th International Conference on Software Engineering 2007 - Future of Software Engineering*, n. 2, p. 37–54, 2007.

FREEMARKER. *jMonkeyEngine*. 2014. Disponível em: <http://freemarker.org/>.

FURTADO, A.; SANTOS, A. Using domain-specific modeling towards computer games development industrialization. ... *Workshop on Domain-Specific Modeling (...*, 2006. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.6173&rep=rep1&type=pdf#page=8>.

FURTADO, A.; SANTOS, A.; RAMALHO, G. SharpLudus revisited: from ad hoc and monolithic digital game DSLs to effectively customized DSM approaches. ... *-located workshops on DSM' ...*, p. 57–62, 2011. Disponível em: <http://dl.acm.org/citation.cfm?id=2095061>.

GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

GAMMA, E.; JOHNSON, R.; HELM RICHARD;VLISSIDES, J. *Padrões de Projeto - Soluções Reutilizaveis de Software Orientado a Objetos*. [S.l.]: Bookman, 2006.

GRONBACK, R. C. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. 1. ed. [S.l.]: Addison-Wesley Professional, 2009. ISBN 0321534077, 9780321534071.

HAHN, C. A Domain Specific Modeling Language for Multiagent Systems. n. Aamas, p. 233–240, 2008.

HESSELLUND, A.; CZARNECKI, K.; WASOWSKI, A. Guided development with multiple domain-specific languages. In: *Model Driven Engineering Languages and Systems (MoDELS 2007)*. [S.l.]: Springer, 2007. p. 46–60.

JMONKEYENGINE. *jMonkeyEngine*. 2014. Disponível em: <http://jmonkeyengine.org/>.

JOVANOVIC, M. et al. Motivation and Multimodal Interaction in Model-Driven Educational Game Design. *Systems Man and Cybernetics Part A Systems and Humans IEEE Transactions on*, v. 41, n. 4, p. 817–824, 2011. ISSN 10834427.

KARAMANOS, C.; SGOUROS, N. M. Automating the Implementation of Games Based on Model-Driven Authoring Environments. p. 524–529, 2012.

KLEPPE, A. G.; WARMER, J.; BAST, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 032119442X.

KOSAR, T. et al. Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. 2010. Disponível em: <http://www.doiserbia.nb.rs/img/doi/1820-0214/2010/1820-02141002247K.pdf>.

KUSTERER, R. *jMonkeyEngine 3.0 Beginner's Guide*. [S.l.]: Packt Publishing Ltd, 2013. ISBN 9781849516464.

LUCREDIO, D. *Uma Abordagem Orientada a Modelos para Reutilização de Software*. Tese (Doutorado) — Universidade de São Paulo, São Carlos, 2009. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-02092009-140533/>.

LUCREDIO, D.; ALMEIDA, E. S.; FORTES, R. P. M. An investigation on the impact of MDE on software reuse. *Software Components ...*, 2012. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6394979>.

MAIER, S.; VOLK, D. Facilitating language-oriented game development by the help of language workbenches. *Proceedings of the 2008 Conference on Future Play Research, Play, Share - Future Play '08*, ACM Press, New York, New York, USA, p. 224, 2008. Disponível em: <http://portal.acm.org/citation.cfm?doid=1496984.1497029>.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 37, n. 4, p. 316–344, dez. 2005. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1118890.1118892>.

MINOVIć, M. et al. Model Driven Development of User Interfaces for Educational Games. 2009.

NOVAK, J. *DESENVOLVIMENTO DE GAMES - Tradução da 2ª edição norte-americana*. [S.l.]: Cengage Learning, 2011.

PETRILLO, F. et al. What went wrong? A survey of problems in game development. *... in Entertainment (CIE)*, v. 7, n. 1, p. 1–22, 2009. Disponível em: <http://dl.acm.org/citation.cfm?id=1486521>.

RADJENOVIC, J.; MILOSAVLJEVIC, B.; SURLA, D. Modelling and implementation of catalogue cards using FreeMarker. 2008.

REYNO, E.; CUBEL, J. Model-driven game development: 2d platform game prototyping. *Proc. Game-On*, p. 2–4, 2008. Disponível em: <http://issi.dsic.upv.es/publications/archives/f-1232450860254/Gameon 2008 - MDGD - 2D Platform Game Prototyping.pdf>.

REYNO, E.; CUBEL, J. A Platform-Independent Model for Videogame Gameplay Specification. *... New Ground: Innovation in Games, Play, ...*, 2009. Disponível em: <http://lmc.gatech.edu/ cpearce3/DiGRA09/Friday 4 September/50 A Platform-Independent Model for Videogame Gameplay Specification.pdf>.

REYNO, E.; CUBEL, J. Automatic prototyping in model-driven game development. *Computers in Entertainment*, v. 7, n. 2, p. 1, jun. 2009. ISSN 15443574. Disponível em: <http://portal.acm.org/citation.cfm?doid=1541895.1541909>.

SCHMIDT, D. Guest editor's introduction: Model-driven engineering. *Computer*, v. 39, n. 2, p. 25–31, 2006. ISSN 0018-9162.

SHULL, F. et al. Victor r. basili's contributions to software quality. *Software, IEEE*, v. 23, n. 1, p. 16–18, Jan 2006. ISSN 0740-7459.

SMITH, T.; COOPER, K.; LONGSTREET, C. Software engineering senior design course: experiences with agile game development in a capstone project. ... *Games and Software Engineering*, p. 9–12, 2011. Disponível em: <http://dl.acm.org/citation.cfm?id=1984679>.

SOLINGEN, R. V.; BERGHOUT, E. *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. [S.l.]: McGraw-Hill Higher Education, 1999. ISBN 9780077095536.

SOMMERVILLE, I. *Engenharia de software, 8ª Edição*. [S.l.]: Pearson Addison-Wesley, 2007.

TANG, S.; HANNEGHAN, M. A Model-Driven Framework to Support Development of Serious Games for Game-based Learning. *2010 Developments in E-systems Engineering*, Ieee, p. 95–100, set. 2010. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=563337>.

TANG, S.; HANNEGHAN, M. Fusing games technology and pedagogy for games-based learning through a model driven approach. *2011 IEEE Colloquium on Humanities, Science and Engineering*, Ieee, n. Chuser, p. 380–385, dez. 2011. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6163756>.

TANG, S.; HANNEGHAN, M. State of the Art Model Driven Game Development : A Survey of Technological Solutions for Game-Based Learning. v. 22, p. 549–604, 2011.

THOMAS, D. MDA: revenge of the modelers or UML utopia? *Software, IEEE*, v. 21, n. 3, p. 15–17, 2004. ISSN 0740-7459.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. A. G. *Introdução à Engenharia de Software Experimental*. Rio de Janeiro, 2002.

VOELTER, M.; GROHER, I. Product line implementation using aspect-oriented and model-driven software development. *Software Product Line Conference, International*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 233–242, 2007.

WALTER, R.; MASUCH, M. How to integrate domain-specific languages into the game development process. *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology - ACE '11*, ACM Press, New York, New York, USA, p. 1, 2011. Disponível em: <http://dl.acm.org/citation.cfm?doid=2071423.2071475>.

WALTER, R.; MASUCH, M.; FUNK, M. 2 nd Workshop on Game Development and Model-Driven Software Development. *Proceedings of the 11th international conference on Entertainment Computing*, Springer-Verlag, Berlin, Heidelberg, p. 500–503, 2012. Disponível em: <http://dx.doi.org/10.1007/978-3-642-33542-6_64>.

WARMER, J.; KLEPPE, A. Building a Flexible Software Factory Using Partial. *Proc. of The 6th OOPSLA Workshop on Domain-Specific Modeling*, n. 612, 2006.

WOHLIN, C. et al. *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5.

Apendice A

INSCRIÇÕES PARA O EXPERIMENTO COM RESPOSTAS DO FORMULÁRIO DE CARACTERIZAÇÃO DO PARTICIPANTE

Tabela A.1: Inscrições - Grupo A - Abordagem Tradicional

Num.	Curso	Graduado	Aprovado em POO	Já desenvolveu algum jogo?	Nível de conhecimento em Java	Nível de conhecimento em MDD	Nível de conhecimento em Jogos 3d
1	CC	Não	Sim	Sim	4	2	3
2	SI	Não	Sim	Não	1	0	0
3	CC	Não	Sim	Não	2	0	0
4	SI	Sim	Sim	Sim	3	0	2
5	CC	Sim	Sim	Não	2	0	0
6	SI	Não	Sim	Não	2	0	1
7	CC	Sim	Sim	Sim	3	0	1
8	CC	Não	Sim	Sim	3	0	1
9	SI	Não	Sim	Não	4	2	2
10	CC	Não	Sim	Sim	3	1	3
11	SI	Não	Sim	Não	0	2	0
12	CC	Não	Sim	Sim	3	0	2
13	SI	Sim	Sim	Não	2	0	1
14	SI	Sim	Sim	Não	3	0	1
15	CC	Não	Sim	Sim	3	0	0
16	SI	Não	Sim	Sim	4	0	2
17	CC	Não	Sim	Sim	3	0	2
18	CC	Não	Sim	Sim	4	0	3
19	SI	Sim	Sim	Não	3	1	2
20	SI	Não	Sim	Não	1	0	0
21	CC	Não	Sim	Sim	2	0	0
22	SI	Não	Sim	Sim	2	0	2
23	SI	Sim	Sim	Sim	2	1	1
24	SI	Não	Sim	Não	3	0	1
25	SI	Não	Sim	Não	3	3	1
26	SI	Não	Sim	Não	3	1	0
						Média	1,44

Tabela A.2: Inscrições - Grupo B - Abordagem MDGD

Num.	Curso	Graduado	Aprovado em POO	Já desenvolveu algum jogo?	Nível de conhecimento em Java	Nível de conhecimento em MDD	Nível de conhecimento em Jogos 3d
27	CC	Não	Sim	Não	2	0	0
28	SI	Não	Sim	Sim	3	0	1
29	SI	Não	Sim	Sim	3	0	0
30	CC	Não	Sim	Não	3	0	2
31	CC	Sim	Sim	Sim	3	3	3
32	CC	Não	Sim	Sim	3	0	1
33	SI	Não	Sim	Não	4	3	1
34	SI	Sim	Sim	Não	2	1	0
35	SI	Não	Sim	Não	1	0	2
36	SI	Não	Sim	Não	1	0	0
37	SI	Não	Sim	Não	3	3	2
38	CC	Não	Sim	Sim	3	0	4
39	CC	Não	Sim	Sim	2	0	0
40	CC	Não	Sim	Não	2	1	2
41	CC	Não	Sim	Sim	3	0	0
42	SI	Não	Sim	Sim	2	3	1
43	SI	Não	Sim	Não	1	0	0
44	CC	Não	Sim	Sim	3	0	2
45	CC	Não	Sim	Sim	3	0	1
46	CC	Não	Sim	Não	4	0	2
47	SI	Não	Sim	Não	3	0	1
48	CC	Não	Sim	Não	2	0	0
49	CC	Não	Sim	Sim	4	0	0
50	SI	Não	Sim	Não	1	0	1
51	CC	Não	Sim	Sim	4	0	0
52	CC	Não	Sim	Sim	3	0	4
						Média	1,44

* SI = Sistemas de Informação
* CC = Ciência da Computação

Apendice B

TUTORIAL JMONKEYENGINE

•**Sobre o JMonkey**

jMonkeyEngine é um motor de jogo 3D para desenvolvedores Java. É open source e multi-plataforma. O jogo é programado inteiramente na linguagem de programação Java, oferecendo assim uma ampla acessibilidade por desenvolvedores que já conhecem esta linguagem. Está sob a licença BSD, garantindo sua utilização de forma gratuita tanto para o desenvolvimento de jogos não comerciais, como também para jogos comerciais.

O jMonkeyEngine SDK (software development kit) é o ambiente de desenvolvimento do jogo recomendado para o desenvolvimento com o jMonkeyEngine 3. O jMonkeyEngine SDK é baseado na plataforma NetBeans, fornecendo além das amplas ferramentas de desenvolvimento da IDE NetBeans, plugins exclusivos para a criação de conteúdo do jogo.

A versão 3.0 do jMonkeyEngine foi lançada de forma estável no dia 15/02/2014. Esta é a versão adotada neste tutorial. O download do JMonkeyEngine SDK pode ser realizado no link: http://hub.jmonkeyengine.org/downloads/

É pré-requisito obrigatório para o desenvolvimento que você tenha instalado em sua máquina o JDK (java development kit). Seu download pode ser relizado através do endereço: http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html

•**Criação de um novo Projeto**

Após instalar o jMonkeyEngine SDK vá no menu “Arquivo-¿Novo Projeto”. Selecione o item “BasicGame” que fica dentro do pacote “JME3”, conforme mostra a Figura B.1.

Clique em “próximo”, digite o nome do projeto e clique em “Finalizar”. Será criado um projeto básico de jogo. Este projeto é separado principalmente pelos diretórios “Project

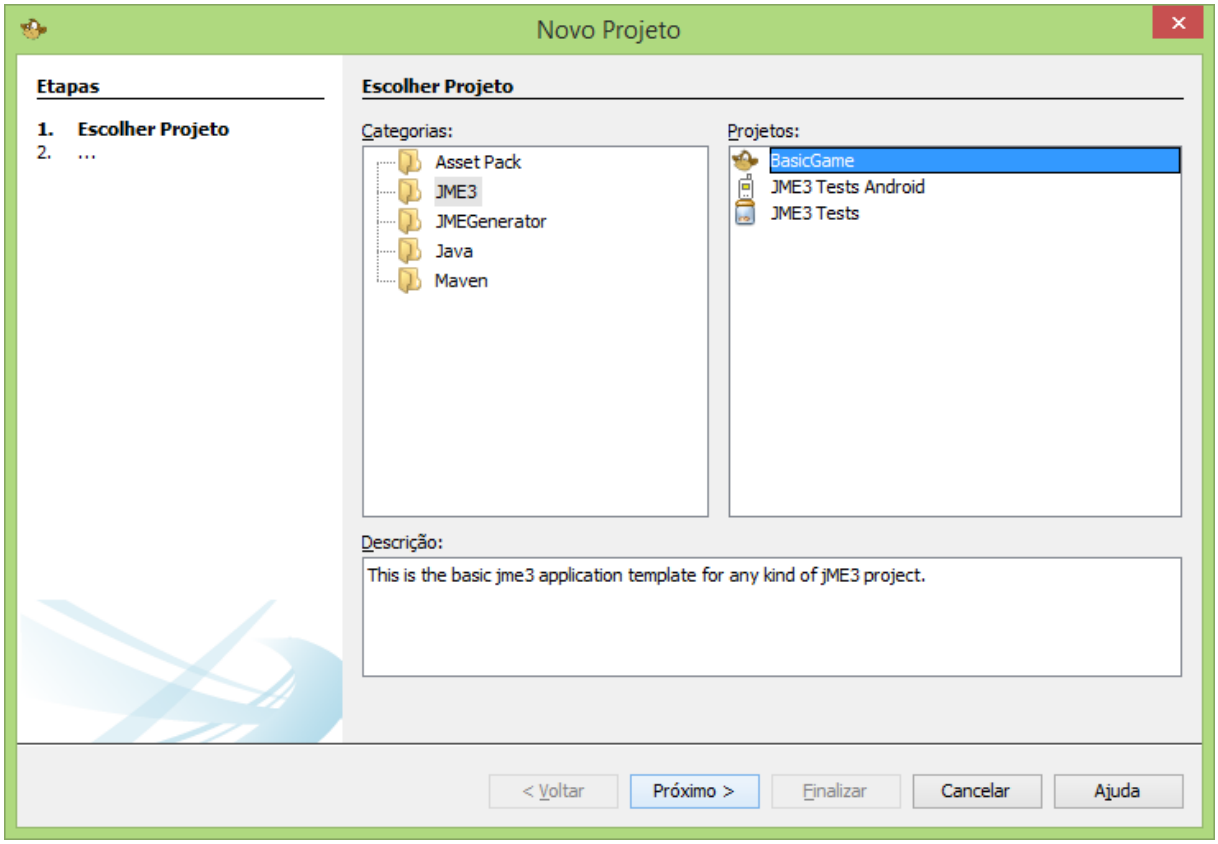

Figura B.1: Novo Projeto JMonkey Engine

Assets" e "Pacotes de Códigos-Fonte" conforme mostra a Figura B.2. Em "Project Assets" deverão ser colocados os arquivos de imagens, sons, objetos 3d, texturas, etc. Já em "Pacotes de Códigos-Fonte" deverão ser editados os código na linguagem Java, através de seus pacotes, classes, interfaces, etc.

A classe "Main" é onde o código deverá ser iniciado. Nela já existem 4 métodos definidos, os quais têm a seguinte funcionalidade:

• public static void main(String[] args): método responsável por inicializar a aplicação;

• public void simpleInitApp(): método responsável por inicializar a cena do jogo;

• public void simpleUpdate(float tpf): método opcional chamado a cada interação do loop do jogo. Neste método poderão ser inseridos códigos para definir a lógica do jogo.

• public void simpleRender(RenderManager rm): método opcional próprio para modificações avançadas na renderização do jogo.

•Orientação no espaço 3D

O jMonkeyEngine utiliza o sistema de coordenação similar ao OpenGL, com posições definidas pelo eixos X, Y e Z, conforme mostrado na Figura B.3.

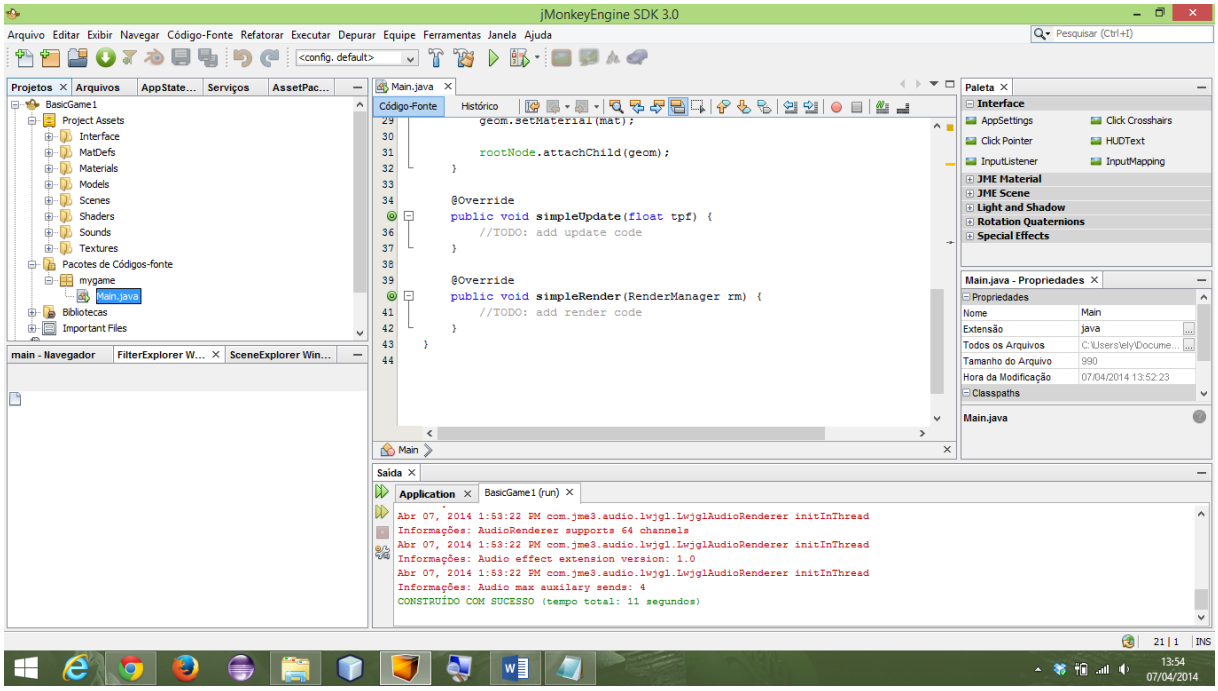

Figura B.2: Estrutura Básica do Projeto

O ponto de origem é a posição (0, 0, 0). Os três eixos de coordenadas estão em ângulos de 90 graus um do outro, e encontram-se no ponto de origem.

O eixo X define posições da esquerda para direita. O eixo Y define posições de baixo para cima. O Eixo Z começa longe de você e vai em sua direção. Em cada eixo poderão ser definidos números positivos e negativos para representar sua posição.

No jMonkeyEngine é possível instanciar um objeto capaz de representar uma posição no espaço 3d. Este objeto é chamado de Vecto3f, e pode ser instanciado passando como parâmetro para seu construtor as posição x, y e z.

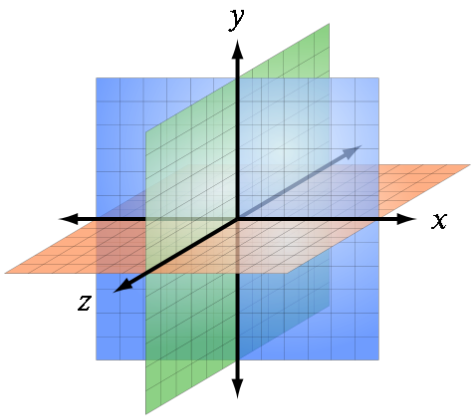

Figura B.3: Orientação no espaço 3d

Exemplo: Vector3f v = new Vector3f(2.0f, 3.0f, -4.0f);

Uma cena do jogo é composta por um conjunto de formas geométricas (Geometry) e nós (Nodes). Um nó é um meio tipicamente usado para agrupar outros nós e formas geométricas, além de poder representar modelos, terrenos e outros objetos visíveis na cena. Para que os nós ou formas geométricas possam ser exibidas na cena, eles devem ser inseridos no nós raiz através do seguinte método:

rootNode.attachChild(geom);

Para compor uma cena faz-se necessário posicionar os nós no ambiente 3d. Para posicionar cada nó, basta chamar o método setLocalTranslation enviando as respectivas posições x, y e z:

floorGeometry.setLocalTranslation(0.5f, -0.3f, 1.0f);

Muitas vezes, queremos saber qual a localização de um elemento dentro do jogo. Para isso podemos utilizar o método getWorldTranslation, capaz de retornar um objeto do tipo Vector3f com as coordenadas do nó:

Vector3f v = node.getWorldTranslation();

Uma funcionalidade interessante quando se trabalha com nós, é que podemos calcular a distância entre 2 nós através do método 'distance', conforme o exemplo a seguir:

float d = node1.getLocalTranslation().distance(node2.getLocalTranslation());

•Física

Para simulação de física em jogos, o jMonkeyEngine disponibiliza um conjunto de bibliotecas iniciado pela classe BulletAppState. Para especificar a utilização da física padrão, deve-se declarar um objeto do tipo BulletAppState. Este objeto deve ser instanciado no método simpleInitApp, e também deve ser atachado ao stateManager, como mostrado no quadro 1.

```
private BulletAppState bulletAppState;
public void simpleInitApp() {
    bulletAppState = new BulletAppState();
    stateManager.attach(bulletAppState);
}
```

Quadro 1 – Aplicação de física

•Iluminação

Um jogo tridimensional deve possuir no mínimo uma fonte de luz, para iluminar o ambiente e para que jogador consiga visualizar os elementos do jogo. Um cenário pode possuir mais de uma fonte de luz, a fim de iluminar melhor o ambiente, e também para trazer bons efeitos de sombra e brilho.

No jMonkeyEngine, para criar um ponto de luz no cenário, deve-se instanciar um objeto do tipo DirectionalLight. Este ponto de luz pode ter seu direcionamento focal modificado pelo método setDirection. Para que o ponto de luz seja anexado ao jogo, e assim passe a ser visível, tem-se que adicioná-lo ao método rootNode.addLight.

Um exemplo de dois pontos de luz adicionados em um cenário pode ser visto no Quadro 2. Esse códigos devem estar no método simpleInitApp.

```
DirectionalLight dl = new DirectionalLight();
dl.setDirection(new Vector3f(-0.5f, -1f, -0.3f).normalizeLocal());
rootNode.addLight(dl);

dl = new DirectionalLight();
dl.setDirection(new Vector3f(0.5f, -0.1f, 0.3f).normalizeLocal());
rootNode.addLight(dl);
```

Quadro 2 – Iluminação

•Assets

Para o desenvolvimento dos exemplos contidos neste tutorial foram utilizados alguns modelos 3d e texturas que podem ser baixados no seguinte endereço:

http://goo.gl/z5gvO9

Após efetuar o download, descompacte todo o conteúdo do arquivo dentro da pasta "assets" de seu projeto.

Na Figura B.4 segue uma representação que descreve cada elemento que iremos utilizar. Os demais arquivos contidos em assets não foram descritos pois não serão utilizados diretamente.

•Terreno

Para dar inicio a construção do cenário, antes é necessário criar um retângulo para representar o chão. Esse retângulo pode receber uma textura, simulando um grama, por exemplo.

Para criação do terreno deve-se instanciar um Material e um Texture. A textura pode ser carregada a partir de qualquer arquivo de imagem. Para definir a posição e o tamanho

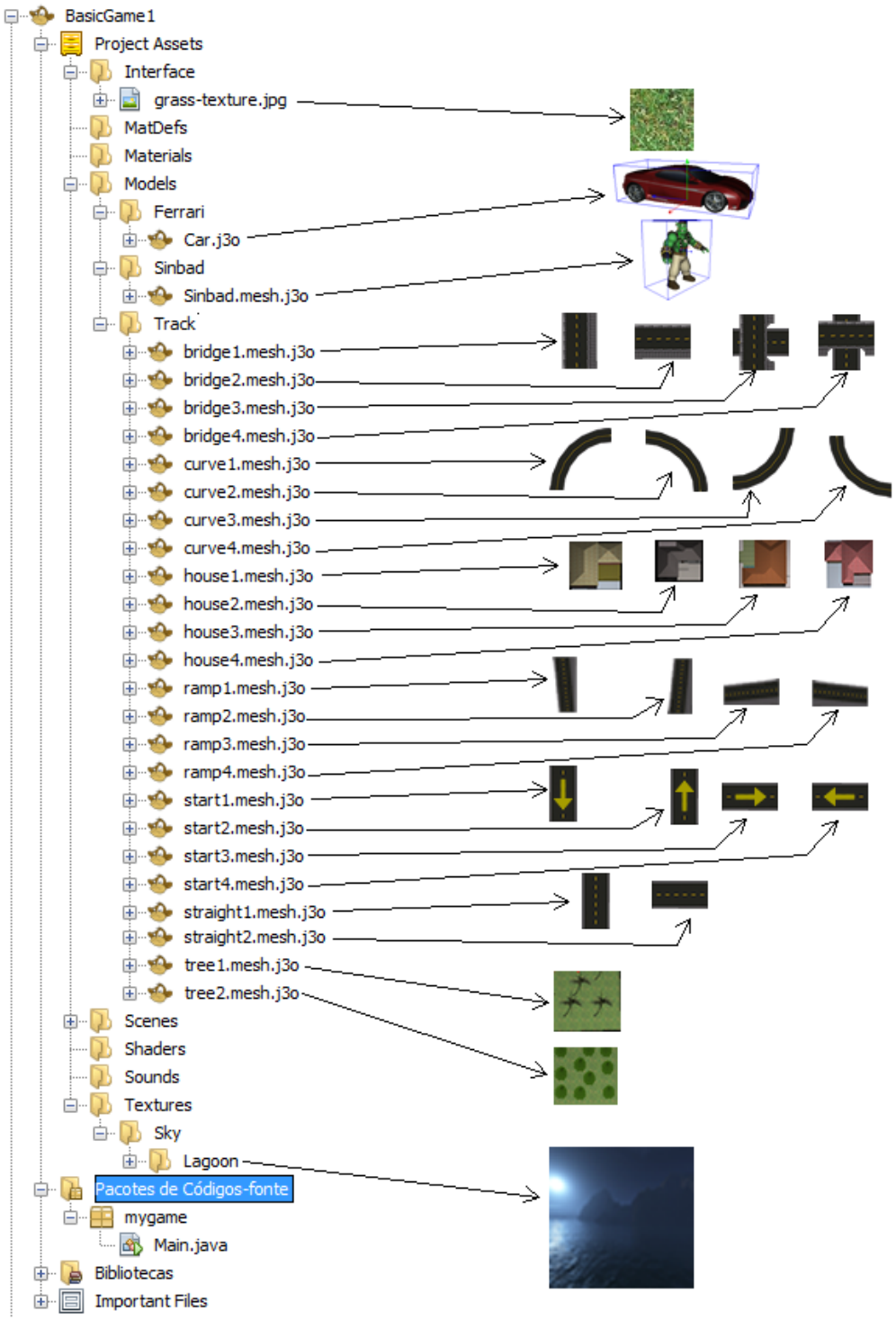

Figura B.4: Assets

do terreno podem ser definidas variáveis do tipo float (x1, x2, z1 e z2). Através dessas variáveis pode-se criar um objeto do tipo Box e outro Geometry. O objeto Geometry deve ser atachado no rootNode para que possa ser exibido no jogo, e também deve ser adicionado ao bulletAppState para que o jogo consiga detectar colisões com o chão.

Um exemplo de implementação do terreno pode ser visto no quadro 3. Esta implementação deve estar dentro do método simpleInitApp.

```
Material material = new Material(assetManager, "Common/MatDefs/Misc/
    Unshaded.j3md");
Texture floortexture = assetManager.loadTexture("Interface/grass-
    texture.jpg");
floortexture.setWrap(Texture.WrapMode.Repeat);
material.setTexture("ColorMap", floortexture);

float x1 = 0;
float x2 = 400;
float z1 = 0;
float z2 = 400;

Box floorBox = new Box(x2 - x1, 0.25f, z2 - z1);
floorBox.scaleTextureCoordinates(new Vector2f(100f, 100f));
Geometry floorGeometry = new Geometry("Floor", floorBox);

floorGeometry.setMaterial(material);
floorGeometry.setLocalTranslation(x1, -0.3f, z1);
floorGeometry.addControl(new RigidBodyControl(0));
rootNode.attachChild(floorGeometry);
bulletAppState.getPhysicsSpace().add(floorGeometry);
```

Quadro 3 – Terreno

•Cenário

Para adicionar um modelo qualquer dos assets para montar o cenário, deve-se instanciar um objeto do tipo Spatial, responsável por carregar o modelo 3d. Para que este objeto interaja com a física (gravidade, inércia, colisão, etc..) deve-se instanciar um objeto do tipo RigidBodyControl, vinculando-se ao objeto da modelagem. Outra definição importante é quanto a posição que o objeto será adicionado no cenário. Para isso deve-se declarar um método do tipo Vector3f, passando-o como parâmetro para o método 'move' do objeto 3d. No objeto Vector3f são passados os valores correspondentes à posição desejada no cenário, conforme explicado na seção 3.1.

Um exemplo de adição de um modelo 3d no cenário pode ser visto no quadro 4. Lembre-se de colocar esses código dentro do método simpleInitApp.

```
Spatial scene = assetManager.loadModel("Models/Track/house1.mesh.j3o")
    ;
RigidBodyControl terrainPhysicsNode = new RigidBodyControl(
    CollisionShapeFactory.createMeshShape(scene), 0);
Vector3f pos = new Vector3f(46, 0, 10);
scene.move(pos);
rootNode.attachChild(scene);
terrainPhysicsNode.setPhysicsLocation(pos);
bulletAppState.getPhysicsSpace().add(terrainPhysicsNode);
```

Quadro 4 – Adição de modelo 3d no cenário

Os elementos bridge1, bridge2, bridge3, house1, house2, house3, house4, start1, start2, start3, start4, straight1, straight2, tree1 e tree2 possuem tamanho de 36 por 36 nas respectivas coordenadas x e z. Já os elementos curve1, curve2, curve3 e curve4 possuem tamanho de 72 por 72 nas coordenadas x e z. Os elementos ramp1 e ramp2 possuem tamanho de 36 no eixo x e 72 no eixo z. Os elementos ramp3 e ramp4 possuem tamanho de 72 no eixo x e 36 no eixo z.

Desta forma é possível formar grandes cenários a partir da união de diferentes modelos. Um exemplo é mostrado a seguir no quadro 5, onde são inseridos no cenário uma reta e duas rampas.

```
Spatial scene1 = assetManager.loadModel("Models/Track/straight1.mesh.
    j3o");
RigidBodyControl terrainPhysicsNode1 = new RigidBodyControl(
    CollisionShapeFactory.createMeshShape(scene1), 0);
Vector3f pos1 = new Vector3f(10, 0, 10);
scene1.move(pos1);
rootNode.attachChild(scene1);
terrainPhysicsNode1.setPhysicsLocation(pos1);
bulletAppState.getPhysicsSpace().add(terrainPhysicsNode1);


Spatial scene2 = assetManager.loadModel("Models/Track/ramp2.mesh.j3o")
    ;
RigidBodyControl terrainPhysicsNode2 = new RigidBodyControl(
    CollisionShapeFactory.createMeshShape(scene2), 0);
Vector3f pos2 = new Vector3f(10, 0, 46);
scene2.move(pos2);
rootNode.attachChild(scene2);
```

```
terrainPhysicsNode2.setPhysicsLocation(pos2);
bulletAppState.getPhysicsSpace().add(terrainPhysicsNode2);

Spatial scene3 = assetManager.loadModel("Models/Track/ramp1.mesh.j3o")
    ;
RigidBodyControl terrainPhysicsNode3 = new RigidBodyControl(
    CollisionShapeFactory.createMeshShape(scene3), 0);
Vector3f pos3 = new Vector3f(10, 0, 118);
scene3.move(pos3);
rootNode.attachChild(scene3);
terrainPhysicsNode3.setPhysicsLocation(pos3);
bulletAppState.getPhysicsSpace().add(terrainPhysicsNode3);
```

Quadro 5 – Exemplo de cenário

•Personagem jogável

Um personagem jogável pode ser definido através do objeto CharacterControl. Este objeto do jMonkeyEngine disponibiliza uma série de métodos para controle do personagem. Também precisamos de um objeto do tipo Node atrelado ao CharacterControl, além de um objeto Vector3f para indicar a direção que o personagem irá andar e outro objeto Vector3d para indicar a direção em que o personagem ficará com o corpo virado. Por fim, outros atributos importante são variáveis para definir o controle do usuário. Lembre-se que a declaração de atributos em uma classe deve ser feita fora dos métodos da classe. Os atributos necessários para a criação do personagem podem ser vistos no quadro 6.

```
private CharacterControl physicsCharacter;
private Node characterNode;
private Vector3f walkDirection;
private Vector3f viewDirection;
private boolean forward = false, backward = false, leftRotate = false,
    rightRotate = false;
```

Quadro 6 – Atributos do personagem

Para inicializar o personagem dentro do método simpleInitApp deve-se carregar um modelo 3d dos assets. Também deve-se definir uma escala do seu tamanho e sua rotação inicial. Por fim deve-se adicionar o objeto physicsCharacter no bulletAppState para que o personagem tenha tratamento de colisão e também adicionar o characterNode no rootNode para que o personagem seja mostrado no jogo. Um exemplo de implementação do personagem pode ser visto no Quadro 7.

```
physicsCharacter = new CharacterControl(new CapsuleCollisionShape(0.5f
    , 1.8f), .1f);
physicsCharacter.setPhysicsLocation(new Vector3f(0, 1, 0));
characterNode = new Node("character node");
Spatial model = assetManager.loadModel("Models/Sinbad/Sinbad.mesh.j3o"
    );
model.scale(0.25f);
model.setLocalRotation(new Quaternion(0, 1, 0, 0));
characterNode.addControl(physicsCharacter);
bulletAppState.getPhysicsSpace().add(physicsCharacter);
rootNode.attachChild(characterNode);

characterNode.attachChild(model);

walkDirection = physicsCharacter.getWalkDirection();
viewDirection = physicsCharacter.getViewDirection();
```

Quadro 7 – Implementação do personagem

Para reposicionar o personagem dentro do cenário é necessário definir as coordenadas desejadas em um objeto do tipo Vector3f, e passado como parâmetro para o método setPhysicsLocation, conforme mostrado no quadro 8.

```
Vector3f vec = new Vector3f(10, 0, 46);
physicsCharacter.setPhysicsLocation(vec);
```

Quadro 8 – Posicionando Personagem

Caso haja a necessidade de remover o personagem do cenário e da física, basta remove-lo do bulletAppState e do rootNode, conforme mostrado no quadro 9.

```
bulletAppState.getPhysicsSpace().remove(physicsCharacter);
rootNode.detachChild(characterNode);
```

Quadro 9 – Removendo Personagem

•**Veículo**

Além do personagem mostrado anteriormente, pode-se definir um veículo como personagem jogável, podendo desenvolver jogos de corrida, por exemplo. Para adicionar um veículo, antes é necessário declarar um método chamado ‘findGeom’, capaz de pesquisar de forma recursiva por partes de um modelo 3d. Este método é mostrado no quadro 10.

```
private Geometry findGeom(Spatial spatial, String name) {
    if (spatial instanceof Node) {
```

```
        Node node = (Node) spatial;
        for (int i = 0; i < node.getQuantity(); i++) {
            Spatial child = node.getChild(i);
            Geometry result = findGeom(child, name);
            if (result != null) {
                return result;
            }
        }
    } else if (spatial instanceof Geometry) {
        if (spatial.getName().startsWith(name)) {
            return (Geometry) spatial;
        }
    }
    return null;
}
```

Quadro 10 – Pesquisa por elementos de um objeto 3d

Para instanciar um veículo é necessário declarar um atributo do tipo VehicleControl disponível na biblioteca do jMonkeyEngine. Este objeto disponibiliza um série de métodos para controle e definição de veículos. Também é necessário declarar um objeto do tipo Node, além de atributos para controle de direção e aceleração do veículo. No quadro 11 são mostrados os atributos para declaração de um veiculo.

```
private VehicleControl car;
private Node carNode;
private boolean turnLeft = false;
private boolean turnRight = false;
private boolean accelerate = false;
private int accelerationValue = 0;
private float steeringValue = 0;
```

Quadro 11 – Atributos do Veiculo

Para instanciar um veículo devem ser definidas configurações para o amortecedor (stiffness, campValue e dampValue) e para o peso do veículo (mass). Depois deve ser carregado o modelo 3d do veículo, e adicionado uma caixa de colisão. O próximo passo é definir as configurações de cada uma das quatro rodas e também cada amortecedor. Um método importante de configuração das rodas é o 'setFrictionSlip' que define o valor de atrito dos pneus e controla a derrapagem do veículo. Como em todos outros elementos do jogo, o veículo deve ser atachado ao rootNode e também adicionado no bulletAppState. Para definir a posição inicial do veículo, basta posicionar seu nó através do método

setPhysicsLocation. Um exemplo de implementação de um veículo pode ser visto no Quadro 12.

```
float stiffness = 120.0f;//200=f1 car
float compValue = 0.2f; //(lower than damp!)
float dampValue = 0.3f;
float mass = 600.0f; //original: 400

//Load model and get chassis Geometry
carNode = (Node) assetManager.loadModel("Models/Ferrari/Car.j3o");
carNode.setShadowMode(RenderQueue.ShadowMode.Cast);
Geometry chasis = findGeom(carNode, "Car");
BoundingBox box = (BoundingBox) chasis.getModelBound();

//Create a hull collision shape for the chassis
CollisionShape carHull = CollisionShapeFactory.createDynamicMeshShape(
    chasis);

//Create a vehicle control
car = new VehicleControl(carHull, mass);
carNode.addControl(car);

//Setting default values for wheels
car.setSuspensionCompression(compValue * 2.0f * FastMath.sqrt(
    stiffness));
car.setSuspensionDamping(dampValue * 2.0f * FastMath.sqrt(stiffness));
car.setSuspensionStiffness(stiffness);
car.setMaxSuspensionForce(10000);

//Create four wheels and add them at their locations
//note that our fancy car actually goes backwards..
Vector3f wheelDirection = new Vector3f(0, -1, 0);
Vector3f wheelAxle = new Vector3f(-1, 0, 0);

Geometry wheel_fr = findGeom(carNode, "WheelFrontRight");
wheel_fr.center();
box = (BoundingBox) wheel_fr.getModelBound();
float wheelRadius = box.getYExtent();
float back_wheel_h = (wheelRadius * 1.7f) - 1f;
float front_wheel_h = (wheelRadius * 1.9f) - 1f;
car.addWheel(wheel_fr.getParent(), box.getCenter().add(0, -
    front_wheel_h, 0),
wheelDirection, wheelAxle, 0.2f, wheelRadius, true);
```

```
Geometry wheel_fl = findGeom(carNode, "WheelFrontLeft");
wheel_fl.center();
box = (BoundingBox) wheel_fl.getModelBound();
car.addWheel(wheel_fl.getParent(), box.getCenter().add(0, -
    front_wheel_h, 0),
wheelDirection, wheelAxle, 0.2f, wheelRadius, true);

Geometry wheel_br = findGeom(carNode, "WheelBackRight");
wheel_br.center();
box = (BoundingBox) wheel_br.getModelBound();
car.addWheel(wheel_br.getParent(), box.getCenter().add(0, -
    back_wheel_h, 0),
wheelDirection, wheelAxle, 0.2f, wheelRadius, false);

Geometry wheel_bl = findGeom(carNode, "WheelBackLeft");
wheel_bl.center();
box = (BoundingBox) wheel_bl.getModelBound();
car.addWheel(wheel_bl.getParent(), box.getCenter().add(0, -
    back_wheel_h, 0), wheelDirection, wheelAxle, 0.2f, wheelRadius,
    false);

car.getWheel(0).setFrictionSlip(16.0f);
car.getWheel(1).setFrictionSlip(16.0f);
car.getWheel(2).setFrictionSlip(12.0f);
car.getWheel(3).setFrictionSlip(12.0f);

rootNode.attachChild(carNode);

bulletAppState.getPhysicsSpace().add(car);

car.setPhysicsLocation(new Vector3f(5, 0.5f, -5));
```

Quadro 12 – Implementação do Veiculo

Para reposicionar o veículo dentro do cenário é necessário definir as coordenadas desejadas em um objeto do tipo Vector3f, e passado como parâmetro para o método setPhysicsLocation, conforme mostrado no quadro 13.

```
Vector3f vec = new Vector3f(10, 0, 46);
car.setPhysicsLocation(vec);
```

Quadro 13 – Posicionando Veículo

•Textura Espacial

A textura espacial é capaz de proporcionar um ambiente mais realista, mostrando uma imagem no horizonte, como o céu, por exemplo. Para definir uma textura espacial devem ser carregadas imagens para serem exibidas no oeste, leste, norte, sul, acima e abaixo. Depois deve ser declarado um objeto do tipo Spatial e atachado no rootNode. Um exemplo de implementação da textura espacial pode ser vista no quadro 14.

```
Texture west =
   assetManager.loadTexture("Textures/Sky/Lagoon/lagoon_west.jpg");
Texture east =
   assetManager.loadTexture("Textures/Sky/Lagoon/lagoon_east.jpg");
Texture north =
   assetManager.loadTexture("Textures/Sky/Lagoon/lagoon_north.jpg");
Texture south =
   assetManager.loadTexture("Textures/Sky/Lagoon/lagoon_south.jpg");
Texture up =
   assetManager.loadTexture("Textures/Sky/Lagoon/lagoon_up.jpg");
Texture down =
   assetManager.loadTexture("Textures/Sky/Lagoon/lagoon_down.jpg");

Spatial sky = SkyFactory.createSky(assetManager, west, east, north,
    south, up, down);
rootNode.attachChild(sky);
```

Quadro 14 – Textura Espacial

•Interação via teclado

Até este ponto do tutorial foi possível definir elementos no cenário do jogo, porém o jogar ainda não faz nenhum tipo de interação com os personagens. Para permitir interação do usuário deve-se fazer implementação da interface ActionListener, conforme mostrado no quadro 15.

```
public class Main extends SimpleApplication implements ActionListener
    {
```

Quadro 15 – Implementação da interface ActionListener

Tome cuidado ao fazer a importação da classe ActionListener para selecionar a biblioteca correta: com.jme3.input.controls.ActionListener. O próximo passo é adicionar o mapeamento das teclas (Mapping) e adicionar uma escuta para as teclas (Listener) no método simpleInitApp, conforme mostrado o quadro 16.

```
inputManager.addMapping("Lefts", new KeyTrigger(KeyInput.KEY_LEFT));
inputManager.addMapping("Rights", new KeyTrigger(KeyInput.KEY_RIGHT));
inputManager.addMapping("Ups", new KeyTrigger(KeyInput.KEY_UP));
inputManager.addMapping("Downs", new KeyTrigger(KeyInput.KEY_DOWN));
inputManager.addMapping("Space", new KeyTrigger(KeyInput.KEY_SPACE));

inputManager.addListener(this, "Lefts");
inputManager.addListener(this, "Rights");
inputManager.addListener(this, "Ups");
inputManager.addListener(this, "Downs");
inputManager.addListener(this, "Space");
```

Quadro 16 – Mapeamento das teclas

Depois de fazer o mapeamento das teclas deve-se implementar o método onAction, que será responsável por executar a ação de cada tecla pressionada. O atributo 'value' recebido por parâmetro tem valor 'true' quando alguma tecla é pressionada e 'false' quando a tecla é solta. O atributo 'binding' tem como valor o nome da tecla pressionada. Um exemplo do método onAction pode ser visto no quadro 17.

```
public void onAction(String binding, boolean value, float tpf) {
    if (binding.equals("Lefts")) {

    } else if (binding.equals("Rights")) {

    } else if (binding.equals("Ups")) {

    } else if (binding.equals("Downs")) {

    } else if (binding.equals("Space")) {

    }
}
```

Quadro 17 – Ação de tecla pressionada

–Controle do Personagem

Para implementar o controle do personagem, dentro do método onAction apenas são alteradas os valores dos atributos definidos na criação do personagem. A partir deste atributos, dentro do método simpleUpdate são feitos os incrementos ou decrementos da rotação e direção do personagem. A única exceção está no evento de pular, onde

dentro do próprio método onAction é possível fazer chamada ao método 'jump' que se encarrega de fazer o personagem pular. Um exemplo de implementação do método onAction está no Quadro 18.

```
public void onAction(String binding, boolean value, float tpf) {
    if (binding.equals("Lefts")) {
        leftRotate = value;
    } else if (binding.equals("Rights")) {
        rightRotate = value;
    } else if (binding.equals("Ups")) {
        forward = value;
    } else if (binding.equals("Downs")) {
        backward = value;
    } else if (binding.equals("Space")) {
        if (value) {
            physicsCharacter.jump();
        }
    }
}
```

Quadro 18 – Ações dos personagens

O método simpleUpdate é invocado a cada interação do game loop, fazendo com que seja possível tratar eventos de forma gradual. Desta forma, quando as variáveis de rotação (leftRotate e rightRotate) têm valor verdadeiro é feito um cálculo da fração do número PI, que é o valor que define o quão rápido o personagem será rotacionado. Após efetuar este calculo é atributo o valor da rotação no método setViewDirection. Da mesma forma quando as variáveis de deslocamento (forward e backward) estão com valor verdadeiro é feito com calculo que corresponde a fração de movimento para frente ou para traz que o personagem irá deslocar. Quando maior for essa fração, mais rápido o personagem irá se locomover. Um exemplo de implementação do simpleUpdate de movimentação do personagem pode ser visto no quadro 19.

```
public void simpleUpdate(float tpf) {
    if (leftRotate) {
        Quaternion rotateL = new Quaternion().fromAngleAxis(
            FastMath.PI * 0.01f, Vector3f.UNIT_Y);
        rotateL.multLocal(viewDirection);

    } else if (rightRotate) {
        Quaternion rotateL = new Quaternion().fromAngleAxis(
            FastMath.PI * -0.01f, Vector3f.UNIT_Y);
        rotateL.multLocal(viewDirection);
```

```
    }


    walkDirection.set(0, 0, 0);

    Vector3f modelForwardDir = characterNode.getWorldRotation().
        mult(Vector3f.UNIT_Z);
    if (forward) {
        walkDirection.addLocal(modelForwardDir.negate().mult(0.3f)
            );
    } else if (backward) {
        walkDirection.addLocal(modelForwardDir.multLocal(0.3f));
    }

    physicsCharacter.setWalkDirection(walkDirection);
    physicsCharacter.setViewDirection(viewDirection);
}
```

Quadro 19 – Movimentação do personagem

–Controle do Veiculo

A implementação do controle do veículo segue a mesma lógica do controle do personagem. No método onAction, quando o usuário pressionar ou soltar as teclas para a esquerda ou direita são alterados os valores das variáveis turnLeft e turnRight. A seta para cima irá alterar a variável accelerate, mas caso esta tecla for solta o carro tem sua aceleração atribuída como zero imediatamente. A seta para baixo tem duas funcionalidades embutidas, frear ou acelerar de ré. Quando o veiculo estiver a uma velocidade superior a 2 km/h, o veículo é freado, já quando a velocidade for inferior a 2 km/h o veículo é acelerado para traz. Tanto os valores de aceleração, quanto de velocidade são negativos para movimentos para frente e positivos para movimentos para traz. Quando o usuário soltar a seta para baixo, independente da velocidade do veiculo, o freio é solto. A implementação deste método é apresentada no quadro 20.

```
public void onAction(String binding, boolean value, float tpf) {
    if (binding.equals("Lefts")) {
        turnLeft = value;
    } else if (binding.equals("Rights")) {
        turnRight = value;
    } else if (binding.equals("Ups")) {
        this.accelerate = value;
        if (!accelerate) {
            accelerationValue = 0;
```

```
            car.accelerate(accelerationValue);
        }
    } else if (binding.equals("Downs")) {
        if (!value) {
            car.brake(0f);
        } else {
            if (car.getCurrentVehicleSpeedKmHour() <= -2) {
                car.brake(140f);
            } else {
                car.accelerate(400);
            }
        }
    } else if (binding.equals("Space")) {
    }
}
```

Quadro 20 – Ações do veiculo

Após ter definido os valores das variáveis de acordo com as teclas pressionadas, pode-se implementar de fato a movimentação do veículo. No caso do evento disparado ao se pressionar a seta para cima, pode-se colocar estruturais condicionais para definir o limite de aceleração e o limite de velocidade do veículo. No exemplo mostrado no quadro 21 o limite de aceleração máxima é de 1200 e o limite de velocidade máxima é de 160 km/h, lembrando que para movimentos para frente esses valores devem ser atribuídos com números negativos.

Para fazer uma direção gradual, deve-se implementar incrementos em um variável que define o quanto a direção está virada (steeringValue). O incremento da direção mostrado no quadro 16 está como 0,008, pode alterar esse valor para mais deixando a direção mais rápida ou para menos deixando a direção mais lenta. Também é importante definir o valor máximo que a direção poderá virar, sendo definido com 0,5 no exemplo mostrado no quadro 21.

```
@Override
public void simpleUpdate(float tpf) {
    if (accelerate) {
        if (accelerationValue > -1200.0) {
            accelerationValue -= 160;
            car.accelerate(accelerationValue);
        }
        if (car.getCurrentVehicleSpeedKmHour() <= -160.0) {
            //velocidade Ã¡mxima
            car.accelerate(accelerationValue / 10);
```

```
            }
        }
        if (turnLeft) {
            if (steeringValue <= .5f) {
                steeringValue += 0.008f;
                car.steer(steeringValue);
            }
        } else {
            //volta a direcao
            if (steeringValue > 0) {
                steeringValue -= 0.008f;
                if (steeringValue < 0.01) {
                    steeringValue = 0;
                }
                car.steer(steeringValue);
            }
        }
        if (turnRight) {
            if (steeringValue >= -.5f) {
                steeringValue -= 0.008f;
                car.steer(steeringValue);
            }
        } else {
            //volta a direcao
            if (steeringValue < 0) {
                steeringValue += 0.008f;
                car.steer(steeringValue);
                if (steeringValue > -0.01) {
                    steeringValue = 0;
                }
            }
        }
    }
```

Quadro 21 – Movimentação do veiculo

•Camera

Por fim, uma funcionalidade muito importante nos jogos em terceira dimensão é a definição da câmera. No jMonkeyEngine é possível 'amarrar' a câmera em um nó, fazendo com que a câmera siga o personagem ou o veiculo. Para isso primeiramente deve-se declarar um atributo do tipo CameraNode, conforme quadro 22.

```
private CameraNode camNode;
```

Quadro 22 – Declaração CameraNode

Depois, dentro do método simpleInitApp deve ser feita a instanciação do objeto CameraNode e amarração deste com o nó do personagem ou veiculo através do método attachChild. O método lookAt, faz com que a câmera seja direcionada para o personagem e logo após e feito um deslocamento da câmera para melhor posicioná-la. Um exemplo de implementação da câmera pode ser vista no quadro 23.

```
flyCam.setEnabled(false);
camNode = new CameraNode("CamNode", cam);          camNode.setControlDir
    (CameraControl.ControlDirection.SpatialToCamera);
characterNode.attachChild(camNode);
camNode.setLocalTranslation(new Vector3f(0, 3, 12));
camNode.lookAt(characterNode.getLocalTranslation(), Vector3f.UNIT_Y);
camNode.move(0, 0, -4);
```

Quadro 21 – Implementação da câmera

Caso haja a necessidade de desanexar a câmera do personagem, basta chamar o método detachChild, como mostrado no quadro 23.

```
characterNode.detachChild(camNode);
```

Quadro 23 – Desanexar câmera do personagem

Apendice C

TUTORIAL JMEGENERATOR

•Sobre o JMEGenerator

JMEGenerator é um projeto desenvolvido no Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação de Ely Fernando do Prado, sob orientação do Prof. Dr. Daniel Lucrédio.

JMEGenerator propõe o desenvolvimento de jogos orietado a modelos, permitindo a fácil geração de código através de suas interfaces intuitivas e seus transformadores de código.

O JMEGenerator é um plugin compatível com o jMonkeyEngine SDK. O download desse plugin pode ser realizado através do endereço: http://goo.gl/NDJtzy.

Após baixar o plugin e instalar o jMonkeyEngine SDK, vá no menu "Ferramentas-¿Plug-ins" e abra a aba "Obtidos por Download". Clique no botão "Adicionar Plug-ins" e selecione o arquivo "org-ufscar-modules-jmegen.nbm". Por fim clique no botão "Instalar" conforme mostrado na Figura C.1, e depois clique em "Avançar", aceite o contrato e clique em "Concluir".

•Criação de um novo Projeto

O JMEGenerator possui um template de projeto, o qual já cria uma infraestrutura preparada de um jogo. Para criar um novo projeto com esse template vá no menu "Arquivo-¿Novo Projeto". Selecione o item "JMEGenerator Project" que fica dentro do pacote "JMEGenerator", conforme mostra a Figura C.2.

Clique em "próximo", digite o nome do projeto e clique em "Finalizar". Será criado um projeto básico de jogo. Este projeto é separado principalmente pelos diretórios "Project

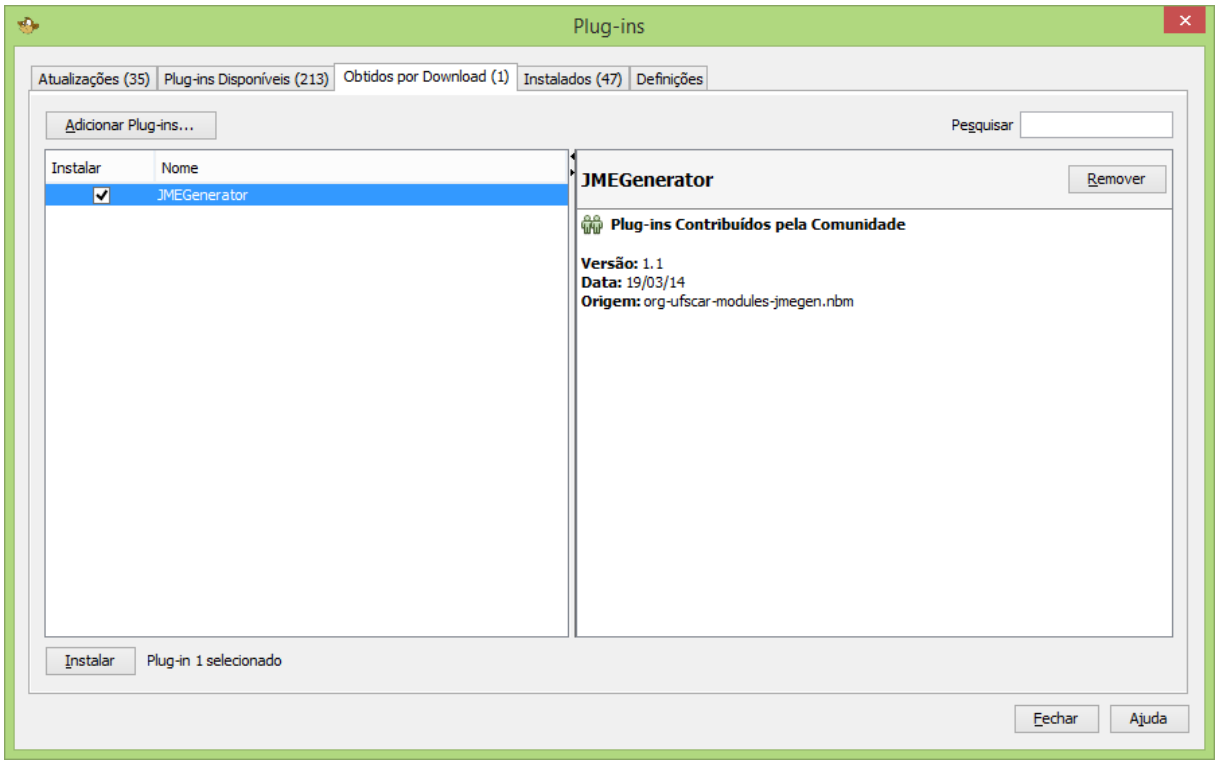

Figura C.1: Instalação JMEGenerator

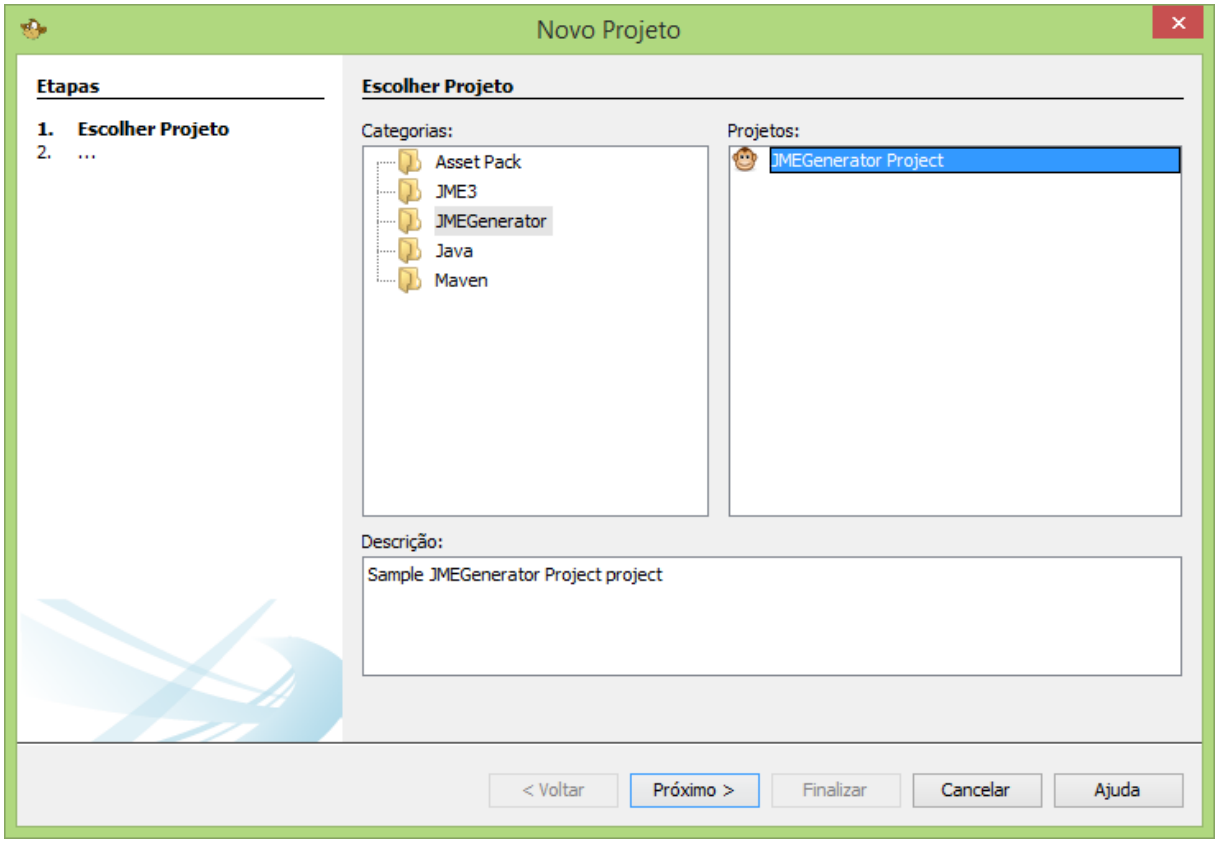

Figura C.2: Novo Projeto JMonkey Engine

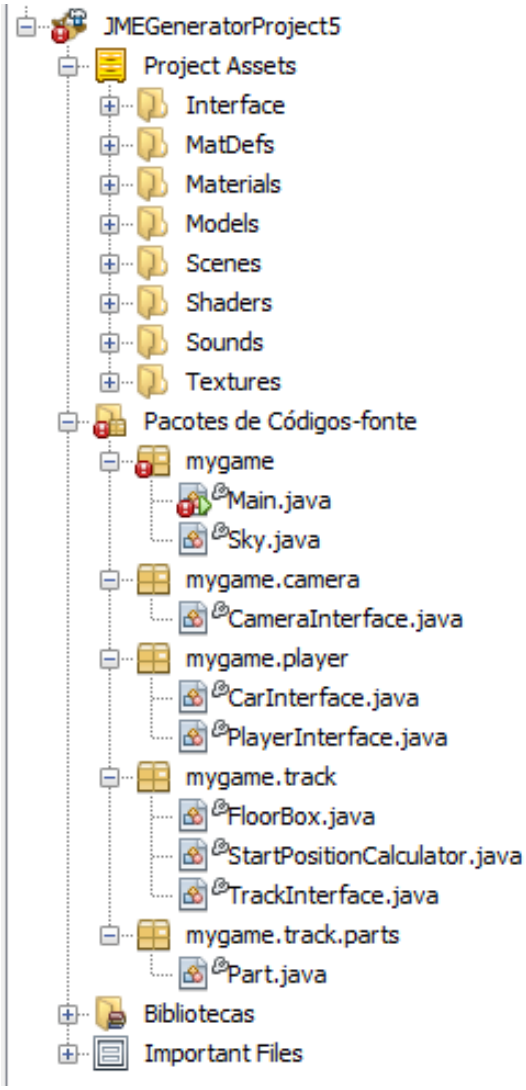

Figura C.3: Estrutura de Projeto Básico

Assets" e "Pacotes de Códigos-Fonte" conforme mostra a Figura C.3. Em "Project Assets" deverão ser colocados os arquivos de imagens, sons, objetos 3d, texturas, etc. Já em "Pacotes de Códigos-Fonte" deverão ser editados os código na linguagem Java, através de seus pacotes, classes, interfaces, etc.

Neste projeto criado pelo template do JMEGenerator, já existem alguns modelos 3d e texturas inseridas dentro de Assets. Estes recursos serão importantes para criar os exemplos demonstrados neste tutorial. Também existem um conjunto de pacotes no código fonte. O pacote mygame possui uma classe Main, que é a classe principal que inicializa o jogo e instancia todos os demais objetos utilizados no código. Nesse mesmo pacote também há a classe Sky que tem como funcionalidade desenhar o plano de fundo do cenário com um céu. O pacote mygame.camera contém uma Interface para câmera. O pacote mygame.player contém uma interface para jogador (PlayerInterface) e outra inter-

face para carro (CarInterface). Dentro do pacote mygame.track estão as classes FlooxBox, responsável por desenhar o chão do cenário e StartPositionCalculator, responsável por calcular a posição inicial do personagem jogável. Nesse mesmo pacote há uma interface utilizada para declaração de cenários (TrackInterface). Por fim temos o pacote mygame.track.parts, onde há uma classe cuja funcionalidade é abrir um modelo 3d, o qual será utilizado para composição do cenário.

Observe que a classe "Main" apresenta defeitos pela inexistência das seguintes classes:

• mygame.player.newPlayer;

• mygame.track.newTrack;

• mygame.camera.newCamera;

Estas classes deverão ser implementadas utilizando os modelos e transformadores descritos nos próximos capítulos.

•Câmera

Para codificação da câmera com JMEGenerator, primeiramente deve ser criado o modelo de câmera onde serão definidas todas as configurações da câmera. O ideal é que o modelo de câmera seja criado no pacote mygame.camera, portanto clique com o botão direito do mouse nesse pacote, vá em "Novo-¿Outros" e selecione o item "Camera.xml" no grupo "JMEGenerator", conforme mostra a Figura C.4.

Para continuar a criação do modelo de câmera, clique no botão "Próximo", defina o nome do seu modelo e clique em "Finalizar". De início, deixe o nome como "newCamera" para que não seja necessário fazer alterações na classe "Main".

Todo modelo do JMEGenerator é definido através de um arquivo XML, que possui opção para edição textual ou visual. Dê preferência para o editor visual, conforme mostrado na Figura C.5, onde as configurações do modelo são definidas mais facilmente.

Dentre os tipos pré-definidos de câmera no JMEGenerator estão os seguintes:

• Free: deixa a câmera livre, podendo ser controlada pelo mouse ou teclado independente da posição do jogador.

• Back: a câmera é posicionada sempre atrás do personagem jogável através de um nó.

• Over: a câmera é posiciona em cima do personagem jogável.

• First Person: a câmera é colocada em visão de primeira pessoa.

• Chase: um tipo especial de câmera disponível no JMonkeyEngine capaz de seguir o personagem jogável.

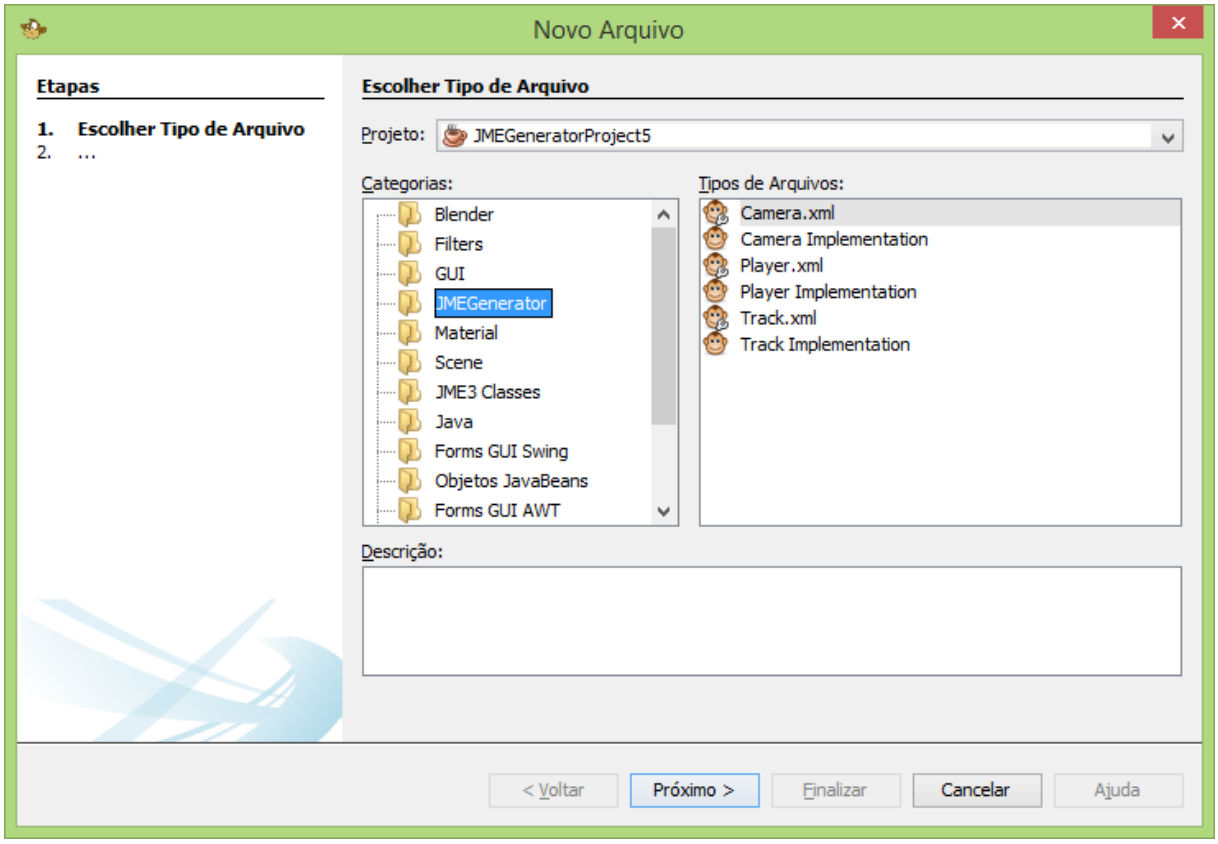

Figura C.4: Criação do modelo de câmera

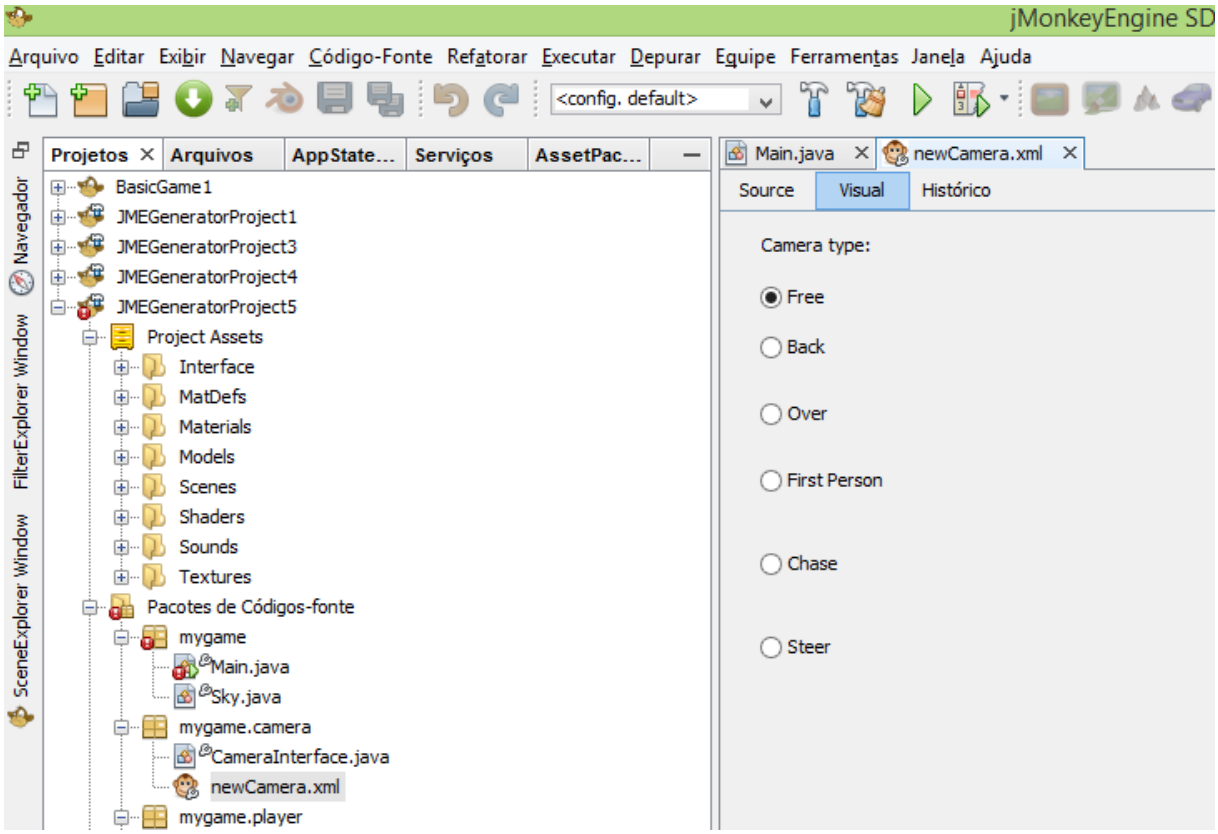

Figura C.5: Editor visual do modelo de câmera

• Steer: implementação de uma câmera muito similar à câmera Back, porém ela apresenta uma leve rotação quando o personagem é um veículo e este está em uma curva. Em jogos de corrida esta é a câmera mais indicada.

Após definir as configurações do modelo de câmera, é possível gerar o código através da implementação desta. Para isso, antes verifique se o modelo foi salvo e depois clique com o botão direito do mouse no pacote "mygame.camera", vá em "Novo-¿Outro" e selecione o item "Camera Implementation" da pasta "JMEGenerator". Clique no botão próximo, selecione o modelo de câmera que você deseja implementar e clique em "Finalizar". Neste momento é criada uma classe Java com a implementação da câmera, seguindo exatamente o mesmo nome de seu modelo. Caso haja alguma alteração no arquivo XML modelo, basta repetir o processo de implementação descrito neste parágrafo, e a classe Java será refeita.

•Player

Para criação do modelo do jogador, deve-se seguir passos semelhantes aos utilizados para criação do modelo de câmera. Primeiramente deve-se clicar com o botão direito do mouse no pacote "mygame.player" e ir no menu "Novo-¿Outro". Selecione o item "Player.xml" dentro da pasta "JMEGenerator", clique me Próximo, defina o nome do modelo e clique em "Finalizar". Em primeiro momento deixe o nome do modelo como "newPlayer", porém pode-se implementar modelos com quaisquer nomes que queira, fazendo assim com que seja possível implementar diferentes tipos de jogadores no mesmo jogo.

No editor visual do jogador há opções para selecionar entre jogador "Character" ou "Car". O jogador do tipo "Character" corresponde a um personagem jogável e poderão ser definidas as seguintes configurações para ele:

• Rotate Increase: número que corresponde ao incremento utilizado para rotacionar o jogador. Quanto maior este número, mais rápido será sua velocidade de rotação. O valor padrão é 0,01.

• Walk Forward: número correspondente ao incremento utilizado para movimentar o personagem para frente. Quando maior este número, maior será a velocidade em que o personagem andará. O valor padrão é de 0,3.

• Walk Backward: valor utilizado para incrementar movimentos do personagem para traz. O valor padrão é de 0,3.

Outro jogador disponível é o "Car" que corresponde a um veiculo. As opções disponível no editor visual para o jogador do tipo "Car" podem ser vistas na Figura C.6.

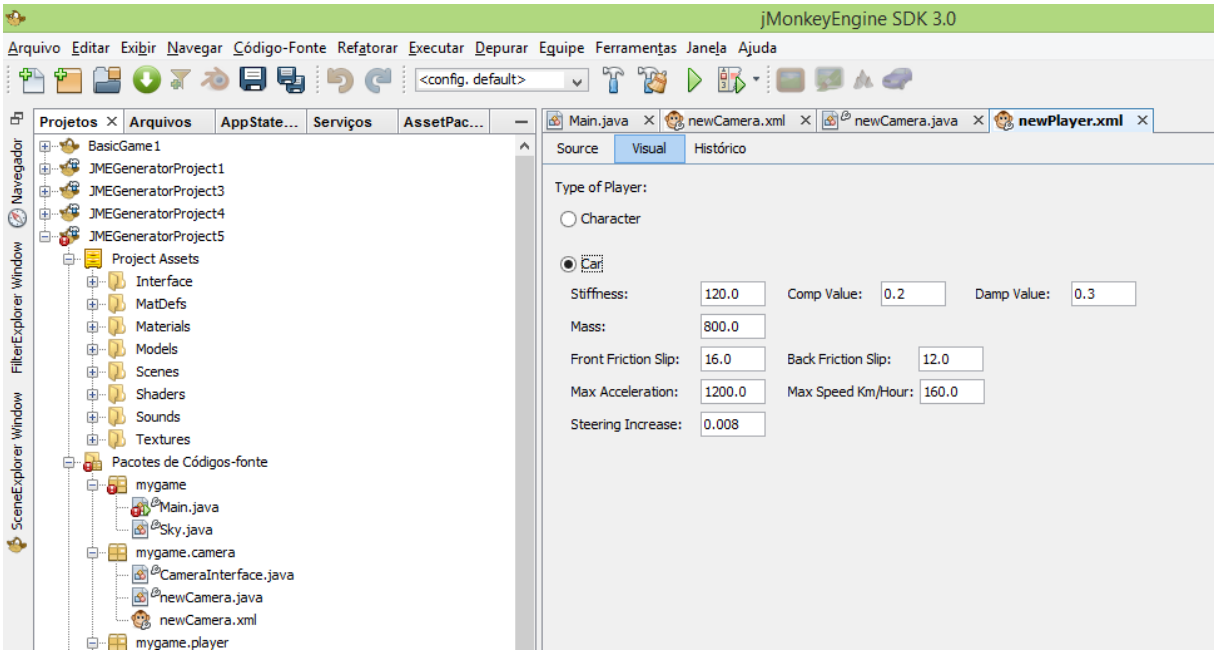

Figura C.6: Editor visual do jogador

As opções do jogador do tipo “Car” são:

• Stiffness: configuração do amortecedor

• Comp Value:

• Damp Value:

• Mass: peso do veículo.

• Front Friction Slip: valor de atrito dos pneus dianteiros. Um valor baixo fará o carro derrapar mais facilmente.

• Back Friction Slip: valor de atrito dos pneus traseiros.

• Max Acceleration: aceleração máxima do veículo.

• Max Speed Km/Hour: velocidade máxima do veículo em km/hora

• Steering Increase: valor de incremento da direção. Quanto maior este número, mais rápida será a direção do veículo.

Depois de definidas todas as configurações do jogador, basta fazer sua implementação clicando com o botão direito do mouse no pacote “mygame.player” e clicando o menu “Novo-¿Outros”. Selecione a opção “Player Implementation” da pasta “JMEGenerator”, clique no botão próximo, selecione o modelo do player que você deseja implementar e clique em “Finalizar”. Neste momento é criada uma classe Java com a implementação do seu jogador, seguindo exatamente o mesmo nome do modelo. Caso haja alguma alteração

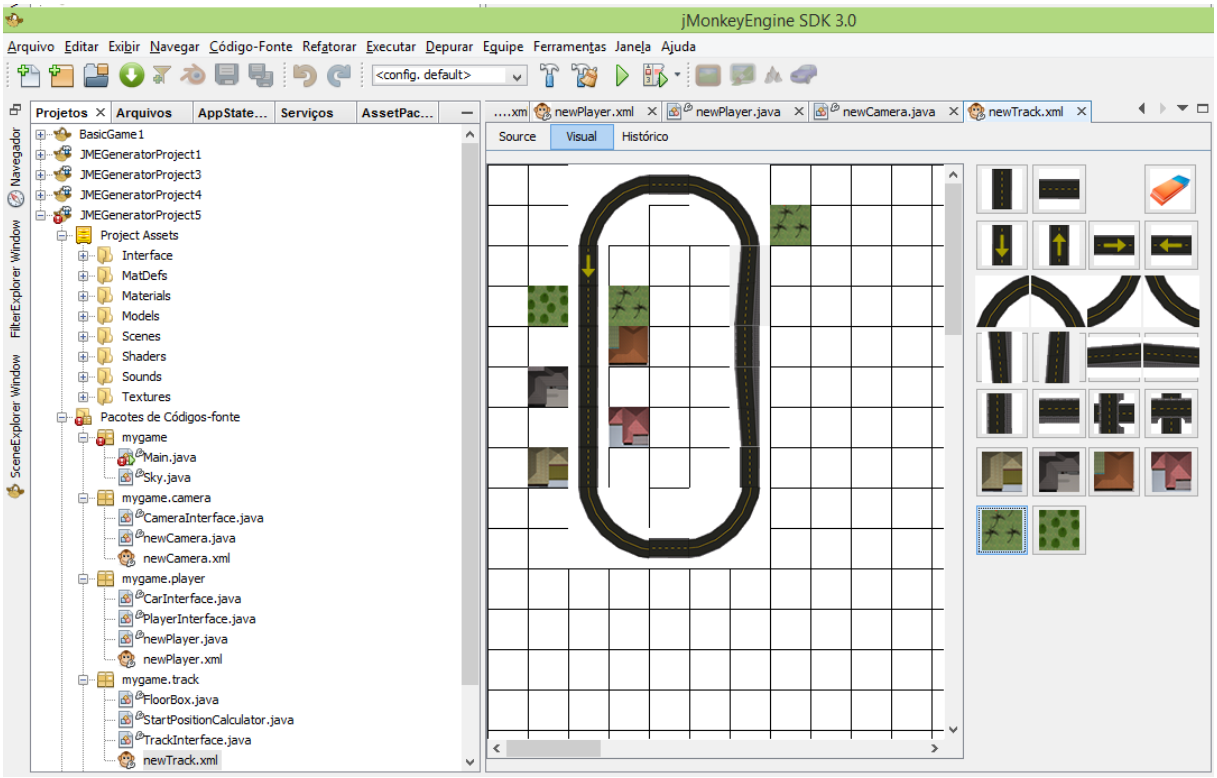

Figura C.7: Editor visual do modelo de cenário

no arquivo XML modelo, lembre-se de salvar o modelo e depois repetir o processo de implementação descrito neste parágrafo, e a classe Java será refeita.

- **Track**

Para criação do modelo de cenário, deve-se seguir passos semelhantes aos anteriores, clicando com o botão direito do mouse no pacote “mygame-¿track” e abrindo o menu “Novo-¿Outros”. Selecione o tipo “track.xml” dentro da pasta “JMEGenerator”, digite o nome do cenário e clique em “Finalizar”. O editor visual do cenário permite o desenho em duas dimensões que simula como deverá ficar o cenário em terceira dimensão, conforme mostrado na Figura C.7.

Depois de definidas todas as configurações do cenário, basta fazer sua implementação clicando com o botão direito do mouse no pacote “mygame.track” e clicando o menu “Novo-¿Outros”. Selecione a opção “Track Implementation” da pasta “JMEGenerator”, clique no botão próximo, selecione o modelo do cenário que você deseja implementar e clique em “Finalizar”. Neste momento é criada uma classe Java com a implementação do cenário. Caso haja alguma alteração no arquivo XML modelo, lembre-se de salvar o modelo e depois repetir o processo de implementação descrito neste parágrafo, e a classe Java será refeita. Podem ser definidos quanto modelos desejar e também podem ser feitas quantas implementações desejar, porém deve-se atentar para qual cenário está

sendo utilizando na classe Main.

Após a implementação da câmera, do jogador e do cenário, o jogo poderá ser executado sem problemas. Caso deseje fazer alguma alterações no código, siga o tutorial do JMonkey, mas fique ciente que a implementação de qualquer parte do jogo pelo JMEGenerator irá sobrescrever suas alterações manuais.

Apendice D

GUIA DE ORIENTAÇÕES GERAIS DO EXPERIMENTO

A ciência da engenharia de software tem dentre seus objetivos estudar as diferentes metodologias utilizadas para o desenvolvimento de software. Este experimento visa avaliar uma abordagem de desenvolvimento dirigido por modelos (MDD) para produção de jogos eletrônicos. Portanto, você não será avaliado ou julgado por aquilo que produziu durante o experimento, muito pelo contrário, você será um colaborador importante para avaliar a viabilidade da utilização desta abordagem de desenvolvimento para jogos, contribuindo positivamente para o meio acadêmico, cientifico e da indústria.

Visando um melhor aproveitamento de sua experiência, peço que siga os tutoriais fornecidos e também faça consulta na internet, porém não consulte os colegas que também estão fazendo o experimento afim de não haver "contaminação", já que nem todos farão o experimento seguindo a mesma abordagem.

Quaisquer dúvidas poderão ser sanadas a qualquer momento através do e-mail elyfprado@gmail.com

Peço que entreguem o resultado das 4 tarefas até o dia 06/06/2014.

Você será classificado pertencente ao grupo A ou grupo B. Aquele que pertencer ao grupo A irá receber o tutorial do JMonkeyEngine, e terá que realizar as 4 tarefas utilizando os recursos descritos no tutorial e internet. Aquele que pertencer ao grupo B irá receber o tutorial do JMEGenerator e terá que realizar as 3 primeiras tarefas com este tutorial, enquanto a tarefa 4 deverá ser realizada com o tutorial sobre o JMonkeyEngine.

Uma informação importante durante cada tarefa, está no tempo despendido para a realização

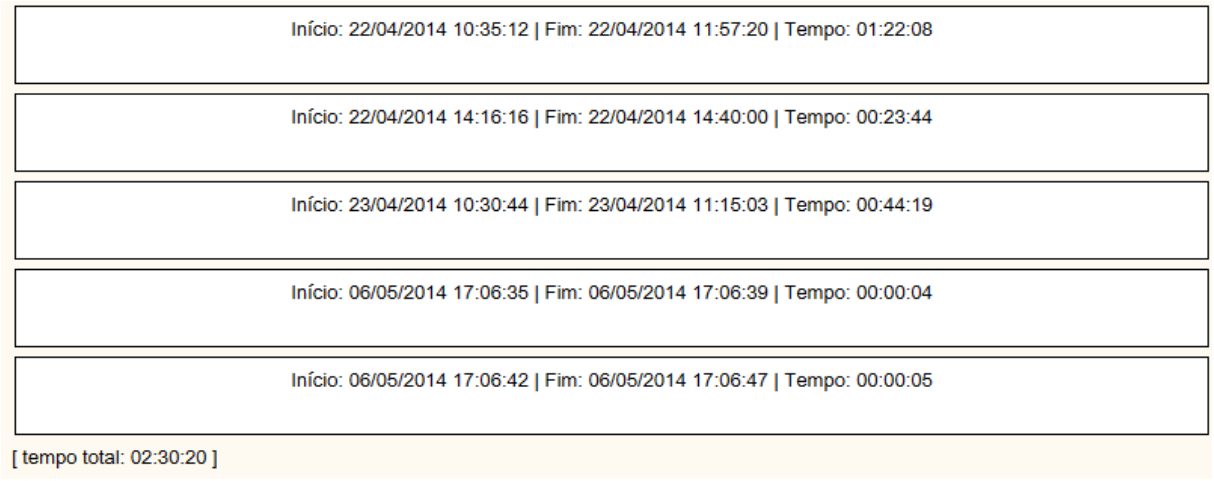

Figura D.1: Exemplo de tempo gasto no experimento

da mesma. Considere a contagem do tempo, inclusive da leitura dos tutoriais, codificação do projeto e teste. O tempo pode ser pausado quantas vezes for necessário para realização do projeto, desde que o tempo seja corretamente preenchido. Por exemplo, caso tenha começado a tarefa de manhã, tenha parado para almoçar, feito um pouco mais da tarefa a tarde e concluído no dia seguinte, poderia ter a tabela de tempo semelhante a mostrada na Figura D.1.

Para facilitar o preenchimento da tabela de tempo foi desenvolvido um sistema na nuvem capaz de contabilizar este tempo. O sistema pode ser acessado pelo endereço: http://trialclock.appspot.com . O usuário é o seu endereço de e-mail e a senha é ‘jme123’. Após efetuar o login você terá as funcionalidades mostradas na Figura D.2.

Após o término de cada uma das 4 tarefas, compacte o código fonte do seu jogo e envie para o site TrialClock. Por medida de segurança, também envie este mesmo arquivo para o e-mail elyfprado@gmail.com com as seguintes informações:

- Número da tarefa
- Tempo total gasto

O título do e-mail deve ser ”Tarefa01-SeuNome”, onde 01 corresponde ao número da tarefa seguido pelo seu nome.

Por fim após terminar cada tarefa, responda um pequeno questionário final da tarefa através deste endereço: http://goo.gl/uZvXOF

Reforço que suas informações serão divulgadas de forma anônima, preservando sua identidade. Muito obrigado pela contribuição!

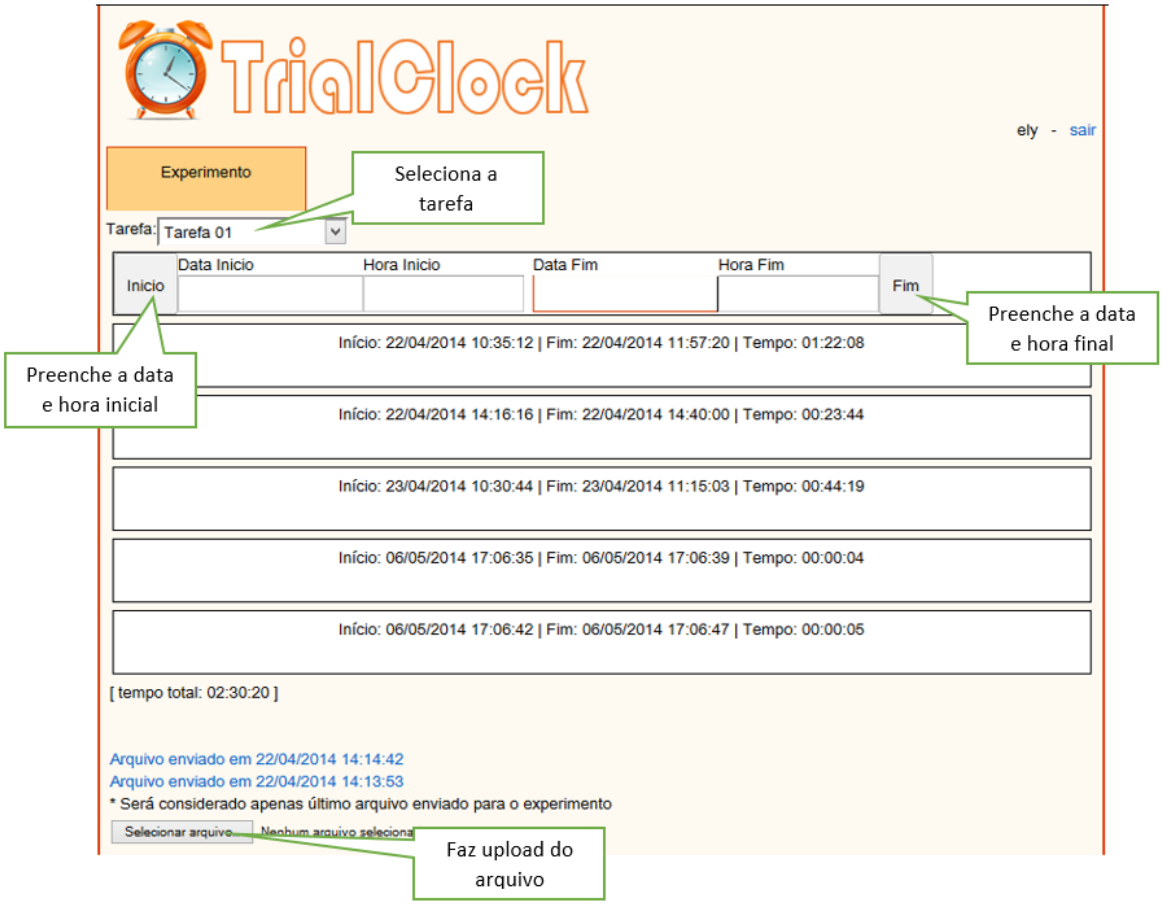

Figura D.2: Trial Clock

Apendice E

DESCRIÇÃO DETALHADA DAS TAREFAS DO EXPERIMENTO

•Descrição da Tarefa 1

–Objetivo

Desenvolver jogo de corrida com pista circular e com câmera atrás do veículo.

–Características esperadas

∗O personagem jogável deve ser um veículo;

∗O jogador poderá efetuar ações de acelerar, frear, virar para a direita e virar para a esquerda;

∗O veículo deve possuir as seguintes configurações:

·Peso de 1200 kg;

·Aceleração máxima de 1400;

·Velocidade máxima de 200 km/h;

∗A câmera deverá estar fixa no nó do veículo, estando posicionado atrás deste;

∗O cenário deve ser composto por elementos dispostos conforme a Figura E.1.

∗A posição inicial do veículo deve ser igual à mostrada na Figura E.1;

∗O cenário deve conter um chão com textura de grama em torno da pista;

∗O jogo deve exibir uma textura de céu no fundo;

∗A aparência geral do jogo deve ser igual à mostrada na Figura E.2;

•Descrição da Tarefa 2

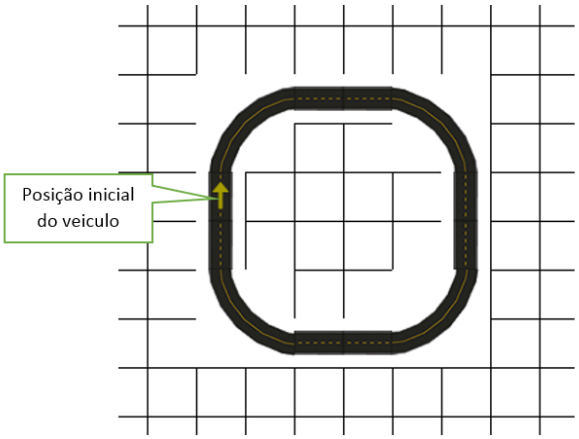

Figura E.1: Cenário 1

Figura E.2: Aparência do jogo 1

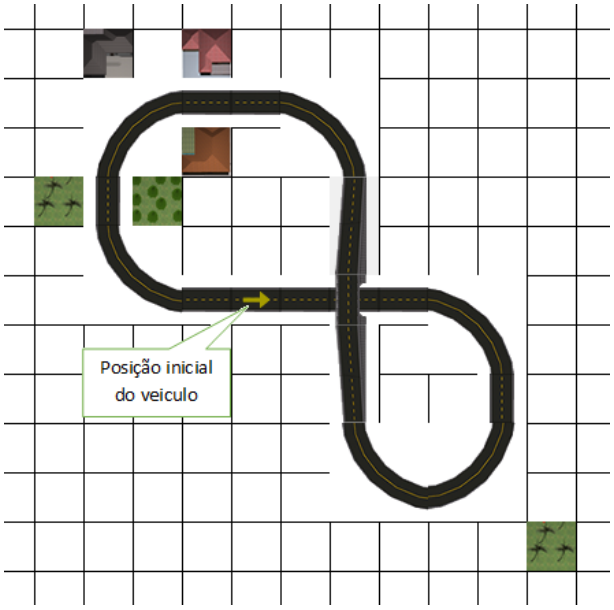

Figura E.3: Cenário 2

Figura E.4: Aparência do jogo 2

–Objetivo

Alterar cenário do jogo criado na tarefa anterior.

–Características esperadas

∗Altere o jogo criado na tarefa anterior para que ele fique com seu cenário formado por elementos dispostos conforme a Figura E.3.

∗A posição inicial do veículo deve ser igual à mostrada na Figura E.3;

∗O cenário deve conter um chão com textura de grama em torno da pista;

∗A aparência geral do jogo deve ser igual à mostrada na Figura E.4;

•Descrição da Tarefa 3

Figura E.5: Aparência do jogo

–Objetivo

Alterar jogador para um personagem.

–Características esperadas

∗Altere o jogo criado na tarefa anterior para que o jogador seja um personagem do tipo caracter;

∗O jogador poderá efetuar ações de andar para frente, andar para trás, girar para a direita, girar para esquerda e pular;

∗A aparência geral do jogo deve ser igual à mostrada na Figura E.5;

•Descrição da Tarefa 4

–Objetivo

Codificar a troca de personagem (veículo e caracter) em tempo de execução.

–Características esperadas

∗Altere o jogo criado na tarefa anterior para que seja possível efetuar a troca entre o personagem caracter pelo veículo durante a execução do jogo;

∗Quando o jogo for iniciado o caracter deve ser o personagem jogável, enquanto o veículo deve ser mostrado ao seu lado como personagem não jogável, conforme mostrado na Figura E.6.

∗O personagem jogável deverá receber os comandos do jogador e ter a câmera "amarrada" ao seu nó. A troca de personagem implica na reconfiguração da câmera e tratamento das entradas de teclado.

Figura E.6: Inicio do Jogo

∗As trocas de personagem deverão ocorrer ao se pressionar a tecla ‘X’ do teclado respeitando as seguintes regras: o A troca do personagem caracter para o veículo somente deverá ocorrer quando a distância entre os nós do caracter com o veículo forem menores que 10. o A troca do personagem veículo para o caracter poderá ocorrer a qualquer momento e em qualquer localização dos objetos.

∗Quando for feita a troca do personagem caracter para o veículo, deve-se efetuar a remoção do personagem caracter do jogo, para simular como se o personagem estivesse dentro do carro, não podendo ser mais visto.

∗Quando for feita a troca do personagem veículo para o caracter, deve-se readicionar o caracter no ambiente físico do jogo e posicioná-lo próximo à posição atual do veículo, para que ele possa ser visualizado no jogo novamente. Caso no momento da troca de personagem, o veículo esteja acelerando, sua aceleração deverá ser desativada.