DANIEL FERREIRA MONTEIRO ALVES

Projeto InVision Framework

Um *framework* para suportar a criação e uso de jogos no ensino

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Mestrado em Ciência da Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Eduardo Simões de Albuquerque

Goiânia
2011

DANIEL FERREIRA MONTEIRO ALVES

Projeto InVision Framework

Um *framework* para suportar a criação e uso de jogos no ensino

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Mestrado em Ciência da Computação, aprovada em 03 de Junho de 2011, pela Banca Examinadora constituída pelos professores:

Prof. Eduardo Simões de Albuquerque
Instituto de Informática – UFG
Presidente da Banca

Prof. Wellington Santos
Instituto de Informática – UFG

Profa. Judith Kelner
Centro de Informática – UFPE

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador(a).

Daniel Ferreira Monteiro Alves

Graduou-se em Ciência da Computação pela Universidade Federal de Goiás. Durante sua graduação, foi monitor de Sistemas Operacionais durante e publicou um artigo internacional, junto com seu orientador Prof. Eduardo Simões de Albuquerque, na IADIS - International Conference Applied Computing, em 2007, como fruto de seu Projeto Final de Curso. Após a conclusão da graduação, trabalhou como Desenvolvedor de Software para a empresa Fibonacci, num projeto envolvendo a construção de um executor de *Workflows*, voltando a universidade um ano após.

Dedico esse trabalho a todos os futuros alunos de Ciência da Computação, na esperança de que se torne uma ferramenta que alivie um pouco do peso, presente nas disciplinas desse curso.

Agradecimentos

Primeiramente, gostaria de agradecer a Deus por me permitir completar mais essa etapa na minha vida.

Agradeço ao meu orientador, professor Eduardo Simões de Albuquerque, ao qual eu trabalhei em conjunto desde minha graduação, sendo meu orientador pela segunda vez. Agradeço pela sua paciência, amizade e confiança; e pelas devidas cobranças e críticas realizadas quando necessárias.

Agradeço ao professor Humberto José Longo, por ser um ótimo professor, mas também um bom amigo, durante as conversas informais, fora de sala de aula. Ao pessoal da secretaria, em especial o Edir, o qual sempre foi muito prestativo e um bom amigo.

Agradeço a todos os meus amigos e colegas, desde os mais próximos até os mais distantes. Várias risadas foram dadas por conta de muita gente, especialmente junto de um grande amigo, Marcelo Ricardo Quinta, o qual além de dividir algumas horas me ajudando com artigos e estudando juntos, também já dividiu broncas junto comigo, por conta de imagens e algumas piadinhas em apresentações, na aula do professor Humberto Longo.

Também agradeço à Atlética Unificada da Computação, formada por algumas pessoas que se tornaram grandes amigos e colegas. Por conta dessa organização de alunos, momentos de stress foram aliviados, e vários amigos foram conhecidos: João Henrique, João Ozório, Rosana, Flávio, Diego (Bozo Star), Rafael (Bolinha), Marcos Vinícius, Jessica, Jeanyson (Maranhas), Jean, além de outros que terei de não citar para não aumentar muito a lista.

Agradeço a minha família, pelo apoio e por tentar entender aquilo que geralmente não conseguem, essa grande fixação e necessidade de estar em frente a um computador.

Por fim, agradeço a Universidade Federal de Goiás pela bolsa de estudos concedida durante o tempo do mestrado.

Qualquer tecnologia suficientemente avançada é indistinguível da mágica.

Arthur C. Clarke,
Terceira "Lei" da Tecnologia.

Resumo

Alves, Daniel Ferreira Monteiro. **Projeto InVision Framework**. Goiânia, 2011. 106p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

A quantidade de pessoas que ingressam no curso de Ciência da Computação a cada ano está diminuindo. Dentre aqueles que entram, poucos conseguem se formar. Há uma grande taxa de retenção e desistência, principalmente entre as disciplinas introdutórias de algoritmos e programação. Dessa forma, a utilização de jogos como fator motivacional é um assunto bastante estudado nos últimos anos, conseguindo bons resultados para tal problema. Contudo, para a aplicação de jogos na educação, numa abordagem construcionista, é requerido que alunos construam jogos. Para isso, várias ferramentas estão disponíveis com tal intuito, mas existe uma grande diferença quanto a usabilidade e recursos disponíveis, entre as ferramentas educacionais (que focam no ensino de programação) e as específicas para a criação de jogos. Deste modo, esse projeto propõe um *framework* para a construção de jogos, sendo apoiado por uma aplicação extensível através de *scripts*, que o permite ser adaptado para a utilização, em várias disciplinas no decorrer do curso, e não somente nas disciplinas introdutórias.

Palavras–chave

Jogos, Framework, Educação, Bindings

Abstract

Alves, Daniel Ferreira Monteiro. . Goiânia, 2011. 106p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

The number of people joining the Computer Science course, in the last years, is decreasing. Among those who enter, just a few are able to graduate, because there is a great retention rate and dropout, particularly among the introductory courses in algorithms and programming. The use of games as a motivational factor is a subject much studied in recent years, achieving good results for this problem. However, for the implementation of games in education, using a constructionist approach, students are required to build games. Several tools are available for this job, but there is a big difference in usability between the educational tools (that focus on educational programming) and those specific for creating games. This work proposes a framework for building games, being supported by an extensible application through scripts which allow it to be adapted for use in various disciplines throughout the course, and not only in introductory courses.

Keywords

Games, Framework, Education, Bindings

Sumário

Lista de Figuras **12**

Lista de Algoritmos **13**

Lista de Códigos de Programas **14**

1 Introdução **15**
- 1.1 Justificativa 16
- 1.2 Contribuições 17
- 1.3 Objetivos 17
- 1.4 Metodologia 18
- 1.5 Estrutura da dissertação 18

2 Jogos eletrônicos **20**
- 2.1 Funcionamento dos jogos internamente 21
- 2.2 Arquitetura dos jogos eletrônicos 22
- 2.3 Motores de Jogos 23
  - 2.3.1 Gênero de jogos 24
- 2.4 Criatividade na criação de jogos 27

3 Jogos na educação **28**
- 3.1 Abordagens para utilização de jogos com fim educacional 29
- 3.2 Ambientes para criação de jogos educativos 30
  - 3.2.1 Scratch 31
  - 3.2.2 GreenFoot 32
  - 3.2.3 Alice 33
  - 3.2.4 Avaliação dos ambientes para criação de jogos educativos 35
- 3.3 Ambientes específicos para a criação de jogos 35
  - 3.3.1 Unity3D 36
  - 3.3.2 Panda3D 37
  - 3.3.3 XNA Framework 38
  - 3.3.4 Outros motores e *frameworks* 38
  - Avaliação dos ambientes específicos para criação de jogos 38

4 Projeto InVision **40**
- 4.1 Preenchendo o vazio entre os ambientes de ensino e os motores de jogos especializados 41
  - 4.1.1 Diretrizes do projeto InVision 42
- 4.2 Arquitetura 43

4.2.1 A plataforma .NET e o Mono 44
4.2.2 Camada de Suporte Nativo 45
4.2.3 *Bindings* para bibliotecas nativas 47
4.2.4 InVision Framework 47
4.2.5 Protótipo extensível (Tutano) 48

5 Construindo o *framework* **50**
5.1 Interoperabilidade em .NET 50
5.1.1 Interoperabilidade entre C/C++ e C# com *P/Invoke* 51
5.2 Construção da camada de Suporte Nativo e camada de Bindings 54
5.2.1 Geração de Wrappers com o SWIG 54
5.2.2 Geração manual de *bindings* 57
5.2.3 *Blittable types* 59
Dependência de contexto para tipos booleanos 59
5.2.4 *VTable* Artificial 60
5.2.5 Problemas com ponteiros e conversões 63
5.2.6 *Layout* de objetos em memória 64
5.3 Construção do *framework* 65
5.3.1 Extensões utilitárias 66
5.3.2 Abstrações matemáticas 67
5.3.3 Modelo de Aplicação para Jogos e Simulações 68
5.3.4 Fluxos de execução customizáveis 69
5.3.5 Suporte a *Scripting* em várias linguagens 73
5.3.6 *API* simplificada para Áudio e Renderização 74
5.3.7 *API* Persistência de objetos 74
5.3.8 *API* Comunicação via Rede de Computadores 75
5.4 Construção do protótipo extensível Tutano 75
5.5 Análise do processo de construção 77

6 Usando jogos em Ciência da Computação **79**
6.1 O Uso do *framework* Invision 79
6.1.1 Escolhendo um fluxo de execução adequado 80
6.1.2 Animações controladas 81
6.2 O ensino de programação 82
6.2.1 Diferentes Paradigmas de Programação 82
6.2.2 Redes e Comunicação via Internet 84
Testando modelos de comunicação 84
Usando MMORPG em Redes 84
6.3 Computação Gráfica 85
6.4 Programação de agentes inteligentes 85
6.4.1 Teste de algoritmos computacionais com visualização 86

7 Conclusão **88**
7.1 Trabalhos futuros 89

Referências Bibliográficas **91**

A Caso de estudo “Karel The Robot” **100**
A.1 A definição do jogo 100
A.2 Papeis e responsabilidades 101
A.3 O projeto e arquitetura da aplicação 102
A.3.1 Passo 1 - O *script* de criação 102
A.3.2 Passo 2 - Interface de Programação para o Karel 103
A.3.3 Passo 3 - Definindo o fluxo de execução 106
A.4 Configuração e execução do Tutano 106

Lista de Figuras

2.1 Modelo generalizado da arquitetura de um jogo. 22
2.2 Jogo Call of Duty para PlayStation 3. 24
2.3 Jogo Red Dead Redemption para PlayStation 3, XBox 360 e PC. 25
2.4 Megaman para PlayStation. 25
2.5 Need for Speed Carbon para PC. 26
2.6 Snapshots de animações em uma luta, no jogo Street Fighter 4 para PlayStation 3. 26

3.1 Programação visual usando o Scratch. 31
3.2 Execução de um programa no GreenFoot. 33
3.3 Programação visual usando o Alice. 34

4.1 Logomarca proposta para o Framework. 40
4.2 Arquitetura do Projeto InVision. 44

5.1 Construção de *wrappers* com vtable artificial. 61
5.2 Empacotamento de dados em memória 64
5.3 Módulos presentes no *framework* InVision 66
5.4 Macro Estados em um Jogo 69
5.5 Diagrama de classes demonstrando o modelo da aplicação e seus estados 70
5.6 Fluxo de execução em um jogo 70
5.7 Delegação de chamadas para um arquivo de script 73

6.1 Representação de uma cidade no jogo City Ville do Facebook. 86

A.1 Rascunho do cenário para o mundo de Karel. 101

Lista de Algoritmos

2.1 Execução de um jogo 21

Lista de Códigos de Programas

5.1 Funções que encapsulam métodos de um objeto em C++ 53
5.2 Criação e manipulação de um dicionário em C++ 55
5.3 Criação e manipulação de um dicionário em C# 56
5.4 Função nativa em C++ que retorna um dicionário 56
5.5 Problema com a conversão de ponteiros em C++ 63
5.6 Exemplo de um fluxo de execução customizado 72
5.7 Exemplo de um GameState implementado em Ruby 77
6.1 Exemplo de algoritmo em Boo, simulando o C-Sheep 81
A.1 Exemplo de um *script* de criação de um mundo de Karel, em Boo. 102
A.2 Programação para o Karel. 103
A.3 Parte do código do controlador do Karel 104
A.4 Parte do código do Karel 105
A.5 Configuração do Tutano para o Karel 106

CAPÍTULO 1

Introdução

Nos últimos anos, o número de ingressos no curso de Ciência da Computação está diminuindo globalmente, mesmo com a necessidade por estes profissionais no mercado. Dentre os alunos que ingressam, muitos não terminam o curso [12, 11]. Alguns pesquisadores já descrevem a atual situação como uma crise nessa área [40, 9, 82]. Por curiosidade, assim como a proporção de ingressos que se formam diminui, a quantidade de mulheres que procuram tal curso também está diminuindo [55].

Os motivos atribuídos para esse cenário são vários, mas há uma convergência que aponta para as disciplinas introdutórias, aplicadas durante o primeiro ano no curso, relativas ao ensino de Algoritmos e Programação [32, 12, 10]. Tais disciplinas estão dentre aquelas com maior índice de retenção e desistência [10].

Para conter esse problema, é necessário transformar o ensino de programação em algo divertido e desafiador, características de jogos, estimulando a motivação do estudante e fazendo-o criar interesse próprio em buscar por tal conhecimento. Dessa forma, devido à grande acessibilidade e aceitação pelos jovens [53], jogos estão sendo introduzidos no ambiente de ensino.

A possibilidade de se utilizar jogos na educação vem sendo objeto de pesquisa há muitos anos, como em [58]. Entretanto, criar um jogo não é uma tarefa fácil, principalmente se o aluno, que está iniciando o aprendizado de algoritmos e programação, for o responsável por desenvolvê-lo. A criação de jogos é uma tarefa interdisciplinar, envolvendo outras áreas além da Computação [25].

Aspectos artísticos e computacionais devem ser imaginados, organizados e sincronizados de forma que funcionem perfeitamente, inclusive aproveitando ao máximo dos recursos oferecidos pelo hardware que irá executá-lo. Do ponto de vista computacional, quanto maior a qualidade das animações, interação entre objetos e com o usuário, efeitos e processamento físico, maior será a necessidade de bons componentes para a realização de determinadas tarefas, inclusive um bom hardware.

Para isso, vários ambientes estão sendo desenvolvidos com o propósito de facilitar o uso de jogos na educação [18, 57, 9, 88, 103]. Cada ambiente segue uma abordagem de trabalho, focando em um dos itens abaixo:

- Construção de animações (como o Scratch [67] e o Alice [22]), utilizando linguagens visuais para apresentar conceitos de programação;
- Demonstração de conceitos de programação e aplicação de algoritmos (GreenFoot [49]);
- Resolução de problemas em jogos já construídos através de extensões (RoboCode [46, 46] e "The Meadow" [9]), ou utilizando de *APIs* bem definidas ("Karel de Robot" [81, 14]);
- No *design* e construção de jogos completos, através de um *framework* com componentes pré-definidos [105], como a Panda3D [79].

Nesse trabalho, é apresentado um novo *framework*, InVision, para auxiliar na construção e utilização de jogos em disciplinas de Ciência da Computação. Tal ambiente provê componentes prontos para a construção de jogos, assim como uma aplicação semi-pronta e extensível, que permite o desenvolvimento de protótipos de forma rápida.

1.1 Justificativa

A utilização de jogos na educação pode permitir ganhos, tanto em aspectos motivacionais, quanto em estímulos sensoriais, psíquicos e motores [29].

Nos últimos anos, ferramentas como o Scratch e o Alice ganharam um papel importante na educação e ensino de programação, com suas linguagens de programação visual e desprovidas de erros de sintaxe. Contudo, a maioria desses ambientes e metodologias foca somente no ensino de programação, como ponto chave de sua aplicação.

A simplicidade de tais ambientes acaba por restringir aquilo que pode ser criado, consequentemente também restringem a criatividade do estudante. Além disso, tais restrições acabam por torná-los inadequados para disciplinas posteriores, que dependem de recursos extras.

Entretanto, só prover recursos ou utilizar ferramentas, como as citadas anteriormente, também pode ocasionar no fracasso da abordagem. Anderson e McLoughlin [9] afirmam que é extremamente difícil manter o interesse dos jovens da geração atual e que o uso de tecnologias 3D, que utilizem de efeitos visuais mais sofisticados e de qualidade, garantem melhores resultados.

Da mesma forma, utilizar de motores jogos e outros *frameworks* especializados será inadequado. Tais ferramentas, muitas vezes, focam no trabalho do artista e em prover meios para construir um produto de qualidade, mas o usuário deve se adaptar à ferramenta, assim como a seu processo de desenvolvimento. Dessa forma, sua utilização iria no mínimo necessitar de uma adaptação para o contexto educacional.

Assim, este projeto realiza uma ponte entre as duas abordagens, preenchendo o espaço entre as ferramentas com propósito educacional e as especializadas na criação de

jogos. Sempre almejando sua utilização como uma ferramenta de auxílio, seja no ensino de programação ou outras disciplinas mais avançadas do curso.

1.2 Contribuições

De maneira geral, considerando os objetivos presentes na seção 1.3, as principais contribuições desse trabalho são:

- **O desenvolvimento de um *framework* para jogos, multiplataforma, baseado em componentes especializados** capazes de dar às criações uma qualidade próxima das encontradas em jogos comerciais, através de componentes de qualidade como o motor de renderização Ogre3D [25]. Tal *framework* foi construído sobre uma arquitetura própria, definida para suportar vários níveis de interoperabilidade de código e ser portável à Sistemas Operacionais como Windows, Linux e MacOSX;
- **A criação de *bindings* portáveis**, escritos em C#, para bibliotecas nativas de alto desempenho e voltadas para simulações e jogos, como a Ogre, OIS e FMod. Tais bibliotecas somente possuíam *bindings* criados utilizando *Managed* C++ e que eram dependentes do Sistema Operacional Windows;
- **A criação da técnica denominada "*vtable* artificial"**, criada para a construção de *wrappers* onde há necessidade de simular a herança direta entre uma classe em C++ e outra em C#. Incluindo um suporte seletivo a métodos que devem realmente sobrescrever a *vtable* artificial, no objeto nativo.
- **A criação do Tutano, uma aplicação extensível para a rápida prototipação de jogos**, suportando várias linguagens de *script*;
- **A reprodução de uma abordagem antiga**, mas funcional, para o ensino de programação, como caso de uso, construída sobre o Tutano. Tal abordagem foi definida originalmente por [81], visando o ensino de algoritmos e programação utilizando uma ideia baseada na movimentação de um robô.

1.3 Objetivos

Tendo em vista a discussão apresenta na seção 1.1, os principais objetivos do projeto InVision *Framework* são:

- **Criar uma arquitetura de software para desenvolvimento de jogos, apoiada por um *framework* próprio**. Tal *framework* deve ser portável aos Sistemas Operacionais: Windows, Linux e Mac OS X;

- **Criar uma aplicação, Tutano**, sobre tal arquitetura que seja extensível através de *scripts* e bibliotecas dinâmicas. Tal aplicação deve dar suporte para estudantes criarem seus jogos com *scripts* de forma rápida e simples;
- **Desenvolver um protótipo funcional**, como caso de uso, para demonstrar sua viabilidade, utilizando o Tutano. Tal protótipo recria o mundo de "Karel The Robot" numa perspectiva moderna, em gráficos 3D e interativa.

1.4 Metodologia

A arquitetura e organização do InVision se baseia em tecnologias específicas para o desenvolvimento de jogos, como motores de jogos e componentes especializados (motor de renderização, reprodução de áudio, física etc).

Além disso, se baseia na ideia do motor de jogo Unity3D [99] e do *framework* Microsoft XNA, onde uma máquina virtual para .NET é utilizada, dando a base para a construção da aplicação e otimizando a execução de *scripts* [30, 31]. Algumas ideias quanto ao molde das *APIs* e facilidades de uso foram retiradas do *framework* Panda3D [79].

Adicionalmente, funcionalidades de componentes especializados, de alta qualidade, para renderização (Ogre [78]), áudio (FMOD [34]) e física (Bullet [27]), devem ser utilizados. Uma avaliação sobre a escolha de tais componentes está em 4.2.2.

O *framework* foi escrito em C#, sendo apoiado em bibliotecas de componentes escritas em C++. A criação dos *bindings*, seus problemas e técnicas utilizadas, também são exploradas capítulo 4.

Como uma forma de facilitar o uso do *framework* e aumentar sua aceitação, foi criado o suporte para execução de *scripts* em várias linguagens, tais como: C#, F#, Boo, Python, Ruby.

1.5 Estrutura da dissertação

O restante desta dissertação está estruturado da seguinte maneira. O capítulo 2 apresenta conceitos sobre o que são jogos eletrônicos e plataformas de execução, conceitos arquiteturais e funcionalidades comuns a esses, assim como a separação em gênero de jogos.

O capítulo 3 apresenta como os jogos são utilizados, com propósitos educacionais. Uma análise crítica é realizada sobre as ferramentas, seus propósitos e limitações.

O capítulo 4 apresenta as visões e diretivas sobre as quais esse projeto foi proposto. Apresentando a arquitetura proposta, o motivo pela escolha do Mono como máquina virtual, e outros detalhes sobre as camadas e suas responsabilidades.

O capítulo 5 apresenta detalhes sobre a construção do *framework* e dos componentes envolvidos, focando principalmente nos problemas encontrados durante a parte de maior esforço do trabalho, a construção dos *bindings*. Também é apresentada a técnica de *vtable* artificial em 5.2.4.

O capítulo 6 apresenta várias abordagens que utilizam jogos na educação, focando em diversas disciplinas e conteúdos, e como adaptá-las para a utilização com os artefatos produzidos por este trabalho.

Por fim, o capítulo 7 conclui a dissertação, discutindo sobre as contribuições, resultados e apresentando propostas para trabalhos futuros.

CAPÍTULO 2

Jogos eletrônicos

Jogos eletrônicos são aplicações de tempo-real, ou programas de computador, executadas sobre uma plataforma, onde o foco principal é prover o entretenimento ao usuário [25]. Tais aplicações constituem uma poderosa indústria cultural que evolui constantemente em seus aspectos tecnológicos [13]; são universos virtuais frequentados por pessoas de várias idades, principalmente abaixo dos 35 anos [29].

Atualmente, temos jogos em diversas plataformas, sejam elas:

- Computadores pessoais;
- Consoles especializados (conhecidos simplesmente como Video Games, no Brasil: Nintendo Wii, Sony PlayStation 3, Microsoft XBox 360 etc);
- Celulares;
- Algumas televisões e outros dispositivos.

"A maioria dos usuários já gastou várias horas se divertindo com algum jogo" [13], em alguma plataforma. Entretanto, seu desenvolvimento não é uma tarefa simples e requer várias técnicas e conhecimento multidisciplinar para se construir um título de qualidade. Dessa forma, surge a necessidade de formar equipes com programadores, *designers* gráficos, criadores de estórias, artistas etc [44].

Os jogos possuem várias características peculiares e precisam de técnicas específicas para o seu desenvolvimento. A forma e a ordem como os componentes visuais são mostrados, por exemplo, é inerente a cada título e gênero de jogo (seção 2.3.1), assim como sua interação com o usuário.

Em jogos mais recentes, a interação deixou de ser somente via teclados e mouses ou *joysticks*. Atualmente, buscando criar novas formas de interação, a indústria de jogos utiliza recursos de Realidade Aumentada para prover inovações quanto ao modo de jogar [90]. Como fruto dessas pesquisas existem controles baseados em movimentos, como: o controle WiiMote para Nintendo Wii [4], o PlayStation Move para PlayStation 3 [2] e o Kinect (Projeto Natal) para XBox 360 [1].

2.1 Funcionamento dos jogos internamente

Os jogos são aplicações que podem variar muito quanto a complexidade de sua criação. O que define o quão complexo será um jogo são: a interatividade que se quer prover ao usuário, o gênero de jogo escolhido (seção 2.3.1) e o quão longe se quer "levar" o jogador (relativo ao tamanho do jogo, que pode ser medido pela quantidade de fases, horas gastas para se cumprir um objetivo etc).

Apesar da variedade quanto a complexidade de sua criação, a maioria dos jogos possui uma estrutura de execução tão simples quanto a apresentada no algoritmo 2.1:

Algoritmo 2.1: Execução de um jogo

```
Abrir janela
executando ← true
enquanto executando faça
|  Limpar desenhos
|  Desenhar conteúdo
|  Verificar entrada do usuário
|  se Usuário apertou a tecla Esc então
|  |  executando ← false
|  fim
fim
Fechar janela
```

Tal estrutura de execução é dividida nas seguintes fases:

Inicialização Nessa fase todos os recursos computacionais e artísticos (imagens, modelos etc) são inicializados e carregados;

Execução em *loop* infinito Nessa fase, o jogo será uma repetição de passos. Cada iteração do *loop*, em geral, representa um quadro (*frame*) que será gerado e mostrado ao usuário. Além disso, pode-se tratar eventos dos dispositivos de entrada e executar a lógica do jogo, antes de realmente renderizar o quadro para o usuário;

Finalização Por fim, quando o jogo estiver finalizando, deve-se liberar a memória e os recursos obtidos durante o processo.

A complexidade por trás dessa simples estrutura de execução está em como realizar a iteração. Um determinado gênero de jogo pode requerer a organização, em camadas ordenadas, das imagens que serão desenhadas em tela; como e quando ler uma entrada do usuário; como iterar um subsistema (executá-lo, passo-a-passo, por *frame* da aplicação) etc. Contudo, essa organização interna depende da arquitetura escolhida para implementar o jogo.

2.2 Arquitetura dos jogos eletrônicos

Cada jogo, assim como um software qualquer, possui suas próprias necessidades quanto a algoritmos, dados e interação. Entretanto, algumas partes e funcionalidades são fundamentais e podem, com um pouco de trabalho, serem desacopladas das funcionalidades do próprio jogo, e utilizadas como componentes reutilizáveis.

Quanto maior o nível de separação de conceitos e funcionalidades, maior a quantidade de componentes e subsistemas, geralmente separados em camadas. A figura 2.1 mostra um modelo de abstração generalizado da arquitetura de um jogo, que está estruturado de acordo com as seguintes camadas:

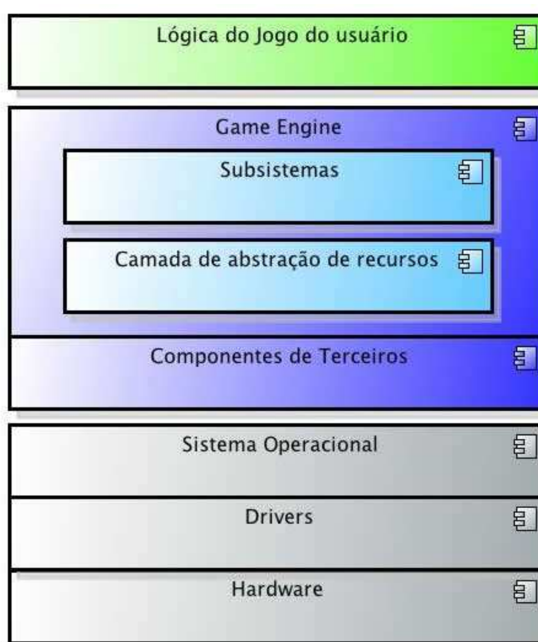

Figura 2.1: *Modelo generalizado da arquitetura de um jogo.*

Plataforma (em cinza) - define a plataforma de destino do jogo. É dividida em:

- Hardware - basicamente é a configuração de hardware de destino;
- Drivers - implementações dependentes do sistema operacional e do hardware de destino, que simplesmente expõem as funcionalidades de um dispositivo através de uma API;
- Sistema Operacional - pode ser realmente um sistema operacional, como Windows, Linux, Mac OS X; ou ser simplesmente uma fina camada de abstração para dar acesso ao hardware e drivers diretamente, como uma API (comum em video-games de sexta geração ou anterior);

Subsistemas de suporte (em azul) - engloba todos componentes e funcionalidades comuns a vários jogos.

Implementação do jogo (em verde) - define todo o código responsável por efetivamente criar o jogo e suas interações, como:

- Definições e manipulação de objetos e cenários;
- Implementação de inteligência artificial para alguns objetos;
- Animações, texturas, *shaders*, arquivos de som e música, *meshes* etc.

Atualmente, existem várias bibliotecas que dão suporte para a construção da segunda camada (subsistemas de suporte). Assim um desenvolvedor pode:

- Selecionar um conjunto de bibliotecas com fins específicos, como: renderização, áudio, inteligência artificial, controle de dispositivos de entrada etc. Assumindo então a responsabilidade de realizar a integração entre as diversas *APIs*, ou;
- Selecionar um motor de jogo existente que dê suporte para a construção de um jogo, baseado num gênero específico (seção 2.3.1). Dessa forma, simplesmente utiliza-se das funcionalidades providas pelo motor e assume-se que são componentes fechados, pronto para uso.

2.3 Motores de Jogos

Motores de jogos (*Game engines*) são componentes de software que proveem funcionalidades especializadas para o desenvolvimento de jogos. Sua função é *"agrupar funções fundamentais para o desenvolvimento de jogos, que podem se estender da interação com os periféricos de entrada até a renderização dos cenários e personagens"* [16]. Comumente, definem também um processo e modelo de projeto para o desenvolvimento, principalmente quando utilizando uma ferramenta de edição.

Sua complexidade pode ser maior do que a necessária para se criar o próprio jogo, mas pode garantir grandes facilidades para o desenvolvimento [44].

A quantidade de componentes providos pelo motor de jogo influencia diretamente na sua manutenção, assim como em sua complexidade. Em geral, como acontece em grandes softwares, existe uma divisão por camadas e subsistemas, sendo alguns:

- Subsistema de renderização, 2D ou 3D;
- Subsistema de áudio, 2D ou 3D;
- Subsistema para controle de física;
- Subsistema para carregamento de modelos de objetos (*meshes*) e animações.
- Subsistema para controle de janelas e entradas do usuário, como: mouse, teclados, joysticks e outros dispositivos;

Adicionalmente, ainda pode-se ter:

- Um camada de abstração para recursos do sistema operacional, como:
  - Threads;
  - Entrada e Saída de dispositivos específicos;
  - Manipulação de arquivos;
  - Componentes para comunicação via rede de computadores.
- Componentes especializados para persistência de dados;
- Componentes especializados para comunicação via rede, implementando algum mecanismo de comunicação específicos, como: comunicação direta, peer-to-peer, em estrela etc;
- Componentes e algoritmos para Inteligência Artificial, como: *path finder*, *redes neurais* etc;
- Um editor visual para construção interativa das cenas;
- Ferramentas para processamento e otimização de modelos de objetos e animações.

Quanto mais completo o motor de jogo menos recursos externos serão necessários e, consequentemente, o foco do desenvolvedor estará direcionado a criação da lógica de sua própria aplicação, seu jogo. É importante salientar que a maioria dos motores de jogos são orientados para um gênero específico de jogo [44]. Isso acontece porque cada gênero requer otimizações específicas para garantir um bom desempenho.

2.3.1 Gênero de jogos

Existem vários gêneros de jogos e cada um requer um conjunto de funcionalidades e otimizações específicas, como descrito a seguir:

Figura 2.2: *Jogo Call of Duty para PlayStation 3.*

Tiro em Primeira Pessoa - tal gênero se apoia principalmente sobre a câmera, que deve conseguir prover respostas rápidas e ser otimizada para ambientes abertos e amplos. Os modelos e principalmente as animações possuem qualidade alta, mas o

personagem principal é considerado ser a própria câmera, tendo no máximo seus membros (geralmente braços) à mostra. A checagem de colisões, realizada entre objetos do cenário e o personagem do jogador, em geral, pode conter leves erros, que passarão despercebidos pelo jogador; mas que dá uma sensação de flutuação sobre o chão, quando em movimento. Exemplo na figura 2.2.

Figura 2.3: *Jogo Red Dead Redemption para PlayStation 3, XBox 360 e PC.*

Terceira Pessoa - esse gênero se distingue do anterior, por mostrar o personagem por inteiro, estando a câmera sempre focada nesse. Os modelos podem conter menos detalhes do que no jogo em primeira pessoa, mas as animações e interações com o ambiente devem ser mais detalhadas. Além disso, deve existir um cuidado adicional com a câmera, pois objetos que estão entre a câmera e o personagem não devem atrapalhar a visão do jogador; esses podem ser removidos, colocados em transparência ou mesmo fazer com que a câmera mude de posição. Exemplo na figura 2.3.

Figura 2.4: *Megaman para PlayStation.*

Plataforma - são basicamente jogos em terceira pessoa, onde o personagem terá que superar obstáculos (plataformas, rampas, escadas etc) durante seu caminho, através

de pulos, por exemplo. Boa parte dos jogos desse gênero são jogos em 2D que fizeram, e ainda fazem, bastante sucesso, como: Sonic, Mario, Prince of Persia, Flashback, Metroid, Megaman etc. As necessidades são basicamente as mesmas do gênero anterior, com suporte a elementos que proveem níveis diferentes de chão e altura para o personagem. Exemplo na figura 2.4.

Figura 2.5: *Need for Speed Carbon para PC.*

Corrida / Simuladores - são variações de jogos em primeira e/ou terceira pessoa, onde a física e principalmente o aspecto visual são especialmente trabalhados. A física deve ser ponderada com a qualidade visual dos modelos e do ambiente, mas deve prover a interatividade a mais próxima do real possível. O uso de volantes e outros dispositivos com *Force Feedback* - reprodução da força exercida por um dispositivo real em um dispositivo de entrada, como a trepidação nos volantes de um carro em alta velocidade - pode auxiliar na reprodução da simulação. Exemplo na figura 2.5.

Figura 2.6: *Snapshots de animações em uma luta, no jogo Street Fighter 4 para PlayStation 3.*

Jogos de luta - são jogos onde o cenário em geral é muito pequeno e somente possuem dois personagens que devem duelar (em jogos mais novos podem haver varições quanto ao número de personagens na tela). A qualidade do cenário pode variar, mas o foco são os lutadores, movimentos e checagem de colisões. Um bom mecanismo para processamento da entrada do usuário (via um *joystick*, por exemplo) é

necessário; principalmente para o acúmulo e reprodução de *combos* (sequências de golpes e movimentos). Com relação a física, pode-se tentar uma reprodução mais fiel ou simplesmente representar as partes de contato por grandes caixas, evitando um processamento muito pesado. Exemplo na figura 2.6.

Outros - existem mais gêneros de jogos que podem ser inovações ou apenas variações dos citados anteriormente, com por exemplo: estratégia em tempo-real, esportes, RPG (*Role-Playing Game*), jogos de tabuleiro, MMOG (*Massively Multiplayer Online Game*) etc. Este último gênero, teve uma variação que se destacou muito nos últimos anos para a plataforma PC, os MMORPG (*Massively Multiplayer Online Role-Playing Game*), uma variação do gênero RPG, onde jogadores, representados por avatares, entram em um mundo virtual para duelarem entre si, formarem guildas e conquistarem territórios, armas e equipamentos[86].

2.4 Criatividade na criação de jogos

A criação de um jogo deve ser realizada visando a interatividade do jogador, mas sem esquecer de um aspecto chave, a diversão.

Entretanto, criar jogos interativos é uma tarefa possivelmente mais fácil, ainda que não seja completamente, do que a criação de um jogo divertido. Isso acontece porque aspectos humanos estão envolvidos e cada pessoa possui a sua própria definição do que é divertido, de acordo com seu próprio gosto.

Um desenvolvedor de jogos deve se preocupar com o conteúdo, público-alvo, faixa etária destinada, elementos de animação e interação, estória do jogo, música e efeitos sonoros etc [25].

Assim, para a criação de um jogo adequado, para determinado público, será necessário exigir do desenvolvedor muito mais do que somente conhecimento técnico e computacional, mas também artísticos e humanos. Tal atividade faz com que os jogos sejam mais do que simples aplicações, mas um misto de conhecimento técnico, intelectual e criações artísticas. O que pode ser devidamente explorado como uma abordagem de ensino, como visto no capítulo 3.

CAPÍTULO 3

Jogos na educação

O sistema educacional busca alternativas para enfrentar um problema: a falta de motivação dos alunos para estudarem e se dedicarem a disciplinas de seus cursos. Tal problema é considerado atualmente sério para o curso de Ciência da Computação, internacionalmente, pois existe um alto grau de retenção e desistência dentre os alunos recém ingressos [9, 11].

Visando uma alternativa para amenizar ou resolver esse problema, pesquisadores e educadores, há alguns anos, estão utilizando os jogos como ferramentas educativas. A aposta está em aumentar a motivação dos alunos e direcionar sua atenção para os problemas tratados pelas disciplinas do curso, enquanto trabalham sua criatividade [41, 11].

A atividade de brincar, por si só, desenvolve a imaginação e a criatividade [18] e o simples ato de jogar pode se tornar uma atividade de aprendizado, pois é possível desenvolver habilidades importantes [29], tais como:

- Percepção e reconhecimento espacial;
- Desenvolvimento de lógico indutiva;
- Desenvolvimento cognitivo e científico;
- Representação espacial.

Ainda que tal abordagem seja temida por alguns [29, 40] e exista resistência quanto a sua aplicação [18, 101], a utilização de jogos na educação já apresenta pontos positivos [41, 24, 42, 9, 36], como:

- Há uma melhora significativa na motivação dos alunos quanto a participação e concentração em sala de aula;
- Há um estímulo próprio do aluno para pesquisar e procurar conteúdo correlato, sem a necessidade da participação direta do educador;
- Segundo Gackenbach [36], as habilidades trabalhadas e suas intensidades variam de acordo com os gêneros dos jogos utilizados (seção 2.3.1), podendo serem selecionadas de acordo com as necessidades, como descrito abaixo:

  - Coordenação motora, técnicas manuais e reflexos são trabalhados principalmente em jogos do gênero *Arcade*, *Role-Play* e Plataformas;
  - Atenção e concentração são forçadas principalmente em jogos do gênero de Lutas em 3D, mas também pelos gêneros: Plataforma (2D), alguns de esportes e jogos modernos de ação e aventura;
  - A capacidade de planejamento é especialmente trabalhada, ao máximo, em jogos de ritmo / música, esportes e corrida; sendo quase nula em jogos do gênero de Ação / Aventura;
  - O "despertar fisiológico" (aumento nos batimentos cardíacos, de respiração e pressão sanguínea) é fortemente trabalhado em jogos de Luta, Plataformas (2D) e Ação / Aventura;
  - A capacidade para processamento de ações sucessivas são trabalhadas fortemente em jogos dos gêneros: Tiro em Primeira Pessoa, Esporte e MMORPG;
  - A capacidade para processamento paralelo (ações ocorrendo simultaneamente) são bem trabalhadas em jogos dos gêneros: Estratégia em Tempo-Real, RPG e MMORPG.

- Vários tópicos do curso de Ciência da Computação podem ser abordados durante a criação de jogos, segundo Clua [24], como: Álgebra Linear, Inteligência Artificial, Computação Gráfica, Redes, Simulações em Tempo-Real, Interação Homem-Computador, Engenharia de Software, dentre outras;

3.1 Abordagens para utilização de jogos com fim educacional

Segundo Kafai [53], pode-se utilizar jogos na educação com duas abordagens, uma instrutivista e outra construtivista. Na primeira, o estudante assume a posição de jogador, ganhando conhecimento ou domínio sobre determinado assunto na medida que avança no jogo, cumprindo suas metas. Na segunda, o estudante assume o papel de um *Designer* de jogos, devendo criar um jogo.

Ambas abordagens requerem diferentes visões e proveem diferentes resultados, mesmo que a diferença entre ambas possa parecer sutil. A instrutivista foca no entretenimento, enquanto provê conhecimento, mas limita o estudante às ações que podem ser tomadas durante o jogo, isto é, ao que o personagem, por exemplo, pode realizar. A construtivista, por outro lado, dá mais liberdade ao estudante, permitindo que esse dite as regras do seu jogo enquanto o cria.

Segundo Gerosa [42], a abordagem construtivista estimula o trabalho em equipe e organização quando aplicada em grupos e também contribui para a desenvoltura do

estudante para resolver problemas. *"Durante o processo de criação do jogo, os estudantes aprendem como ser um bom projetista: como conceituar um projeto, como fazer uso dos recursos disponíveis, como persistir e achar alternativas para problemas inesperados e como colaborar com os outros."* [87].

O desenvolvimento de jogos que sejam divertidos e ainda efetivamente instrutivos é uma tarefa difícil e complexa, mas que causa motivação dentre os envolvidos [11]. Adicionalmente, a abordagem construtivista também contribui para que o estudante possa ter fluência digital, que basicamente é a capacidade de criar conteúdo digital e não somente consumi-lo [88].

Hicks [50] também apresenta uma hipótese, quanto ao modo de se utilizar jogos na educação, de que a socialização também deve ser estimulada e deve estar presente nas atividades dos estudantes. Segundo ele, os jogos conseguem criar laços de "camaradagem" mais fortes que qualquer outra mídia. Além disso, a competição e torcida entre os próprios amigos serviriam para aumentar a motivação de sua utilização.

3.2 Ambientes para criação de jogos educativos

Boa parte das pesquisas que visam a aplicação de jogos na educação, em um curso de Ciência da Computação, focam na fonte inicial da dificuldade dos alunos, disciplinas como "Introdução a Algoritmos e Programação" (IAP). A ideia por trás de tal escolha é diminuir a taxa de retenção e desistência, aumentando a motivação e o interesse do aluno, assim que esse entre na faculdade.

O aprendizado de algoritmos é considerado algo fundamental para o curso, e futuramente para aprender programação, mas disciplinas como IAP estão entre as que mais reprovam e retém alunos [10]. As causas são diversas [51, 43, 84]:

- Existe falta de competência para resolução de problemas, por parte dos alunos;
- Os métodos pedagógicos são inadequados ao estilo de aprendizagem dos estudantes;
- As linguagens de programação possuem sintaxes complexas ou não adequadas para estudantes, num primeiro momento. Por exemplo, a utilização de Java ou C++ como linguagem de programação inicial se torna inadequada para novatos, pois tais linguagens não foram criadas com propósito educacional, em contraste com outras como Python, Logo, Eiffel e Pascal [83];
- Existem diferenças entre os alunos: origens, experiências e habilidades;
- O nível de abstração envolvido é muito alto para que o estudante assimile facilmente;
- A situação é agravada pela *"impossibilidade de um acompanhamento individualizado do aluno"*;

- O ato de programar é uma atividade dinâmica, mas usualmente trabalhada com materiais estáticos.

A geração atual de jovens, denominada como "geração Plug&Play" por Anderson em [9], tem a tecnologia como um fator constante em suas vidas. Tal geração está acostumada a aparatos eletrônicos, ao compartilhamento de informações e a internet [54]. Dessa forma, é necessário utilizar-se dessas novas tecnologias para conseguir despertar o interesse e aumentar a motivação desses alunos, pois métodos tradicionais de ensino unicamente já não darão mais resultados tão significativos [40].

Atualmente, alguns ambientes para criação de conteúdo interativo estão sendo utilizados na tentativa de minimizar as diferenças entre alunos e os problemas citados anteriormente. Os principais ambientes são apresentados, abaixo:

3.2.1 Scratch

Ferramenta para programação visual, onde *scripts* de execução são criados agrupando peças de código, como blocos de LEGO (figura 3.1). Tal aplicação age como um incentivador para estimular a fluência digital, provendo um meio simples para o aprendizado de algoritmos e construção de conteúdo digital interativo.

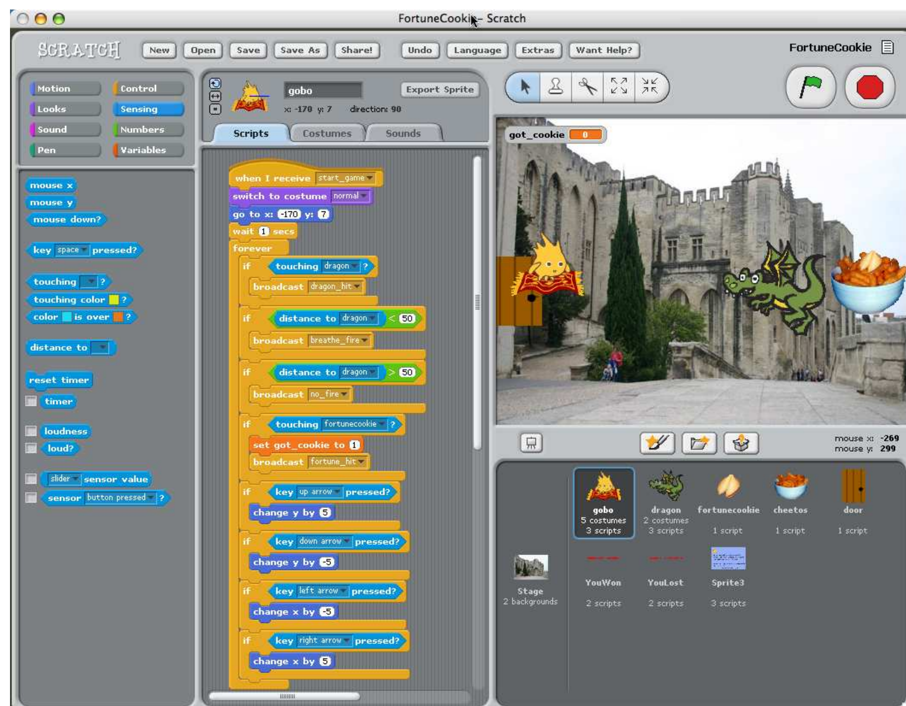

Figura 3.1: *Programação visual usando o Scratch.*

Pontos fortes:

- A sintaxe deixa de ser um problema para o usuário, pois é formada por blocos encaixáveis, que lembram peças de LEGO [88];
- É possível visualizar o fluxo do *script*, visto que as estruturas da linguagem formam um grande bloco;
- É possível aprender sobre tratamento de eventos, pois o bloco de código só será executado se estiver “escutando” um evento de um determinado objeto no cenário;
- Conceitos rudimentares de Computação Gráfica 2D são apresentados ao usuário;
- Pode-se compartilhar o conteúdo criado na internet com outros usuários, assim como realizar um trabalho em grupo [103, 88];

Pontos fracos:

- Somente aplicações muito simples em 2D podem ser criadas. Além disso, o sistema de renderização não é muito eficiente;
- O tamanho do mundo virtual é muito pequeno para representar alguns tipos de visualizações, como uma rede de objetos ou a visualização de arrays [103];
- Códigos redundantes são necessários para realizar algumas tarefas, pois não há um modo para se criar métodos [103];
- Eventos do teclado e do mouse podem ser tratados, mas de forma rudimentar, sem muita interatividade;
- Não possui suporte geral para dispositivos de entrada e saída;
- Não possui suporte para redes;
- O foco do Scratch, segundo os próprios desenvolvedores, é prover algo para aprendizagem de programação e construção de raciocínio lógico [88]. Sendo assim, não é uma ferramenta que deve ser usada para construção de jogos mais sérios ou mais complexos, nem foca em detalhes de como as aplicações devem ser construídas [100].

3.2.2 GreenFoot

Ferramenta para criação de jogos em 2D, diretamente em código, com uma linguagem similar ao java [33] (figura 3.2). O uso de uma linguagem de programação textual para a programação pode ser vista tanto como um ponto positivo, quanto um ponto negativo.

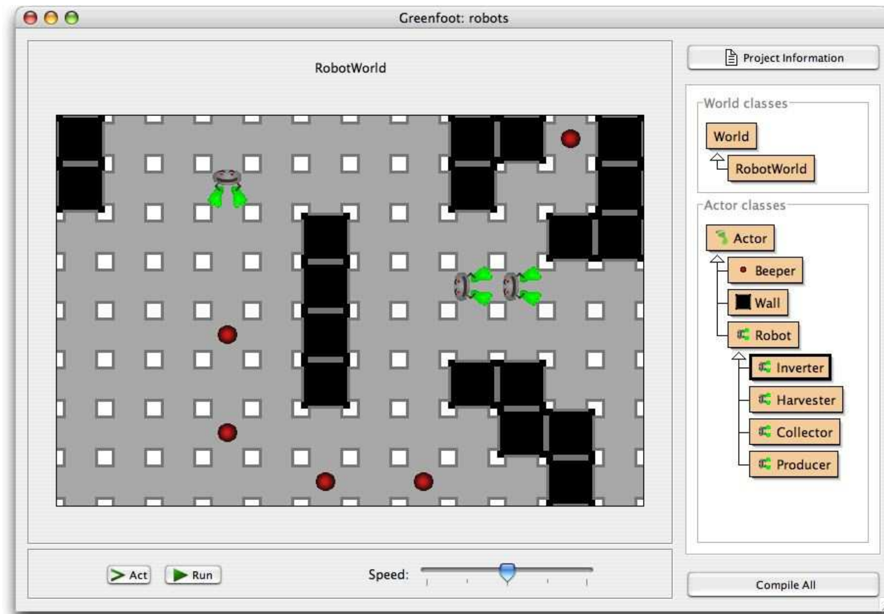

Figura 3.2: *Execução de um programa no GreenFoot.*

Pontos fortes:

- É direcionado para mundos e objetos visuais, assim como o Scratch. O rastreamento de problemas se torna quase que imediato [100];
- Existe a possibilidade de se criar métodos definidos pelo próprio usuário, diferentemente do Scratch. Isso permite que estudantes não fiquem presos somente aos recursos oferecidos pelo ambiente de programação [100];
- São apresentadas aos usuário processos comuns no desenvolvimento de software, como: compilação, execução e depuração passo-a-passo. Tais processos são ocultados no Scratch [100];

Pontos fracos:

- Sua linguagem requer cuidados com sintaxe e se torna não adequada para crianças [100].
- A execução da aplicação não é fluída (ocorre pequenas pausas durante a execução);
- Não é possível criar jogos com gráficos mais avançados e apelativos para o usuário.

3.2.3 Alice

Ferramenta para criação interativa de animações 3D (figura 3.3). O usuário programa toda a movimentação de objetos no cenário e aprende conceitos introdutórios

de orientação a objetos [33]. A análise abaixo foi realizada com base na versão 2.0 do Alice.

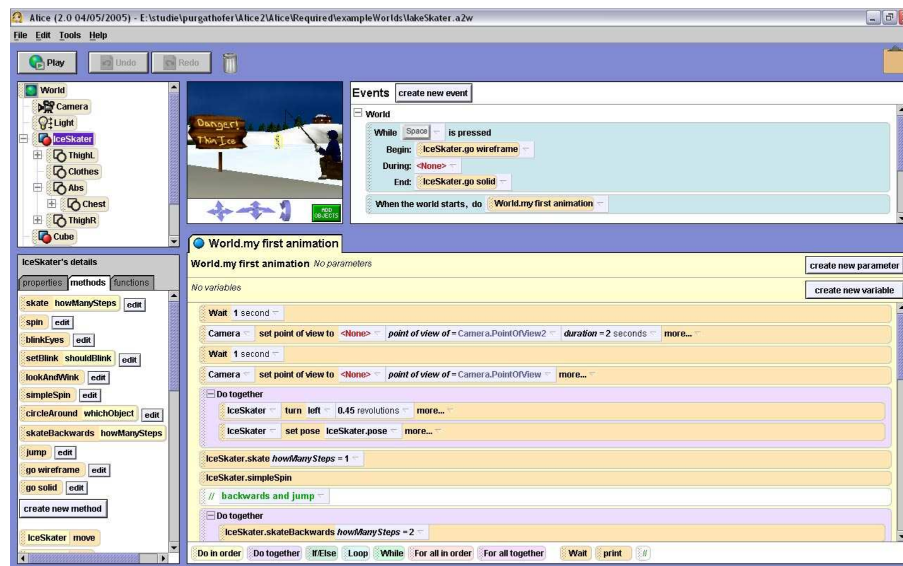

Figura 3.3: *Programação visual usando o Alice.*

Pontos fortes:

- Provê um ambiente de desenvolvimento com uma interface *drag-and-drop*, evitando erros de sintaxe [19, 33];
- As animações construídas são animações em 3D, consideradas mais estimulantes para o aluno da geração “Plug&Play” [71, 9, 103];
- O foco da aplicação é criar animações e tal atividade é considerada criativa e recompensadora, além de estimular a motivação do aluno [19];

Pontos fracos:

- A aplicação é demasiadamente pesada e consome muita memória RAM, ocasionando o uso constante do disco para paginação de memória [19];
- Possui suporte fraco para herança. *“Classes são extendidas modificando uma instância; essa instância modificada pode então ser salva a parte, como um exemplar de uma subclasse”*. [19];
- A aplicação pode travar quando trabalhando com uma quantidade relativamente grande de objetos [103];
- Não possui suporte geral para dispositivos de entrada e saída [19];
- Não possui suporte para redes [19];
- Comparativamente, o Alice é menos estável que o Scratch [103];
- A transição entre a programação visual do Alice e o modo texto, de linguagens de programação como Java e C# não é uma tarefa fácil; principalmente por conta dos erros de sintaxe.

3.2.4 Avaliação dos ambientes para criação de jogos educativos

Uma apresentação das ferramentas citadas e seus contextos de aplicação está em [33]. Tais ferramentas trazem bons resultados quanto a auxiliar e tornar o aprendizado de programação divertido. Kereki afirma que para o aprendizado inicial de algoritmos e programação o Scratch é uma ferramenta viável; ressaltando também que para tal contexto, ele é uma opção melhor, pela sua simplicidade, do que o Alice e o GreenFoot [32].

Entretanto, tais ambientes de desenvolvimento não resolvem o problema do aprendizado de linguagens de programação, apesar de serem boas alternativas para o aprendizado dos conceitos envolvidos [57]. Em todas as metodologias de uso, tais aplicações estão presentes somente no início do curso, como uma espécie de pré-introdução à Computação.

3.3 Ambientes específicos para a criação de jogos

Na medida que a experiência do programador (aluno) aumenta, e o rigor das disciplinas também, os ambientes (seção 3.2) se tornam inadequados e acabam limitando a criatividade, pelas suas restrições.

Continuar forçando o aluno a utilizar tais ferramentas quando este já alcançou suas limitações é algo que possivelmente irá desmotivá-lo. Sendo um efeito colateral indesejado e que deve ser evitado. Nesse ponto, pode-se recorrer ao ensino de linguagens de programação como C, C# e Java.

Assim, para continuar aplicando jogos na educação, principalmente utilizando da abordagem construcionista (ver 3.1), é necessário realizar uma troca de tais ferramentas por algo mais poderoso. Na internet, é possível encontrar vários componentes espalhados para a criação de um jogo, como:

- Bibliotecas de renderização e tratamento de imagens;
- Bibliotecas para reprodução de áudio;
- Bibliotecas para simulação de física em tempo-real;
- Bibliotecas com algoritmos eficientes e estruturas de dados complexas para a construção da lógica do próprio jogo;
- Bibliotecas para auxiliar na criação de Inteligência Artificial;
- Bibliotecas para comunicação via rede.

Entretanto, organizar e sincronizar todos os componentes em uma única aplicação não é uma tarefa simples [44]. Além disso, muitas vezes requer conhecimentos que o aluno somente irá adquirir em disciplinas do final do curso.

Dessa forma, utilizar um *framework* ou um motor de jogo que disponha de funcionalidades já prontas, se torna uma opção melhor do que realizar a montagem manualmente.

Existem diversas bibliotecas e motores de jogos que servem a tal propósito. Em geral, um motor de jogo completo engloba todas as funcionalidades descritas anteriormente e ainda provê ferramentas, como editores, para a organização, visualização e montagem do ambiente virtual, programação de *scripts* e publicação para uma ou mais plataformas [44, 80].

Abaixo, uma descrição e análise sobre algumas opções bem conhecidas, por entusiastas e profissionais na criação de jogos, é apresentada:

3.3.1 Unity3D

A Unity [99] é um motor de jogo completo, provendo abstrações para funcionalidades diversas, incluindo renderização e física. Seu editor é capaz de importar diversos tipos de modelos, incluindo arquivos do Blender (uma ótima opção de código livre para criação de modelos 3D).

Pontos fortes:

- Possui um editor (disponível para Windows e Mac OS X), que permite a publicação da aplicação para as plataformas: Windows, Mac OS X, Android, iOS, Nintendo Wii, XBox 360 e PlayStation 3;
- Provê um sistema de renderização independente de plataforma, que abstrai detalhes da implementação (seja OpenGL [45] ou DirectX [63]);
- Provê um sistema de luzes diversificado com reprodução de sombras em tempo real e efeitos especiais de lentes;
- Suporte para geração de terrenos e vegetação;
- Suporte a física através do motor de física PhysX;
- Suporte a *scripting*, através do Mono [75] (plataforma para .NET open-source). Suportando 3 linguagens: C#, Javascript e Boo. Também provê integração com o ambiente de desenvolvimento integrado, MonoDevelop;
- Suporte à comunicação via rede e sincronização de estados em objetos distribuídos. Além da possibilidade de se utilizar dos recursos da plataforma .NET para comunicação via protocolos como UDP e TCP/IP.

Pontos fracos:

- É um motor de jogo proprietário, de código fechado; apesar de haver licenças para uso gratuitas;

- É direcionada para o trabalho profissional e em equipe. Existe uma distinção entre o papel do artista e do programador, sendo que o editor é direcionado principalmente para o artista;
- Só existe um meio para extensão e trabalho em cada uma das linguagens disponíveis, que algumas vezes requer conhecimentos avançados, que pode inviabilizar sua utilização em disciplinas do começo do curso.

3.3.2 Panda3D

Panda [79] é um motor de jogo gratuito, disponibilizado e mantido pela Universidade Carnegie Mellon. Provê suporte para programação em C++ e Python.

Pontos fortes:

- É utilizada para prototipação rápida de jogos, sendo suportada por uma Universidade de renome;
- Provê boa parte dos recursos necessários para o desenvolvimento de jogos, assim como alguns disponibilizados pela Unity3D;
- Provê suporte para programação em duas linguagens: C++ e Python;
- Seu motor de física é disponibilizado através da biblioteca ODE [76], uma biblioteca de código aberto;
- A criação de jogos, principalmente em Python, é rápida e relativamente fácil de se trabalhar. Entretanto, requer que certa organização do projeto seja seguida;
- Provê suporte para as plataformas: Windows, Linux e Mac OS X.

Pontos fracos:

- Não existe um editor visual, assim como na Unity3D (o que não chega a ser exatamente um problema, mas perde-se na facilidade de gerenciamento de conteúdo);
- Apesar dos cálculos físicos serem realizados pela ODE, essa é considerada menos estável do que os motores PhysX ou Bullet [27], causando erros no cálculo da gravidade e outros efeitos físicos;
- Boa parte dos recursos estão implementados somente em Python. Sendo assim, apesar de disponibilizar uma interface para desenvolvimento em C++, na prática, somente utilizando Python é que pode-se utilizar todos os recursos do ambiente;
- O desempenho de Python (mesmo em sua última versão, a 3.0) é menor que o desempenho de um programa executado em C# com o Mono (informação obtida analisando uma tabela de comparação disponível no

site `http://shootout.alioth.debian.org/u32/benchmark.php?test=all&lang=csharp&lang2=python3`).

3.3.3 XNA Framework

O XNA *framework* [66] é um ambiente disponibilizado pela Microsoft, apoiado sobre o Microsoft Visual Studio, que provê serviços e funcionalidades para a criação de jogos e aplicações interativas.

Pontos fortes:

- Suportado diretamente pelo .NET *framework*, permite que todos os recursos de programação deste estejam disponíveis ao usuário;
- Abstrai detalhes de programação do DirectX. Entretanto, permite que *shaders*, pequenos programas executados sobre a *GPU*, em HLSL sejam executados;
- Organiza o projeto e gerencia os recursos permitindo uma checagem em tempo de compilação;
- Permite que jogos sejam criados para Windows, XBox 360 e Windows Phone sem grandes alterações no código. Em geral, basta uma recompilação do projeto definindo a plataforma de destino.

Pontos fracos:

- Não provê um motor de física integrado;
- Provê componentes para a renderização de objetos, mas não define algoritmos de gerenciamento dos objetos em cena, ficando a cargo do programador implementá-los;
- Não provê um mecanismo de `scripting` diretamente. As linguagens de programação suportadas são, basicamente, C# e Visual Basic;
- Requer que o projeto esteja organizado num determinado formato.

3.3.4 Outros motores e *frameworks*

Além dos ambientes citados, existem outros motores de jogos e *frameworks*, como: Crystal Space [95], jMonkeyEngine [52], UDK [38] etc. Tais motores foram também foram analisados, mas devido a baixa popularidade ou desempenho, restrições de licença ou portabilidade (UDK), não foram considerados.

Avaliação dos ambientes específicos para criação de jogos

Os principais pontos das ferramentas analisadas são:

- Quanto melhor o motor de jogo, mais ele estará direcionado à utilização de ferramentas; porque o sucesso de um jogo está intimamente ligado à produção artística. Consequentemente, a maioria de tais ferramentas são direcionadas aos artistas e não exatamente a Cientistas da Computação. Esses fatores tornam a Unity3D e Unreal SDK, que estão dentre os melhores motores de jogos, escolhas pouco adequadas para a utilização em um curso de Ciência da Computação;
- A Panda3D seria uma boa opção por prover vários componentes prontos, *scripting* e ainda permitir uma certa liberdade quanto ao modo como o projeto pode ser criado. Entretanto, ainda existem restrições quanto a como o fluxo do programa deve ser montado, organização de pastas e formatos de arquivos. Adicionalmente, ainda que negligenciável, a utilização de Python, como linguagem de *script*, gera um impacto no desempenho da aplicação. Tal desempenho poderia ser melhorado através de algumas técnicas adicionais ou utilizando o Mono como máquina de virtual. Por fim, Python possui um suporte fraco a orientação a objetos, baseado em convenções, tanto para a declaração de métodos, construtores ou mesmo campos [17];
- O XNA *framework* seria uma boa opção pois, como o próprio nome indica, é um *framework* orientado para a criação de jogos. Assim, provê somente componentes para a criação dos mesmos, deixando a organização do projeto e diversos outros recursos extensíveis e disponíveis para o usuário. Entretanto, por ser orientado somente a plataformas da Microsoft, não provendo suporte para outros sistemas operacionais, se torna uma opção inadequada para várias universidades públicas e federais, que muitas vezes utilizam Linux. Além disso, seu ambiente de desenvolvimento requer o Visual Studio, que está presente somente no Windows também.

CAPÍTULO 4

Projeto InVision

No capítulo 2 uma breve introdução sobre o que são jogos, motores de jogos (ver 2.3) e gêneros de jogos (ver 2.3.1) foi apresentada. Neste capítulo, são mostradas as diretrizes do projeto e a arquitetura para a construção de um *framework* de desenvolvimento de jogos, que seja: simples, robusto e adaptável ao meio acadêmico.

Como apresentado na seção 3.2, a utilização de ambientes com linguagens de programação visual para o ensino e aprendizado de algoritmos é um bom meio para manter a motivação, desenvolver o raciocínio lógico e apresentar novos conceitos de Computação [42]. *"É importante que os estudantes desenvolvam habilidades de programação, assim eles podem criar programas que resolvam problemas reais"* [43].

Entretanto, mesmo podendo utilizar jogos para apresentar e trabalhar conceitos de outras disciplinas, continuar a utilizar tais ambientes de programação se torna uma limitação visível para um aluno que já detenha conhecimento em programação. Torna-se uma necessidade a utilização de uma ferramenta mais poderosa ou com suporte mais amplo para não limitar a capacidade e criatividade desse aluno, mantendo a aplicação de jogos e simulações como fator motivacional.

Figura 4.1: *Logomarca proposta para o Framework.*

O *framework* InVision, cujo logotipo é apresentado na figura 4.1, tenta preencher o vazio, em relação a usabilidade e recursos disponíveis, entre as ferramentas para ensino de programação (ver 3.2) e os *frameworks* e motores de jogos especializados (ver 3.3), seguindo algumas diretivas que serão apresentadas nas seção 4.1, com o propósito de

dar suporte à construção de jogos e simulações de forma mais fácil e com aplicação educacional.

É importante frisar que a diferença entre jogos e simulações é quase nula, pois um jogo pode ser basicamente uma simulação. Contudo, uma característica difere ambas, segundo Bittencourt: *"as diferenças conceituais entre os jogos e as simulações podem ser caracterizadas pelo fato de que o jogo é um processo intrinsecamente competitivo (em que co-existem a vitória e a derrota) e uma simulação é a simples execução dinâmica de um modelo previamente definido"* [18].

O projeto InVision é direcionado para a utilização em um curso de Ciência da Computação. Esse tenta criar mecanismos para que disciplinas existentes no curso possam utilizá-lo; auxiliando na apresentação de conceitos, construção de animações, simulação ou para construção de protótipos, por alunos com um conhecimento básico ou avançado de programação, e não somente em disciplinas introdutórias.

4.1 Preenchendo o vazio entre os ambientes de ensino e os motores de jogos especializados

O termo "limitado", usado para descrever os ambientes de programação visual para aprendizado de algoritmos e programação, não deve ser levado como algo pejorativo. Afinal, vários pesquisadores, como [87, 32, 57, 59, 100], já os utilizaram e afirmam que suas utilizações garantem bons resultados.

Tais ferramentas focam na simplicidade e no aprendizado de conceitos de programação, em geral. Um dos próprios envolvidos no projeto do Scratch, Resnick, afirma que para a construção de jogos e aplicações mais completas deve-se aprender e utilizar uma linguagem adequada a tal fim, como Java, pois o Scratch não possui tal finalidade [88].

Se tais ambientes são inadequados para a construção de aplicações mais complexas, então os motores de jogos também se tornam inadequados para a utilização e criação de aplicações que não sejam tão complexas quanto os jogos comerciais.

O problema em si está em como o projeto é executado. A maioria dos motores de jogos define um processo e uma arquitetura final para o produto, além de prover ferramentas que muitas vezes são destinadas a artistas visuais e *game designers* [44]. Trabalhar em tal arquitetura muitas vezes requer um conhecimento que vai além daquele que um aluno, do segundo ou terceiro semestre de curso, possua. Além disso, pode ocasionar vícios e criar uma visão orientada para jogos somente, não sendo uma boa opção para um curso de Ciência da Computação.

Por mais que os motores de jogos especializados disponibilizem ferramentas de qualidade para a criação de artefatos e jogos, o que seria um ótimo recurso para um

curso de “Desenvolvimento de jogos”, não é algo inteiramente desejado para um curso de Ciência da Computação. Os motivos por tal afirmação estão descritos abaixo:

- Motores de jogos focam na qualidade final do jogo, consequentemente possuem ferramentas, como editores, que facilitam muito o trabalho de um *game designer* ou artista. A programação e *scripting* torna-se algo suplementar, trabalhando-se muito com abstrações e cálculos, mas os resultados só poderão ser “visualizados” quando tiverem tal processamento ligado a um objeto visual, por um *game designer*. Logo, a figura do artista e o trabalho em equipe sempre estão presentes. Numa classe de Ciência da Computação, o trabalho em equipe pode ser aprimorado, mas nem todos tem dons para a arte. Se um trabalho for realizado, mesmo em grupo, alguns jogos serão melhores que outros e se destacarão, por sua apelação visual, mas não demonstrarão, em geral, a capacidade do programador e nem refletirão as decisões de projeto, ocultando o trabalho intelectual da construção;
- O modo de trabalho e formas de construção são pré-definidos. É possível se ter uma certa liberdade quanto a como um determinado algoritmo será implementado, mas não quanto a organização do projeto e o processo de construção. Contudo, isso é algo que pode variar de motor para motor. *Frameworks* são uma opção melhor por proverem componentes e não tentarem definir como a aplicação será construída.

É esperado de um Cientista da Computação que este seja capaz de criar, abstrair, construir e, se possível, estender os limites da Computação e não somente utilizar ferramentas.

Dessa forma, a seção 4.1.1 apresenta um conjunto de diretrizes do projeto.

4.1.1 Diretrizes do projeto InVision

- **O *framework* deve ser simples**. Significando que este deve ser de fácil uso para estudantes de Computação entre o primeiro e o último semestre. O termo “fácil”, nesse caso, é condicionado às facilidades providas para criação:
  - O usuário pode utilizar *scripts* para uma rápida prototipação e testes, assim como é feito na Panda3D [79];
  - Para a programação é permitida a utilização de várias linguagens, amplamente utilizadas na construção de aplicações comerciais e para web, como: C#, F#, Boo, Visual Basic, Python (Iron Python), Ruby (Iron Ruby) etc. A possibilidade de escolher várias linguagens permite que abordagens mais simples ou mais complexas sejam executadas. Pode-se utilizar de certas linguagens seguindo um paradigma procedural e então avançar para o paradigma orientado a objetos ou o funcional, por exemplo.

  - Quando necessário pode-se compilar o código, gerando bibliotecas em CIL que utilizem das funcionalidades do *framework* ou que complementem essas;
  - Componentes prontos encapsulando funcionalidades presentes em jogos podem ser utilizados para agilizar na construção.

- **O *framework* deve ser robusto**, isto é, deve prover funcionalidades que garantam a construção de jogos simples, mas também com recursos mais avançados e qualidade visual próxima das presentes em jogos comerciais. Dessa forma, a utilização de bibliotecas nativas e com qualidade reconhecida para prover tal suporte é essencial. Alguns exemplos de tais bibliotecas são:
  - O motor de renderização OGRE3D [78];
  - O motor de física Bullet [27];
  - A biblioteca OIS [39], para o tratamento de dispositivos de entrada como: teclado, mouse, *joysticks*, WiiMote etc;
  - A biblioteca para reprodução de áudio FMod [34].

- **O *framework* deve ser adaptável**, isto é, não deve forçar o usuário a seguir um único padrão de projeto. O usuário pode usar partes do *framework* para a construção de sua própria aplicação ou usar um protótipo auto-gerenciável, que permite a organização do fluxo de execução da aplicação. Assim, esse pode escolher entre os fluxos existentes ou criar a sua própria maneira de trabalhar;
- **O *framework* deve ser multi-plataforma**. Como boa parte do *framework* é criado em .NET, mantendo compatibilidade com o Mono, a limitação quanto as plataformas de execução são as próprias limitações do *runtime* e das bibliotecas nativas, que nesse caso já garantem os Sistemas Operacionais: Windows, Linux e Mac OS X [6]. Estuda-se também a possibilidade de adaptar o projeto para que também execute sobre as plataformas iOIS e Android, permitindo a construção de jogos e simuladores para plataformas móveis.

4.2 Arquitetura

A figura 4.2 apresenta um esquema completo da arquitetura proposta. A camada do topo deve ser construída pelo usuário e consiste no jogo propriamente dito, autoria do usuário.

Seguindo as orientações presentes na tese de Plummer [85], a criação e organização de uma arquitetura baseada em componentes permite que tal arquitetura seja também flexível e expansível. Este projeto é dividido em quatro partes:

1. Uma camada de suporte nativo;

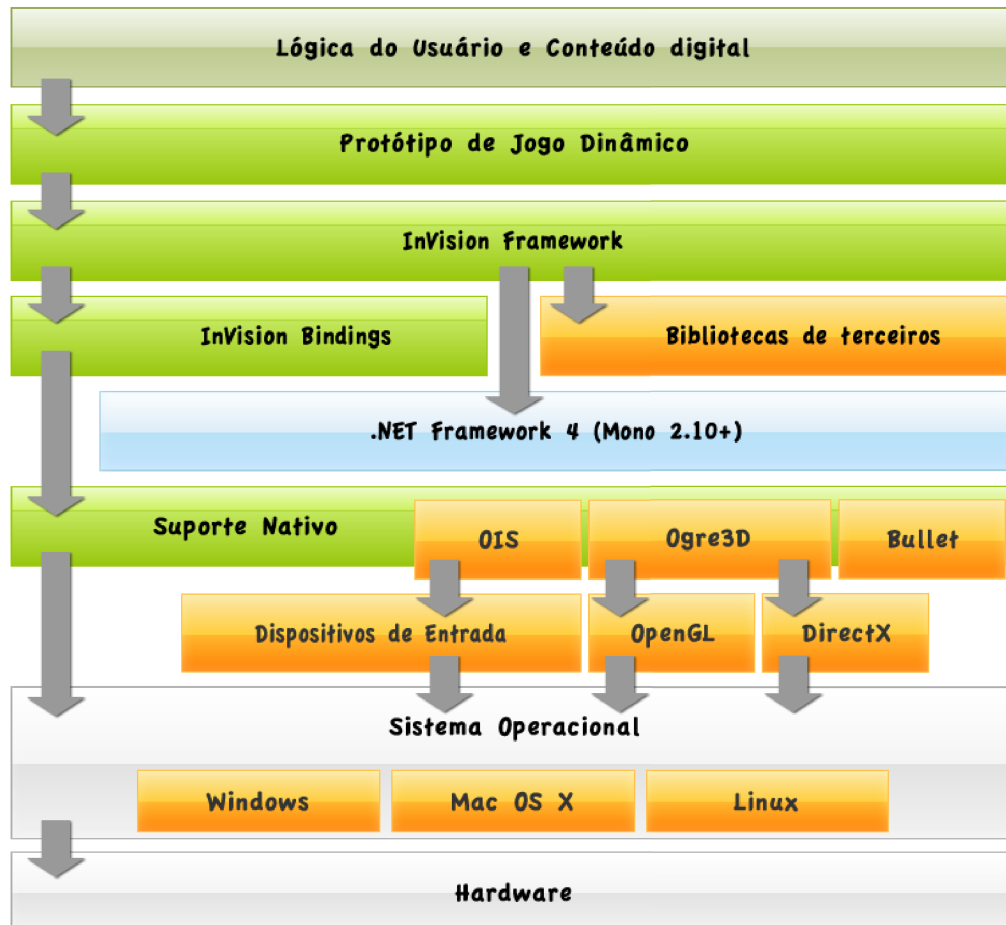

Figura 4.2: *Arquitetura do Projeto InVision.*

2. Uma camada de abstração para a camada de suporte nativo, transformando as chamadas de funções em objetos e ações;
3. Um conjunto de bibliotecas e serviços que usam e expandem as funcionalidades providas pela camada de suporte nativo, descrita como a base do *framework*;
4. Uma aplicação auto-gerenciável, suportada por *scripts* e compilação dinâmica, para a construção rápida de protótipos (semelhante a como o motor de jogo Panda3D funciona).

4.2.1 A plataforma .NET e o Mono

O Mono [75] é um *runtime* para a plataforma .NET, atualmente mantido pela Novel como *open-source*. A escolha do Mono como plataforma para a execução do *framework* se deve a um conjunto de características, descritas abaixo:

- A plataforma .NET, e consequentemente o Mono, permite a utilização de várias linguagens nativamente;

- Quase todos os *assemblies* puros (bibliotecas ou executáveis criados em .NET que não realizam chamadas nativas ao Sistema Operacional) são compatíveis com o Mono;
- O Mono provê suporte para várias plataformas, incluindo algumas que vão além daquelas escolhidas como alvos do projeto. Tornando o *framework* uma opção multiplataforma, semelhante ao java;
- Empresas como a Unity e a Second Life estão utilizando o Mono como executor dos seus *scripts*. Isso permite que a maioria das funcionalidades já providas pelo .NET *framework* seja utilizadas diretamente nesses, além de se ter um ganho substancial de desempenho (mais detalhes em [56, 31]);
- O Mono se especializa cada vez mais para prover suporte para o desenvolvimento de jogos, incluindo meios para execução de trechos de código usando instruções SIMD, de forma transparente ao usuário [30]. Isso garante um aumento no desempenho de aplicações que requerem processamento de dados em lote, como processamento vetorial e de matrizes, muito utilizados em jogos;
- As plataformas Android e iOS possuem suporte do Mono atualmente. Permitindo a possibilidade de se portar o *framework* InVision para a execução nesses dispositivos móveis.

4.2.2 Camada de Suporte Nativo

Para prover serviços de qualidade e manter as restrições quanto a robustez do projeto, algumas bibliotecas bem conhecidas, utilizadas no desenvolvimento de jogos, foram selecionadas:

Ogre3D [78] - uma poderosa biblioteca *open-source*, escrita em C++, para renderização gráfica 3D. A Ogre suporta nativamente OpenGL e Direct X, as duas principais bibliotecas para renderização com aceleração de hardware. Algumas das características dessa biblioteca estão listadas abaixo (retiradas de [77]);

- Provê interface de programação orientada a objetos em C++, construída para ser independente de implementação da API 3D (Direct X, OpenGL etc);
- Provê suporte nativo para Direct X e OpenGL;
- Provê funcionalidades comuns como gerenciamento para estados de renderização, *culling* hierárquico e transparências;
- Provê suporte para os sistemas operacionais Windows, Linux, Mac OS X. Recentemente portada para iOS e PlayStation 3;
- Provê uma poderosa linguagem para declaração de materiais, ajudando a manter os recursos fora do código;

- Provê suporte a *Vertex* e *Fragment Shaders* em baixo-nível com *assembly* ou em alto-nível com: NVidia Cg, Direct X HLSL ou OpenGL GLSL;
- Provê suporte para a técnica de Nível de Detalhes (LoD - Level of Detail);
- Provê carregamento de texturas em diversos formatos, como: PNG, JPEG, TGA, BMP, PVRTC e DDS; ou ainda formatos nada comuns como texturas 1D, texturas volumétricas, mapas cúbicos, HDR (*High Dynamic Range*) e texturas comprimidas (DXT / S3TC);
- Provê Gerenciadores de Cena customizáveis;
- Provê suporte a efeitos especiais como: sistema de partículas, *skyboxes*, *skyplanes*, *skydomes*, *billboarding* para gráficos em *sprites*, gerenciamento automático de transparência para objetos, *overlays* para criação de interfaces de usuário e menus usando 2D ou 3D, controle de neblina flexível, *bump mapping* e efeitos de pós-processamento (aparência de filme velho, HDR etc);

OIS [39] - uma pequena biblioteca orientada a objetos para tratar eventos de dispositivos de entrada, como: teclados, mouses, controladores de jogo etc. Permitindo que dispositivos com *force feedback* possam ser tratados.

Bullet [27] - uma biblioteca para simulação de física em objetos em 3D. Suporte multiplataforma para: consoles (PlayStation 3, XBox 360, Wii) e os Sistemas Operacionais Windows, Linux e Mac OS X (mais informações em `http://www.ogre3d.org/tikiwiki/OgreBullet`);

FMOD [34] - uma poderosa biblioteca de áudio. Possui código fechado, mas é gratuita para projetos *open-source*. Provê suporte para várias plataformas.

Cada uma das bibliotecas anteriores são nativas e dependentes do Sistema Operacional. Assim, será necessário prover pacotes contendo os binários já compilados para cada uma das plataformas de destino, combinados com os devidos *bindings* escritos em C#, como bibliotecas gerenciáveis (*managed assemblies*). Motores de jogos multiplataformas, em geral, realizam uma tarefa parecida com essa, onde certos componentes adicionais ao Sistema Operacional (bibliotecas nativas) são instalados ou acompanham os executáveis do próprio jogo.

A escolha das bibliotecas acima não foi realizada somente pela quantidade de funcionalidades e qualidade em geral, mas também levando em consideração aspectos como portabilidade e estabilidade.

Por exemplo, existem vários motores de física atualmente disponíveis, como a ODE e a NVidia PhysX. A escolha pela Bullet, ao invés dessas duas opções, se deve aos seguintes fatos:

- A Bullet se especializou e provê inúmeros recursos avançados, presente nos motores de física mais conceituados e utilizados no mercado de jogos, tais como o

NVidia PhysX e o Havok [47]. Além disso, é portável entre várias plataformas e existem vários títulos AAA (conceito de ultra qualidade para jogos) que a utilizam atualmente;

- A NVidia PhysX é uma excelente alternativa à Bullet, principalmente pelo desempenho, mas sua utilização iria contra a diretiva de manter o *framework* multiplataforma. Isso acontece porque para tal motor funcionar é requerido uma placa de vídeo da NVidia, sendo que tal dispositivo não se encontra em todas as plataformas de destino visadas, como um dispositivo obrigatório.
- Em termos práticos, a utilização da ODE ao invés da Bullet é uma opção melhor, pois a primeira exporta todas as suas funcionalidades através de uma interface simples, escrita em C, o que facilita a criação dos *bindings*. Entretanto, tal biblioteca teve sua última atualização em 2009. Além disso, existem poucas alterações no código fonte do projeto, posteriormente, sendo a última há mais de 3 meses, o que indica que esse projeto foi abandonado. De forma contrária, a Bullet está em constante evolução, incluindo novas funcionalidades e efeitos, projetos e parceiros, além de ser portável para várias plataformas, incluindo consoles;
- Por fim, a AMD na tentativa recuar o avanço do PhysX e tornar o mercado mais competitivo contra a NVidia, anunciou que irá contribuir para o projeto da Bullet [7]. A contribuição é uma alteração do motor de física, que o permite executar sobre OpenCL [96], fazendo-o executar de forma mais rápida sobre uma *GPU* genérica e sem restrições quanto ao fabricante desta.

4.2.3 *Bindings* para bibliotecas nativas

Essa camada está totalmente apoiada sobre o Mono e a camada de suporte nativo. Sua função é expor e gerenciar as funcionalidades das bibliotecas nativas ao mundo gerenciável (*managed world*), como é conhecido os componentes que rodam em .NET.

Para cada uma das bibliotecas nativas será criado um *assembly* em .NET, permitindo sua utilização por outros desenvolvedores em escopos externos ao do *framework*.

Um detalhe importante, é que nem todas as funcionalidades providas pelos *bindings* possuem um paralelo nativo, pois quanto menos invocações a métodos externos (nativos) melhor será o desempenho. Alguns cálculos e funcionalidades podem ser diretamente portadas para .NET, ganhando-se em tempo de execução pela eliminação da sobrecarga presente na chamada do método externo.

4.2.4 InVision Framework

O papel do *framework* InVision é organizar as funcionalidades providas pela camada nativa, mesclando com outras requeridas para o desenvolvimento de jogos.

Podemos organizar o *framework* em *namespaces*, similar a pacotes da linguagem Java, como a seguir:

InVision.Game - Provê um modelo de aplicação extensivo através de fluxos de execução customizáveis (ver 5.3.4);

InVision.GameMath - Provê um conjunto de funcionalidades matemáticas necessárias para a utilização em jogos, como: vetores, quatérnions, matrizes, utilitários para conversão entre unidades;

InVision.Rendering - Provê abstrações para a utilização da camada de renderização, através da biblioteca nativa Ogre;

InVision.Audio - Provê abstrações para a reprodução de áudio, através da biblioteca nativa FMod;

InVision.Input - Provê abstrações para o tratamento de dispositivos de entrada, através da biblioteca nativa OIS;

InVision.Persistence - Provê um mecanismo para serialização de objetos e armazenamento;

InVision.Scripting - Provê o suporte básico para o carregamento e processamento utilizando *scripts*;

InVision.Network - Provê um conjunto de implementações para a comunicação via rede e transferência de estados de objetos entre a aplicação;

InVision.IA - Provê um conjunto de algoritmos básicos para a organização e criação de Inteligência Artificial para jogos.

O *framework* InVision pode ser utilizado por qualquer aplicação .NET como provedor de funcionalidades, em uma das plataformas de destino, desde que as dependências nativas estejam presentes.

4.2.5 Protótipo extensível (Tutano)

Um protótipo extensível, denominado Tutano, é apresentado junto ao *framework* InVision. Sua função é prover uma organização para um projeto de um jogo que possa ser estendido através de novas bibliotecas de serviços ou *scripts*.

Nessa primeira fase do projeto, tal protótipo somente dá suporte para uma organização baseada em diretórios pré-definidos; algo semelhante ao que a Panda3D realiza.

Existem basicamente cinco diretórios:

Bin/ - onde todos os executáveis estão presentes. Inclusive as bibliotecas nativas e gerenciáveis;

Libraries/ - contém bibliotecas desenvolvidas pelos usuários, que estendem as funcionalidades do *framework*;

Config/ - diretório contendo arquivos em XML para configuração avançada do ambiente, quando necessário;

Content/ - diretório contendo os *assets* (texturas, meshes, materiais etc) para a construção da aplicação;

Scripts/ - diretório contendo os *scripts* com a lógica do usuário para o jogo. Todos os *scripts* serão interpretados ou compilados dependendo da linguagem escolhida.

O funcionamento do Tutano é simples e pode ser dividido em passos:

1. Carregamento das bibliotecas dinâmicas, e consequentemente das bibliotecas nativas referenciadas;
2. Leitura dos arquivos de configuração;
3. Compilação dos *scripts* para execução;
4. Execução do *script* inicial, definido nos arquivos de configuração.

O Tutano é uma aplicação pré-configurada para auxiliar na construção de protótipos de jogos de forma mais rápida, visto que tudo é construído para executar sobre *scripts* e sem muita configuração. A execução do programa se torna algo transparente, o que facilita sua adoção principalmente por estudantes recém ingressos, com pouco conhecimento sobre computação, linguagens e compiladores.

CAPÍTULO 5

Construindo o *framework*

No capítulo 4, algumas diretrizes para a construção do projeto foram apresentadas em 4.1.1, assim como uma proposta de arquitetura em 4.2 e algumas das funcionalidades previstas para o *framework*, como: suporte para bibliotecas nativas conceituadas em 4.2.2, *bindings* para tais bibliotecas em 4.2.3, o *framework* InVision em 4.2.4 e um modelo de protótipo extensível 4.2.5.

Nesse capítulo, serão apresentadas as decisões de projeto e formas de implementação, seguindo a arquitetura proposta em 4.2. Uma avaliação crítica sobre os possíveis mecanismos para a construção das duas camadas de base (camada de suporte nativo e de *bindings*) é apresentado em 5.1, sendo que os detalhes de sua construção estão descritos na seção 5.2. A construção do *framework* InVision será apresentada na seção 5.3 e a construção do protótipo extensível em 5.4.

5.1 Interoperabilidade em .NET

A máquina virtual Mono e o .NET *Runtime* executam *bytecodes* em CIL (*C Intermediate Language* [5]), compilando tais instruções em tempo de execução (*just-in-time compilation* [28]). Tais máquinas simulam um ambiente de execução, geralmente independente do hardware que o está executando. Contudo, algumas vezes é necessário utilizar recursos dependentes do Sistema Operacional, como bibliotecas nativas. Alguns motivos para a utilização desses recursos são:

- Existência de um grande conjunto de funcionalidades já disponíveis e compiladas em bibliotecas dinâmicas escritas em C e C++;
- Necessidade acessar diretamente dispositivos de entrada e saída;
- Garantia de desempenho, pois alguns processamentos dependem de um controle de alocação de memória e algumas otimizações que só podem ser obtidas através de linguagens de baixo nível, como C / C++ e Assembly.

Em Sistemas de Informação, o termo **interoperabilidade** significa trocar ou fazer uso da informação, isto é, a capacidade de um sistema se relacionar com outro [61].

Em .NET, usa-se muito esse termo quando se deseja acessar recursos do ambiente de execução da máquina virtual, o Sistema Operacional.

Dessa forma, a plataforma .NET disponibiliza meios para expor tais recursos nativos do Sistema Operacional (ambiente não gerenciado ou *unmanaged*) em seu próprio ambiente de execução (ambiente gerenciado ou *managed*). Uma biblioteca gerenciada que realiza chamadas nativas é definida com o termo *binding* ou ligação, que basicamente descreve a ligação entre ambos ambientes. Ocasionalmente, *wrappers* são criados sobre as chamadas nativas para encapsulá-las em classes, seguindo o paradigma principal da plataforma .NET, a orientação a objetos. Contudo, isso é algo relacionado às linguagens de uso e não exatamente um requerimento.

A plataforma .NET permite a interoperabilidade entre o ambiente gerenciado e não gerenciado, através de alguns mecanismos [69, 68, 73, 72]:

***P/Invoke ou Platform Invoke* ou invocação explícita** - um simples atributo é definido sobre um método com a notação `extern`. Tal atributo contém o nome da biblioteca dinâmica a ser utilizada, o nome do método a ser acessado e outros detalhes. Tal mecanismo tem uma limitação quanto ao seu uso, a biblioteca deve ser escrita em C ou em C++ (através da notação `extern "C"`). Além disso, somente funções podem ser chamadas e as estruturas de dados para a troca de informação, no ambiente gerenciado, devem ser compatíveis (em tamanho de *bytes* e forma de armazenamento: *little-endian* ou *big-endian*).

***Managed* C++ ou invocação implícita** - esse mecanismo requer que um *assembly* seja escrito em .NET, utilizando a linguagem C++ *Managed*, que faz uso direto e encapsula as chamadas nativas em classes e/ou chamadas de função. Quando compilado, um *assembly* contendo código misto (*mixed code*) de .NET e intruções nativas do Sistema Operacional é criado. Tal abordagem permite que *bindings* para código em C++ sejam facilmente criados, além disso existe ganho de desempenho sobre o mecanismo de *P/Invoke*;

COM - esse mecanismo permite a interoperabilidade entre C++ e componentes do Sistema Operacional de forma fácil, apenas sabendo a interface do componente que se deseja trabalhar. Define-se a interface para a comunicação com alguns atributos de marcação sobre o tipo e seus métodos e uma fábrica para se criar tais objetos (por exemplo, utilizando-se de *P/Invoke* para a criação do componente).

5.1.1 Interoperabilidade entre C/C++ e C# com *P/Invoke*

A base do *framework* InVision tem como suporte a plataforma Mono e algumas bibliotecas nativas, apresentadas em 4.2.2. Entretanto, existe uma preocupação sobre como gerar os *bindings* e *wrappers* para prover as funcionalidades externas, pois as

aplicações que serão construídas são aplicações de tempo-real, que demandam alto desempenho. Além disso, o requisito de portabilidade deve ser mantido (diretrizes do projeto em 4.1.1).

Dentre os mecanismos para a realização das chamadas nativas, citados em 5.1, a utilização de *Managed* C++ fere diretamente o requisito de portabilidade, descartando sua utilização, pois o Mono não provê suporte para *assemblies* em modo misto. Um *assembly* em modo misto carrega, misturada em seus `bytecodes`, instruções dependentes do Sistema Operacional, tornando-o não portável.

O mecanismo que utiliza COM foi testado sem sucesso, mesmo para os exemplos pré-definidos providos pelo projeto Mono. Contudo, utilizar COM para a comunicação requer que *wrappers* em C++ também sejam criados no lado nativo, encapsulando a funcionalidade desejada, e que todos os métodos sejam virtuais [37]. Também não há um modo de obter valores de campos, forçando a utilização de métodos *get/set* para obter tais valores, o que pode provocar problemas de desempenho, onde a chamada externa custa mais tempo que o processamento externo [20].

Utilizar *P/Invoke* é uma boa alternativa para chamadas de funções em C, que garante a portabilidade, mas para C++ torna-se uma atividade desafiadora. Cada método de uma classe em C++ deve ser traduzido como uma função em C, tornando o trabalho de conversão manual muito extenso.

Analisando o funcionamento do mecanismo de *P/Invoke*, escolhido para o projeto InVision, seu funcionamento é descrito abaixo:

- Todas as funcionalidades devem ser expostas como funções em C, no lado nativo. Para cada função, será criado um método estático em C# (*binding*), em uma classe qualquer, não genérica;
- Certas funcionalidades podem ser agrupadas e mascaradas sob um tipo qualquer, criando algo como um objeto e seus métodos (*wrapper*);
- Para chamadas a funções em C, não existem grandes problemas, pois os parâmetros são enviados ao lado nativo e tem-se o resultado retornado. Seu funcionamento equivale ao de um método estático qualquer, exceto pelo cuidado com a compatibilidade entre os tipos de dados e as formas de *marshalling* (conversão de um tipo gerenciado para um não gerenciado e vice-versa);
- Quando funcionalidades de um objeto em C++ são expostas existem complicações. Cada método desse objeto (inclusive os construtores, destrutores, sobrecarga de operadores etc) deve ser exposto como uma função em C também. O código 5.1 demonstra como criar as funções para cada método da classe `HelloWorld`. Com tal código, pode-se criar um classe `HelloWorld` como um *wrapper* em C#, simulando o mesmo comportamento e funcionalidades do objeto `HelloWorld` em C++, pois para cada método em C# será invocado um método do objeto em C++;

Código 5.1 Funções que encapsulam métodos de um objeto em C++

```cpp
#include <iostream>

/**
 * Diz olá para você!
 */
class HelloWorld {
public:
        /** Construtor */
        HelloWorld();

        /** Destrutor */
        ~HelloWorld();

        /** Escreve olá no console */
        void say();
};

/* O código das funções será colocado diretamente no header
 * para fins de demonstração.
 */
extern "C" {
        /** Retorna uma instância de HelloWorld */
        HelloWorld* new_HelloWorld() {
                return new HelloWorld();
        }

        /** Deleta a instância de HelloWorld */
        void delete_HelloWorld(HelloWorld* self) {
                delete self;
        }

        /** Invoca o método say do objeto HelloWorld */
        void HelloWorld_say(HelloWorld* self) {
                self->say();
        }
}
```

Entretanto, existe um problema quanto a abordagem do último item. Por exemplo, se um usuário estende a classe `HelloWorld`, criando a classe `HelloWorldBR` e so-

brescreve o método `Say`. Quando o método `Say` for invocado do lado gerenciado, a nova implementação será executada, sem problemas. Contudo, se algum outro método em C++ invocar o método `say`[1] da classe em C++, essa irá executar a implementação original e não o novo código. Isso acontece porque a *vtable* do objeto, mecanismo utilizado pela linguagem para simular métodos virtuais [21], não foi sobrescrita pela nova implementação. Para que seja possível simular esse tipo de comportamento é necessário alterar a *vtable* manualmente ou realizar uma simulação de herança, que o SWIG chama de `Director` (para mais informações veja [98]). O projeto InVision criou um mecanismo baseado na manipulação de *vtables*, simulando uma herança através da geração de uma *proxy* e outras estruturas (veja 5.2.4).

5.2 Construção da camada de Suporte Nativo e camada de Bindings

A camada de Suporte Nativo está intimamente ligada à camada superior, a de *Bindings*. A construção dessa camada implica em prover funções no estilo da linguagem C, abstraindo as chamadas de métodos em C++. Adicionalmente, deve-se prover meios para que a camada superior possa construir os *wrappers* de forma otimizada.

A seção 5.2.1 apresenta a primeira tentativa do projeto e seus pontos de falha. A seção 5.2.2 apresenta a alternativa escolhida para a construção dessa camada. A seção 5.2.3 define o que são tipos *blittable* para a plataforma .NET. Em seguida, a seção 5.2.4 apresenta uma técnica utilizada para construir *wrappers* que simulem o sistema de herança entre uma classe em C++ e outra em C#. Por fim, um problema com relação ao manuseio de ponteiros é apresentado em 5.2.5. Tal problema influencia diretamente em como as funções em C devem ser criadas.

5.2.1 Geração de Wrappers com o SWIG

O SWIG é uma ferramenta capaz de gerar *wrappers* e *bindings* em diversas linguagens que visam fazer uso e funcionalidades em bibliotecas nativas em C ou C++ (mais detalhes em [98]). A atratividade quanto a utilização do SWIG deve-se ao fato de que uma biblioteca nativa, em C ou C++, pode ser portada rapidamente para um ambiente de programação qualquer, com velocidade e relativa praticidade. O SWIG é bem flexível e existem algumas formas de se adaptar o código gerado para a linguagem de destino, mas nem tudo pode ser resolvido facilmente.

[1] Métodos em C++ devem ser nomeados em *Camel Case*, ao contrário da linguagem C#, onde usa-se *Pascal Case*

Apesar de conseguir gerar boa parte do código sem muito problema, o foco do SWIG é prover as funcionalidades da biblioteca nativa, mas nem sempre os *wrappers* se adequam totalmente à linguagem de destino. Alguns *wrappers* fogem à regra quanto as construções e modos de trabalho da linguagem.

O conjunto de bibliotecas padrão em C++, presente no *namespace* `std`, é constantemente usada na OGRE, OIS e outras bibliotecas. Entretanto, as classes que representam coleções de objetos, como listas (`std::vector`), dicionários (`std::map` e `std::multimap`) e outras estruturas de dados desse *namespace*, geradas pelo SWIG em C#, não geram equivalentes em funcionalidades no ambiente .NET, e acabam desrespeitando certos detalhes do ambiente de destino, como a herança de interfaces para coleções; isso fere quanto ao uso e facilidade do usuário, uma das diretrizes do projeto InVision.

Por exemplo, para criar e preencher um dicionário do tipo `std::map`, onde a chave é uma enumeração do tipo `MyEnum` e o valor é do tipo `std::string`, em C++, usa-se o seguinte código:

Código 5.2 Criação e manipulação de um dicionário em C++

```
// declaração do objeto
std::map<MyEnum, std::string> map;

// inserindo o primeiro elemento <MyEnum.First, "First Item">
map.insert( std::pair<MyEnum, std::string>(
                MyEnum.First,
                std::string("First Item") ));

// inserindo o segundo elemento <MyEnum.Second, "Second Item">
// usando sobrecarga de operador para a criação da string
map.insert( std::pair<MyEnum, std::string>(
                MyEnum.Second,
                "Second item" ));

```

O mesmo exemplo, convertendo-se os tipos para os tipos nativos em C# (`std::string` para `string`), está abaixo:

Código 5.3 Criação e manipulação de um dicionário em C#

```
1 // declaração do objeto
2 Dictionary<MyEnum, string> map;
3
4 // inserindo o primeiro elemento <MyEnum.First, "First Item">
5 map.Add(MyEnum.First, "First Item");
6
7 // inserindo o segundo elemento <MyEnum.Second, "Second Item">
8 map.Add(MyEnum.Second, "Second item");
9
```

Entretanto, o tipo `std::map<MyEnum, std::string>` em C++ será processado e gerado pelo SWIG com o nome `SWIGTYPE_p_std__mapT_MyEnum_std__string_t`, se retornado por uma função como a seguinte:

Código 5.4 Função nativa em C++ que retorna um dicionário

```
1 #include <map>
2 #include <string>
3
4 using namespace std;
5
6 enum MyEnum
7 {
8         First,
9         Second
10 };
11
12 typedef std::map<MyEnum, std::string> MyDictionary;
13
14
15 MyDictionary* someFunction();
```

Além de nomes ilegíveis ou muito grandes para alguns tipos, certas funcionalidades do dicionário também não estarão disponíveis, pois o SWIG não irá portar alguns métodos do dicionário para .NET. Esse é um problema que ocorre quando manipula-se alguns tipos com o SWIG. É possível resolver esse problema definindo-se *typemaps* e extensões específicas para cada caso. Entretanto, quando o código a ser gerado é muito grande, isso se torna uma tarefa extremamente penosa.

Outro agravante, é que tais *wrappers* realizam chamadas ao ambiente nativo em cada operação, penalizando o desempenho. Quando trabalhando com coleções, em

alguns casos, manter o conteúdo na memória gerenciada e então enviá-lo em uma única chamada para o lado nativo se torna uma abordagem melhor. O SWIG somente converte cada método nativo em uma chamada externa, não permitindo que soluções heterogêneas como a dita anteriormente, sejam utilizadas em alguns casos.

Contudo, talvez o ponto mais crítico, é quanto aos tipos gerados em .NET. Todo tipo, classe ou estrutura, em C++ é representado como uma classe em C#, tendo seus devidos métodos como chamadas nativas. Apesar de ser uma abordagem simplista, que funciona bem para a maioria dos casos, tal mecanismo não gera tipos de valor em C# (`structs`), que em muitos casos poderiam garantir um desempenho melhor para a aplicação.

Um exemplo da dificuldade de uso do SWIG para certas aplicações foi o projeto OgreDotNet, uma biblioteca que provia *bindings* e *wrappers* para a Ogre. Tal projeto foi abandonado e os motivos foram a dificuldade para manter a biblioteca e o surgimento de outra biblioteca (Mogre) que garantia melhor desempenho por utilizar *bindings* implícitos em C++.

5.2.2 Geração manual de *bindings*

Ainda que existam problemas com a geração de código do SWIG (apresentada em 5.2.1), ganha-se muito em tempo. Contudo, para a implementação do *framework* InVision, os *bindings* e *wrappers* estão sendo criados manualmente, com o auxílio de uma ferramenta criada pelo projeto que gera um código parcial, garantindo ao usuário do *framework* as seguintes características:

- Algumas classes e estruturas mais simples em C++ foram convertidas como tipos de valor (`struct`) em C#, principalmente quando são imutáveis e/ou seguem algumas regras contidas em 5.2.3;
- Alguns tipos imutáveis que possuem muitos dados podem ser carregados sob demanda, realizando uma única chamada ao método externo durante a primeira solicitação de valor, e mantendo o valor localmente no objeto. Dessa forma, partes do objeto (campos) serão carregadas à medida que forem utilizados;
- Tipos como `Vector3`, `Vector4`, `Quaternion` e `Matrix` são utilizados para processamento vetorial e matemático constantemente. Logo, a maioria das funcionalidades relacionadas a tais tipos foram portadas diretamente para C#. Ainda que o processamento em C++ seja mais rápido, o *overhead* para a execução da chamada nativa já ultrapassa o tempo para a mesma execução localmente. Além disso, a criação de tais tipos utilizando o *assembly* Mono.Simd permite que a máquina virtual acelere o processamento em C#, utilizando instruções *SIMD* para sua execução, garantindo quase o mesmo desempenho que o código nativo [30].

- Uma análise quanto ao modo de uso das coleções foi realizado, separando-as em duas categorias: as de transporte imediato e as de transporte tardio.

  – **Coleções com transporte imediato** realizam uma chamada ao método externo para cada operação, realizando o *marshalling* dos dados sempre que necessário. São boas alternativas quando a coleção sofre poucas mudanças, possuem muitos dados ou somente possuem tipos *blittable* (veja 5.2.3).

    Ponto forte:

    * Uma vantagem clara de sua utilização é que os dados já estão presentes na memória nativa e basta passar o `handle` (referência ou ponteiro para a coleção nativa) para o método que for utilizá-la.

    Ponto fraco:

    * A desvantagem dessa abordagem é que para adicionar ou obter um item da coleção é realizada uma chamada externa, onde muitas chamadas podem afetar drasticamente o desempenho.

  – **Coleções com transporte tardio** realizam a transferência somente quando necessário. Os dados são enviados pelo método `Flush` e são recuperados pelo método `Load`. O envio e carregamento dos dados é realizado em lotes e pode ocorrer em uma ou mais chamadas externas. A quantidade de objetos a serem enviados é baseada no tamanho em *bytes* desses objetos, sendo 4KB o tamanho máximo do lote de objetos, por padrão.

    Pontos fortes:

    * Ocorre uma redução no número de chamadas externas, ganhando-se tempo;
    * Pode-se controlar quando tais dados serão enviados ou carregados para o lado nativo.

    Pontos fracos:

    * Objetos podem ser duplicados, estando na coleção nativa e na coleção gerenciada;
    * Alterações diretas no objeto nativo só estarão no objeto gerenciado se esse for recarregado, através do método `Load`. Da mesma forma, alterações no objeto gerenciado só estão presentes no objeto nativo se o método `Flush` for executado.

- Pode-ser utilizar da técnica denominada como *vtable* artificial (veja 5.2.4) para se criar um *wrapper* que seja exatamente uma implementação (por herança) de uma classe nativa. Além disso, com o uso de tal técnica é possível que dados sejam obtidos e transferidos do lado nativo diretamente, e vice-versa, somente realizando *marshalling*, quando necessário.

5.2.3 *Blittable types*

Tipos *blittable* são tipos de dados que podem ser transferidos entre o ambiente gerenciado e nativo com o mínimo de esforço computacional, através de uma simples cópia de memória.

Uma tabela contendo os tipos *blittable* e suas respectivas representações nativas é apresentada a seguir:

C#	C / C++	<stdint.h>
sbyte	char	int8_t
byte	unsigned char	uint8_t
short	short	int16_t
ushort	unsigned short	uint16_t
int	int	int32_t
uint	unsigned int	uint32_t
long	long (Win32), long long (GCC)	int64_t
ulong	unsigned long (Win64), unsigned long long (GCC)	uint64_t
char	unsigned short	uint16_t
float	float	
double	double	
IntPtr	void* ou outro ponteiro	
UIntPtr	void* ou outro ponteiro	
bool	depende do contexto (veja 5.2.3)	

Tipos mais complexos, seguem as seguintes regras:

- Uma estrutura que contenha somente tipos *blittable* também é considerada *blittable*. Isso se deve porque uma estrutura em memória é apenas um bloco de *bytes*, onde seu tamanho é a soma do tamanho dos tipos internos em *bytes*;
- Um *array* unidimensional de um tipo *blittable* também é um tipo blittable;
- Todos os demais tipos não são *blittable* e requerem alguma forma de *marshalling* para a transferência de dados.

Dependência de contexto para tipos booleanos

O tipo booleano recebe um tratamento especial. Dentro de estruturas, por padrão, o tamanho de um booleano é 4 *bytes*. Como parâmetro em uma função, seu tamanho se torna 2 *bytes*.

Entretanto, é possível especificar o tamanho do tipo booleano explicitamente, utilizando o atributo `MarshalAsAttribute` sobre a definição do tipo ou sobre o método

de utilização. O tipo booleano pode assumir os seguintes tipos: `UnmanagedType.Bool` (4 *bytes*), `UnmanagedType.VariantBool` (2 *bytes*) e `UnmanagedType.U1` (1 *byte*).

5.2.4 *VTable* Artificial

Algumas vezes existe a necessidade de se estender o comportamento de determinada classe nativa, como por exemplo, quando implementando uma classe *listener* ou uma `ManualObject` da Ogre. A primeira serve para tratar eventos de alguma natureza (teclado, mouse etc) e a segunda para criar um objeto que pode desenhar formas manualmente.

Em geral, os *bindings* e *wrappers* encapsulam somente funcionalidades e não permitem um sistema de herança de tipos de objetos completo, como descrito em 5.1.1.

Nessa seção, é apresentada uma técnica, baseada no conceito de `vtables` (ou tabela de métodos virtuais), criada por esse projeto para simular a herança entre uma classe em C++ e outra em C#. Andreia Gaita, uma contribuidora do projeto Mono, e Alex Corrado, um candidato ao Google Summer of Code de 2011, estão trabalhando em algo similar, mas numa abordagem dinâmica. A ideia original de Corrado realiza mudanças diretas na *vtable* de um objeto em C++, em tempo de execução, baseada em geradores e uma alteração no compilador.

A principal diferença entre as duas técnicas, a de Corrado e a criada por esse projeto, a qual foi denominada de *vtable* artificial, é que a primeira altera diretamente a *vtable* do objeto, enquanto a segunda, utilizada no projeto, simula o comportamento através de indireções de chamadas de métodos. Além disso, a implementação contida no projeto InVision torna-se independente de plataforma e não depende da criação de um dialeto personalizado para cada compilador de C++ (por exemplo: G++ e o Microsoft C++).

A figura 5.1 exemplifica o funcionamento da técnica, que é descrita abaixo:

- A classe `Vector3` é uma classe nativa em C++ que deve ser estendida em C#. Para que isso seja possível, será necessário a construção de um Proxy (classe `Vector3Proxy`) que herda diretamente de Vector3 e permite sua ligação com a classe em C#;
- Para cada método virtual da classe `Vector3` cria-se uma variável (A) e dois métodos, um estático (B) e outro que sobrescreve o método original (C);
- A variável A é um ponteiro para um método que possui a mesma assinatura do método B. Tal variável faz o papel do registro do método na `vtable` e possui, inicialmente, o endereço do método B;
- O método B possui basicamente a mesma assinatura do método C, exceto que adiciona como primeiro parâmetro um ponteiro para o tipo do proxy, que apontará

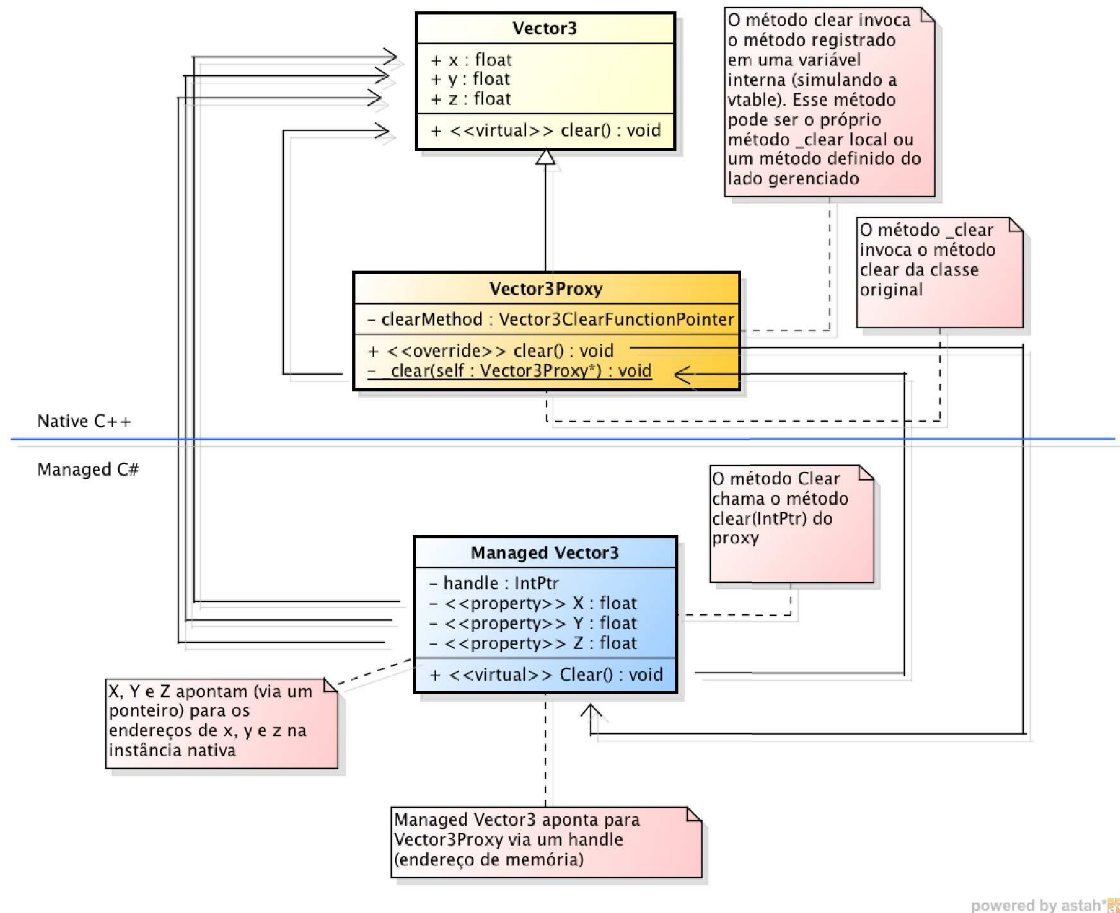

Figura 5.1: *Construção de* wrappers *com vtable artificial.*

para o objeto a ser invocado. Ao ser invocado o método *B* invoca o método original, isto é, o método `Vector3::clear`, no exemplo;

- O método *C* invoca a função registrada na sua *vtable* artificial, isto é, a função apontada por *A*;
- Cria-se mais um tipo de valor que serve para transferir as informações do objeto criado para o lado gerenciado, para que o tipo seja montado em tempo de execução. O objeto contendo as informações do proxy possui os seguintes dados:
  - Um ponteiro para o próprio proxy (handle);
  - Um ponteiro contendo o endereço de cada campo do objeto (público ou protegido);
  - Um ponteiro para cada variável *A*, criada anteriormente, definindo o conteúdo da *vtable* artificial.
- Para que os dados sejam transferidos do lado nativo para o gerenciado, uma chamada externa comum é realizada no lado gerenciado, criando a instância e retornando as informações desta;
- A classe `Vector3` em C# obtém as informações, durante a construção do objeto

gerenciado. Em seguida, armazena tais informações e cria uma cópia dos valores da *vtable* original transferida. Após a cópia, sobrescreve os mesmos, fazendo a *vtable* no proxy apontar para um *delegate* (ponteiro seguro para um método) no ambiente gerenciado, que por sua vez invoca o método local;

- O tipo em C# possui propriedades que representam os campos no objeto original. Cada propriedade possui o tipo nativo em C# e encapsula qualquer transformação necessária entre o valor do campo original e o local. Usa-se ponteiros em C#, um recurso da linguagem, para que seja possível ler e alterar os dados originais diretamente, sem a necessidade de realizar uma chamada externa para isso, somente realizando *marshalling*, quando necessário;
- A grande diferença dessa abordagem é que para invocação de métodos ou leitura de dados não é necessária a realização de uma chamada externa, pois os objetos em C# manipulam diretamente os dados originais, através de ponteiros, melhorando o desempenho. O ganho de tempo e processamento em relação aos métodos é muito pequeno, mas ganha-se em relação aos dados, pois a manipulação é feita diretamente na memória, sem a necessidade de uma função e, consequentemente, da criação de sua pilha de execução.

Essa técnica possui um ponto negativo que é visível. Por exemplo, se houver uma chamada nativa para um *proxy*, esse irá redirecionar a chamada para sua versão gerenciada em C#. Contudo, se o método não foi sobrescrito pelo usuário, esse continuará com a implementação padrão que irá direcionar novamente a chamada para a função invocadora no lado nativo, que então irá chamar o método original. Assim, várias indireções de chamadas acontecem até a funcionalidade ser invocada, que pode deteriorar o desempenho da aplicação, principalmente quando for necessário a realização de *marshalling* (uso de *strings* como parâmetros da função, por exemplo) para enviar e obter os dados.

Para evitar tal problema, os métodos gerados são marcados com um atributo interno que não pode ser herdado, quando o método for sobrescrito. Assim, toda vez que um método virtual na classe em C# é sobrescrito, o atributo é perdido. Dessa forma, com o uso de reflexão, é possível selecionar, durante a construção do objeto, quando se deve ou não sobrescrever um método na *vtable* artificial.

Os métodos que ainda possuem o atributo, não foram sobrescritos, sendo assim, não há necessidade de sobrescrevê-los na *vtable*. De forma contrária, os métodos que perdem o atributo foram sobrescritos, assim também é necessário que estes sejam sobrescritos na *vtable*. Ao final, evita-se pelo menos duas indireções de chamadas, quando não há necessidade.

5.2.5 Problemas com ponteiros e conversões

Um problema encontrado quando utilizando a técnica de *vtable* artificial (veja 5.2.4) está na manipulação de objetos em memória, através de ponteiros.

Código 5.5 Problema com a conversão de ponteiros em C++

```
1 Hello* new_HelloWorld()
2 {
3         HelloWorldBR* instance = new HelloWorldBR();
4         Hello* ret = instance;
5
6         return ret;
7 }
```

O código 5.5 mostra uma simples função, onde uma instância de `HelloWorldBR` é criada e então converte-se esta instância para o tipo base `Hello`, retornando a conversão para o usuário.

Uma vez que a instância de `HelloWorldBR` herda de `Hello`, aparentemente não há problema nenhum.

Se instância de `HelloWorldBR` declarar mais algum campo ou método em sua estrutura, logicamente ela terá um tamanho em `bytes` maior que a primeira e é nesse ponto que reside o problema. Esse tamanho adicional pode influenciar no endereçamento dos ponteiros.

Voltando ao exemplo anterior, suponha que uma instância de `Hello` use 4 *bytes* de memória e uma instância de `HelloWorldBR` use 12 *bytes* (4 *bytes* (Hello) + 8 *bytes* local), o que resulta numa diferença de 8 *bytes*. Suponha também que quando a instância de `HelloWorldBR` foi criada, um ponteiro com endereço de valor 1000 foi retornado. Ao se fazer a conversão para o tipo base, na linha 4 do programa, o endereço retornado não será o endereço original, pois o ponteiro será movido em 8 *bytes* (a diferença), para mais (1008) ou para menos (992).

Tal diferença no endereçamento pode ocasionar vários problemas, como a escrita em uma área de memória que exceda os limites do objeto ou ainda a execução de métodos com dados errados, pois os valores na instância serão lidos com base em um endereço diferente do original.

O *framework* InVision resolve esse problema com duas abordagens diferentes. Na primeira, quando manipulando ponteiros diretamente, evita-se chamar métodos que requerem conversões de ponteiros entre tipos hierárquicos, mesmo que isso ocasione na criação de métodos que façam uma indireção de chamadas. Dessa forma, é criada uma função que recebe o tipo referente ao ponteiro (objeto completo), sendo a conversão (*casting*) de ponteiros realizada em C++ automaticamente e de forma segura.

A segunda abordagem utiliza *handles*, que são apenas valores inteiros, como chaves de identificação de uma tabela *hash*, que são passados como identificadores de objetos [44]. Assim, o valor do ponteiro não é alterado e somente um inteiro que identifica tal objeto é passado ao ambiente gerenciado. Essa abordagem apesar de mais segura, causa um impacto no desempenho devido ao processo de *lookup*, procura do ponteiro original dado o *handle*, e a necessidade de execução dinâmica, em alguns casos.

5.2.6 *Layout* de objetos em memória

Layout de objetos em memória é uma expressão usada para se definir como os dados desses objetos se encontram armazenados fisicamente. Os dados dos objetos são armazenados em blocos de *bytes* na memória principal e a leitura eficiente desses blocos depende do empacotamento dos dados [44].

Processadores leem dados da memória em blocos múltiplos de um *byte* (8-bit), geralmente quatro `bytes`, e dependendo do tamanho dos dados e sua posição em memória, será requerido processamento adicional. *"Por exemplo, se um programa requer que um inteiro de 32-bit (quatro* bytes*) seja lido do endereço 0x6A341174, o controlador de memória irá carregar os dados feliz, porque o endereço está alinhado em quatro* bytes *(nesse caso, sua parcela menos significativa é 0x4). Entretanto, se uma requisição é feita para carregar um inteiro de 32-bit do endereço 0x6A341173, o controlador de memória agora tem que ler* dois *blocos de quatro* bytes*: um em 0x6A341170 e o outro de 0x6A341174."* [44]. Após a leitura, o controlador de memória irá ajustar os bits lidos, movendo bits, para que então possa carregar ao registrador, somente o valor requerido.

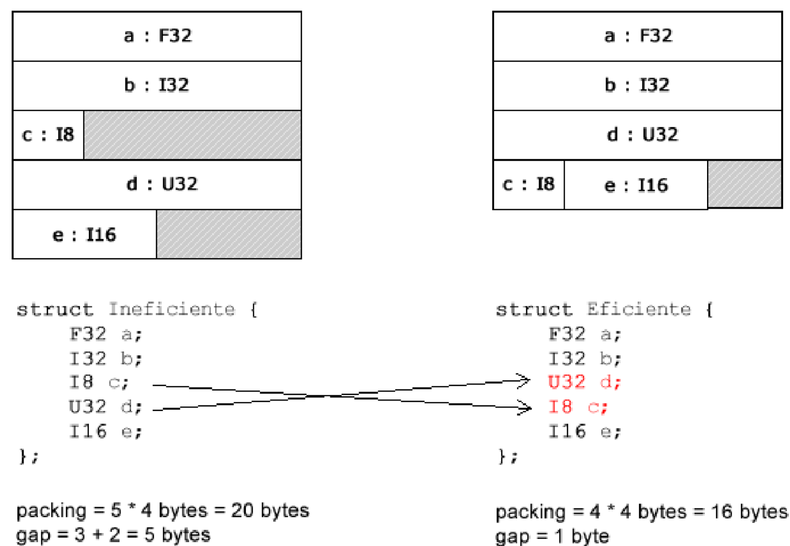

Figura 5.2: *Empacotamento de dados em memória*

A figura 5.2 demonstra o problema de empacotamento. Ambas estruturas (`struct Ineficiente` e `struct Eficiente`) possuem um tamanho de quinze *bytes*,

sendo os tipos F32, I32 e U32 de quatro *bytes* (32-bit), o tipo I8 (8-bit) de um *byte* e o tipo I16 (16-bit) de dois *bytes*.

Em memória, a estrutura `Ineficiente` irá ocupar um espaço de 20 *bytes* em memória, ocorrendo um desperdício de 5 *bytes*. A estrutura `Eficiente` irá ocupar 16 *bytes*, desperdiçando somente 1 *byte*. Tal desperdício não é tratado pelo compilador, pois o alinhamento dos dados deve ser respeitado, garantindo a interoperabilidade entre sistemas, como acontece durante a construção dos *bindings*.

Segundo Gregory [44], o ideal é que até o espaço vazio (*gap*) também seja mapeado na estrutura, tornando-a uma opção melhor que a `Eficiente`. A razão para tal situação é fácil de ser imaginada, pois resolve o problema do exemplo anterior. Se duas estruturas `Eficientes` fossem armazenadas em sequência, então durante a leitura da segunda, ocorreria o problema de posicionamento em memória, pois a segunda estrutura estaria em uma posição que não é múltipla de dois, ocasionando a movimentação de `bytes` (`shift`) pelo controlador. Se o espaço é preenchido, então força-se a próxima estrutura a estar em um endereço de memória que é múltiplo de dois, facilitando para o controlador de memória, durante a leitura.

Dessa forma, as estruturas de valor usadas para transferir os dados entre o lado gerenciado e não gerenciado, foram criadas respeitando as observações de Gregory e mapeando inclusive os espaços vazios, para uma alocação ideal.

5.3 Construção do *framework*

O papel principal de um *framework* é prover componentes prontos, ou semi-prontos, para que uma nova aplicação possa ser construída.

O *framework* InVision se divide em vários módulos (*assemblies*). A árvore de dependência desses módulos, incluindo as duas camadas inferiores, está representada na figura 5.3, onde:

- Os blocos em vermelho representam os artefatos gerados para a camada de Suporte Nativo;
- Os blocos em verde representam os artefatos gerados para a camada de *Bindings*;
- Os blocos em azul representam os artefatos gerados para a camada do *framework*.

Boa parte das funcionalidades do projeto proposto foram baseadas no motor de jogo Panda3D e no Microsoft XNA Framework. O *assembly* InVision provê um conjunto de funcionalidades que complementam as já providas pelos `bindings` e bibliotecas das camadas inferiores. Uma lista de tais características está abaixo:

- Extensões utilitárias;

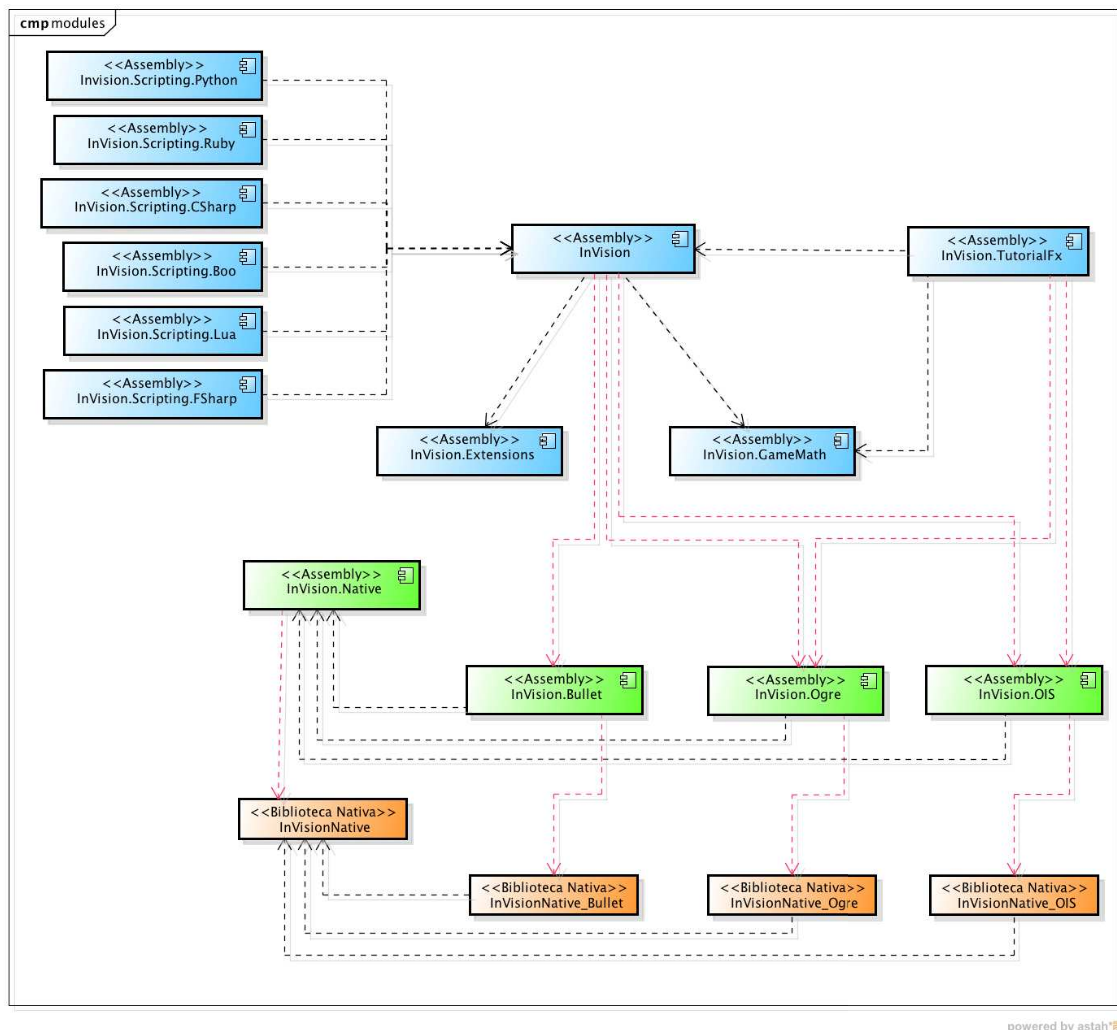

Figura 5.3: *Módulos presentes no* framework *InVision*

- Abstrações matemáticas para cálculos vetoriais, relacionados à Computação Gráfica, com suporte SIMD;
- Um modelo de jogo pré-definido para jogos e simulações, com suporte a subsistemas;
- Fluxos de execução customizáveis;
- Suporte a *Scripting* em várias linguagens;
- *API* simplificada para áudio e renderização;
- *API* para persistência de objetos;
- *API* para comunicação entre aplicações via Rede de Computadores;

5.3.1 Extensões utilitárias

"Encapsulamento é uma das características mais importantes da programação orientada a objetos... Ela reforça a separação de responsabilidades, implementação e políticas de segurança [para objetos]" [17].

Contudo, muitas vezes ao se trabalhar com determinados objetos, criamos novas funcionalidades, dependentes do contexto de trabalho, para estender as ações do objeto original. Geralmente, tais funcionalidades são métodos que se encontram na classe de trabalho, e não no objeto utilizado, pois nem sempre temos domínio sobre as definições dessa instância.

Algumas linguagens dinâmicas, como Ruby, permitem que classes sejam estendidas em qualquer ponto de execução, bastando uma nova redefinição do tipo que se deseja estender. O que permite a inserção e alteração de métodos dinamicamente [3].

A linguagem C#, apesar de conter recursos de linguagens dinâmicas, não é uma linguagem dinâmica e, portanto, não permite que seus tipos sejam estendidos posteriormente. Contudo, ainda existe a possibilidade de extensão através de um mecanismo construído para a linguagem, que simula essa execução, os *métodos de extensão*.

Apesar de ser somente um recurso estético sobre a sintaxe, o uso de extensões permite que novas funcionalidades sejam aplicadas a grupos de objetos de um mesmo tipo, simulando a chamada como se fosse um método definido pelo próprio objeto.

Por exemplo, seja o método `IEnumerable<int> Enumerable.Range(int from, int to)` responsável por criar uma enumeração de inteiros no intervalo especificado, $[from, to[$. Para se utilizar tal funcionalidade é necessário chamar o método diretamente, como em `var values = Enumerable.Range(0, 10)`. Uma utilização mais amigável ao programador pode ser obtida, transformando o método `Range` em uma extensão para números inteiros, apenas com a alteração de sua assinatura para `IEnumerable<int> Enumerable.Range(this int from, int to)`. Assim, todos os inteiros possuem agora o método Range, o que permite que a definição anterior seja escrita como `var values = 0.Range(10)`. E ao renomear o método de `Range` para `UpTo`, a expressão se torna `var values = 0.UpTo(10)`.

O *assembly* InVision.Extensions basicamente provê extensões para objetos que são utilizados constantemente pelo *framework*. Basicamente, são extensões ligadas a reflexão, manipulação de *strings*, checagem de tipos e validação. Tal extensões, podem ser somente utilizadas pela linguagem C#, de modo natural, por isso sua utilização está focada no uso pelo próprio *framework*, mas não está limitada a este.

5.3.2 Abstrações matemáticas

> *"Um jogo é um modelo matemático de um mundo virtual simulado em tempo-real em um computador de algum tipo"* [44].

Pela própria natureza de um jogo, várias estruturas e representações matemáticas de conceitos presentes em Álgebra Linear e Cálculo Numérico se fazem necessários, não obstante aos conceitos de Computação Gráfica, em si.

Dessa forma, utilizar de abstrações para os modelos matemáticos mais utilizados é algo essencial em jogos e simulações.

A construção de um mundo virtual começa pelo seu sistema de coordenadas, onde vértices definem posições no espaço, mas também podem representar direções, forças, coordenadas para a construção de uma malha ou polígono etc. Além disso, tem-se *quaternions* para representação de rotações sobre um eixo. Matrizes quadradas de tamanho 4, que podem representar translações, rotações e escalas, numa única instância.

O *assembly InVision.GameMath* encapsula vários tipos que representam conceitos matemáticos, principalmente os utilizados em Computação Gráfica, e provê um modo simples para realizar processamentos matemáticos com tais objetos. Tal *assembly* foi construído com base no código da biblioteca *open-source* `Mono.GameMath`, mantendo-se os devidos créditos. Entretanto, o conteúdo original continha funcionalidades incompletas e com erros, assim alguns métodos foram reimplementados posteriormente (ainda será criado um *patch* para a biblioteca original com tais alterações).

A biblioteca matemática do projeto InVision é a mesma da biblioteca *Mono.GameMath* mas com algumas correções e funcionalidades adicionais. Por utilizar o *assembly Mono.Simd* internamente, para a definição dos tipos matemáticos, ganha-se o suporte à aceleração de cálculos com instruções SIMD, por consequência. Esse suporte é provido pelo Mono para alguns tipos, como: `Vector2`, `Vector3`, `Vector4` etc. Quando utilizando-os, por exemplo, para realizar uma multiplicação entre dois vetores, o Mono *Runtime* irá converter as instruções em `bytecode` para código nativo, mas otimizando o código para que esse utilize instruções SIMD.

5.3.3 Modelo de Aplicação para Jogos e Simulações

O *assembly* InVision provê um modelo de aplicação suportado pelos *bindings* e *wrappers* providos pela camada inferior. Tal modelo possui um fluxo de execução padrão, que pode ser substituído por um implementado pelo usuário (veja 5.3.4);

A aplicação principal é representada pela classe `GameApplication`. Cada instância de uma `GameApplication` possui um conjunto de estados, chamados de `GameState`, que definem transições entre si, formando um grafo direcionado. Cada instância de `GameState` pode ser imaginada como uma cena, como no exemplo da figura 5.4.

Cada `GameState` é responsável por carregar os objetos daquela cena (modelos, *meshes*, animações etc), preparar o ambiente (definir o gerenciador de cenários, vincular os objetos à cena etc), processar as entradas do usuário (oriundas de um dispositivo suportado pela OIS), dentre outras possíveis realizações.

As ligações entre os `GameStates` são realizadas por instâncias de `GameStateTransition`. Cada transição se assemelha ao próprio `GameState` em ter-

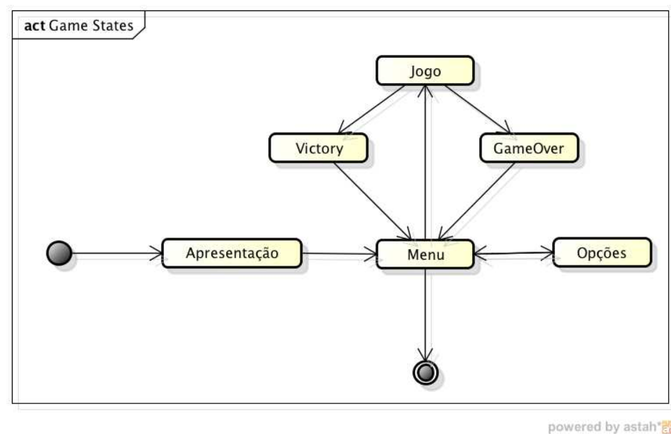

Figura 5.4: *Macro Estados em um Jogo*

mos de funcionalidades, mas são responsáveis por levar de um estado a outro, podendo aplicar efeitos durante o trajeto. Assim, um `GameState` pode conter várias transições para outros `GameStates`, desde que cada transição seja identificada com uma mensagem única.

Ao se enviar uma mensagem para um `GameState`, esse irá checar se existe alguma `GameStateTransition` que aceita a mensagem especificada. Se houver, a transição é ativada e assume a responsabilidade pela renderização atual, controlando até quando o `GameState` de origem será executado e quando iniciar o próximo `GameState`. Se não houver transição, uma mensagem de erro será apresentada e o jogo finalizado.

Por questões de facilidade para usuários iniciantes, pode-se criar um jogo com somente um `GameState` e sem transições.

O conjunto de instâncias de `GameStates` e `GameStateTransitions` são controlados pela classe `GameStateManager`, referenciada pela classe `GameApplication`. A figura 5.5 apresenta um diagrama de classes demonstrando a organização da aplicação e seus estados.

5.3.4 Fluxos de execução customizáveis

A execução de um jogo, em geral, funciona como apresentada na figura 5.6. Entretanto, é possível criar variações, por exemplo:

- Durante a atividade de simulação do mundo virtual, pode-se separar objetos a serem processados e paralelizar suas ações;
- Pode-se paralelizar tudo, incluindo o tratamento de entradas, simulação do mundo e renderização. O que requer um cuidado de sincronização;
- Pode-se criar serviços que executem em paralelo, como a comunicação via rede ou outro serviço qualquer, e deixá-los executando em *threads* paralelas à *thread* principal.

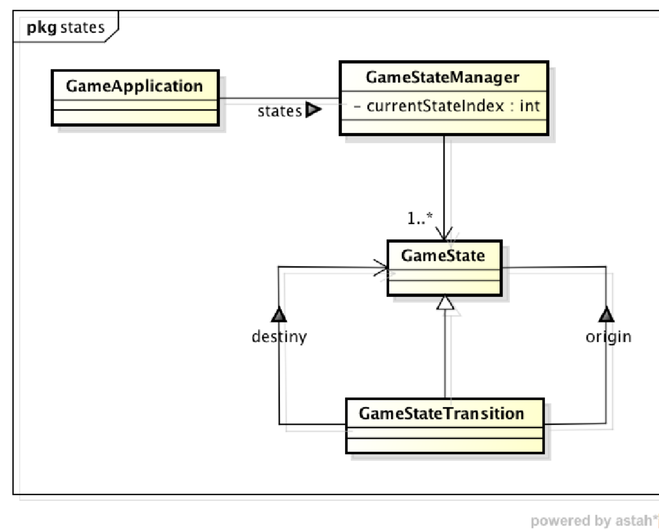

Figura 5.5: *Diagrama de classes demonstrando o modelo da aplicação e seus estados*

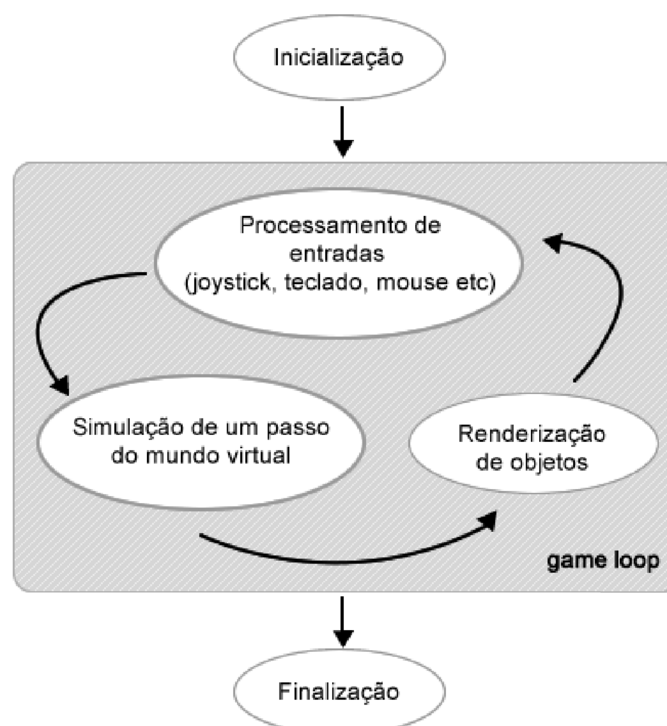

Figura 5.6: *Fluxo de execução em um jogo*

Devido as possíveis variações quanto ao modo de execução do jogo, e principalmente suas táticas de paralelização e otimização, o conceito de fluxo de execução customizável foi criado.

Fluxos de execução customizáveis definem basicamente como que uma `GameApplication` deve ser executada, incluindo como os serviços registrados por tal aplicação serão gerenciados.

Um fluxo de execução customizável é definido pela classe abstrata `GameFlow`. Tal classe possui somente um método, com a seguinte assinatura: `void Run(GameApplication app)`. Pode-se implementar um fluxo de execução e substituir o fluxo de execução principal da aplicação (`DefaultGameFlow`), apenas alterando um item nos arquivos de configuração ou manualmente, durante a inicialização da aplicação.

Um exemplo de fluxo de execução monolítico, implementado em C#, está no código 5.6.

Código 5.6 Exemplo de um fluxo de execução customizado

```
using System;
using InVision;
using InVision.Configuration;
using InVision.Ogre;
using InVision.OIS;

public class SimpleGameFlow : GameFlow
{
        public override void Run(GameApplication app)
        {
                // Inicialização da OGRE
                var config = new Configuration();
                config.Screen.Width = 800;
                config.Screen.Height = 600;
                config.Screen.Background = ColourValue.Black;

                app.Initialize(config);

                // Game loop
                while (app.IsRunning) {
                        var state = app.States.Current;

                        if (state == null)
                                break;

                        app.BeginScene();

                        var gameTime = app.Timer;

                        state.BeginFrame();

                        foreach (var obj in state.Components) {
                                obj.Update(gameTime);
                        }

                        state.EndFrame();
                        app.EndScene();
                }
        }
}
```

5.3.5 Suporte a *Scripting* em várias linguagens

A possibilidade de se escolher uma determinada linguagem para se trabalhar com o *framework* permite que esse se adapte às necessidades ou preferências do usuário. Dentre as linguagens para *scripting* selecionadas, temos: Boo [93], IronRuby [92], IronPython [65], Lua [60], C# Script [70] e F# [64] .

Pode-se escolher entre interpretar ou compilar e executar os *scripts*. A responsabilidade de configurar (compilando ou somente interpretando) cada *script* fica a cargo do gerenciador de *scripting* (`IScriptManager`). Cada gerenciador possui uma implementação específica para a linguagem utilizada, estando em conformidade com a interface `IScriptManager`.

O módulo principal (InVision) define a interface `IScriptManager`, com uma assinatura que permite integrar novos mecanismos de *scripting* em .NET, que é implementada pelos *assemblies* que possuem o nome *InVision.Scripting.(linguagem de destino).dll*. Entretanto, o próprio módulo InVision utiliza os serviços dessa interface. Dessa forma, os gerenciadores de *scripts* são carregados em tempo de execução, através de carregamento dinâmico por reflexão.

Scripts que utilizam padrões de convenção (seção 5.4) são importados para dentro das classes especificadas (`ScriptableGameState`, `ScriptableGameObject` etc) e executados sobre um contexto seguro, expandindo as funcionalidades de um determinado objeto.

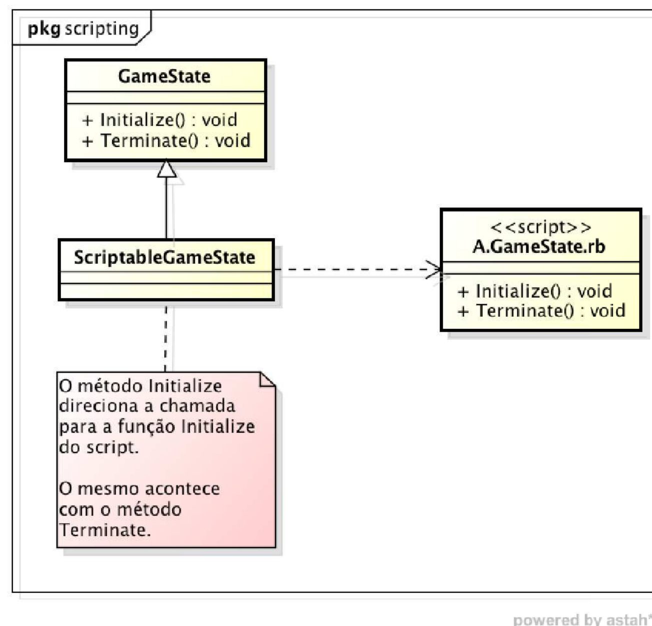

Figura 5.7: *Delegação de chamadas para um arquivo de script*

É importante frisar que algumas linguagens de *script* do *framework*, como *Boo*, permite que se defina no *script* um tipo que herde diretamente de um tipo qualquer.

Isso permite a extensão do *framework* por *scripts*, sem a necessidade da delegação de chamadas, como mostrado no exemplo 5.7, tudo é realizado por herança direta entre tipos.

5.3.6 *API* simplificada para Áudio e Renderização

Algumas vezes, o usuário quer somente reproduzir um arquivo de áudio ou desenhar algo na tela. O *framework* InVision provê *wrappers* e *bindings* para as bibliotecas FMod e Ogre, para reprodução de áudio e renderização, respectivamente.

Entretanto, tais bibliotecas requerem configurações avançadas ou muitos passos de configuração. Dessa forma, o *framework* também provê *APIs* simplificadas para tais serviços. Tais *APIs* não permitem muitas configurações, sendo somente algo para facilitar o desenvolvimento por pessoas que não têm muito conhecimento da área, ou querem criar algo mais rapidamente.

Existem basicamente duas abstrações simplificadas:

- As classes `Sound` e `Music`, que tem suas instâncias criadas pela classe `SoundFactory`. Permitem a reprodução simplificada de sons e músicas, através de métodos simples de *playback*, como: `Play`, `Pause` e `Stop`;
- A classe `Graphics`, que permite um meio simplificado para a renderização de formas na tela, de forma direta, mas ainda independente da *API* de desenho.

5.3.7 *API* Persistência de objetos

Assim como qualquer software, jogos também necessitam armazenar dados de forma confiável. Cada gênero de jogo pode requerer estratégias diferentes quanto à velocidade de armazenamento e consulta, implicando em novas pesquisas sobre armazenamento, como a utilização de bancos de dados declarativos [104] ou ainda de novos mecanismos que garantam o desempenho e grande quantidade de usuários conectados [106].

A proposta principal do *framework* é prover um conjunto de funcionalidades para jogos, que garantam desempenho, mas que ainda sejam fáceis de utilizar. O uso de *SQL* diretamente pode se tornar algo mais complicado ao usuário, além de requerer possíveis mapeamentos dos dados relacionais para objetos e o uso de um servidor *SQL* ou uma *API* embutida, como o SQLite [94].

O ambiente ideal, para iniciantes em Computação ou para aqueles que querem focar diretamente na lógica do seu projeto, seria aquele que permitisse facilidade para a integração. Isso é possível utilizando um banco de dados orientado a objetos, como o Db4o [102]. O sistema de persistência do *framework* InVision é baseado nesse banco de dados. Algumas de suas características estão descritas abaixo:

- Armazenamento de objetos nativos (entidades), sem a necessidade de mapeamentos ou quaisquer outros mecanismos intermediários (comuns em mapeadores Objeto-Relacional, como o NHibernate [35]);
- Suporte à evolução do banco de dados (adição/remoção de campos, tipos etc), manualmente ou automaticamente;
- Suporte à consultas nativas (através da definição de métodos de pesquisa);
- Suporte à consultas via *Linq* [62];
- Suporte à consultas dinâmicas, através da *API* SODA;
- Permite utilizar um banco de dados local ou remoto.

Existem algumas limitações quanto ao que pode ser persistido. Por exemplo, tipos que possuem ligações diretas para objetos de renderização ou *wrappers* não podem ser persistidos, pois isso invalidará suas referencias ao final da aplicação. Outra limitação, é quanto ao *design* da classe, que deve definir atributos claramente, e não utilizar do recurso de "Propriedades Automáticas", presente em C#, se quiserem usar a *API SODA*, de forma efetiva.

5.3.8 *API* Comunicação via Rede de Computadores

Atualmente, psicólogos estudam os efeitos dos jogos on-line no cotidiano das pessoas, e como tal interação afeta a vida social de cada uma dessas pessoas [23, 86]. Existem também conferências com o tema jogos e saúde, como a *Games for Health Conference*.

Entretanto, construir um jogo que suporte comunicação e interação via Rede de Computadores não é uma atividade simples. Cada gênero de jogo requer níveis diferentes de desempenho e diferentes modos para a realização da comunicação.

Assim, dois modelos de comunicação em rede são apresentados pelo projeto:

- Comunicação baseada numa arquitetura cliente e servidor. Onde servidores processam requisições de clientes, através de pacotes de informação pré-definidos;
- Comunicação ponto-a-ponto. Todos estão conectado a todos, em um grupo de clientes, realizando comunicações diretas.

5.4 Construção do protótipo extensível Tutano

A construção do protótipo extensível Tutano é uma parte requerida do projeto para que este seja de fácil uso e adaptável para o ensino, principalmente pelas disciplinas introdutórias. Tal ambiente, baseado fortemente no *framework* InVision, provê meios para se criar jogos e simulações facilmente, através de *scripts* em diversas linguagens.

Como definido em 4.2.5, o Tutano gerencia cinco diretórios principais:

- **Bin/** - que contém os binários do projeto (bibliotecas gerenciadas e nativas);
- **Libraries/** - onde novas bibliotecas podem ser colocadas, ficando disponíveis automaticamente para a utilização pelos *scripts*;
- **Config/** - que contém os arquivos para a configuração;
- **Content/** - que contém os *assets* (imagens, arquivos de sons etc) e recursos utilizados para a criação dos objetos visuais e sonoros;
- **Scripts/** - que contém a lógica do jogo, na forma de *scripts* escritos em modo texto, em diversas linguagens.

A diferença entre o Tutano e o serviços providos pelo *framework* está no modo de uso. O *framework* disponibiliza funcionalidades para a construção de ambientes virtuais, englobando bibliotecas nativas que proveem e garantem desempenho e qualidade, no produto final. Entretanto, requer que configurações sejam realizadas manualmente ou através de arquivos de configuração, o que dificulta sua adoção para iniciantes em Ciência da Computação.

O Tutano visa diminuir a quantidade de configurações, provendo uma aplicação semi-pronta, baseada em *convenções sobre configurações*, padrão conhecido em ambientes de linguagens dinâmicas [89].

As convenções estão vinculadas ao nome do arquivo de *script* e/ou diretório onde esse se encontra. Todos os arquivos de *script* devem possuir a seguinte assinatura: `(Nome do arquivo).(Convenção).(Extensão da linguagem)`.

O *nome do arquivo* é um identificador para a sua funcionalidade. Comumente, cada arquivo possui o mesmo nome do tipo que esse descreve, numa abordagem orientada a objetos, ou uma categoria de funcionalidades, quando definindo um conjunto de funções correlatas.

A *convenção* é opcional e pode ser uma das seguintes opções: *GameObject* ou *GameState*. Se um *script* não definir uma convenção válida, esse será considerado como somente um arquivo contendo lógica do domínio, e não terá o ambiente de execução customizado pelo executor (com a injeção de variáveis comuns, uso de *namespaces* e facilidades para a implementação). Nesse caso, o *script* será importado por um `ScriptableObject`, que irá executá-lo somente uma vez.

A *extensão da linguagem* é obrigatória e se refere à uma das linguagens de *script* suportadas, tais como:

.cs - Linguagem C#;
.py - Linguagem IronPython;
.rb - Linguagem IronRuby;
.fs - Linguagem em F#;
.boo - Linguagem Boo.

Um exemplo de uso do uso de *script* pode ser visto no código 5.7, definido com o nome `Main.GameState.rb`. Tal exemplo, define duas funções de configuração (`Initialize` e `Terminate`), que serão chamadas durante a configuração e finalização do `GameState`.

Código 5.7 Exemplo de um GameState implementado em Ruby

```
1  def Initialize()
2    EnableDevice("mouse", DeviceType.Mouse)
3    EnableDevice("keyboard", DeviceType.Keyboard)
4  end
5
6  def Terminate()
7    DisableDevice("mouse")
8    DisableDevice("keyboard")
9  end
```

O *script* 5.7 será importado para um `GameState`, mais precisamente uma classe `ScriptableGameState`, durante a execução do jogo.

5.5 Análise do processo de construção

A construção de um jogo ou um simulador é algo que, apesar de prazeroso pelos resultados finais, requer muito trabalho [25]. Da mesma forma, construir um *framework* para auxiliar na construção de jogos é uma atividade extensa. Inclusive, pode se tornar maior do que a implementação de muitos jogos [44].

O motivo é o simples fato de que um *framework* deve prover componentes semi-prontos para a construção da aplicação. Isso resulta num grande trabalho intelectual e manual, que varia entre uma análise sobre como um usuário gostaria de utilizar tal recurso, até qual tecnologia, processo e modo de implementação, seriam os melhores para criá-lo.

Um bom projeto começa por um bom trabalho de Engenharia de Software, levantamento de requisitos e usabilidade [74]. Nesse projeto, optou-se por tentar prover as mesmas funcionalidades já existentes em outros projetos, mas com abordagens novas.

Por exemplo, a utilização do Mono como mecanismo de *script* não é algo novo, inclusive já existem softwares comerciais criados com base nesse. Entretanto, boa parte desses softwares, por terem sido construídos há alguns anos atrás, não utilizam recursos como o suporte a objetos dinâmicos, presentes na atual versão 4.0 do .NET *Framework*, que permite integração com mais linguagens dinâmicas, permitindo várias linguagens de *script* e com bom desempenho.

Outro exemplo, foi a técnica de *vtable* artificial, criada a partir da adaptação de métodos para criação de *bindings* e *wrappers* que ainda estão sendo trabalhados. A *vtable* artificial é uma técnica baseada em uma ideia antiga e simples, mas adaptada a um novo contexto.

Para esse trabalho, que ainda se encontra em andamento, muito do que foi feito se baseia em trabalhos de pesquisadores e empresas ao redor do mundo. Tentou-se obter as melhores técnicas e tecnologias para prover um *framework* capaz de construir aplicações, que mesmo sendo educacionais, sejam capazes de se aproximar em qualidade dos jogos comerciais.

A construção do projeto é um processo lento. Todo o *framework* se baseia, quase que inteiramente, na camada de Suporte Nativo (veja 4.2.2) e de *Bindings* (veja 4.2.3). Essas duas camadas são a fonte de maior esforço gasto durante o projeto. Primeiramente, porque é necessário se entender como certas funcionalidades executam, e não somente ver o código, para que se possa pensar num modo de provê-las em *wrappers*. Segundo, porque deve-se escolher o que irá ser portado ou somente disponibilizado, resultando em várias opções: uma simples chamada externa, um ponteiro, uma classe com suporte a herança ou uma simples estrutura. Adicionalmente, ainda existe o trabalho de gerenciamento de objetos em memória, no ambiente nativo.

Construir o *framework* para uma plataforma específica, como o Sistema Operacional Windows, facilitaria bastante o trabalho, pois já existem *wrappers* de qualidade para a *OGRE* e outras bibliotecas, nessa plataforma. Entretanto, são projetos que utilizam C++ *Managed*, o que os impedem de serem utilizados pelo Mono, em plataformas não Windows. E a independência de plataforma é um requerimento para o projeto.

Para se ter uma ideia do problema real, na construção dos *wrappers*, atualmente temos quatro bibliotecas, que realizam chamadas às bibliotecas nativas: Ogre, OIS, FMod e Bullet. Todas as bibliotecas são escritas em C++ e precisam da análise de funcionalidades para sua criação. Uma breve análise na documentação da Ogre irá relevar 768 classes. Multiplicando a quantidade de classes pela quantidade de métodos em cada, aproximadamente teríamos 9000 entradas, que talvez virassem chamadas externas a serem implementadas; e tudo isso somente para biblioteca Ogre.

O trabalho de automação para geração dos *bindings* com o SWIG se mostrou ineficaz nesse caso, por não garantir desempenho. Além disso, certos tipos são exportados de forma não eficaz e as interfaces geradas para sua manipulação não são naturais ao desenvolvedor .NET.

CAPÍTULO 6

Usando jogos em Ciência da Computação

Esse capítulo apresenta como os jogos estão sendo utilizados por pesquisadores e professores, como ferramenta didática, e como poderiam ser utilizados com o novo *framework*, focando principalmente no uso do protótipo. Será apresentada uma comparação entre o ambiente original e o porte para o InVision, apresentando os ganhos e perdas dessa abordagem.

A seção 6.1 apresenta as ideias, com foco na utilização pelos alunos, que levaram à construção da proposta desse projeto. A seção 6.1.1 apresenta o motivo, de forma prática, para se criar um mecanismo que permita a customização de como a execução é realizada.

A seção 6.2 apresenta como o *framework* poderia ser utilizado para o aprendizado de algoritmos e programação. A seção 6.4 traz alguns exemplos de simulações que poderiam ser criadas e como elas poderiam trazer benefícios ao ensino das disciplinas relacionadas. E por fim, a seção 6.4.1 apresenta algumas ideias de como se poderia utilizar do *framework* para trabalhos em disciplinas como Grafos e Otimização.

6.1 O Uso do *framework* Invision

A ideia inicial do projeto InVision era criar uma nova ferramenta que auxiliasse no aprendizado de algoritmos e programação, com foco na visualização do fluxo de execução da aplicação. No entanto, o que seria criado se baseava em linguagens de programação visual, similares às providas pelo Scratch, Alice e GreenFoot; com diferenças apenas sobre a forma de utilização.

Desse ponto, analisou-se o que tais ferramentas possuem de características, modos de utilização e o ganhos relativos à sua aplicação, no curso de Ciência da Computação. Também foram procuradas outras ferramentas e projetos com o mesmo intuito.

Após uma análise, foi constatado que apesar de estarem utilizando jogos na educação, tal uso se resume, basicamente, às disciplinas introdutórias, assim como o esforço para a construção de novas ferramentas. Disciplinas mais avançadas, que poderiam em-

pregar jogos para melhorar o entendimento, continuavam a utilizar o método tradicional de ensino.

Um dos motivos constatados, foi o fato de que as ferramentas para programação visual se tornavam inadequadas, pela sua simplicidade, para tais disciplinas. Da mesma forma, a utilização de motores de jogos acaba por demandar de conhecimento e trabalho extra, além de serem direcionadas à utilização através de ferramentas específicas para a criação de jogos.

Assim, o projeto InVision tentou criar um *framework* que tivesse o poder para gerar jogos com alta qualidade visual, mas que pudesse ser adaptado às disciplinas do curso de Ciência da Computação. Adicionalmente, tal produto deveria tentar facilitar ao máximo sua utilização para estudantes recém ingressos, permitindo inclusive a sua utilização para o aprendizado de algoritmos.

Enquanto não é possível concorrer com as ferramentas de programação visual, no mesmo nível de facilidade, a flexibilidade do *framework* permite que projetos futuros o complementem. Além disso, uma pequena camada de abstração para a construção de jogos 2D foi projetada, sobre a Ogre, para que seja possível a criação de tais jogos de forma rápida e simplificada, em relação ao esforço de se usar a Ogre diretamente.

O *framework* permite que o seu *pipeline* de execução seja alterado. Isso permite novas alternativas quanto ao seu uso, assim como a criação de diferentes projetos e processos de construção para a aplicação (veja 6.1.1).

Por fim, a disponibilização de um pacote de software que permite programação genérica e ainda a criação de jogos, através de bibliotecas integradas e o .NET *Framework*, resolve parcialmente um problema citado por [8]. Este diz que uma das principais frustrações de estudantes, quando criando jogos, são as inconsistências e incompatibilidades de tantos pacotes de software diferentes, envolvidos na criação. A solução parcial para tal problema se deve ao fato de que para a construção de *assets*, ainda são necessários pacotes de softwares externos, providos para artistas.

6.1.1 Escolhendo um fluxo de execução adequado

O protótipo do *framework* permite que seu fluxo de execução seja alterado (veja 5.3.4). Com tal característica é possível adaptar seu uso de acordo com as necessidades do usuário.

Por exemplo, pode-se definir um fluxo de execução extremamente simples, que somente realizará as inicializações necessárias, como talvez abrir uma janela. Isso permitiria que, junto com *scripts* de execução sem contexto (veja 5.4), abordagens mais simplistas fossem executadas, forçando o aluno a aprender como configurar e inicializar sua própria aplicação, seguindo seus próprios modelos e processo de desenvolvimento.

Da mesma forma, tal abordagem permite que o aluno, numa possível tentativa de otimizar o desempenho de sua aplicação, crie mecanismos de execução mais elaborados, utilizando paralelismo por processador, *threads* ou mesmo *micro-threads* (*Continuations e Coroutines* [48]).

Por ser suportado pelo .NET *Framework*, o *framework* permite a utilização de alguns tipos para a criação de *threads* (`Thread` e *Task*), além de vários mecanismos para sincronização (`Mutex`, `Semaphore`, *lock* etc). Adicionalmente, pode-se simular *micro-threads* através do uso da interface *IEnumerable*, numa abordagem já utilizada pela Unity3D [99] e pelo próprio *framework*, em `GameObjects`.

6.1.2 Animações controladas

Uma abordagem ao estilo do Alice (veja 3.2.3), que se baseia na programação de animações, também pode desenvolvida. É possível que um professor crie um ambiente, como o da aplicação em [9], o qual é um labirinto, em uma paisagem com efeitos visuais e climáticos sofisticados. A meta do jogo é que o estudante crie um algoritmo, em C-Sheep (uma variação simplificada da linguagem ANSI C), que leve a ovelha (*sheep*), presente na entrada do labirinto, até a sua bola, "perdida" em algum lugar deste.

O ambiente pode ser realizada facilmente com a Ogre, permitindo inclusive efeitos visuais tão aprimorados quanto os utilizados no "The Meadow". Para a programação, pode-se escolher uma das linguagens de *script* (veja 5.3.5) suportada pelo *framework*. Para o algoritmo do aluno que contém a lógica para encontrar a bola da ovelha, pode-se criar um *script* que adiciona um comportamento a um `GameObject`, como no exemplo de código em 6.1.

Código 6.1 `Exemplo de algoritmo em Boo, simulando o C-Sheep`

```
1  sheep = GameState.sheep
2
3  def IEnumerable Update(time):
4          while true:
5                  yield sheep.MoveForward()
6
7                  if sheep.GotBall():
8                          yield sheep.Found()
9                          break
10
11                 if !sheep.CanMoveForward():
12                         yield sheep.TurnRight()
```

Abordagens como essa, oriundas da metáfora da tartaruga de sistemas de aprendizado em Logo, podem ser criadas para a visualização direta de algoritmos. Tal mecanismo ainda pode ser aperfeiçoado, utilizando animações para a movimentação dos atores em cena, e adicionando obstáculos. Além disso, pode-se modificar a aplicação para uma proposta que utilize realidade aumentada ou interfaces tangíveis (através do controle WiiMote, suportado pela biblioteca OIS) [91]. O usuário poderia, através de movimentos, brincar, enquanto especula em como transformar seus pensamentos, em algoritmos.

6.2 O ensino de programação

São várias as dificuldades para o aprendizado de algoritmos e de programação (veja 3.2). Aprender a construir algoritmos é aprender a pensar de forma lógica; é conseguir organizar as próprias ideias em passos, os quais serão transformados em código posteriormente.

A utilização dos ambientes de programação visual é algo que se faz necessário para iniciantes, porque aquilo que é codificado pode ser visualizado diretamente. Tais ambientes facilitam para adquirir conceitos de programação, mas não ajudam, diretamente, a aprender uma linguagem de programação.

Uma linguagem não visual perde no aspecto de facilidade e talvez legibilidade, mas ganha em flexibilidade. Contudo, quando utilizada em conjunto com um ambiente, como os propostos na seção 6.1.2, pode-se visualizar o fluxo da aplicação através das ações do ator (a ovelha, no caso). Além disso, a utilização de *scripts* é uma vantagem para o aluno, pois esse não precisa compilar o código, para então ver sua aplicação funcionando.

6.2.1 Diferentes Paradigmas de Programação

O projeto InVision provê suporte para várias linguagens de programação, suportadas pelo .NET *Framework* 5.3.5. Cada uma dessas linguagens possuem seus próprios paradigmas de programação ou podem simular outros.

Aulas de laboratório permitem apresentar, de forma prática, os detalhes quanto às linguagens de programação e seus paradigmas [15]. E como o *framework* provê um meio para a utilização de *scripts* em várias linguagens, pode-se analisar soluções criadas em cada linguagem, seguindo um determinado paradigma, para comparação. Uma compilação sobre as linguagens de *script* providas pelo InVision e seus paradigmas de programação, retiradas de [97], é apresentada a seguir:

C# - linguagem estática, recentemente com suporte dinâmico também. Paradigmas suportados (multi-paradigma):

- Estruturado;
- Imperativo;
- Orientado a objetos;
- Orientado a eventos;
- Funcional (através do Linq);
- Reflexivo;
- Genérico.

Ruby - linguagem dinâmica de propósito geral. Paradigmas suportados (multi-paradigma):

- Estruturado;
- Imperativo;
- Orientado a objetos;
- Reflexivo;
- Funcional.

Python - linguagem dinâmica de programação em alto nível interpretada. Paradigmas suportados (multi-paradigma):

- Estruturado;
- Orientado a objetos;
- Imperativo;
- Reflexivo;
- Funcional.

Boo - linguagem estática de programação, inspirada fortemente no Python. Paradigmas suportados (multi-paradigma):

- Estruturado;
- Imperativo;
- Orientado a objetos;
- Reflexivo;
- Genérico;
- Funcional.

Lua - linguagem de programação multi-paradigma e leve, criada para ser utilizada principalmente como linguagem de *scripts*. Suporta os seguintes paradigmas:

- Imperativo;
- Funcional;
- Orientado a objetos (a simulação não é muito natural, em termos de usabilidade).

6.2.2 Redes e Comunicação via Internet

Para o aprendizado de Redes, pode-se utilizar o protótipo do *framework*. Como esse é extensivo, pode-se criar novas implementações para comunicação de jogos em rede, entre um ou mais clientes.

Com tal abordagem, o estudante irá se deparar com problemas comuns da área, como:

- Serialização de dados;
- Comunicação via UDP ou TCP;
- Construção de *proxies*;
- Desempenho e limites de banda em uma rede de computadores;
- *Firewalls* e bloqueios.

Além disso, estará em um ambiente restrito, onde o desempenho para as respostas é algo que não deve ser ignorado, senão afetará diretamente o jogo.

Testando modelos de comunicação

Um professor pode separar grupos de alunos em sala, pedindo para que cada um construa um pequeno *middleware* de comunicação para os jogos, seguindo algum modelo baseado na topologia da rede ou apenas na comunicação:

- Comunicação em estrela;
- Comunicação circular;
- Peer-to-peer.

Usando MMORPG em Redes

Criação de um jogo no estilo MMORPG (veja 2.3.1). Para isso, será necessário a criação uma aplicação servidor e um protocolo de comunicação para os clientes. Tal aplicação deve ser, por natureza, focada na transmissão de dados em tempo-real e criada de tal forma a suportar milhares de clientes.

Possivelmente, a arquitetura com um único servidor não irá conseguir suportar tantos clientes. Dessa forma, uma melhoria no servidor será necessária, utilizando técnicas de replicação, *clusters* e outras. Por exemplo, pode-se conseguir um número maior de clientes *online* com a criação de vários servidores em paralelo, cada um atendendo a um número fixo de clientes. Contudo, também aparece o problema de sincronia entre os servidores.

Outro aspecto importante que pode ser trabalhado, inclusive usando detalhes do jogo, é como agrupar as conexões de clientes. Por exemplo, pode-se querer agrupar em

um servidor aqueles clientes que se encontram virtualmente mais perto um do outro, no mapa do jogo, ou pela área de origem da conexão (cidade, estado, país etc).

Por fim, pode-se criar um *middleware*, com foco em desempenho, que permita a transparência de conexão entre tais clientes, permitindo a transferência de um cliente entre servidores, sem a necessidade de perda de conexão.

Entretanto, tal abordagem requer a construção do ambiente de jogo, o mundo virtual MMO, o que não é provido pelo *framework* atualmente.

6.3 Computação Gráfica

Um jogo visualmente é basicamente artefatos de Computação Gráfica. Dessa forma, é natural a sua utilização dentro dessa área.

Contudo, o *framework* InVision é baseado na Ogre, que abstrai o uso de tecnologias 3D, como OpenGL e o DirectX. Ainda assim, pode-se explorar tal área de conhecimento, com algoritmos e técnicas de renderização, além da criação e reprodução de efeitos visuais.

Pode-se estender a Ogre ou usar objetos de desenho manual para a criação e manipulação de formas dinâmicas em 3D. Criar novos tipos de objetos visuais, como malhas e roupas (inclusive trabalhando com física para prover efeitos mais realistas).

Outro detalhe importante é testar novas técnicas de *culling*, assim como novos algoritmos para o gerenciamento de cenas, implementadas pela Ogre como um `SceneManager`. O gerenciamento de cenas visa organizar os objetos em memória de forma que seja fácil percorrê-los e que otimize a renderização, ocultando objetos fora da visão da câmera (*culling*).

6.4 Programação de agentes inteligentes

A área de Inteligência Artificial (IA) é uma área que está intimamente ligada aos jogos. A criação de algoritmos de IA em jogos pode facilmente ser acoplada a atores virtuais. Tais atores, após o "raciocínio", desempenhariam as ações necessárias para cumprir uma meta.

O uso dessa área em jogos está intimamente ligada à seção 6.1.2, onde existem agentes, com movimentos e ações pré-definidas, que devem realizar alguma tarefa. Entretanto, tais algoritmos devem ser preferencialmente divididos em etapas ou executados em segundo plano, para que não afete o andamento do jogo.

6.4.1 Teste de algoritmos computacionais com visualização

Ao estudar Grafos e Otimização Combinatória, problemas como o "problema da mochila", "agendamento", "escolha do melhor caminho" e "controle de tráfego" são logo apresentados.

Segundo Conrad [26], a utilização do paradigma orientado a objetos pode melhorar o entendimento de conceitos matemáticos. Assim, expandindo tal representação e dando forma para tais objetos, pode-se garantir resultados ainda melhores.

Por exemplo, alunos estão estudando Grafos, tentando entender como o algoritmo de *Dijkstra* é melhor que o algoritmo Guloso. Em geral, a primeira coisa que um aluno fará, será desenhar um grafo que represente o problema. Então, tentará simular o comportamento do algoritmo, visualizando as alterações de estado e andamento do algoritmo.

Figura 6.1: *Representação de uma cidade no jogo City Ville do Facebook.*

Agora imagine que tal estudante pudesse aplicar o algoritmo que aprendeu, para movimentar um carro ou pessoa, de um ponto a outro, em uma cidade, como a apresentada na figura 6.1. Algoritmos para "roteamento" poderiam testar sua eficácia nesse ambiente simulado, bastando adicionar algumas regras e objetos a mais, como semáforos, pessoas e carros; até a representação do caos oriundo de um determinado problema no trânsito ou em parte da cidade poderia ser simulado.

Um outro exemplo seria a própria representação visual do grafo, onde a medida que o algoritmo avança, os vértices vão mudando de cor ou apresentando novas informações.

Para se chegar a esse ponto, utilizando o *framework*, um trabalho de construção da aplicação é requerido. E o *framework* já provê recursos e funcionalidades para facilitar

no seu desenvolvimento, deixando o desenvolvedor mais focado na lógica do seu mundo virtual, baseando-se na arquitetura já existente.

CAPÍTULO 7

Conclusão

Essa dissertação apresentou o projeto InVision, um *framework* para a criação e utilização de jogos com foco educativo. Tal *framework* se propôs a preencher a distância, de usabilidade, entre as ferramentas específicas para a criação de jogos, que visam a criação de títulos (jogos) de qualidade, e as educacionais, usadas para o ensino de programação. Servindo, inicialmente, de um intermediário entre os dois pontos, mas que pode ser estendido para chegar mais perto de um ou outro extremo.

Inicialmente, pensou-se na criação de mais uma ferramenta para facilitar no aprendizado de algoritmos, utilizando os mesmos conceitos existentes nos ambientes com o mesmo propósito, como o Scratch e o Alice. A nova ferramenta permitiria retirar as restrições e limitações das anteriores, provendo um ambiente de programação mais rico. Contudo, tal proposta iria somente prover um ambiente um pouco menos restrito que o anterior. Assim, o projeto evolui, seguindo uma visão mais ampla, onde a simplificação para a construção de abordagens já existentes e funcionais, num ambiente sem grandes limitações quanto ao que pode ser criado, fosse o verdadeiro produto do trabalho.

O trabalho, então, produziu os seguintes artefatos:

- **Um conjunto de *bindings* compatíveis com o Mono**, escritos em C#, para bibliotecas nativas de alto desempenho, escritas em C++. Tais bibliotecas somente possuíam *bindings* dependentes do Sistema Operacional Windows;
- **Um *framework* genérico para a construção de jogos**. O qual utiliza técnicas e tecnologias atuais, como processamento suporte a SIMD e execução de linguagens dinâmicas, que permitem construir um produto de qualidade e com otimizações para ganho de desempenho na execução de cálculos matemáticos pelos jogos desenvolvidos;
- **Uma aplicação extensível, denominada de Tutano**, capaz de ser estendida através de *scripts* e baseada no conceito de convenções sobre configurações. Tal aplicação permite a rápida prototipação de jogos, mantendo aspectos de qualidade audiovisual e interativa, mas de uma forma simples e rápida. Isto a torna uma boa alternativa em um curso, para apresentação de conceitos de algoritmos ou simulações, que são usados em diversas disciplinas de um curso de Computação;

- **Uma segunda aplicação, construída sobre o Tutano**, seguindo a abordagem definida em “Karel de Robot” [14] e adaptando conceitos do projeto “The Meadow” [9]. Tal aplicação permite o ensino de programação de forma fácil, através da metáfora de controle dos movimentos de um robô e desafia o estudante a construir um algoritmo que leve em consideração as adversidades do mundo virtual, obstáculos presentes na rua. Além disso, devido ao suporte multilinguagem do projeto, pode-se escolher dentre as linguagens providas, aquela que se achar mais indicada para o primeiro contato e ir evoluindo até, por exemplo, a mais complexa, quanto a estruturas de sintaxe, a linguagem C#.

A criação dos *bindings*, a qual é a base do projeto, resultou na técnica de *vtable* artificial, capaz de unir por herança duas classes em sistemas distintos: um gerenciado em C# e outro não gerenciado em C++. Entretanto, devido a grande quantidade de *bindings* que deveriam ser criados para as bibliotecas nativas, assim como a análise requerida para tal, essa camada ainda se encontra em desenvolvimento.

Por fim, o projeto reuniu tecnologias e construiu um *framework* que pode ser estendido e adaptado às necessidades de uso do aluno, do educador ou da instituição de ensino.

7.1 Trabalhos futuros

O *framework* é um conjunto de bibliotecas, que proveem componentes com funcionalidades comuns a jogos. Sendo assim, qualquer pessoa pode estender suas funcionalidades criando novas bibliotecas que utilizem desse núcleo principal.

A aplicação extensível também permite o carregamento dinâmico das funcionalidades das novas bibliotecas criadas, o que requer apenas colocar tais bibliotecas na pasta *Libraries*. Mudanças no comportamento de execução do protótipo podem ser conseguidas através da criação de fluxos de execução customizáveis e da customização de `GameStates`.

Dessa forma, existem algumas possibilidades de trabalho, tais como:

- **A construção de um ambiente, englobando a aplicação extensível, que organize de forma mais fácil e gerencie o conteúdo do jogo a ser produzido**. Tal ambiente poderia ser criado dentro da *IDE*, o que permitiria a correção de erros de sintaxe no código, depuração e o auto-completar de código;
- **A criação de um editor para linguagem visual, como no Scratch**. Tal editor poderia somente transcrever o código gerado para uma das linguagens suportadas ou salvar em um formato intermediário. Usando um formato intermediário, o

código poderia ser convertido para uma linguagem suportada ou ser interpretado diretamente, através de um novo gerenciador de *script* que o entenda;

- **Adaptar abordagens existentes e/ou novas para o ensino de programação**, construídas com base no *framework* e disponibilizadas como uma extensão do Tutano. Exemplos de tais abordagens são a aplicação "The Meadow" e a RoboCode;
- **A construção de um motor de jogo completo**, criado sobre o *framework*, o que permite que jogos mais complexos sejam criados e gerenciados mais facilmente;
- **A criação de suporte para dispositivos de Interface Tangível**, como o Wiimote, através da nova versão da biblioteca OIS. Isto permite a criação de jogos e simuladores que façam uso, ou criem novas formas de interação, de Interfaces Tangíveis (mais detalhes sobre o assunto em [90, 91]).
- **Portar os *bindings* e as devidas bibliotecas nativas, para os Sistemas Operacionais Android e iOS e consoles como o PlayStation 3**. Tais bibliotecas já são portáveis a tais sistemas, com poucas alterações. Portanto, deve-se checar quais seriam as mudanças necessárias para que o *framework* também pudesse ser utilizado em tais dispositivos móveis e consoles.

Referências Bibliográficas

[1] **Microsoft kinect**. Wikipedia http://pt.wikipedia.org/wiki/Kinect, março 2011.

[2] **Playstation move**. Wikipedia http://pt.wikipedia.org/wiki/PlayStation_Move, março 2011.

[3] **Ruby - a programmer's best friend**. http://www.ruby-lang.org/, 2011.

[4] **Wiimote**. Wikipedia http://pt.wikipedia.org/wiki/Wii_Remote, março 2011.

[5] ADAMS, E. J. **Using microsoft's .net platform in cs1 and cs2**. *J. Comput. Small Coll.*, 21:41–41, February 2006.

[6] ALVES, D. F. M.; DE ALBUQUERQUE, E. S. **A framework to support game development**. *IADIS - International Conference Applied Computing*, p. 644–688, 2007.

[7] AMD. **Amd open physics initiative expands ecosystem with free dmm for game production and updated version of bullet physics**. http://www.amd.com/us/press-releases/Pages/amd-ecosystem-2010mar8.aspx, agosto 2010.

[8] AMORY, A. **Game object model version ii: a theoretical framework for educational game development**. *Educational Technology Research and Development*, 55(1):51–77, 2007.

[9] ANDERSON, E. F.; MCLOUGHLIN, L. **Critters in the classroom: a 3d computer-game-like tool for teaching programming to computer animation students**. In: *SIGGRAPH '07: ACM SIGGRAPH 2007 educators program*, p. 7, New York, NY, USA, 2007. ACM.

[10] BARCELOS, R. J.; TAROUCO, L. **O uso de mobile learning no ensino de algoritmos**. http://www.cinted.ufrgs.br/renote/dez2009/index.html, Dezembro 2009.

[11] BARNES, T.; POWELL, E.; CHAFFIN, A.; LIPFORD, H. **Game2learn: improving the motivation of cs1 students**. In: *GDCSE '08: Proceedings of the 3rd international conference on Game development in computer science education*, p. 1–5, New York, NY, USA, 2008. ACM.

[12] BARNES, T.; RICHTER, H.; POWELL, E.; CHAFFIN, A.; GODWIN, A. **Game2learn: building cs1 learning games for retention**. *SIGCSE Bull.*, 39:121–125, June 2007.

[13] BATTAIOLA, A. L. **Jogos por computador - histórico, relevância tecnológica e mercadológica, tendências e técnicas de implementação**. *Anais da XIX Jornada de Atualização em Informática, SBC*, 2:83–122, 2000.

[14] BECKER, B. W. **Teaching cs1 with karel the robot in java**. In: *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, SIGCSE '01, p. 50–54, New York, NY, USA, 2001. ACM.

[15] BEHESHTI, M.; GUNAWARDENA, A. D. **Teaching programming paradigms using a laboratory approach**. *J. Comput. Small Coll.*, 16:144–148, March 2001.

[16] BESSA, A.; SOUSA, C.; BEZERRA, C.; MONTEIRO, I.; BANDEIRA, H.; SOUZA, R. **O desenvolvimento de um motor multiplataforma para jogos 3d**, 2007.

[17] BICZÓ, M.; PÓCZA, K.; PORKOLÁB, Z. **Runtime access control in c# 3.0 using extension methods**. In: *Proceedings of the 10th Symposium on Programming Languages and Software Tools (SPLST 2007), Dobogóko (Hungary)*, p. 45–60. Citeseer, 2007.

[18] BITTENCOURT, J. R.; GIRAFFA, L. M. **Modelando ambientes de aprendizagem virtuais utilizando role-playing games**. *XIV Simpósio Brasileiro de Informática na Educação*, 2003.

[19] BROWN, P. H. **Some field experience with alice**. *J. Comput. Small Coll.*, 24:213–219, December 2008.

[20] BUSBY, S.; JEZIERKSI, E. **Microsoft .net/com migration and interoperability**. http://msdn.microsoft.com/en-us/library/ee817653.aspx#cominterop_topic5b, agosto 2001.

[21] CAMPBELL, R. H.; ISLAM, N.; RAILA, D.; MADANY, P. **Designing and implementing choices: an object-oriented system in c++**. *Commun. ACM*, 36:117–126, September 1993.

[22] CARNEGIE MELLON, U. **Alice**. http://www.alice.org/, April 2010.

[23] CHEN, K.-T.; LEI, C.-L. **Network game design: hints and implications of player interaction**. In: *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM.

[24] CLUA, E.; FEIJÓ, B.; ROCCA, J.; SCHWARTZ, J.; DAS GRAÇAS, M.; PERLIN, K.; TORI, R.; BARNES, T. **Game and interactivity in computer science education**. In: *SIGGRAPH '06: ACM SIGGRAPH 2006 Educators program*, p. 2, New York, NY, USA, 2006. ACM.

[25] CLUA, E. W. G.; BITTENCOURT, J. R. **Desenvolvimento de jogos 3d**. *Anais da XXIV Jornada de Atualização em Informática do Congresso da Sociedade Brasileira de Computação*, p. 1313–1356, Julho 2005.

[26] CONRAD, M.; FRENCH, T. **Using the synergies between the object-oriented paradigm and mathematics in joint mathematics/computer science programs**. *SIGCSE Bull.*, 36:254–254, June 2004.

[27] COUMANS, E. **Bullet physics engine**. www.bulletphysics.com/, 2011.

[28] CUNI, A.; ANCONA, D.; RIGO, A. **Faster than c#: efficient implementation of dynamic languages on .net**. In: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, p. 26–33, New York, NY, USA, 2009. ACM.

[29] DE AGUILERA, M.; MENDIZ, A. **Video games and education: (education in the face of a "parallel school")**. *Comput. Entertain.*, 1(1):1–10, 2003.

[30] DE ICAZA, M. **Mono's simd support: Making mono safe for gaming**. http://tirania.org/blog/archive/2008/Nov-03.html, nov 2008.

[31] DE ICAZA, M. **Scripting with mono**. http://www.mono-project.com/Scripting_With_Mono, abril 2011.

[32] DE KEREKI, I. F. **Scratch: Applications in computer science 1**. *38th ASEE/IEEE Frontiers in Education Conference*, p. 5, October 2008.

[33] FINCHER, S.; COOPER, S.; KÖLLING, M.; MALONEY, J. **Comparing alice, greenfoot & scratch**. In: *SIGCSE '10: Proceedings of the 41st ACM technical symposium on Computer science education*, p. 192–193, New York, NY, USA, 2010. ACM.

[34] FIRELIGHT. **fmod - interactive audio middleware**. http://www.fmod.org/, abril 2011.

[35] FORGE, N. **Nhibernate**. http://nhforge.org/, 06 2011.

[36] GACKENBACH, J.; ROSIE, M. **Cognitive evaluation of video games: players' perceptions**. In: *Proceedings of the 2009 Conference on Future Play on @ GDC Canada*, Future Play '09, p. 23–24, New York, NY, USA, 2009. ACM.

[37] GAITA, A. **Binding c++ apis, the com way**. http://blog.worldofcoding.com/2009/08/binding-c-apis.html, agosto 2009.

[38] GAMES, E. **Udk - unreal development kit**. http://www.udk.com/, abril 2011.

[39] GAMES, W. **Object oriented input system**. http://sourceforge.net/projects/wgois/, abril 2011.

[40] GEE, J. P. **High score education: Games, not school, are teaching kids to think.** On Newsstandes Now. http://www.wired.com/wired/archive/11.05/view.html, May 2003.

[41] GEE, J. P. **What video games have to teach us about learning and literacy**. *Comput. Entertain.*, 1(1):20–20, 2003.

[42] GEROSA, L. M.; CURY, D. **Uma abordagem construcionista no uso de jogos eletrônicos na educação**. *Anais da SBGames 2007*, 2007.

[43] GOMES, A.; MENDES, A. J. **An environment to improve programming education**. In: *Proceedings of the 2007 international conference on Computer systems and technologies*, CompSysTech '07, p. 88:1–88:6, New York, NY, USA, 2007. ACM.

[44] GREGORY, J. **Game engine architecture**. A K Peters, Wellesley, Mass., 2009.

[45] GROUP, K. **Open graphics layer - opengl**. http://www.opengl.org/, abril 2011.

[46] HARTNESS, K. **Robocode: using games to teach artificial intelligence**. *J. Comput. Small Coll.*, 19:287–291, April 2004.

[47] HAVOK. **Havok physics**. http://www.havok.com/index.php?page=havok-physics, abril 2011.

[48] HAYNES, C. T.; FRIEDMAN, D. P.; WAND, M. **Continuations and coroutines**. In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, p. 293–298, New York, NY, USA, 1984. ACM.

[49] HENRIKSEN, P.; KOLLING, M. **Greenfoot**. http://www.greenfoot.org, 2010.

[50] HICKS, A. **Towards social gaming methods for improving game-based computer science education**. In: *FDG '10: Proceedings of the Fifth International Conference on the Foundations of Digital Games*, p. 259–261, New York, NY, USA, 2010. ACM.

[51] JENKINS, T. **On the difficulty of learning to program**. In: *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, p. 53–58, 2002.

[52] JMONKEYENGINE. **jmonkeyengine**. http://jmonkeyengine.com/, 2011.

[53] KAFAI, Y. B. **The educational potential of electronic games: From games-to-teach to games-to-learn**, October 2001.

[54] KAFAI, Y. B. **Children designing software for children: what can we learn?** In: *Proceedings of the 2003 conference on Interaction design and children*, IDC '03, p. 11–12, New York, NY, USA, 2003. ACM.

[55] KATZ, S.; ALLBRITTON, D.; ARONIS, J.; WILSON, C.; SOFFA, M. L. **Gender, achievement, and persistence in an undergraduate computer science program**. *SIGMIS Database*, 37:42–57, November 2006.

[56] LIFE, S. **Mono for second-life**. http://wiki.secondlife.com/wiki/Mono, abril 2011.

[57] MALAN, D. J.; LEITNER, H. H. **Scratch for budding computer scientists**. In: *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, p. 223–227, New York, NY, USA, 2007. ACM.

[58] MALONE, T. W. **What makes things fun to learn? heuristics for designing instructional computer games**. In: *SIGSMALL '80: Proceedings of the 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems*, p. 162–169, New York, NY, USA, 1980. ACM.

[59] MALONEY, J. H.; PEPPLER, K.; KAFAI, Y.; RESNICK, M.; RUSK, N. **Programming by choice: urban youth learning programming with scratch**. In: *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08, p. 367–371, New York, NY, USA, 2008. ACM.

[60] MASCARENHAS, F.; IERUSALIMSCHY, R. **Efficient compilation of lua for the clr**. In: *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, p. 217–221, New York, NY, USA, 2008. ACM.

[61] MATTHEWS, J.; FINDLER, R. B. **Operational semantics for multi-language programs**. *ACM Trans. Program. Lang. Syst.*, 31:12:1–12:44, April 2009.

[62] MEIJER, E.; BECKMAN, B.; BIERMAN, G. **Linq: reconciling object, relations and xml in the .net framework**. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, p. 706–706, New York, NY, USA, 2006. ACM.

[63] MICROSOFT. **Directx**. http://www.gamesforwindows.com/en-US/directx/, abril 2011.

[64] MICROSOFT. **F# language for .net**. http://fsharp.net, abril 2011.

[65] MICROSOFT. **Ironpython**. http://www.ironpython.net/, maio 2011.

[66] MICROSOFT. **Microsoft xna 4**. http://en.wikipedia.org/wiki/Microsoft_XNA, 2011.

[67] MIT. **Scratch**. http://scratch.mit.edu/, March 2010.

[68] MONO. **Mono com interop**. http://www.mono-project.com/COM_Interop, abril 2011.

[69] MONO. **Mono interoperability**. http://www.mono-project.com/Interop_with_Native_Libraries, abril 2011.

[70] MONO. **Mono.csharp**. http://docs.go-mono.com/Mono.CSharp, abril 2011.

[71] MOSKAL, B.; LURIE, D.; COOPER, S. **Evaluating the effectiveness of a new instructional approach**. *SIGCSE Bull.*, 36:75–79, March 2004.

[72] MSDN, M. **Using c++ interop (implicit p/invoke)**. http://msdn.microsoft.com/pt-br/library/2x8kf7zx%28v=VS.100%29.aspx, abril 2011.

[73] MSDN, M. **Using explicit p/invoke in c++ (dllimport attribute)**. http://msdn.microsoft.com/pt-br/library/eyzhw3s8%28v=VS.100%29.aspx, abril 2011.

[74] NAKAMOTO, P.; CARRIJO, G.; CARDOSO, A. **Construção de ambientes educacionais com realidade aumentada: processo centrado no usuário**. *RENOTE*, 7(3):244–252, 2010.

[75] NOVELL. **Mono - open-source .net framework runtime**. http://www.mono-project.com, 2011.

[76] ODE. **Open dynamics engine - ode**. http://www.ode.org, 2010.

[77] OGRE. **Current ogre features: What ogre can do**. http://www.ogre3d.org/tikiwiki/Current+Ogre+Features, abril 2011.

[78] OGRE3D. **Ogre3d - 3d rendering engine**. http://www.ogre3d.org, lastaccessApril/2010, April 2010.

[79] PANDA3D. **Panda3d game engine**. http://www.panda3d.org, 2010.

[80] PASSOS, E.; DA SILVA JR, J.; RIBEIRO, F.; MOURÃO, P. **Tutorial: Desenvolvimento de jogos com unity 3d**. *VIII Brazilian Symposium on Games and Digital Entertainment*, 2009.

[81] PATTIS, R. E.; ROBERTS, J.; STEHLIK, M. **Karel the robot: a gentle introduction to the art of programming**. Wiley, New York, 2nd ed. edition, 1995.

[82] PEARCE, J.; NAKAZAWA, M. **The funnel that grew our cis major in the cs desert**. *SIGCSE Bull.*, 40:503–507, March 2008.

[83] PEARS, A.; SEIDMAN, S.; MALMI, L.; MANNILA, L.; ADAMS, E.; BENNEDSEN, J.; DEVLIN, M.; PATERSON, J. **A survey of literature on the teaching of introductory programming**. *SIGCSE Bull.*, 39:204–223, December 2007.

[84] PIMENTEL, E.; OMAR, N. **Ensino de algoritmos baseado na aprendizagem significativa utilizando o ambiente de avaliação netedu**. *SBC*, p. 79, 2008.

[85] PLUMMER, J. **A Flexible and Expandable Architecture for Electronic Games. Vol**. PhD thesis, Master Thesis. Arizona State University, Phoenix, 2004.

[86] REEVES, B.; READ, J. L. **Total engagement: using games and virtual worlds to change the way people work and businesses compete**. Harvard Business Press, Boston, Mass., 2009.

[87] RESNICK, M. **Rethinking learning in the digital age**. *The Global Information Technology Report 2001–2002*, p. 32–37, 2002.

[88] RESNICK, M.; MALONEY, J.; MONROY-HERNÁNDEZ, A.; RUSK, N.; EASTMOND, E.; BRENNAN, K.; MILLNER, A.; ROSENBAUM, E.; SILVER, J.; SILVERMAN, B.; KAFAI, Y. **Scratch: programming for all**. *Commun. ACM*, 52(11):60–67, 2009.

[89] RICHARDSON, C. **Orm in dynamic languages**. *Queue*, 6:28–37, May 2008.

[90] ROBERTO, R.; TEIXEIRA, J. M.; LIMA, J. P.; DA SILVA, M. M. O.; DE ALBUQUERQUE, E. S.; ALVES, D. F. M.; TEICHRIEB, V.; KELNER, J. **Jogos educacionais baseados em realidade aumentada e interfaces tangìveis**. In: *SBGames 2010 - Tutorials*, Florianópolis, Santa Catarina, nov 2010.

[91] Roberto, R. A.; Teixeira, J. M. X. N.; do Monte Lima, J. P. S.; da Silva, M. M. O.; Alves, D. F. M.; de Albuquerque, E. S.; Teichrieb, V.; Kelner, J. **Jogos educacionais baseados em realidade aumentada e interfaces tangíveis**. Tutorial na SVR 2011 - XIII Simpósio de Realidade Virtual e Aumentada, maio 2011.

[92] Schementi, J. **Embedding ironruby**. http://blog.jimmy.schementi.com/2009/12/ironruby-rubyconf-2009-part-35.html, dezembro 2009.

[93] Schürmann, T. **Exploring the boo scripting language**. Linux Magazine Issue #73, dezembro 2006.

[94] Siva, S.; Wang, L. **A sql database system for solving constraints**. In: *Proceeding of the 2nd PhD workshop on Information and knowledge management*, PIKM '08, p. 1–8, New York, NY, USA, 2008. ACM.

[95] Space, C. **Crystal space 3d game engine**. http://www.crystalspace3d.org/main/Main_Page, abril 2011.

[96] Stone, J.; Gohara, D.; Shi, G. **Opencl: A parallel programming standard for heterogeneous computing systems**. *Computing in Science and Engineering*, 12:66–73, 2010.

[97] Sucupira, I.; da Silva, F. **Programação por propagação de restrições: teoria e aplicações**. *São Paulo*, 2003.

[98] SWIG. **Swig developer documentation**. http://www.swig.org/Doc2.0/index.html, abril 2011.

[99] Unity3D. **Unity3d game engine**. http://unity3d.com/unity/, 2011.

[100] Utting, I.; Cooper, S.; Kölling, M.; Maloney, J.; Resnick, M. **Alice, greenfoot, and scratch – a discussion**. *Trans. Comput. Educ.*, 10:17:1–17:11, November 2010.

[101] Velasquez, C. **Modelo de engenharia de software para o desenvolvimento de jogos e simulações interactivas**. Master's thesis, Universidade Fernando Pessoa, 2009.

[102] Versant. **Db4o - database for objects**. http://www.db4o.com/, abril 2011.

[103] Ward, B.; Marghitu, D.; Bell, T.; Lambert, L. **Teaching computer science concepts in scratch and alice**. *J. Comput. Small Coll.*, 26:173–180, December 2010.

[104] WHITE, W.; SOWELL, B.; GEHRKE, J.; DEMERS, A. **Declarative processing for computer games**. In: *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, Sandbox '08, p. 23–30, New York, NY, USA, 2008. ACM.

[105] WHITEHEAD, J. **Introduction to game design in the large classroom**. In: *GDCSE '08: Proceedings of the 3rd international conference on Game development in computer science education*, p. 61–65, New York, NY, USA, 2008. ACM.

[106] ZHANG, K.; BETTINA, K.; DENAULT, A. **Persistence in massively multiplayer online games**. In: *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '08, p. 53–58, New York, NY, USA, 2008. ACM.

APÊNDICE A

Caso de estudo "Karel The Robot"

Esse capítulo apresenta informações sobre como construir o jogo "Karel The Robot" utilizando o *framework* InVision e o Tutano[1]. Dessa forma, alguns aspectos serão omitidos para manter o conteúdo mais leve e direto. Informações mais detalhadas podem ser encontradas junto ao código do *framework*[2].

A seção A.1 explica brevemente como a aplicação original funciona. A seção A.2 separa os papeis do instrutor e do aluno. A seção A.3 detalha as decisões de projeto e pontos de extensão sobre o *framework* para a construção da aplicação.

A.1 A definição do jogo

"Karel The Robot" é uma aplicação que utiliza uma abordagem dinâmica para o ensino de programação. O objetivo do jogo se resume em construir um algoritmo que faça com que o robô Karel pegue todos os sinalizadores, presentes no cenário, e os leve para um local designado.

O mundo de Karel é basicamente um labirinto. Numa visão superior do cenário, este se resume a um tabuleiro, com blocos de passagem livres (pisos sem obstáculos) e bloqueados por outros objetos (pisos ocupados pelos obstáculos). Um exemplo do cenário pode ser visto na figura A.1.

O personagem pode ser visto como um protótipo de um robô, que pode se movimentar somente passo-a-passo, executando movimentos simples, como:

- Dar um passo para frente;
- Girar a esquerda;
- Pegar um sinalizador;
- Soltar um sinalizar.

[1]Código do projeto: https://github.com/danfma/InVision
[2]Módulo Karel: https://github.com/danfma/InVision/tree/master/Karel

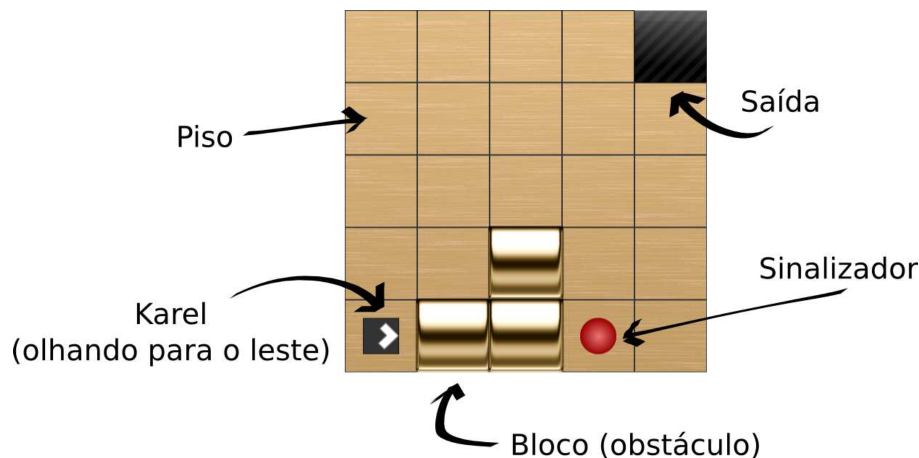

Figura A.1: *Rascunho do cenário para o mundo de Karel.*

Adicionalmente, Karel possui uma bússola e consegue informar se está apontando para um dos quatro polos: norte, sul, leste ou oeste. Dado essas ações básicas, o personagem não pode transpassar os obstáculos, somente contorná-los.

A.2 Papeis e responsabilidades

Dois papeis são desempenhados no contexto da aplicação:

O criador do cenário - o qual será desempenhado por um professor ou instrutor do curso. Seu objetivo é auxiliar no aprendizado de programação do estudante, construindo os cenários a serem resolvidos e impondo regras para a programação do Karel;

O programador do robô - o qual será desempenhado por um estudante. Seu objetivo é fazer com que o Karel alcance sua meta, dado um determinado cenário pré-definido e algumas premissas (regras) definidas pelo instrutor.

Ambos papeis irão trabalhar com *scripts* para interagir com o sistema, entretanto o contexto de cada *script* é diferenciado. Para o "criador", o *script* produzido terá como intuito a criação do cenário. Para o "programador", o *script* terá a função de estender e direcionar o robô Karel para que essa cumpra seu objetivo.

É importante ressaltar que num primeiro momento, onde o estudante está aprendendo os conceitos de programação, o cenário deve ser pré-amostrado. Assim, o estudante poderá focar em como resolver o problema, definindo ações para o robô, passo-a-passo; para então, pensar em como traduzir suas ideias em comandos de programação para o Karel. Posteriormente, tal abordagem pode evoluir para que os *scripts* sejam rotinas de inteligência artificial para solucionar qualquer cenário.

A.3 O projeto e arquitetura da aplicação

A aplicação é implementada como um módulo do Tutano (ver 5.4). Isso permite usar sua estrutura de carregamento dinâmico de *scripts* e pré-organização da estrutura da aplicação.

Módulos são apenas bibliotecas pré-compiladas (*assemblies* em .NET) e colocadas na pasta “Libraries” do Tutano. Tais bibliotecas são carregadas automaticamente pelo Tutano durante sua inicialização.

Os passos para a criação da aplicação estão descritos nas próximas seções.

A.3.1 Passo 1 - O *script* de criação

Para criar o cenário, o “criador” tem uma interface de programação simples, na qual poderá, através de comandos, realizar as seguintes funções:

- Criar o mundo de Karel, definindo-o como uma matriz de blocos;
- Adicionar blocos a blocos;
- Adicionar sinalizadores a blocos;
- Definir o *checkpoint* (ponto de saída do jogo);
- Posicionar o Karel em algum bloco, com a frente voltada para alguma das seguintes direções: norte, sul, leste ou oeste.

O código A.1 mostra, como exemplo, um *script* de criação.

Código A.1 Exemplo de um *script* de criação de um mundo de Karel, em Boo.

```
1  import System
2  import InVision
3  import InVision.GameMath
4  import Karel
5
6  world = World(4, 4)
7  world.AddBlockAt(3, 1)
8  world.AddBlockAt(2, 2)
9  world.AddBlockAt(3, 2)
10 world.AddBeeperAt(3, 3)
11 world.SetDepositAt(0, 3)
12 world.SetCheckpointAt(0, 3)
13 world.SetKarelAt(3, 0, KarelDirection.East)
14 world.Save()
```

O *script* cria uma instância de `World`, um objeto que define um mundo em Karel, posiciona os obstáculos, ponto de saída, posiciona o Karel e salva as alterações. Essa etapa é uma pré-configuração do mundo de Karel. A classe `World` é implementada como um `GameComponent`, isto é, um objeto que possui uma rotina de atualização a ser executada por *frame* do jogo. Sua responsabilidade é criar e posicionar os outros componentes e objetos visuais nas suas posições.

O *script* de criação é executado uma única vez durante a inicialização da aplicação.

Um ponto interessante a ser apresentado é sobre a capacidade de *scripting* do *framework*, onde os *scripts* são extensões e não somente chamadas de funções existentes. Assim, é possível criar, definir e até registrar novas classes e objetos para serem utilizados pelo *framework* e outros scripts durante a execução da aplicação.

A.3.2 Passo 2 - Interface de Programação para o Karel

Código A.2 `Programação para o Karel.`

```
1  import Karel.Controller
2
3  karel = Karel()
4  karel.TurnOn()
5  karel.Move()
6  karel.Move()
7  karel.TurnLeft()
8  karel.Move()
9  karel.Move()
10 karel.Move()
11 karel.TurnLeft()
12 karel.Move()
13 karel.Move()
14 karel.PickBeeper()
15 karel.TurnLeft()
16 karel.TurnLeft()
17 karel.Move()
18 karel.Move()
19 karel.Move()
20 karel.PutBeeper()
21 karel.TurnOff()
```

A interface de programação do Karel é tão intuitiva quanto a interface de criação. O código A.2 é um exemplo para solucionar o mundo definido no código A.1, o qual pode

ser visualizado na figura A.1.

Entretanto, essa simplicidade para o usuário tornou parte da implementação complexa e com uso pesado dos recursos de *microthreading* criados para a execução. O problema ocorrido se deve ao fato de termos dois fluxos de execução que devem ser sincronizados, mas executados em tempos diferentes, com velocidades diferentes.

O fluxo da aplicação principal, executa sobre um *loop* infinito que executa próximo aos 60 *frames* por segundo. Já o fluxo de programação deve executar uma ação do Karel a cada 1 segundo. Sendo assim, temos dois fluxos, o fluxo de renderização e animação e o fluxo de execução da programação. Existem várias formas de se implementar tal situação de paralelismo e sincronia, mas optou-se por utilizar o recurso de *microthreading* para testar o desempenho da aplicação.

Sendo assim, o que o programador define são ações executados sobre um controlador, que funciona como um controle remoto do Karel (parte do controlador pode ser visto em A.3).

Código A.3 Parte do código do controlador do Karel

```
1  public void Move()
2  {
3          KarelRobot.Move();
4          WaitPendingActions();
5  }
6
7  private void WaitPendingActions()
8  {
9          while (KarelRobot.HasPendingActions) {
10                 Thread.Sleep(100);
11         }
12 }
```

Cada ação do controle remoto irá criar várias ações pendentes no Karel, que então irá executá-las em fila, até que possa aceitar novas ações, como mostrado no código A.4.

Código A.4 Parte do código do Karel

```
public partial class KarelRobot
{
  private readonly ConcurrentQueue<UpdateAction> _pendingActions;

...
  public void Move()
  {
    _pendingActions.EnqueueAll(GetMoveActions());
  }

  private IEnumerable<UpdateAction> GetMoveActions()
  {
    const int steps = 60;

    Matrix matrix = Matrix.CreateFromQuaternion(
      SceneNode.Orientation);

    Vector3 forward = matrix.Forward / steps;

    for (int i = 0; i < steps; i++) {
      yield return DelayedWork(delegate {
            SceneNode.Position += forward;
          });

      yield return WaitBy(30);
    }
  }

  public override IEnumerable<UpdateAction> UpdateBySteps()
  {
    while (!_pendingActions.IsEmpty) {
      UpdateAction action;
      _pendingActions.TryDequeue(out action);
      yield return action;
    }
  }
...
}
```

A.3.3 Passo 3 - Definindo o fluxo de execução

Como um último passo para a construção é necessário a criação do fluxo de execução customizado. Para se estender as funcionalidades do Tutano, criando um novo fluxo de execução, é necessário estender a classe `TutanoGameFlow`. Assim, o novo fluxo de execução irá:

- Carregar o *script* de criação para o mundo de Karel;
- Carregar a programação do Karel, contido em outro *script*;
- Simular a execução do Karel passo-a-passo, em paralelo a execução do jogo e animações;
- Ao final do *script* de programação, verificar se o *checkpoint* foi alcançado.

A.4 Configuração e execução do Tutano

Para executar o Karel, basta executar o Tutano normalmente, com algumas alterações no seu arquivo de configuração. Os *scripts* de criação do mundo virtual e contendo a solução devem estar dentro da pasta *Scripts* do Tutano. O código A.5 mostra o que deve ser adicionado ao final do arquivo de configuração do Tutano para que esse execute a aplicação.

Código A.5 Configuração do Tutano para o Karel

```
1 <!-- KAREL -->
2 <game-flow-type>Karel.Flow.KarelGameFlow,Karel</game-flow-type>
3
4 <custom-configuration>
5         <config name="Karel.Problem" value="Scripts/Worlds/World1" />
6         <config name="Karel.Resolution" value="Scripts/Resolved/World1" />
7 </custom-configuration>
```

A linha 5 define qual o *script* que contém o mundo a ser solucionado. A linha 6 define a programação que irá resolver o problema.