Desenvolvimento de um Jogo *Multitouch* para iPhone: um estudo de caso

**Eduardo Coelho, George Ruberti Piva,
Paulo César Rodacki Gomes, Dalton Solano dos Reis**

[1]Departamento de Sistemas e Computação, FURB – Universidade Regional de Blumenau, Campus I, 89012-900, Blumenau, SC – Brasil

{eduardocoelho,piva,rodacki,dalton}@inf.furb.br

Abstract. *The recent release of Apple's iPhone SDK opened new possibilities for mobile game development. iPhone has some unique features among existing mobile platforms such as it's programming language, development environment, device's hardware resources, operating system and even the business model for application distribution. The present paper describes the development of a multitouch, multiplayer AirHockey game for iPhone. We begin presenting a short description of the development platform, followed by a discussion about the application's architecture and it's class organization. We also present the collision resolution approach adopted along with the behavior model of the computer-controlled character, then we present conclusions and discuss future work.*

Resumo. *O lançamento do framework de desenvolvimento para iPhone da Apple abriu novas possibilidades na área de jogos para dispositivos móveis. O iPhone possui algumas características únicas entre as plaformas de desenvolvimento atualmente existentes, tais como a linguagem de programação adotada, o ambiente de desenvolvimento, os recursos de hardware do dispositivo, seu sistema operacional, e até o modelo de negócios para a comercialização de aplicações. Este artigo descreve o desenvolvimento de um jogo multi-toque e multi-player para iPhone, no estilo AirHockey. É feita uma breve descrição da plataforma de desenvolvimento, em seguida é apresentada a arquitetura geral da aplicação desenvolvida. Logo após, é feita uma discussão sobre o tratamento de colisão e o modelo de comportamento do jogador controlado pelo computador. Por fim, são apresentadas conclusões e as expectativas de trabalhos futuros.*

1. Introdução

Com a popularização do iPhone e a recente disponibilização da SDK pela Apple, houve um aumento no desenvolvimento de aplicações para esta plataforma. O iPhone possui características diferenciadas em relação aos outros dispositivos eletrônicos tais como tela *multitouch* de 480x320 pixels, acelerômetros, uso de OpenGL ES e OpenAL, entre outros. Estas características favorecem o desenvolvimento de aplicações inovadoras, especialmente jogos.

Visando explorar melhor esta plataforma, este artigo apresenta um estudo de caso referente ao desenvolvimento de um jogo do tipo AirHockey para iPhone. O AirHockey é relativamente simples, tratando-se de um jogo do tipo *pong* onde os jogadores têm forma

de discos planos chamados de *mallets* e o objetivo é marcar gols no campo adversário utilizando os *mallets* para golpear a bola representada por um disco denominado *puck*, conforme ilustrado na figura 1. O projeto do jogo procura explorar ao máximo certos recursos do dispositivo uma vez que permite partidas entre dois jogadores simultaneamente no mesmo dispositivo utilizando *multitouch* ou em dispositivos diferentes utilizando dos seus recursos de rede. Um modo *single player* também foi implementado, e portanto foi agregada Inteligência Artificial ao *mallet* controlado pelo computador.

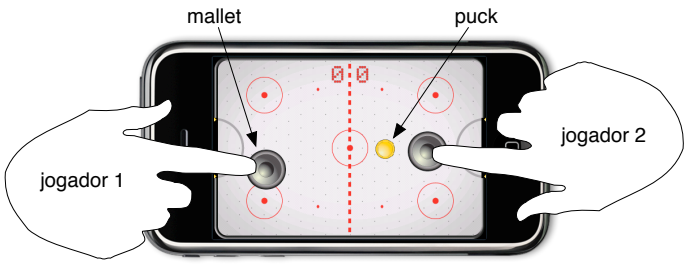

Figura 1. AirHockey com dois jogadores.

Na próxima seção é feita uma introdução ao iPhone como dispositivo e como plataforma de desenvolvimento de aplicações. a seção 3 apresenta de forma resumida a especificação do jogo. Na seção será dada uma ênfase no *gameloop* implementado. Na terceira seção serão apresentados os métodos utilizados para tratar as rotinas de colisão e simulação dos objetos e a inteligência artificial do jogo. Por fim, a última seção apresenta a modelagem do jogo desenvolvido, seguido do protocolo de comunicação implementado.

2. Tecnologias

O iPhone 3G é um *smartphone* com processador ARM de 620Mhz com 128Mb de memória RAM, 8 a 16Gb de memória *flash*. Além disso, possui tela *multitouch*, sensor de proximidade e de luminosidade, além de um conjunto de acelerômetros. O seu sistema operacional, chamado iPhone OS, foi concebido a partir do Apple Mac OS X, sistema operacional dos computadores da Apple Computer, que por sua vez foi construído tendo o Unix como base [Allen and Appelcline 2008]. Em termos de desenvolvimento, a Apple disponibilizou em junho de 2008 o iPhone SDK, a plataforma para desenvolvimento para iPhone e iPod Touch, composta por um simulador de iPhone, bibliotecas de classes e uma IDE com compilador para linguagem Objective-C [Dalrymple and Knaster 2008]. As bibliotecas disponíveis no iPhone SDK permitem acesso a inúmeras funcionalidades úteis para o desenvolvimento de jogos, tais como: suporte a eventos *multitouch*, renderização 2D e 3D, acesso aos acelerômetros, serviços de rede, manipulação e persistência de dados, abstrações para a manipulação de *views* e *windows* entre outras.

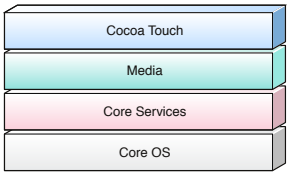

Figura 2. Camadas do iPhone OS

As camadas mais baixas do iPhone OS (fig. 2) consistem dos serviços fundamentais do sistema, utilizados por todas as aplicações, enquanto que camadas mais altas possuem serviços e tecnologias mais sofisticadas – tratam-se de *frameworks* orientados a objetos que são abstrações das camadas mais inferiores, mas que não necessariamente restringem o acesso a elas [Apple Computer 2008]. Uma das camadas mais importantes é a Cocoa Touch, composta pelos *frameworks* UIKit e Foundation que provêm as ferramentas e a infra-estrutura básica necessárias para desenvolver aplicações gráficas e dirigidas a eventos. A camada Media provê os frameworks necessários para a criação de aplicações que exploram o máximo da capacidade do dispositivo com relação aos recursos de multimídia. Os *frameworks* de alto nível permitem a criação rápida de animações com recursos gráficos e animações avançadas, enquanto *frameworks* de baixo nível provêm o acesso as ferramentas necessárias para desenvolver aplicativos altamente customizáveis, a exemplo do OpenGL ES [Munshi et al. 2008]. A camada Core Services fornece os serviços fundamentais de sistema e, portanto, é utilizada por todas as aplicações direta ou indiretamente. Esta camada contém os *frameworks* Core Foundation – responsável o gerenciamento básico dos dados e serviços do iPhone e CFNetwork – que contém abstrações para protocolos de rede, entre outros. Por fim, a camada mais inferior do iPhone OS, a Core OS abrange o *kernel*, *drivers* e interfaces básicas do sistema operacional. É a camada responsável pelo gerenciamento de memória, *threads*, sistema de arquivos, rede e comunicação entre processos. Apenas um conjunto restrito de *frameworks* tem acesso ao *kernel* e *drivers* desta camada.

3. Especificação

O jogo Air-Hockey foi especificado seguindo a orientação a objetos, visto que no iPhone todas as aplicações são escritas em Objective-C. Sua arquitetura é constituída por diferentes pacotes, os quais são: (i) o pacote *Model* que contém as entidades do domínio do problema, (ii) o pacote *Engine* que contém as classes e protocolos do motor de jogos, além do sub-pacote denominado *Networking* que agrupa as classes referentes a comunicação em rede e (iii) o pacote *Game* que contém a implementação do jogo, através de classes que promovem a interação entre objetos do domínio, utilizando protocolos definidos no pacote *Engine*. O propósito desta arquitetura é permitir um bom reuso de código e fácil prototipação visando o desenvolvimento rápido de novos jogos sem grandes alterações estruturais.

3.1. Pacote Engine

O pacote *Engine* é o motor do jogo propriamente dito, seu principal objetivo é tratar da funcionalidade geral do jogo 2D fornecendo classes que para instanciar o jogo, receber e tratar eventos *multitouch* do(s) usuário(s), gerenciar diferentes janelas de visualização, entre outros. Embora o SDK do iPhone forneça subsídios para trabalhar com os acelerômetros do dispositivo, este motor de jogos não adotou tal funcionalidade. Por outro lado, o motor provê subsídios para desenvolver jogos 2D que utilizem comunicação via rede. A seguir é apresentado o diagrama de classes referente a este pacote.

Na figura 3 a classe **NXController** é a classe principal, responsável por controlar a exibição do jogo na tela, fazer o mapeamento dos eventos de toques das camadas de visualização para o jogo e gerenciar as transições entre as classes descendentes da classe NXView controlando a memória por elas alocadas. Além disso, a classe **NXController**

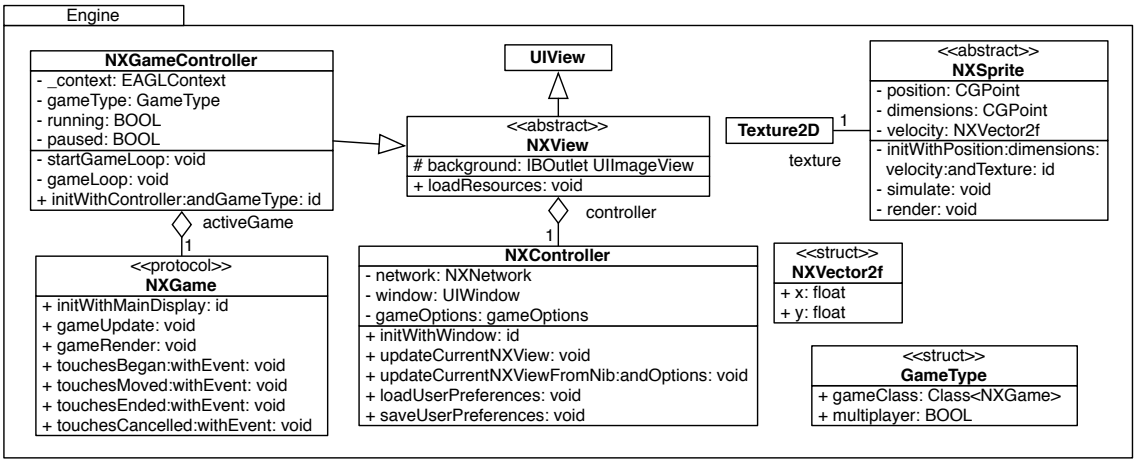

Figura 3. Pacote Engine

possui operações para carregar e apresentar telas provenientes de um de descrição de interface. No *framework* Cocoa, tais arquivos são chamados de arquivos NIB. Apenas um único objeto desta classe é instanciado, e seu ciclo de vida se estende por todo o ciclo de vida da própria aplicação. Esta classe também contém uma referência para um objeto da classe NXNetwork, fornecendo às demais classes a infra-estrutura de comunicação em rede para o modo de jogo multiusuário. Por fim, a classe também fornece mecanismos de persistência de dados, permitindo o armazenamento de informações da aplicação que podem ser utilizados em uma futura execução.

A classe abstrata **NXView** herda todas as funcionalidades da classe UIView do *framework* UIKit do iPhone SDK, adicionando como atributo uma referência ao objeto da classe NXController. Assim, todas as suas classes descendentes têm acesso a NXController podendo utilizar a rede bem como informar ao mesmo sobre transições entre NXViews, que ocorrem, por exemplo, quando se muda da tela do jogo para a tela de configurações. **NXGame** é um protocolo[1] que possui as operações necessárias a serem implementadas por um jogo 2D. Tais como as operações que são invocadas durante o *gameloop* – entrada de eventos, simulação e redesenho – e as operações referentes ao controle de memória. O protocolo ainda garante que as classes que o adotam recebam uma referência a uma instância de NXGameController, dando a classe o poder de suspender ou finalizar o *gameloop*. A classe **NXGameController** é uma especialização de NXView para um *layer* do tipo CAEAGLLayer. Este *layer* cria um contexto OpenGL ES. Seu objetivo principal é o de gerenciar o *gameloop*, através de atributos de controle e estado. O jogo a ser gerenciado deve ser uma classe que adota o protocolo NXGame e é armazenado pelo atributo *activeGame*. Como sua modelagem é genérica, seu construtor requer uma estrutura do tipo GameType que define a classe do jogo a ser criado e define a possibilidade de ele usar a rede, cabendo a ela criar uma nova instância deste jogo para então inicializar o *gameloop*. Por fim, a classe abstrata **NXSprite** apresenta as mínimas características comuns a um objeto gráfico de um jogo 2D. As operações de simulação e renderização de tais objetos estão definidas como métodos abstratos, devendo ser implementados caso necessário.

[1]Em Objective-C, um protocolo define as assinaturas de métodos que devem ser implementados pelas classes que adotam o protocolo.

3.2. Pacote Model

O objetivo deste pacote é armazenar as classes que representam as entidades do domínio do jogo. Para um jogo do tipo Air-Hockey foram delimitadas as entidades apresentadas na figura 4. A classe **Disc** é uma classe base que estende a classe NXSprite não sobrescrevendo nenhuma de seus métodos. Uma vez que as entidades interativas do jogo são discos, foi acrescentado apenas um atributo para o seu raio.

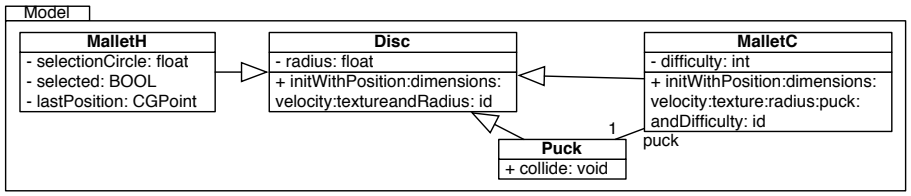

Figura 4. Pacote Model

As classes **Puck**, **MalletH** e **MalletC** representam respectivamente a "bola" do jogo, o disco controlado pelo jogador humano e o disco controlado pelo computador. As três classes estendem a classe Disc, com as seguintes especializações: (i) o puck implementa um método para testar sua colisão com as demais entidades do jogo; (ii) o malletH adiciona alguns atributos de controle para identificar quando o usuário está pressionando o mallet em questão, assim o círculo de seleção é ou não apresentado dependendo do usuário e (iii) o malletC adiciona alguns atributos que são utilizados para definir seu comportamento. Para tal foi implementado o módulo de inteligência artificial dentro da operação de simulação. Seus atributos consistem em uma referência para o puck – leitura de sua posição para tomada de decisão – e o nível de dificuldade que determina o comportamento do *malletC*.

3.3. Pacote Game

Este pacote tem como objetivo delimitar as classes referentes ao jogo de AirHockey propriamente dito. A classe NXAirHockey adota o protocolo NXGame e é a classe base para as demais classes do jogo. Diversas especializações desta classe foram criadas para representar os difrentes modos de jogo conforme ilustrado no diagrama de classe da figura 5.

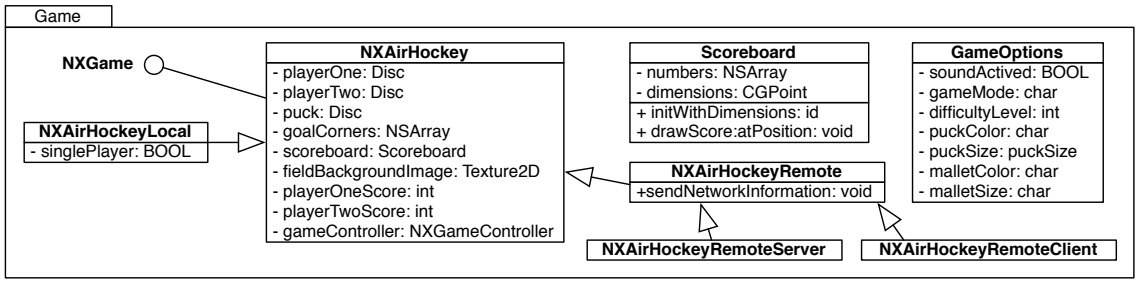

Figura 5. Pacote Game

No diagrama, a classe **NXAirHockey** agrega todas as entidades descritas no modelo, faz a interação entre os objetos e controla o desenvolvimento do jogo. Por fim, ela implementa as regras do jogo e faz o tratamento dos eventos reportados pela classe NXGameController. A arquitetura do jogo foi concebida para diferenciar de forma clara

e desacoplar a implementação dos modos com ou sem rede. Por exemplo, a classe **NXAirHockeyLocal** é uma especialização da classe NXAirHockey para um jogo que acontece localmente. Este jogo pode ser tanto para um jogador quando para dois jogadores. No primeiro caso, o oponente será uma instância de malletC, controlado através de Inteligência Artificial. No segundo caso, ambos os oponentes são intâncias de malletH, e devem ser controlados pelos jogadores. A classe **NXAirHockeyRemote** é uma especialização da classe NXAirHockey para jogos em rede com dois dispositivos. Esta classe permite que as suas sub-classes enviem e recebam *streams* de dados para gerenciar a comunicação entre os dispositivos.

O modo de jogo em rede conta com as classes **NXAirHockeyRemoteServer** e **NXAirHockeyRemoteClient** que estendem NXAirHockeyRemote. A classe servidora é responsável pela simulação das entidades do jogo, invocando a operação *simulate* de cada objeto. Esta classe recebe as coordenadas do mallet cliente, trata os eventos locais, simula o ambiente localmente e então envia as coordenadas de cada objeto do mundo para o cliente (exceto as coordenadas do mallet cliente). O controle de fim de jogo bem como o de pontuação, é feito localmente em cada dispositivo, cabendo somente ao servidor realizar a simulação do jogo. A classe cliente é responsável pelo tratamento dos eventos locais e o envio das coordenadas de seu mallet para que o servidor faça a simulação e retorne a nova posição dos demais objetos do jogo. A instância desta classe é responsável pelo controle de final de jogo e pontuação localmente em seu dispositivo. Por fim, a classe **Scoreboard** imprime os pontos de cada jogador no formato de um placar eletrônico no campo.

4. Simulação e Inteligência Artificial

A seguir, apresentamos alguns aspectos relevantes do desenvolvimento do jogo, incluindo comentários sobre a simulação física do *puck*, que compreende o tratamento de colisão e sobre a implementação do comportamento do jogador controlado pelo computador.

4.1. Tratamento de colisão

A parte mais importante da jogabilidade do AirHockey é a simulação física do movimento do *puck*, que é feita com base no Princípio da Conservação da Quantidade de Movimento. A quantidade de movimento de um corpo j é dada por $Q_j = m_j \times v_j$, onde m_j é a massa do corpo e v_j é sua velocidade. Na colisão ideal entre dois corpos rígidos a soma da quantidade de movimento dos dois corpos antes da colisão é igual à quantidade de movimento total após a colisão, ou seja: $m_p v_{pi} + m_m v_{mi} = m_p v_{pf} + m_m v_{mf}$. Nesta equação, os índices p e m referem-se ao *puck* e ao *mallet* respectivamente, e os índices i e f referem-se aos instantes inicial (antes da colisão) e final (após a colisão).

Para melhorar o realismo da colisão, tratamos o *puck* e o *mallet* como objetos elásticos. Assim, a velocidade dos objetos após a colisão é afetada por um coeficiente de restituição elástica E, onde $0 \leq E \leq 1$ [Stahler 2005]. Para uma colisão elástica ideal, $E = 1$, e para uma colisao inelástica ideal, $E = 0$. Assim, temos a seguinte relação de velocidades inicial e final dos dois objetos colidindo, considerando que suas massas são iguais:

$$(v_{pf} - v_{mf}) = -E\,(v_{pi} - v_{mi}) \tag{1}$$

Como o *mallet* é controlado pelo jogador, consideramos sua velocidade final igual à velocidade inicial ($v_{mf} = v_{mi}$), portanto, chegamos à seguinte expressão para a velocidade final do *puck*:

$$v_{pf} = (1 + E)\, v_{mi} - E v_{pi} \tag{2}$$

Na simulação do comportamento físico do *puck*, dependendo da velocidade v_{mi} imposta pelo jogador, é possível que a velocidade calculada do *puck* resulte em um valor muito alto. Assim, o usuário poderia marcar um gol contra ou a favor sem mesmo ver para onde o *puck* foi. Para resolver tal problema, após o cálculo de v_{pf} na equação 2, é imposto ao *puck* um limite de velocidade máximo aceitável, inibindo este tipo de comportamento.

4.2. Simulação do Puck

Em temos de implementação, durante o processo de modelagem do jogo, decidiu-se que as classes das entidades que o constituem e que possuem representação gráfica devem herdar a classe NXSprite e podem sobrescrever suas operações de simulação e renderização. Desta forma o objeto da classe Puck em nosso jogo AirHockey é responsável por atualizar-se. O algoritmo 1 apresenta o pseudo-código da classe Puck que sobrescreve a operação de simulação da classe NXSprite. Quando um gol acontece durante a simulação, o puck é posicionado fora do campo e conseqüentemente novas simulações não serão realizadas. Após o gol, quando o jogo estiver pronto para continuar, o puck volta a ser posicionado no campo e passar a ser simulado normalmente.

```
1  se puck dentro do campo então
       // atualiza posição do puck com vel da simulação
2      pos_{x,y} ← pos_{x,y} + v_{x,y}Δt;
3      se puck colidir com borda lateral da mesa então
           // mantém o puck na borda e inverte vel em x
4          v_x ← (−v_x) × E;
5      se gol então
6          posiciona o puck fora do campo;
7          v_x ← v_y ← 0;
8      senão
9          se puck colidir com borda inferior ou superior da mesa então
               // mantém o puck na borda e inverte vel em y
10             v_y ← (−v_y) × E;
```

Algoritmo 1: Simulação do puck

4.3. Inteligência Artificial - simulação do malletC

O módulo de inteligência artificial do jogo modela o comportamento do *malletC*, o jogador controlado pelo computador. Todo o comportamento foi modelado empiricamente e é feito com base em tomadas de decisão regidas por questões geométricas acerca da posição do *malletC* e do *puck*. A operação[2] *simulate* da classe MalletC, implementa este

[2]Em Objective-C, um método comumente é chamado de operação.

comportamento. A principal responsabilidade desta classe é calcular o vetor de deslocamento $\overrightarrow{d} = (d_x, d_y)$ do *malletC*. O comportamento do *malletC* está dividido em dois modos: defesa e ataque, que são definidos levando-se em consideração a posição do *puck* em relação a uma área de abrangência do *malletC*. Se o *puck* estiver abaixo da base da área de abrangência, aciona-se o modo de defesa, caso contrário, aciona-se o ataque. A figura 6 apresenta os dois casos (ataque e defesa), a área de abrangência é representada pelo retângulo em torno do *mallet* no campo superior, e configura uma folga lateral para que o *puck* não fique trancado pelo *malletC*. Esta folga no eixo x tem o tamanho igual à soma do raio do *malletC* com o diâmetro do *puck*.

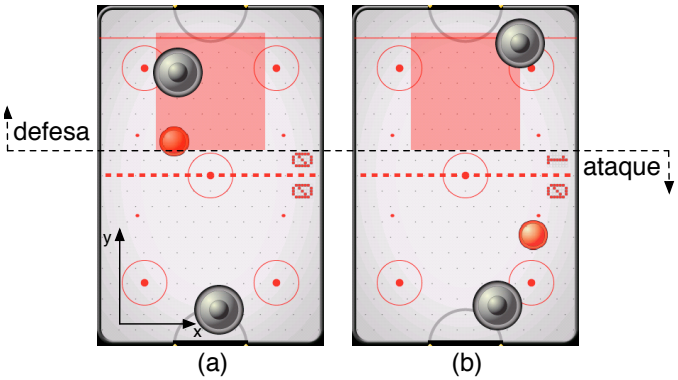

Figura 6. (a) malletC no modo defesa e (b) no modo ataque

A área de abrangência do *malletC* é determinada dinamicamente, uma vez que a medida de seu raio pode ser escolhida pelo jogador ao configurar o nível de dificuldade do jogo. O modo de defesa consiste em fazer o *malletC* acompanhar a posição do *puck* em relação ao eixo x, respeitando uma tolerância de τ_x pixels que varia de acordo com o nível de dificuldade (o menor nível de dificuldade possui uma tolerância maior). Se a diferença das abcissas de ambos os objetos for maior que a τ_x então o *malletC* move-se em direção à mesma coordenada x do *puck* , senão a componente x de seu vetor de deslocamento permanece nula ($d_x = 0$). Com relação à componente y do vetor de deslocamento, consiste em manter o *malletC* sempre na linha de defesa, representada por uma coordenada y próxima ao gol do *malletC*. Semelhantemente com relação ao eixo x, é tomada a decisão tendo como base uma tolerância de τ_y pixels que varia de acordo com o nível de dificuldade. Um menor nível de dificuldade apresenta uma tolerância maior, ou seja, o *malletC* fica mais tempo fora da linha de defesa, deixando o seu gol mais desprotegido. Se a diferença das ordenadas de ambos os objetos for maior que τ_y então o *mallet* move-se para a linha de defesa, senão a componente y de seu vetor de deslocamento permanece nula ($d_y = 0$).

O modo de ataque consiste em impulsionar o *malletC* para atingir o *puck* visando lançá-lo para o campo adversário. O cálculo de d_x recebe o mesmo tratamento do modo defesa, enquanto que o cálculo de d_y sofre algumas alterações. Se o *puck* encontra-se acima do *mallet*, então verifica-se o ângulo entre o *mallet* e o *puck* em relação ao eixo y para evitar que o *malletC* atinja o *puck* em direção ao próprio gol. O *mallet* só atingirá o puck se este encontrar-se em um ângulo acima da tolerância mínima, que é de $45°$, caso contrário deve se mover na direção oposta, desviando-se do *puck*. Se o *puck* estiver abaixo do *malletC*, é atingido visando jogá-lo para o campo do adversário. Se o ângulo θ_{mp} entre

os dois objetos for menor que $30°$, o *puck* é atingido com mais força aumentando-se o fator de escala de velocidade γ_v. O fator de escala de velocidade γ_v cresce linearmente à medida que θ_{mp} se aproxima de $0°$. A figura 7 ilustra esquematicamente tais ângulos de tolerância.

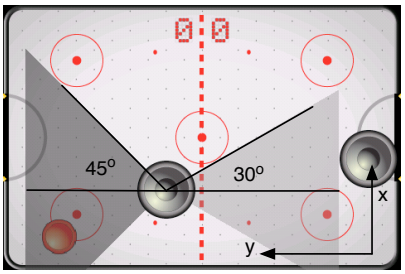

Figura 7. Limites de ângulos para cálculo do vetor de deslocamento do malletC

O cálculo do ângulo θ_{mp} entre os dois objetos é realizado pela da equação 3:

$$\theta_{mp} = \arctan\left(\frac{x_p - x_m}{|y_p - y_m|}\right) \tag{3}$$

Após definido o vetor de deslocamento $\overrightarrow{d}$, basta atribuir o fator de escala inicial da velocidade γ_v do *malletC*, que varia de acordo com o nível de dificuldade do jogo, configurada pelo jogador (nível mais fácil possui valores menores). O pseudo-código do método *simulate* é apresentado pelo algoritmo 2. Na linha 11, $\overrightarrow{v}^{i}_{m}$ é o vetor de velocidade do *malletC* na iteração atual (i), calculado em função do vetor de direção normalizado do *malletC*, do fator de escala de velocidade e da velocidade na iteração anterior $(i-1)$.

```
1  Calcule a nova posição do malletC;
2  se malletC estiver fora de seus limites então
3      ajuste posição do malletC para dentro dos limites;
4  γ_v ← kFatorVelInicialNivelDificuldade;
5  se modo == defesa então
6      Calcule d⃗ para proteger gol e acompanhar o puck;
7  senão
8      Calcule d⃗ para atacar o puck em direção ao gol adversário;
9      se (θ_mp ≤ 30°) e (y_puck < y_malletC) então recalcule γ_v;
10     v⃗_m^(i) ← d⃗/|d⃗| × γ_v × v⃗_m^(i−1);
```

Algoritmo 2: Simulação do malletC

5. Considerações Finais

O presente artigo tratou do desenvolvimento de um jogo de AirHockey *multitouch* e *multiplayer* para iPhone. Foram apresentados aspectos gerais da arquitetura da aplicação, bem como discussões a respeito do tratamento de colisão dos objetos e implementação do comportamento do jogador controlado pelo computador. Foram escritas aproximadamente 5.000 linhas de código fonte em linguagem Objective-C, dividido em cerca de 22

classes da *engine* e 10 classes complementares para implementação de tela de abertura, menus de configuração de opções de jogo, etc. O arquivo executável final ficou com tamanho de 804 Kb. Sua execução ocupa cerca de 23 Mb de memória RAM, o que representa uma pequena parcela dos 128 Mb de memória RAM disponíveis no iPhone 3G. Isso demonstra que a plataforma tem potencial para o desenvolvimento de jogos mais complexos, visto que o iPhone OS não permite que duas aplicações rodem simultaneamente. Tal fato garante que a maior parte dos recursos do harware estejam sempre disponíveis para a plicação que esteja executando. Em testes com o simulador, a aplicação rodou em 55 fps (*frames* por segundo), enquanto que no dispositivo, esta taxa foi de 60 fps. Em relação à simulação da física no tratamento de colisão e à inteligência artificial implementados, os resultados foram bastante satisfatórios, causando um alto grau de realismo para o jogador humano.

O jogo procura utilizar recursos do iPhone que favorecem o desenvolvimento de jogos, tais como uso do OpenGL ES como *framework* para renderização, interface *multitouch*, conexão em rede via *Wi-fi* entre outros. A interface *multitouch* é particularmente interessante, pois permite que duas pessoas se enfrentem jogando no mesmo dispositivo, conforme ilustrado na figura 1. Para a equipe envolvida, o desenvolvimento deste jogo foi recompensador do ponto de vista de aquisição de conhecimento sobre a tecnologia do iPhone. Foi dada ênfase nas questões de uso das bibliotecas fornecidas pela Apple, exploração da usabilidade do dispositivo e uso do OpenGL. Até o presente momento a aplicação não está disponível para *download*, pois nas próximas etapas do projeto será feita investigação sobre os recursos de áudio, criação de trilha sonora e efeitos de áudio e melhorias no *design* dos elementos gráficos para tornar o jogo mais próximo do que se esperaria de uma aplicação comercial.

6. Agradecimentos

Os autores agradecem à Universidade Regional de Blumenau e ao Projeto ACREDITO do curso de Bacharelado em Ciências da Computação da FURB pelo apoio financeiro para aquisição de equipamentos para realização do trabalho.

Referências

Allen, C. and Appelcline, S. (2008). *iPhone in Action: Introduction to Web and SDK Development*. Manning Publications Co., Greenwich, CT, USA.

Apple Computer (2008). *iPhone OS Programming Guide*. Apple Computer Inc, Disponível em: http://developer.apple.com.

Dalrymple, M. and Knaster, S. (2008). *Learn ObjectiveC on the Mac*. Apress, Berkely, CA, USA.

Munshi, A., Ginsburg, D., and Shreiner, D. (2008). *OpenGL ES 2.0 Programming Guide*. Addison-Wesley Professional.

Stahler, W. (2005). *Fundamentals of Math and Physics for Game Programmers (Game Design and Development Series)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.