UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

SIMULADOR DE UMA PARTIDA DE FUTEBOL COM ROBÔS VIRTUAIS

FÁBIO SCHULTER

BLUMENAU
2007

2007/1-12

FÁBIO SCHULTER

SIMULADOR DE UMA PARTIDA DE FUTEBOL COM ROBÔS VIRTUAIS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. José Roque Voltolini da Silva - Orientador

BLUMENAU
2007

2007/1-12

SIMULADOR DE UMA PARTIDA DE FUTEBOL COM ROBÔS VIRTUAIS

Por

FÁBIO SCHULTER

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. José Roque Voltolini da Silva – Orientador, FURB
Membro:	Prof. Jomi Fred Hübner, Doutor – FURB
Membro:	Prof. Dalton Solano dos Reis, Mestre – FURB

Blumenau, 10 de julho de 2007

Dedico este trabalho a todos os amigos, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, que sempre me incentivou na conclusão deste curso. Especialmente a minha esposa Jeni pelo apoio. Aos meus pais Getúlia e Adelmo pela educação e dedicação que me criaram juntamente com os meus dois irmãos.

Aos amigos e colegas do curso, pois sempre que precisei eles estavam presentes.

Ao meu orientador, José Roque Voltolini da Silva, por ter acreditado em mim, na minha proposta e na conclusão deste trabalho.

Eduquem as crianças de hoje e não será preciso punir os homens no amanhã.

Pitágoras

RESUMO

Este trabalho descreve a implementação de software para simular de uma partida de futebol com robôs virtuais. Este software possui módulos para criar as equipes, jogadores, estratégias, táticas e executar a simulação. No módulo da simulação da partida, primeiramente, é definido quais são as equipes que irão jogar, juntamente com a estratégia (lista de táticas) que cada uma vai utilizar. Os jogadores serão posicionados em campo conforme a tática definida para o momento da partida. Nas disputas das jogadas e tomadas de decisões serão levadas em conta os atributos dos jogadores e aleatoriedade. A partida é visualizada em 2D. Para prover a visualização da partida foi usada a biblioteca OpenGL.

Palavras-chave: Simulador. Futebol. Robôs virtuais.

ABSTRACT

This work describes the implementation of software to simulate a soccer match with virtual robots. This software have modules to create the teams, players, strategies, tactics and to execute the simulation. In the simulation module, it is defined which are the teams that will play, together with the strategy (list of tactics) that each one goes to use. The players will be located in agreement field as the tactics defined for the moment of the match. In the disputes of the plays and taking of decisions the attributes of the players will be taken in account. The match is visualized in 2D mode. To provide the visualization of match the OpenGL library was used.

Keywords: Simulator. Soccer. Virtual robots.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

FIFA – *Fédéraration Internationale de Football Association*

OpenGL - *Open Graphics Library*

a.C. – Antes de Cristo

2D – Duas dimensões

3D – Três dimensões

LARF - Agentes Robôs Jogadores de Futebol

GUI - *Graphical User Interface*

API - *Application Programming Interface*

UML - *Unified Modeling Language*

IA – Inteligência artificial

SUMÁRIO

1 INTRODUÇÃO

Uma simulação em computação consiste, de maneira geral, em utilizar técnicas matemáticas em computadores com a finalidade de imitar processos ou operações do mundo real. Para que exista uma simulação é necessário construir um modelo computacional que corresponda a situação do mundo real (SIMULAÇÃO, 2006). Estes modelos permitem a exploração de situações fictícias, de situações com risco, de experimentos que são muito complicados, caros ou que levam muito tempo para serem executados.

Na área de jogos, a simulação é amplamente utilizada. Na internet, por exemplo, existem vários simuladores de vários gêneros, como o simulador de corrida de carros F1 Race Brasil (LOGOX, 2006) ou de *hockey* no gelo (MANAGERZONE, 2007).

Mas especificamente verifica-se que existem simuladores para conduzir, controlar e visualizar uma partida de futebol, o esporte coletivo mais praticado no mundo. O futebol é disputado por duas equipes, de 11 jogadores que têm como objetivo colocar a bola entre as traves adversárias mais vezes que o time adversário sem usar mãos e braços. O ato de colocar a bola entre as traves adversárias é chamado gol. A meta ou baliza é um retângulo formado por duas traves ou postes verticais, perpendiculares ao solo, e uma trave ou travessão paralela ao solo. Ali fica posicionado o goleiro, que é o único jogador com permissão para colocar as mãos na bola, mas somente dentro da sua área. O futebol tem dezessete (17) regras que devem ser obedecidas (FUTEBOL, 2007).

Simuladores de uma partida de futebol podem ser vistos em Hattrick (2007), Managerzone (2007) e Gamegol (2007). Nestes simuladores os usuários não interferem durante a partida, cabe a eles somente a tarefa de escolher e posicionar no campo os jogadores disponíveis em seu time, conforme julguem a estratégia mais adequada e aos atributos que cada jogador possui, como por exemplo, velocidade, resistência, inteligência, passe, chute, desarme, cabeceio, domínio de bola e defesa para goleiro.

Num simulador, o jogador pode ser visto como um robô autônomo. A primeira vez que foi mencionada a idéia de robôs jogando futebol foi no artigo "*On Seeings Robots*" (MACKWORTH, 1993 apud ALEGRETTI, 2004, p. 13). Neste artigo Mackworth descreve o uso de futebol de robôs para testar um sistema de visão robótica e desenvolver algorítimos de controle de movimentos e planejamento de rotas.

Existem organizações internacionais que promovem a competição de futebol de robôs e uma das principais é a RoboCup, que tem como objetivo desenvolver robôs humanóides

totalmente autônomos até o ano 2050, capaz de ganhar do time humano de futebol campeão do mundo. Mas, por enquanto, ela só promove campeonatos de robôs de menor porte, com várias categorias diferentes (BOTELHO, 2006).

Visto o acima descrito, este trabalho implementa um simulador de futebol em computador baseado nos simuladores Hattrick (2007), Managerzone (2007) e Gamegol (2007), onde os robôs serão definidos em forma de software (virtual). Cada robô será modelado através de uma *thread* (unidade concorrente), visto que o mesmo em determinados momentos pode executar atividades independentes dos demais jogadores, citando como exemplo, um novo posicionamento em campo. Os robôs serão autônomos e não terão nenhuma interferência externa direta, obedecendo apenas a estratégia determinada pelo usuário do time (técnico), as quais poderão ser alteradas durante a simulação. Os atributos, como por exemplo velocidade e controle de bola, serão determinados para cada robô (jogador), em uma escala de 0 (zero) a 10 (dez), os quais determinarão as características do mesmo. Uma visualização do jogo em andamento também será disponibilizada pelo software, com informações complementares, como número de gols marcados pelas equipes e tempo de simulação.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um simulador de uma partida de futebol de campo, sem a interferência direta nos jogadores (robôs).

Os objetivos específicos do trabalho são:

a) determinar o posicionamento dos jogadores segundo estratégias definidas;
b) todas as ações dos jogadores durante a partida serão influenciadas pelos seus atributos;
c) permitir modificar táticas durante a partida de futebol;
d) visualizar o andamento do jogo no computador.

1.2 ESTRUTURA DO TRABALHO

O trabalho está dividido em quatro (4) capítulos.

O primeiro capítulo apresenta a introdução e os objetivos pretendidos com a elaboração do trabalho.

O segundo capítulo apresenta temas relacionados com o simulador desenvolvido.

O terceiro capítulo apresenta o desenvolvimento do trabalho, incluindo a descrição da especificação, implementação e a operacionalidade do simulador.

O quarto capítulo finaliza o trabalho, com as conclusões, limitações e sugestões para extensões que poderão ser realizadas objetivando melhorar o simulador.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capitulo são apresentados os assuntos relacionados com o trabalho, os quais são: jogo de futebol, simulação, simulação de futebol, teoria dos jogos, processos concorrentes, computação gráfica e a biblioteca OpenGL. Na última seção são descritos os trabalhos correlatos.

2.1 JOGO DE FUTEBOL

Muitos países se dizem os inventores do futebol. Os primeiros registros surgiram na China entre os anos 3.000 e 2.500 a.C. Em 1700, o futebol foi ganhando aspectos mais modernos e em 1710, na Inglaterra, as escolas de *Covent Garden*, *Strand* e *Fleet Street* passaram a adotar o futebol como atividade física. Com a difusão do esporte, o jogo passou a ter diferentes tipos de regras em cada escola. As duas regras diferentes que mais ganharam destaque na época foram: uma jogada só com os pés e uma com os pés e as mãos. Assim surgia o *football* e o *rugby*, em 1846 (HISTÓRIA DO FUTEBOL, 2007).

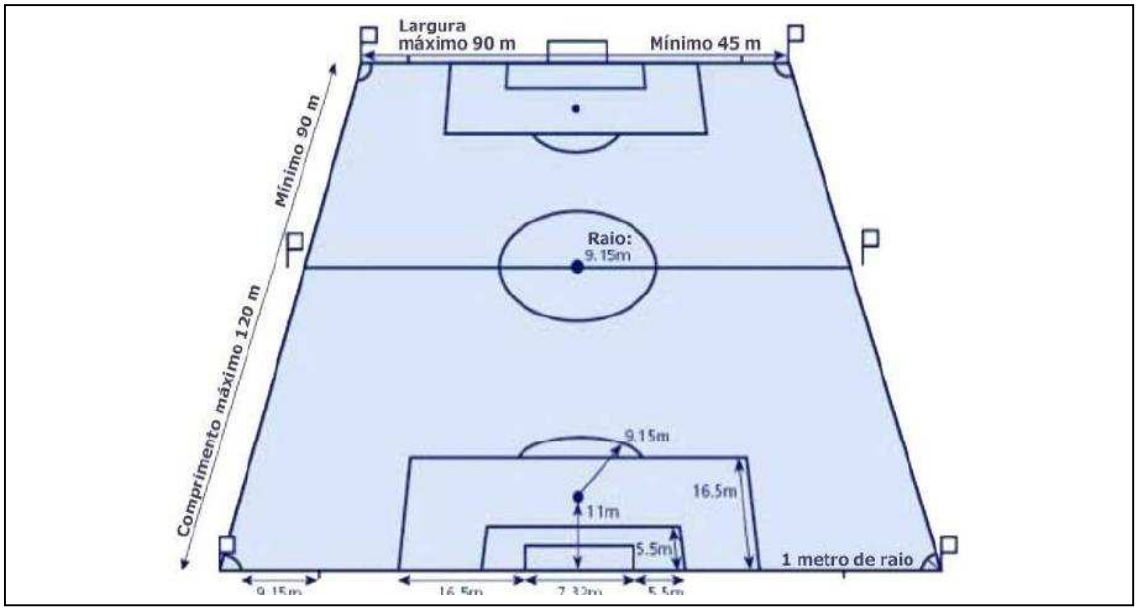
Atualmente o futebol é o esporte coletivo mais praticado e prestigiado no mundo. É disputado num campo retangular entre duas equipes com onze (11) jogadores cada. O principal objetivo do futebol é colocar a bola dentro da baliza da equipe adversária o maior número de vezes, isso sem que nenhum jogador (exceto o goleiro) utilize as mãos e os braços (FUTEBOL, 2007).

2.1.1 Especificações do campo de futebol

O campo de futebol possui as seguintes dimensões (Figura 1) (FIFA, 2006, p. 8):

a) comprimento de 90 a 120 metros;
b) largura de 45 a 90 metros;
c) o raio do círculo central é de 9,15 metros;

d) as áreas de meta (pequena área) possuem 18,32 metros de largura por 5,5 metros de comprimento;
e) as áreas penal (grande área) possuem 40,32 metros de largura por 16,5 metros comprimento;
f) em cada área penal é marcado um ponto penal a 11 metros de distância da linha de meta (linha de fundo);
g) no exterior de cada área penal será traçado um semicírculo com um raio de 9,15 metros do ponto penal;
h) as metas possuem 7,32 metros de largura por 2,44 metros de altura.



Fonte: FIFA (2006, p. 12).
Figura 1 - Dimensões de um campo de futebol

2.1.2 Regras do jogo de futebol

No futebol existem dezessete (17) regras que são definidas pelo *International Football Association Board*, órgão da FIFA. Estas regras definem os seguintes itens:

a) campo de jogo;
b) bola;
c) equipes;
d) uniformes;

e) árbitro;
f) auxiliares;
g) tempo de partida;
h) início e reinício de partida;
i) bola fora de jogo;
j) gol;
k) impedimento ou fora-de-jogo;
l) falta;
m) tiro livre;
n) pênalti;
o) arremesso lateral;
p) tiro de meta;
q) escanteio ou córner.

2.1.3 Funções exercidas pelos jogadores de futebol e o posicionamento em campo

As regras do futebol não determinam especificamente outras posições além do goleiro, porém, com o desenvolvimento do jogo, um certo número de posições especializadas foi criada. As posições principais no futebol são (FUTEBOL, 2007):

a) o goleiro tem como única função proteger o gol. É o único jogador que pode usar as mãos, e mesmo assim só pode usá-las dentro da área. Sua função é impedir que a bola passe pelas traves;
b) os zagueiros possuem a função de ajudar o goleiro a proteger o gol, mas não podem usar as mãos. Eles ficam posicionados no campo de defesa;
c) os laterais ficam posicionados nas laterais do campo. Têm a função de ajudar o goleiro a proteger o gol e normalmente são eles que repõem a bola em jogo quando ela sai pelas linhas laterais do campo;
d) os meias, médios e meio campista têm basicamente a função de fazer a conexão entre a defesa e o ataque do time, atuando tanto na marcação como nas jogadas ofensivas;
e) os atacantes têm a função fundamental de fazer o gol.

2.2 SIMULAÇÃO

Historicamente a simulação, como técnica, originou-se dos estudos de Von Neumann e Ulan. Estes estudos ficaram conhecidos como análise ou técnica de Monte Carlo. A simulação começou a ser mais utilizada como técnica para solução de problemas, principalmente para o tratamento dos problemas eminentemente probabilísticos, cuja solução analítica é, geralmente, muito mais árdua e difícil, senão impossível (CORNÉLIO FILHO, 1998).

É importante ressaltar que existem vários conceitos e definições de simulação, dependendo da abordagem de cada autor (CORNÉLIO FILHO, 1998). Segundo Naylor (1971 apud CORNÉLIO FILHO, 1998), simulação é uma técnica para realizar experiências em um computador digital. Martinelli (1987 apud CORNÉLIO FILHO, 1998) afirma que "a simulação é um meio de se experimentar idéias e conceitos sob condições que estariam além das possibilidades de se testar na prática, devido ao custo, demora ou risco envolvidos". Ainda, de acordo com Schriber (1974 apud CORNÉLIO FILHO, 1998) "simulação implica na modelagem de um processo ou sistema de tal forma que o modelo imite as respostas do sistema real numa sucessão de eventos que ocorrem ao longo do tempo".

2.3 SIMULAÇÃO DE FUTEBOL

Em simuladores de futebol, como descritos em Hattrick (2007), Manazerzone (2007) e Gamegol (2007), o usuário comanda o seu time da seguinte forma: ele possui um time com um plantel de jogadores. Tendo isto, o usuário pode definir os onze (11) jogadores que participarão da partida, o posicionamento de cada jogador conforme os seus atributos e a forma que o time de uma maneira geral vai se comportar durante a simulação, como por exemplo se vai jogar de maneira defensiva, ofensiva, jogadas pelas laterais ou contra ataque. Esta configuração para a partida é denominada de tática.

O usuário monta as táticas conforme ele julgar o mais adequado. Um jogador com bom chute deve ser colocado de forma mais avançada como atacante. No caso de um jogador com mais desarme deve ser colocado como zagueiro. Estas táticas ficam persistidas e o usuário agenda as suas partidas contra times de outros usuários, selecionando a tática que será utilizada em cada jogo (MANAGERZONE, 2007).

2.4 TEORIA DOS JOGOS

A teoria de jogos baseia-se principalmente nos trabalhos desenvolvidos por Von Neumann. Ela trata com situações de tomada de decisão em que dois ou mais oponentes possuem objetivos conflitantes (NOGUEIRA, 2006).

Em um jogo, dois oponentes podem ter o número finito ou infinito de alternativas ou estratégias. Associado com cada par de estratégias há um valor de pagamento (*payoff*) que um jogador é obrigado a pagar ao seu oponente. Estes jogos são conhecidos como jogos de soma zero e dois jogadores, porque o ganho de um é igual a perda de outro (NOGUEIRA, 2006).

Segundo Azevedo et al. (2002, p. 3), uma das abordagens para analisar o jogo é feita através de análises das estratégias que levam aos seus possíveis equilíbrios. Os equilíbrios podem ser basicamente de dois tipos: o equilíbrio de estratégias dominantes e o equilíbrio de Nash. Rasmusen (1989 apud AZEVEDO et al., 2002, p. 3) define uma estratégia dominante como sendo a resposta mais adequada para qualquer estratégia que possa ter sido escolhida pelos oponentes, que resulta o maior *payoff*. Quando estratégias dos jogadores permanecem inalteradas, quer dizer que um equilíbrio de estratégias foi montado. Tavares (1995 apud AZEVEDO et al., 2002, p. 3) define o equilíbrio de Nash como sendo a melhor estratégia de cada jogador, ou seja, a melhor ação em resposta às estratégias definidas pelos oponentes. “Uma vez atingido o equilíbrio de Nash, nenhum jogador tem incentivo para desviar-se dele, dados que os outros jogadores também não desviam” Tavares (1995 apud AZEVEDO et al., 2002, p. 3).

2.5 ESTRATÉGIA E TÁTICA

Estratégia consiste em ações de longo e médio prazo necessárias para se atingir a visão (ESTRATÉGIA, 2007). Segundo Ferreira (1999, p. 841) estratégia pode ser definida como “Arte de aplicar meios disponíveis com vista a consecução de objetivos disponíveis”.

Tática consiste em ações de curto prazo conduzidas sobre os meios para que se atinjam as metas desejadas (TÁTICA, 2007). Segundo Ferreira (1999, p. 1930) tática pode ser definida como “Meios postos em prática para sair-se bem em qualquer coisa”.

2.6 PROCESSOS CONCORRENTES

Os métodos de controle concorrentes aumentam em muito a flexibilidade de programação. Originalmente eles foram desenvolvidos para serem utilizados em problemas enfrentados em sistemas operacionais, mas podem ser usados para diversas aplicações de programação. “Por exemplo, muitos sistemas são projetados para simular sistemas físicos reais, cuja boa parte consiste em múltiplos subsistemas concorrentes. Sendo assim, a execução concorrente pode ocorrer fisicamente em processadores separados ou usando alguma forma de tempo fatiado” (SEBESTA 2000, p. 468).

Uma técnica para visualizar o fluxo de execução de um programa denomina-se *thread*, onde cada instrução é atingida em uma execução particular e independente. “Uma *thread* de controle é a seqüência de pontos do programa atingidos a medida que o controle flui por ele” (SEBESTA, 2000, p. 470).

Ainda que a execução de programas concorrentes tenha somente uma única *thread* de controle, eles sempre terão que ser analisados e projetados como se houvessem múltiplas *threads* de controle. Quando um programa com várias *threads* é executado em um único processador ele se torna, neste caso, um programa virtualmente *multithreaded* (SEBESTA, 2000, p. 470).

Se as *threads* compartilham dados ou que necessitam de uma sincronização nas suas tarefas, é necessário que elas se comuniquem umas com as outras (SEBESTA, 2000, p. 471). Existem basicamente três (3) mecanismos de comunicação entre os processos concorrentes, os quais são:

a) semáforos: é um mecanismo muito simples, para promover sincronização entre os processos. O conceito de semáforo baseia-se na colocação de uma proteção em torno de um código que acessa uma determinada estrutura, limitando a sua manipulação. As duas únicas operações oferecidas por semáforos foram originalmente esperar (esperar a liberação do recurso ou estrutura) e liberar (liberar o recurso ou estrutura para que outro processo possa acessar) (SEBESTA, 2000, p. 475);
b) monitores: é um método para encapsular as estruturas de dados compartilhados, transformando as estruturas de dados compartilhados em tipos de dados abstratos, e todas as operações de sincronização em dados compartilhados são reunidas em uma única unidade de controle (SEBESTA, 2000, p. 479);

c) passagem de mensagens: foi desenvolvida para manipular o problema de o que fazer quando muitos pedidos simultâneos são feitos por muitas tarefas, para comunicar-se com uma em especial. O não-determinismo é necessário neste caso, para assegurar justiça na escolha de qual pedido seria atendido primeiro. Essa justiça pode ser definida de várias maneiras conforme o contexto da aplicação, mas em geral significa que todos os solicitantes recebem uma oportunidade igual de se comunicarem com a tarefa (SEBESTA, 2000, p. 484).

2.7 COMPUTAÇÃO GRÁFICA E A BIBLIOTECLA OPENGL

Segundo Gomes e Velho (2003, p. 2), a computação gráfica é, de uma forma genérica, o conjunto de técnicas e métodos que tratam da manipulação de dados ou imagens no computador. A computação gráfica abrange as seguintes áreas: modelagem, visualização, processamento de imagens, visão computacional e animação. Existem bibliotecas para auxiliar na construção de softwares que usam computação gráfica, entre as quais cita-se a OpenGL.

OpenGL é uma biblioteca que permite a criação de aplicações que utilizam recursos de computação gráfica, como rotinas gráficas de modelagem, manipulações de objetos e exibição tridimensional. Utilizando ela, o usuário pode criar objetos gráficos rapidamente, além da possibilidade de incluir recursos avançados de animação, manipulação de imagens, texturas e ter a visualização de vários ângulos de um mesmo objeto (WANGENEHIM, 2006).

A Silicon Graphics criou a biblioteca OpenGL em 1992, com o objetivo principal de criar uma *Application Programming Interface* (API) gráfica independente do sistema operacional. A OpenGL facilitou na abstração das rotinas de exibição e de processamento de imagens implementadas em dispositivos (hardware) e sistemas operacionais específicos, abstraindo as peculiaridades de cada um. Uma função da OpenGL desenvolvida para o sistema operacional Windows tem o mesmo nome, parâmetros e produz o mesmo resultado na OpenGL desenvolvida para o ambiente Linux, por exemplo (WANGENEHIM, 2006).

“Diante das funcionalidades providas pelo OpenGL, tal biblioteca tem se tornado um padrão amplamente utilizado na indústria de desenvolvimento de aplicações” (WANGENEHIM, 2006). Diversos jogos, aplicações científicas, comerciais e grande parte dos fabricantes de placas de vídeo destinadas aos consumidores domésticos utilizam a

OpenGL (WANGENEHIM, 2006).

2.8 TRABALHOS CORRELATOS

Nas subseções seguintes são apresentados os simuladores Managerzone, Hattrick, e GameGol. A Robocup, a linguagem LARF e arquitetura Robopet também são apresentadas. Uma comparação entre os simuladores estudados é mostrada.

2.8.1 Managerzone

Em Managerzone (2007) é descrito um gigantesco *game multiplayer online* que permite milhares de jogadores interagirem uns com os outros. O usuário pode possuir um time de futebol e de *hockey* no gelo, sendo que em qualquer um deles ele pode vender e comprar jogadores, tanto brasileiros como estrangeiros, e melhorar os atributos através de treinamentos. O usuário pode desafiar outros times (amistosos) ou participar de ligas e copas. Cada time tem uma lista de jogos com as competições que o usuário escolheu participar, e ele deve selecionar a tática a ser utilizada em cada jogo até duas (2) horas antes do início da partida. Cada time pode ter três táticas definidas (a, b e c), onde o usuário define as posições de cada jogador, estilo de jogo (jogadas pelas laterais, passes longos e passes curtos) e agressividade do time (agressivo, normal ou passivo) e as possíveis substituições, que podem ser pré-definidas conforme o cenário da partida. Na maioria das competições o usuário deve pagar para que o seu time possa participar, e em algumas são dados prêmios aos times campeões. O usuário é responsável também pela administração do clube, pois ele pode vir a falência. Para isso ele deve se ater as receitas e despesas do seu clube.

De Figura 2 a Figura 6 é demonstrado o funcionamento da parte tática do jogo.

Fonte: Managerzone (2007).

Figura 2 - Atributos dos jogadores no Managerzone

Fonte: Managerzone (2007).

Figura 3 – Tela em que o usuário monta as táticas A, B e C no Managerzone

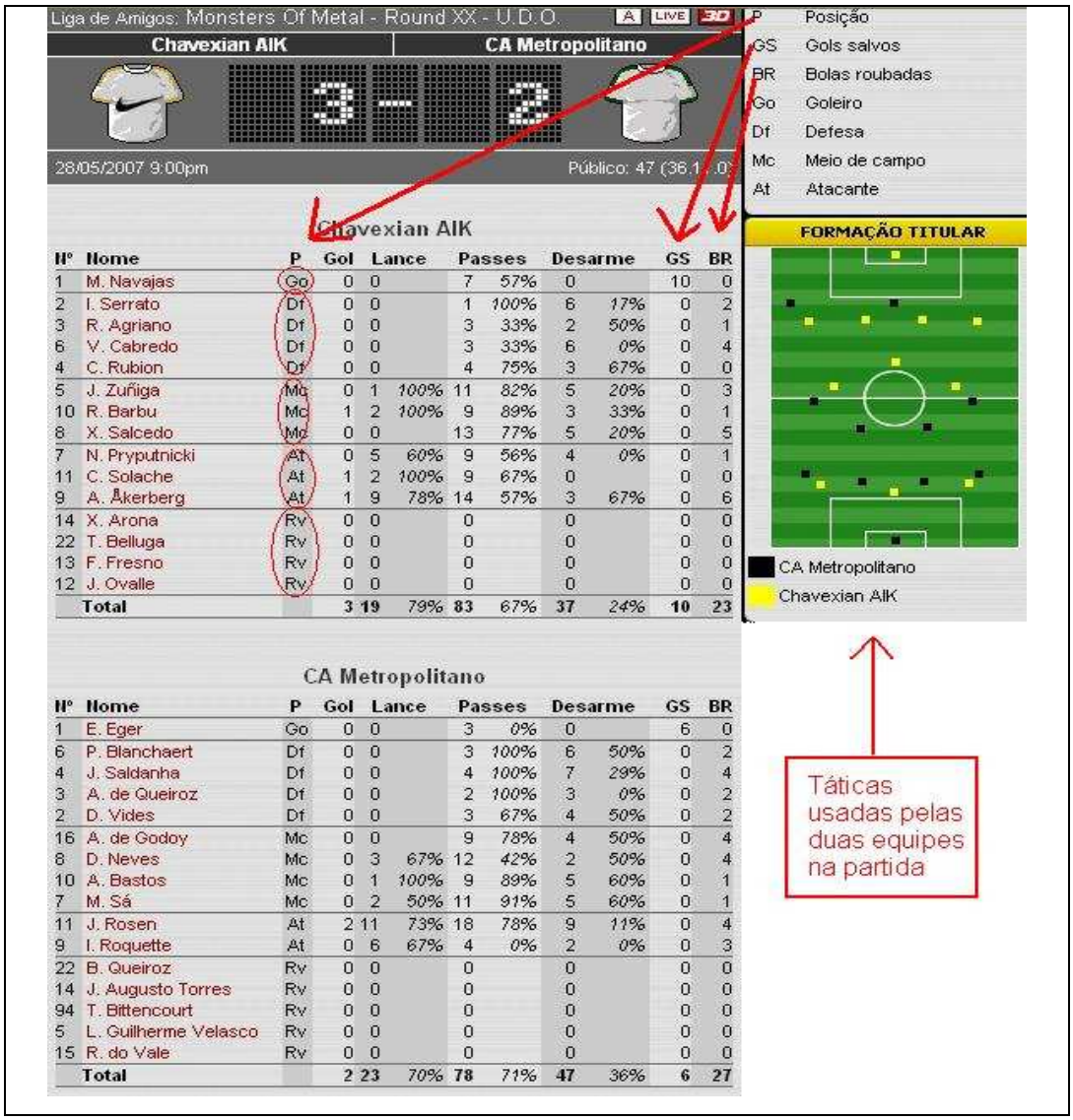
Fonte: Managerzone (2007).

Figura 4 – Tela para a definição da tática a ser utilizada e lista dos jogos do usuário no Managerzone

Fonte: Managerzone (2007).

Figura 5 – Tela de visualização da partida em 2D no ManagerZone

Após a partida, o simulador Managerzone (2007) fornece as estatísticas do jogo (Figura 6). É mostrada a posição que cada jogador atuou, percentual de aproveitamento nos lances, percentual de aproveitamento nos passes, percentual de sucesso nos desarmes, gols salvos (somente goleiro), bolas roubadas e a tática usada pelas duas equipes.



Chavexian AIK

Nº	Nome	P	Gol		Lance	Passes		Desarme		GS	BR
1	M. Navajas	Go	0	0		7	57%	0		10	0
2	I. Serrato	Df	0	0		1	100%	6	17%	0	2
3	R. Agriano	Df	0	0		3	33%	2	50%	0	1
6	V. Cabredo	Df	0	0		3	33%	6	0%	0	4
4	C. Rubion	Df	0	0		4	75%	3	67%	0	0
5	J. Zuñiga	Mc	0	1	100%	11	82%	5	20%	0	3
10	R. Barbu	Mc	1	2	100%	9	89%	3	33%	0	1
8	X. Salcedo	Mc	0	0		13	77%	5	20%	0	5
7	N. Pryputnicki	At	0	5	60%	9	56%	4	0%	0	1
11	C. Solache	At	1	2	100%	9	67%	0		0	0
9	A. Åkerberg	At	1	9	78%	14	57%	3	67%	0	6
14	X. Arona	Rv	0	0		0		0		0	0
22	T. Belluga	Rv	0	0		0		0		0	0
13	F. Fresno	Rv	0	0		0		0		0	0
12	J. Ovalle	Rv	0	0		0		0		0	0
	Total		3	19	79%	83	67%	37	24%	10	23

CA Metropolitano

Nº	Nome	P	Gol		Lance	Passes		Desarme		GS	BR
1	E. Eger	Go	0	0		3	0%	0		6	0
6	P. Blanchaert	Df	0	0		3	100%	6	50%	0	2
4	J. Saldanha	Df	0	0		4	100%	7	29%	0	4
3	A. de Queiroz	Df	0	0		2	100%	3	0%	0	2
2	D. Vides	Df	0	0		3	67%	4	50%	0	2
16	A. de Godoy	Mc	0	0		9	78%	4	50%	0	4
8	D. Neves	Mc	0	3	67%	12	42%	2	50%	0	4
10	A. Bastos	Mc	0	1	100%	9	89%	5	60%	0	1
7	M. Sá	Mc	0	2	50%	11	91%	5	60%	0	1
11	J. Rosen	At	2	11	73%	18	78%	9	11%	0	4
9	I. Roquette	At	0	6	67%	4	0%	2	0%	0	3
22	B. Queiroz	Rv	0	0		0		0		0	0
14	J. Augusto Torres	Rv	0	0		0		0		0	0
94	T. Bittencourt	Rv	0	0		0		0		0	0
5	L. Guilherme Velasco	Rv	0	0		0		0		0	0
15	R. do Vale	Rv	0	0		0		0		0	0
	Total		2	23	70%	78	71%	47	36%	6	27

Fonte: Managerzone (2007).

Figura 6 – Estatísticas do jogo no Managerzone

2.8.2 Hattrick

Em Hattrick (2007) é descrito um jogo de futebol *online* onde o usuário compra, vende e treina jogadores enquanto compete contra centenas de milhares de oponentes em todo o mundo, cerca de novecentos mil (900.000). Tem muitas características em comum com o Managerzone, como transferências, treinamentos de jogadores e escolha de táticas. É totalmente gratuito e não possui premiação. Na Figura 7 pode-se visualizar uma das diferenças destes dois simuladores, que é os atributos dos jogadores. Além dos atributos serem diferentes, os mesmos são mensurados de forma diferente.

Fonte: Hattrick (2007).

Figura 7 – Atributos dos jogadores no simulador Hattrick

2.8.3 Gamegol

Em Gamegol (2007) é descrito um jogo de futebol *online*, onde o usuário também é o dirigente e técnico de um time. Ele possui muitas características em comum com o Managerzone e Hattrick, como transferências, treinamentos de jogadores e escolha de táticas. Assim como o Hattrick, o Gamegol é gratuito. Na Figura 8 pode-se ver a semelhança na maneira de montar á tática nos simuladores Managerzone e Gamegol.

Fonte: Gamegol (2007).

Figura 8 - Tela em que o usuário monta as táticas no Gamegol

2.8.4 Robocup

A Robocup é uma organização internacional de pesquisa e educação através do futebol de robôs. Ela determina regras de padrões a serem adotados nas competições, de modo a fornecer um problema igual a todos os participantes, para que depois seja avaliado o melhor resultado alcançado. Além de regulamentar, a Robocup promove campeonatos mundiais, conferências técnicas, propõe desafios e cria programas na área de educação. Dentro da Robocup há divisões de categorias, cada uma com um objetivo principal diferente e com regras próprias. Atualmente são elas: Simulation League, Small Robot League (F-180), Full Set F-180, Middle Size League (F-2000), Sony Legged Robot League e Humanoid League (Figura 9) (BOTELHO, 2006).

Fonte : RoboCup (2007)

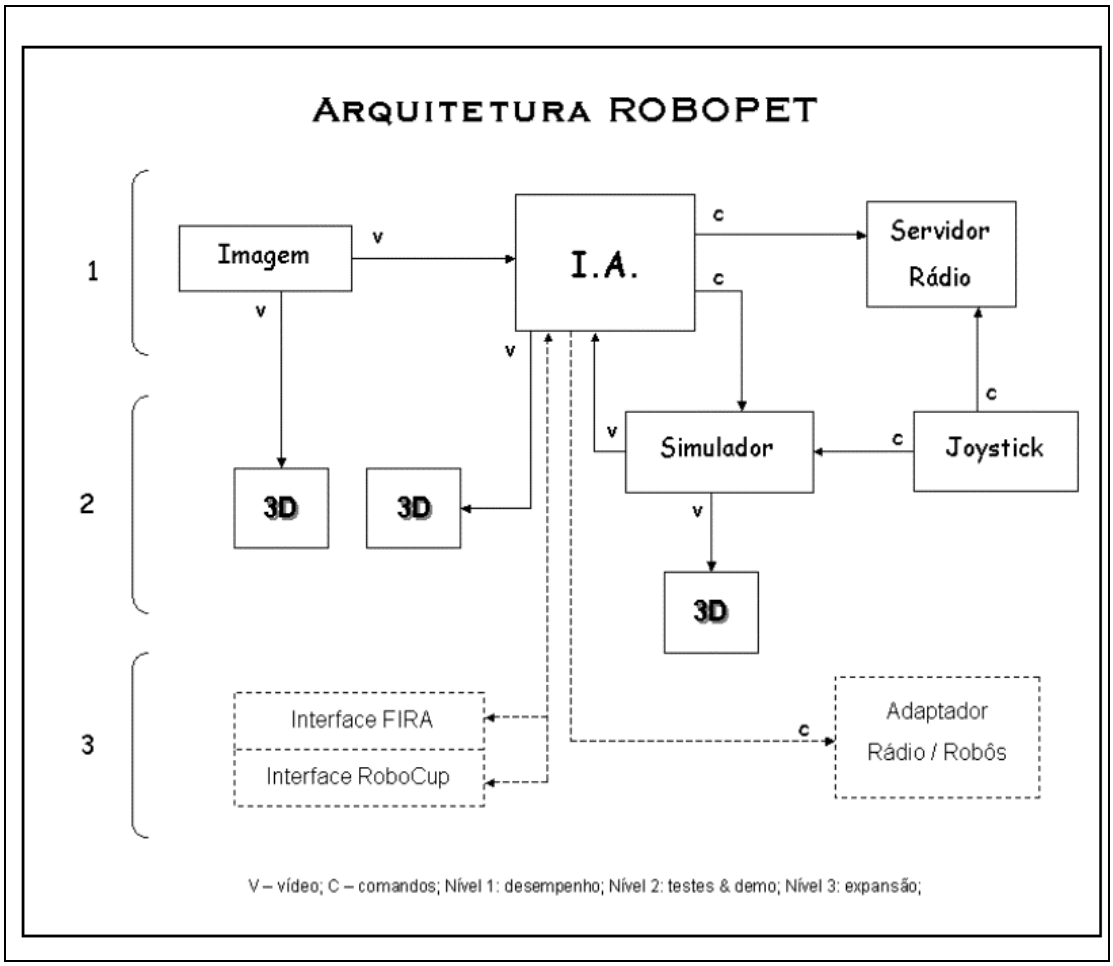
Figura 9 – Humano contra o robô `humainoid`

2.8.5 Larf

Em Jacobsen (2003, p. 17) é apresentada uma extensão da Linguagem para Definição de Agentes Robôs Jogadores de Futebol (LARF), definida em Schlei (2002). A extensão da LARF consiste na adição de comandos para utilização de variáveis e comunicação entre os jogadores de futebol (robôs). Ainda é definido e implementado um time que executa uma jogada ensaiada utilizando a comunicação para se coordenarem.

2.8.6 Robopet

Em Alegretti (2004, p. 24) é descrito o ROBOPET, um sistema de software distribuído para o futebol de robôs. O autor desenvolve uma arquitetura completa para o funcionamento de um time de Futebol de Robôs. A arquitetura ROBOPET é baseada em módulos. Cada módulo constitui um programa independente, que pode ser implementado em qualquer linguagem de programação. Os módulos do ROBOPET são: inteligência artificial, visão computacional, simulador, comunicação via rádio e sistema de controle remoto. A arquitetura ROBOPET pode ser vista na Figura 10.



Fonte: Alegretti (2004, p 26).

Figura 10 – Arquitetura ROBOPET.

2.8.7 Comparação dos ambientes estudados

No Quadro 1 é demonstra uma comparação entre os simuladores estudados. As avaliações foram realizadas mediante uso dos simuladores e em fórum de discussões (MANAGERZONE, 2007; GAMEGOL, 2007; HATTRICK, 2007).

	Managerzone	**Hattrick**	**Gamegol**
Aplicação	Web	Web	Web
Operacionalidade	Simples	Complexa	Simples
Nível de aproximação com as atribuições de um clube no mundo real (gerenciamento do time).	Muito próximo	Próximo	Muito próximo
Nível de aproximação da simulação com uma partida real	Muito próximo	Próximo	Muito próximo
Substituições durante a partida	Possui	Não possui	Possui
Troca de tática durante a partida	Não possui	Não possui	Não Possui
Visualização da partida	2D e 3D	Não possui	2D

Quadro 1 – Comparação entre os ambientes estudados

3 DESENVOLVIMENTO DO TRABALHO

Inicialmente neste capítulo são apresentadas as definições utilizadas no trabalho, como estratégia e tática. A seguir é mostrado os requisitos principais do problema, especificação, implementação, resultados e discussões.

3.1 DEFINIÇÕES UTILIZADAS NO TRABALHO

A seguir é descrito o que é estratégia e o que é tática no simulador desenvolvido.

3.1.1 Estratégia e Tática

A estratégia é o conjunto de táticas que a equipe poderá utilizar durante a partida. Para se realizar uma simulação é necessário selecionar as duas equipes juntamente com a suas respectivas estratégias. Cada estratégia possui uma tática inicial, que poderá ser alterada dependendo da situação prevista antes da partida. Um exemplo é se a equipe estiver perdendo de dois gols de diferença aos trinta (30) minutos do segundo tempo, pode-se trocar a tática para uma que o usuário julgue ser mais ofensiva que a atual. Fica totalmente a critério do usuário o tempo de jogo e a diferença de gols para que a troca de tática seja efetuada. Não existe um número limite de táticas que podem ser usadas em uma estratégia.

Neste trabalho na tática são definidos os jogadores do plantel que vão jogar quando a mesma estiver sendo usada. Para cada jogador escalado defini-se a posição base em campo.

3.2 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os Requisitos Funcionais (RF) e os Requisitos Não Funcionais (RNF) previstos para o software são:

a) o sistema deverá permitir que o usuário possa criar os times juntamente com os seus jogadores. O usuário informará o valor de cada atributo do jogador, conforme considere ideal para a função que o mesmo exercerá nas partidas. Cada atributo poderá conter de zero (0) a dez (10), mas a soma de todos os atributos não poderá passar de trinta (30). Então, se o usuário deseja que o jogador seja um atacante, os atributos relacionados com o ataque devem ser valorizados (RF);
b) um time poderá possuir quantas táticas de jogo o usuário desejar construir. Para ser utilizada numa partida a tática deverá estar dentro de uma estratégia (RF);
c) o simulador deve permitir que uma equipe possa usar várias táticas em uma partida, mas sem a interferência do usuário durante a mesma. Ele deve prever anteriormente na estratégia uma possível situação de jogo para poder alterar a tática (RF);
d) o sistema deverá permitir a visualização gráfica em 2D da partida (RF);
e) o sistema deverá possuir um árbitro (que não será visível como os jogadores) na partida, para validar os gols, marcar as faltas, aplicar os cartões (amarelo e vermelho), saídas de bola do campo e impedimentos (RF);
f) o sistema deverá determinar que todas as disputas, lances e atitudes dos jogadores sejam decididos com base nos seus atributos e com aleatoriedade, pois numa disputa de dois jogadores pela bola nem sempre o melhor é que levará vantagem na jogada (RF);
g) o sistema deverá simular todas as ações do jogador, como posicionamento, correr (com e sem a bola), passar a bola, chutar, desarmar e defender o gol. Não haverá fatores físicos como peso e altura ou psicológicos (RF) ;
h) o sistema deve implementar os movimentos da bola, como rolando no chão (RF);
i) o sistema será implementado com a ferramenta Delphi 7.0 utilizando a biblioteca OpenGL para a visualização da partida (RNF).

3.3 ESPECIFICAÇÃO

A especificação da ferramenta utiliza a técnica de orientação a objetos, com os diagramas de casos de uso, classes e seqüência da UML. A ferramenta usada para fazer a modelagem foi a Enterprise Architect.

3.3.1 Diagramas de casos de uso

Na Figura 11 são mostrados os casos de uso do usuário.

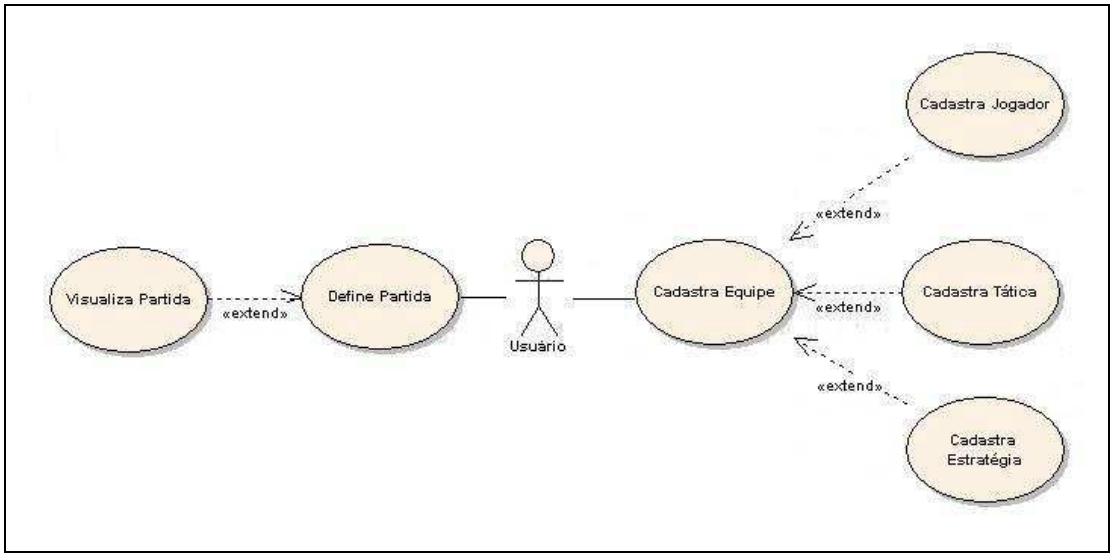


Figura 11 – Diagrama de casos de uso

Os casos de uso do usuário descritos na Figura 11 são:

a) `Cadastra Equipe`: permite ao usuário criar, alterar e excluir a equipe;

b) `Cadastra Jogador`: permite ao usuário criar, alterar e excluir jogadores;

c) `Cadastra Tática`: permite ao usuário criar, alterar e excluir táticas, vinculando os jogadores que irão fazer parte da mesma;

d) `Cadastra Estratégias`: permite ao usuário criar, alterar e excluir estratégias, vinculando as táticas que irão fazer parte da mesma;

e) `Define Partida`: define as duas equipes que irão jogar, juntamente com a estratégia que cada uma delas irá usar;

f) `Visualiza Partida`: permite ao usuário visualizar a partida definida.

3.3.2 Diagramas de classes

O diagrama das classes do simulador é apresentado na Figura 12. As classes que compõem este diagrama são: `TSimulador`, `TfrSimuladorGUI`, `TTempoPlacar`, `TCampo`, `TArbitro`, `TBola`, `TMovimento`, `TMovimentoJogador`, `TEquipe` ,`TEquipePartida`, `TEstrategia`, `TTaticaEstrategia`, `TTatica`, `TJogador`, `TJogadorEscalado` e `TJogadorPartida`.

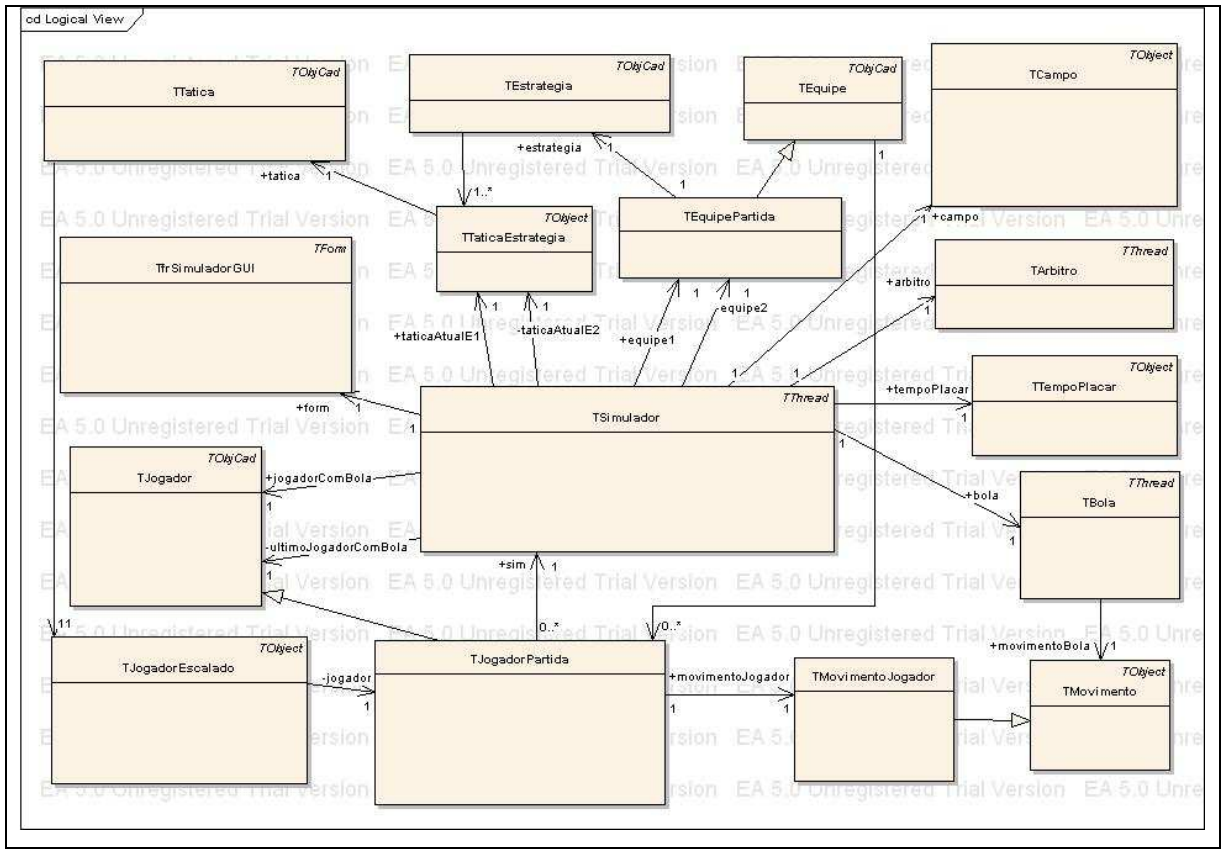


Figura 12 - Diagrama de classes da simulação da partida

3.3.2.1 Classe `TSimulador`

A `TSimulador` (Figura 13) é a principal classe do simulador. É nela que estão ligadas todas as demais classes da simulação. As duas equipes, a estratégia que cada uma delas vai utilizar e o *form* de visualização já são passados como parâmetro no seu construtor. Todos os demais objetos são criados dentro do simulador, como: tempo, placar, árbitro, campo e bola.

O simulador também é responsável em verificar qual a tática a ser usada pelo time. Inicialmente o simulador identifica a tática a ser usada, inicializa os jogadores (*threads*) e inicia a partida. Na inicialização dos jogadores o simulador passa a sua referência para os jogadores, com isso é possibilitado aos jogadores obter informações sobre um outro objeto através do simulador, pois cada um tem um atributo que é uma referência do simulador. Um exemplo disso é o jogador querer saber onde está a bola, ou um jogador da sua ou de outra equipe, ou então saber qual é o jogador que está com a bola.

Um atributo importante do simulador é o `estadoPartida`, que informa a todos os componentes da partida qual é o estado atual da partida. Os estados da partida são:

a) `epIniciandoComponentes`: todos os objetos necessários para a partida não foram instanciados;
b) `epAguardandoInicio`: todos os objetos já estão criados e os jogadores já estão em campo. Agora eles devem se posicionar para o começo da partida;
c) `epEmAndamento`: a bola está em jogo;
d) `epGol`: ocorreu um gol na partida. Os jogadores devem se posicionar para o recomeço da partida;
e) `epIntervalo`: fim do primeiro tempo. Os jogadores devem de posicionar para o começo da próxima etapa;
f) `epFalta`: ocorreu uma falta na partida. Os jogadores se posicionam e um jogador cobra a falta;
g) `epLateral`: houve uma saída da bola pela linha lateral. Um jogador vai cobrar o aremesso lateral, enquanto isto os demais se posicionam;
h) `epLinhaDeFundo`: a bola saiu pela linha de fundo;
i) `epFinal`: fim da partida.

O método `getQueryPerformanceCounter` da classe `TSimulador` é utilizado pelos jogadores e árbitro no momento em que eles necessitam de um número aleatório para definir uma jogada. Este número de zero (0) a nove (9) é o último algarismo do resultado da função `QueryPerformanceCounter` da API do Windows, que retorna em milisegundos o tempo em que o micro computador está ligado.

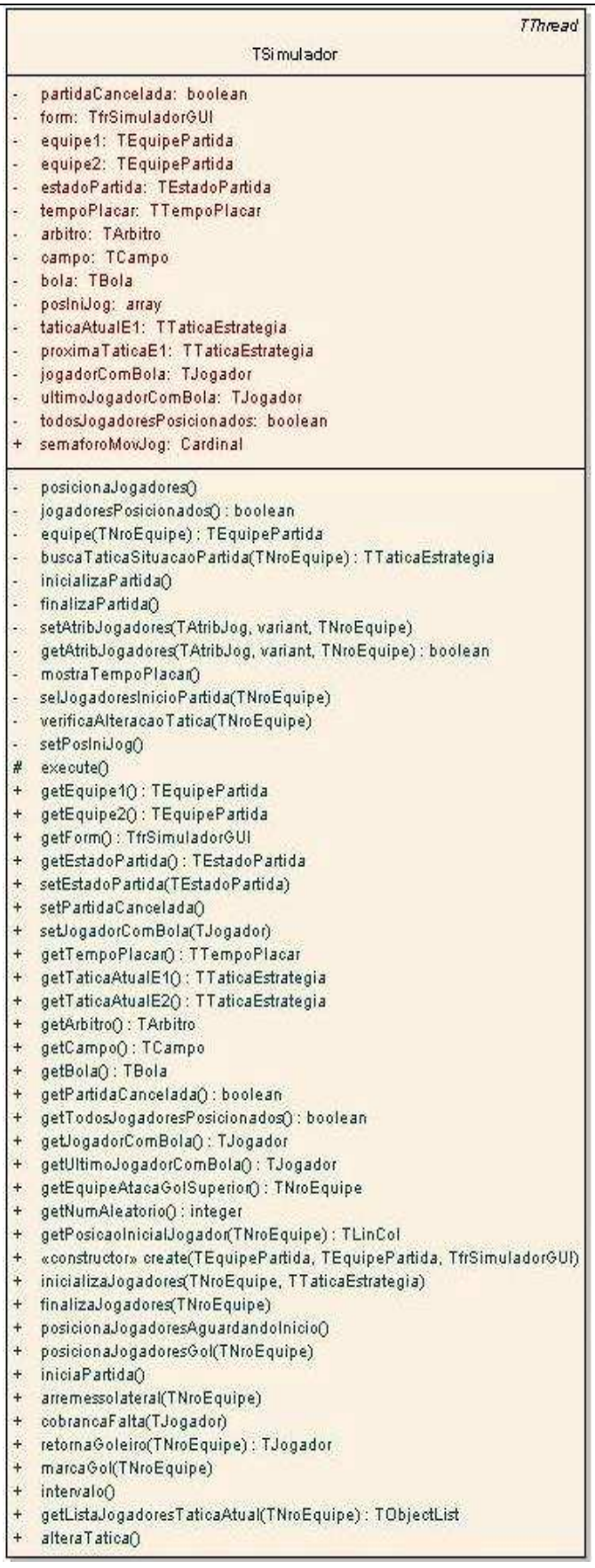


Figura 13 - Classe `TSimulador`

3.3.2.2 Classe `TfrSimuladorGUI`

A classe `TfrSimuladorGUI` (Figura 14) é a classe (*form*) que disponibiliza a visualização da partida e é nela que estão implementadas as rotinas que utilizam a biblioteca OpenGL. Ela possui um componente `TTimer` que a cada cem[1] (100) milisegundos redesenha toda a tela da simulação.

Figura 14 – Classe `TfrSimuladorGUI`

3.3.2.3 Classe `TTempoPlacar`

A classe `TTempoPlacar` (Figura 15) é responsável em fazer a contagem do tempo de jogo e em anotar o placar do jogo.

[1] Intervalo ideal para um micro Pentium 2.4 com 512mb RAM.

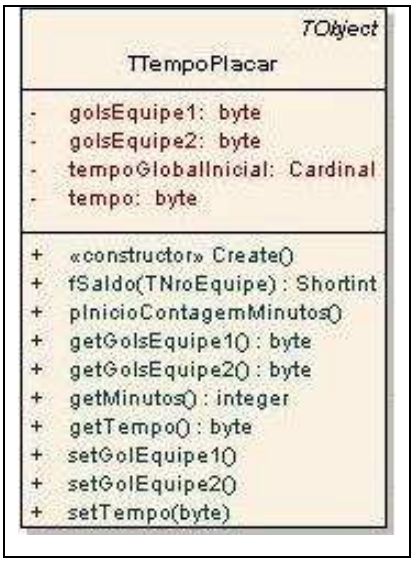


Figura 15 – Classe `TTempoPlacar`

3.3.2.4 Classe `TCampo`

A Classe `TCampo` (Figura 16) contém todas as medidas do campo, que são proporcionais as medidas reais oficiais. É com base nos seus valores que a `TfrSimuladorGUI` desenha o campo, a bola e os jogadores. Aqui também são armazenadas as constantes de deslocamento dos objetos, tamanho da bola e jogadores, distâncias de marcação, passe e área de chute ao gol, além de funções para conversão de `TLinCol` para `TPosicao` e cálculo de distância entre dois pontos.

Uma função importante do campo é não deixar que nenhum jogador sobreponha o outro. Para isso, existe um semáforo para cada ponto do campo. Se um jogador requisitar um ponto que já esteja ocupado por outro, o método do campo `alteraPosicaoJogador` resulta o lado do jogador que ocorreu a colisão, e não permite o deslocamento. Se a posição requisitada estiver livre, este método retorna informando que não houve colisão.

Figura 16 - Classe `TCampo`

3.3.2.5 Classe `TArbitro`

O árbitro (Figura 17) é uma *thread* e fica constantemente verificando se algo acontece durante a partida, como por exemplo, se a bola sai de campo, se ocorreu um gol, ou quando ocorre uma falta. O árbitro detecta o que ocorreu e altera o estado da partida e chama um método do simulador que vai posicionar os jogadores conforme a situação.

Figura 17 - Classe `TArbitro`

3.3.2.6 Classe `TBola`

A bola é passiva enquanto ela está sob domínio de um jogador. Quando o jogador passa ou chuta a bola ela entra em movimento, passando a ser ativa e locomove-se até a posição que o jogador desejou. Se ela chega até o destino ou um outro jogador intercepta-a, ela passa novamente para o modo passivo. Esse estado da bola é controlado pelo atributo `emMovimento` da classe `TBola` (Figura 18).

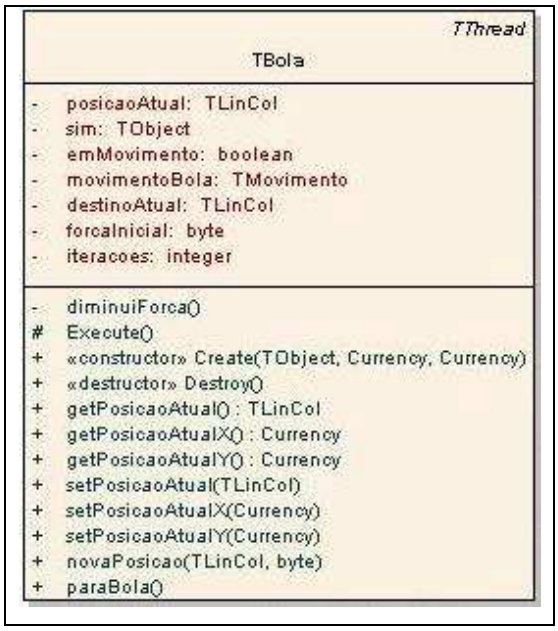


Figura 18 – Classe `TBola`

3.3.2.7 Classe `TMovimento`

A classe `TMovimento` (Figura 19) é utilizada na classe `TBola`. No caso do jogador é utilizada a classe `TMovimentoJogador`, que é descendente da classe `TMovimento`. Tanto a bola ou jogador quando em qualquer movimentação, têm um objetivo que é um ponto específico. Este objetivo não necessariamente é atingido num único tempo, pois cada jogador ou bola é modelado como sendo uma *thread*, e cada *thread* é executada durante uma fatia de tempo. Sendo assim, só é possível dar um passo por vez e a classe `TMovimento` informa para onde é o próximo passo do objeto com base no seu destino final e nas últimas dez (10) movimentações realizadas. As duzentas (200) últimas movimentações são guardadas objetivando auxiliar na escolha da rota com menos custo, o mais próximo possível de uma linha reta. São guardadas as últimas duzentas (200) porque dificilmente o número de iterações de um movimento da bola ou jogador será maior que este valor.

Figura 19 - Classe `TMovimento`

3.3.2.8 Classe `TMovimentoJogador`

A classe `TMovimentoJogador` (Figura 20) é descendente da classe `TMovimento`. A única diferença é o método `novaPosicaoJogador`. Este método é responsável pela tomada de decisão do caminho a ser seguido pelo jogador que está com a bola no momento.

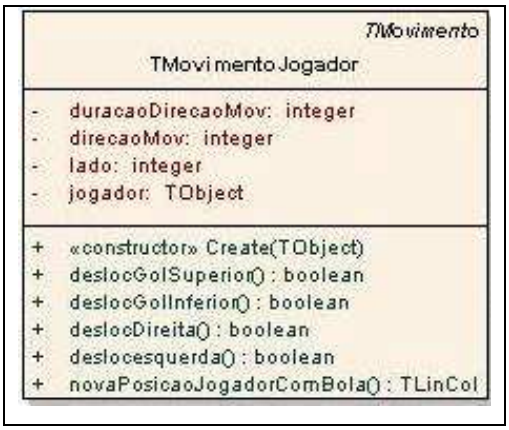


Figura 20 – Classe `TMovimentoJogador`

3.3.2.9 Classe `TEquipePartida`

A classe `TEquipePartida` (Figura 21) é descendente da classe `TEquipe`, que é utilizada no cadastro. A classe `TEquipePartida` contém a lista com todos os jogadores da equipe, juntamente com a estratégia escolhida pelo usuário.

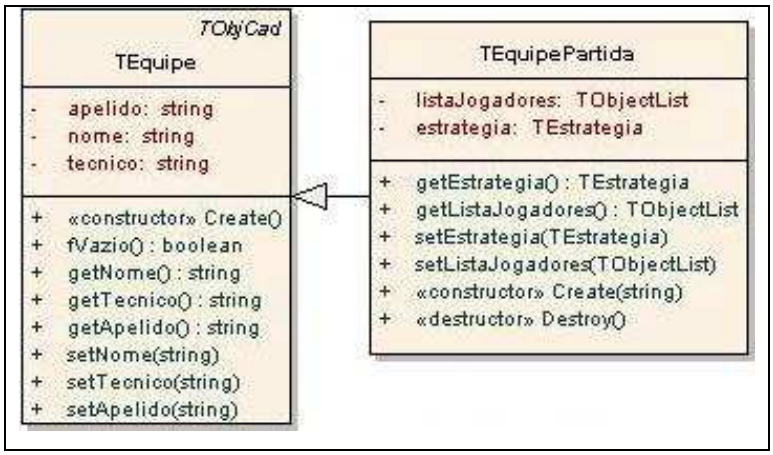


Figura 21 - Classes `TEquipe` e `TEquipePartida`

3.3.2.10 Classe `TEstratégia`

A classe `TEstrategia` (Figura 22) contém a lista com as táticas (`TTaticaEstrategia`) que o usuário poderá utilizar durante a partida e a lista de todos jogadores (`TJogadorPartida`) do plantel da equipe.

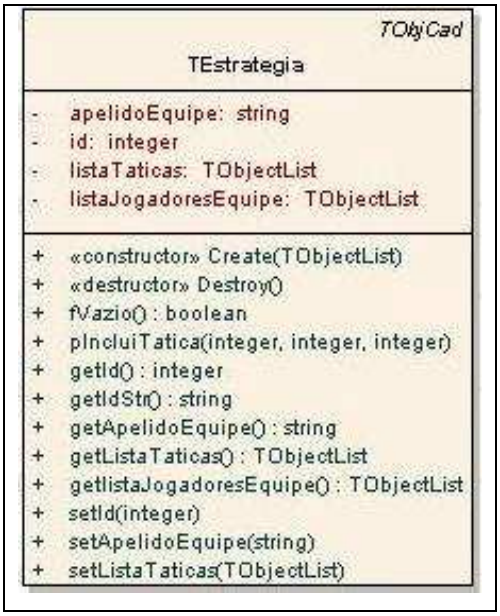


Figura 22 – Classe `TEstrategia`

3.3.2.11 Classe `TTaticaEstrategia`

A classe `TTaticaEstrategia` (Figura 23) contém os atributos minuto, saldo e a tática. A tática somente é usada durante o jogo. Ela é programada para ser ativada num determinado tempo (minutos de jogo), somente se o saldo de gols definidos seja igual ao da partida atual.

O simulador verifica a cada minuto se existe alguma troca de tática programada para o minuto atual da partida. Se tiver programado, somente na próxima parada de bola o simulador vai trocar a tática da equipe, pois pode haver substituições de jogadores.

Figura 23 - Classe `TTaticaEstrategia`

3.3.2.12 Classe `TTatica`

A classe `TTatica` (Figura 24) contém a lista com os jogadores escalados (`TJogadorEscalado`) e a posição base de cada um em campo. A lista de jogadores da equipe já é passado como parâmetro no seu construtor.

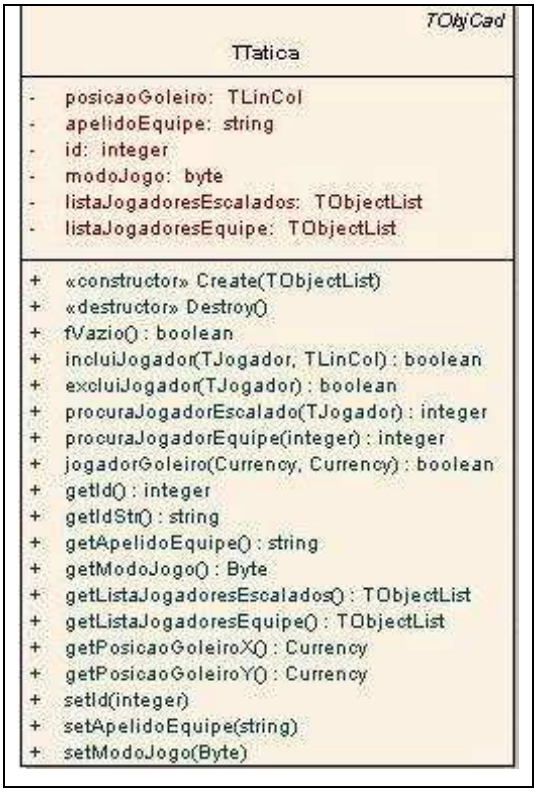


Figura 24 – Classe `TTatica`

3.3.2.13 Classe `TJogador`

A classe `TJogador` (Figura 25) é a que vai conter os atributos básicos dos jogadores. Os atributos podem conter de zero (0) a dez (10). Quanto maior o valor, melhor ele é neste item. Os atributos são:

a) `velocidade`: índice de velocidade de deslocamento do jogador;
b) `resistência`: índice para se medir o cansaço do jogador durante a partida;
c) `controle de bola`: índice de domínio de bola (quanto maior, mais habilidoso é o jogador);
d) `desarme`: índice de eficiência no roubo de bola do adversário;
e) `passe`: índice de eficiência dos passes efetuados pelo jogador;
f) `chute`: índice de acertos dos chutes ao gol feitos pelo jogador;
g) `defesa a gol`: índice de eficiência do goleiro.

Figura 25 – Classe `TJogador`

3.3.2.14 Classe `TJogadorEscalado`

A Classe `TJogadorEscalado` (Figura 26) possui dois atributos, `jogador` que aponta para qual jogador refere-se, e `posicaoBase`, que aponta a posição base do jogador na tática que este está escalado.

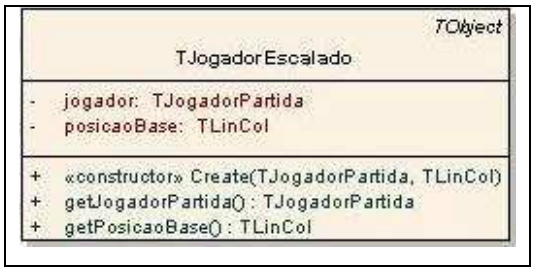


Figura 26 - Classe `TJogadorEscalado`

3.3.2.15 Classe `TJogadorPartida`

Todos os jogadores em campo são instâncias da classe `TJogadorPartida` (Figura 27). É nesta classe que estão implementadas todas as ações e o comportamento dos jogadores em campo (exceto a movimentação com a bola, que está na classe `TMovimentoJogador`). Métodos como `chuta`, `passa`, `desarma`, `acaoGoleiro` e `dominaBola` estão nesta classe.

Os atributos dos jogadores influenciam diretamente nesta classe, pois todas as ações dos jogadores sofrerão interferências dos métodos `getResistenciaAtual`, `getVelocidadeAtual`, `getContrBolaAtual`, `getPasseAtual`, `getChuteAtual`, `getDesarmeAtual` e `getDefesaGolAtual`. Todos os atributos vão decrementando durante a partida, porque os mesmos estão ligados a resistência, que está diretamente ligada ao tempo de jogo.

A detecção da situação do jogo pelo jogador é feita no método `execute`, no qual está o laço principal da classe. Exemplos de situação do jogador são:

a) se um companheiro está com a bola;
b) se é um adversário que está com a bola;
c) se nenhum jogador está com a bola;
d) se ele deve ir marcar ou ficar na sua posição;
e) se ocorreu um gol.

É importante ressaltar que cada jogador é totalmente independente. Ele decide as ações a serem realizadas. O único caso em que os jogadores não são totalmente independentes é no caso de uma falta, gol ou bola fora de campo. Nestes casos o árbitro altera o estado da partida e o simulador determina que cada jogador se posicione na sua posição base. Determina também qual dos jogadores é que vai recolocar a bola em jogo.

Figura 27 – Classe `TJogadorPartida`

3.4 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas, a implementação e a operacionalidade da implementação.

3.4.1 Técnicas e ferramentas utilizadas

A implementação foi realizada usando a programação orientada a objetos no ambiente Delphi 7, juntamente com a biblioteca OpenGl para a visualização da partida.

3.4.2 Código implementado

No método de construção do simulador (pertencente a classe `TSimulador` (Quadro 2)) é informado como parâmetro as duas equipes. Juntamente com as equipes estão definidos os jogadores, a estratégia e as táticas que serão usadas na partida. A referência da instância da classe `TSimuladorGUI` que cuida da visualização também é informada.

```
constructor TSimulador.Create(const prEquipe1 : TEquipePartida;
    const prEquipe2: TEquipePartida; const prForm : TfrSimuladorGUI);
begin

  // cria a thread TSimulador em modo de execução
  inherited Create(false);

  // as equipes, jogadores, táticas e estratégias são passadas como parâmetro
  equipe1 := prEquipe1;
  equipe2 := prEquipe2;

  // o form de vizualização também é passado
  form := prForm;

end;
```

Quadro 2 – Construtor da classe `TSimulador`

Após a criação do simulador é executado o método `execute` (Quadro 3), o qual concorre com as outras *threads*. No método `execute` (Quadro 3) primeiramente é executado método `inicializaPartida` (Quadro 4), que além de iniciar a partida é responsável também em instanciar todas as demais classes que fazem parte da partida. Após, o simulador fica num

laço até que o contador de minutos do objeto `TTempoPlacar` chegar em noventa (90) ou o usuário cancelar a partida. Durante este laço o simulador verifica se uma alteração de tática deve ser feita e atualiza o placar na tela de visualização. Ao término da partida, os objetos são finalizados.

```
procedure TSimulador.Execute;
begin

  // seta o atributo de cancelamento da partida como false
  partidaCancelada := false;

  // inicializa todos os componentes que farão parte da partida
  inicializaPartida;

  // o tempo de duração da partida é de 90 minutos, ou até o momento que o
  // usuário cancelar a partida fechando a tela de visualização
  while (tempoPlacar.getMinutos <= 90) and (not partidaCancelada) do
    begin

      sleep(500);

      if tempoPlacar.getMinutos > 0 then
         begin
           // vefifica se a tática deve ser alterada
           verificaAlteracaoTatica(e1);
           verificaAlteracaoTatica(e2);
         end;

      // atualiza o tempo e placar na tela
      mostraTempoPlacar;

    end;

  // finaliza todos os componentes
  finalizaPartida;

end;
```

Quadro 3 – Método `execute` da classe `TSimulador`

No método `inicializaPartida` (Quadro 4) são instanciados os objetos `TCampo`, `TTempoPlacar`, `TBola` e `TArbitro`. Os jogadores são inicializados e colocados em campo no método `iniciaJogadores`, conforme a tática inicial da estratégia escolhida pelo usuário. Os jogadores buscam a sua posição inicial após o árbitro alterar a partida para o estado `epAguardandoInicio`. Com os jogadores já posicionados é iniciada a partida e a contagem dos minutos, com o árbitro alterando o estado da partida para `epEmAndamento`.

```
procedure TSimulador.inicializaPartida;
begin

  // inicializa variáveis
  proximaTaticaE1 := nil;
  proximaTaticaE2 := nil;
  jogadorComBola := nil;
  ultimoJogadorComBola := nil;

  // instância o campo
  campo := TCampo.Create(self);

  // instância o Tempo e Placar
  tempoPlacar := TTempoPlacar.Create;

  // mostra o campo e placar
  mostraTempoPlacar;

  // instância a bola
  bola := TBola.Create(self,0,0);

  // define a posição inicial dos jogadores
  setPosIniJog;

  // inicializa os jogadores das duas equipes
  inicializaJogadores(e1,buscaTaticaSituacaoPartida(e1));
  inicializaJogadores(e2,buscaTaticaSituacaoPartida(e2));

  // instância o arbitro
  arbitro := TArbitro.Create(self);

  // altera o estado da partida para AguardandoInicio
  arbitro.alteraEstadoPartida(epAguardandoInicio);

  // posiciona os jogadores para começar a partida
  posicionaJogadoresAguardandoInicio;

  // é inicida a contagem dos minutos
  tempoPlacar.pInicioContagemMinutos;

  // o arbitro inicia a partida
  arbitro.alteraEstadoPartida(epEmAndamento);

end;
```

Quadro 4 – Método `inicializaPartida` da classe `TSimulador`

O método `inicializaJogadores` (Quadro 5) recebe como parâmetro a tática da equipe e executa o método `iniciaJogadorPartida` (Quadro 6) da classe `TJogadorEscalado`.

```
procedure TSimulador.inicializaJogadores(const prEquipe : TNroEquipe ;
     const prTatica : TTaticaEstrategia);
var i : integer;
    jogEsc : TJogadorEscalado;
begin

  // atribui a nova tática aos atributos que correspondem as táticas atuais das duas equipes
  if prEquipe = e1 then
     begin
       taticaAtualE1 := prTatica;
     end
  else
     begin
       taticaAtualE2 := prTatica;
     end;

  // atribui a posição base de cada jogador e inicializa cada um deles
  for i := 0 to prTatica.getTatica.getlistaJogadoresEscalados.Count-1 do
    begin
      jogEsc := TJogadorEscalado(prTatica.getTatica.getListaJogadoresEscalados.Items[i]);
      jogEsc.iniciaJogadorPartida(prEquipe,self);
    end;

end;
```

Quadro 5 – Método `inicializaJogadores` da classe `TSimulador`

O método `iniciaJogadorPartida` (Quadro 6) da classe `TJogadorEscalado` seta a posição inicial do jogador, executando o método `iniciaJogador` (Quadro 7) da classe `TjogadorPartida`, o qual seta a equipe na qual o jogador pertence, a equipe 1 (`e1`) ou equipe 2 (`e2`), e informa a referência do jogador ao simulador da partida. Ao final é executado o método `resume` do jogador, o qual ativa a execução da *thread* jogador, sendo o método `execute` acionado. Com isso o jogador está pronto para a partida.

```
procedure TJogadorEscalado.iniciaJogadorPartida(const prEquipe: TNroEquipe;
  const prSim: TObject);
begin

  // seta a posicao base do jogador
  jogador.setPosicaoBaseX(getPosicaoBase.x / 1000);
  jogador.setPosicaoBaseY(getPosicaoBase.y / 1000);

  // prepara o jogador para a partida
  jogador.iniciaJogador(prEquipe,TSimulador(prSim));

  // inicializa a thread do jogador
  jogador.Resume;

end;
```

Quadro 6 – Método `iniciaJogadorPartida` da classe `TJogadorEscalado`

```
procedure TJogadorPartida.iniciaJogador(const prEquipe : TNroEquipe;
    const prSim: TSimulador);
var wPosIni : TLinCol;
begin

  // atribui o simulador
  sim := prSim;

  // identifica se o jogador faz parte da equipe 1 ou da equipe 2
  equipe := prEquipe;

  // seta o jogador como posicionado
  jogadorPosicionado := true;

  // como na construção da tática o time e configurado como  atacando o gol superior
  // se ele esta atacando para o gol inferior a posição base deve ser "invertida"
  setPosicaoBaseX(getPosicaoBaseX * multLadoCampo);
  setPosicaoBaseY(getPosicaoBaseY * multLadoCampo);

  // instância o objeto de movimentação
  movimentoJogador := TMovimentoJogador.Create(self);

  // se for inicio do primeiro ou segundo tempo a posicao inicial do jogador é
  // configurada para ele ficar em fila
  if (sim.getTempoPlacar.getMinutos = 0) or (sim.getEstadoPartida = epIntervalo) then
     begin
       wPosIni := prSim.getPosicaoInicialJogador(equipe);
       posicaoAtual.x := wPosIni.x;
       posicaoAtual.y := wPosIni.y;
     end
  else
     begin
       // se for troca de tática seta a posição base
       posicaoAtual.x := posicaoBase.x;
       posicaoAtual.y := posicaoBase.y;
     end;

  // inicializa os pontos ocupados
  SetLength(pontosPosicaoAtual,0);

end;
```

Quadro 7 - Método `iniciaJogador` da classe `TJogadorPartida`

No método `execute` do `TJogadorPartida` o jogador fica num laço até o final da partida. Ali o jogador analisa que atitude deve-se tomar conforme o cenário atual da partida.

Primeiramente no método `execute` é verificado se o jogador está posicionado (Quadro 8). Ele deve posicionar-se quando ocorre uma falta, gol, lateral, tiro de meta ou início da partida. O árbitro detecta o evento e solicita que o simulador informe aos jogadores o que ocorreu através do estado da partida. O simulador seta todos os jogadores como não posicionados, fazendo com que os jogadores busquem o seu posicionamento.

```
procedure TJogadorPartida.Execute;
var linColMov : TLinCol;
    jogComBola : TJogadorPartida;
begin

  // faz enquanto a partida não chegar ao final
  while (sim.getEstadoPartida <> epFinal) do
    begin

      // se o jogador não está posicionado quer dizer que a partida não está em
      // andamento, e o jogador deve se posicionar conforme o que for definido no
      // método posicionaJogadorPosicaoBase
      if not jogadorPosicionado then
         begin

           // da uma acelerada na movimentação, pois o delay normal de movimentação é
           // maior que este
           sleep(floor(15 - (getVelocidadeAtual * 0.1)));

           // o jogador se posiciona conforme a situação da partida
           posicionaJogadorPosicaoBase;

           // se o jogador está posicionado ele aguarda que todos os jogadores se posicionem
           if jogadorPosicionado then
              begin

                // aguarda os jogadores se posicionarem
                while not sim.getTodosJogadoresPosicionados do
                  begin
                    sleep(10);
                  end;

                // requisita os novos pontos (requisita exclusividade dos semáforos)
                alteraPosicao(posicaoAtual);

              end;

         end
```

Quadro 8 – Parte de posicionamento do jogador do método `execute` do `TJogadorPartida`

No Quadro 9 tem-se um detalhamento do método `posicionaJogadorPosicaoBase` da classe `TJogadorPartida`.

```
procedure TJogadorPartida.posicionaJogadorPosicaoBase;
var posFinal : TLinCol;
begin

  case sim.getEstadoPartida of
    // se o estado da partida for aguardando inicio (1º ou 2º tempo), ou se ocorreu um gol
    epAguardandoInicio, epGol :
       begin

         // se o jogador for o que vai dar o primeiro toque na bola vai se posicionar no meio de campo
         if iniciaPartida[1] then
            begin
              posFinal.x := sim.getBola.getPosicaoAtualX + sim.getCampo.getTamanhoBola + sim.getCampo.getTamJogGridDiv2;
              posFinal.y := sim.getBola.getPosicaoAtualY;
            end
         else
         // se o jogador for o que vai dar o segundo toque na bola vai se posicionar no meio de campo
         if iniciaPartida[2] then
            begin
              posFinal.x := sim.getBola.getPosicaoAtualX - sim.getCampo.getTamanhoBola - sim.getCampo.getTamJogGridDiv2;
              posFinal.y := sim.getBola.getPosicaoAtualY;
            end
         else
            begin

              // senão sele vai se posicionar na sua posicao base,
              // mas não podendo ir para o campo de ataque
              posFinal.x := posicaoBase.x;
              posFinal.y := posicaoBase.y;

              if atacaGolSuperior then
                 begin
                   if posFinal.y > 0 then
                      posFinal.y := 0
                   else
                   if posFinal.y = 0 then
                      posFinal.y := -0.05;
                 end
              else
                 begin
                   if posFinal.y < 0 then
                      posFinal.y := 0
                   else
                   if posFinal.y = 0 then
                      posFinal.y := 0.05;
                 end;

              // não pode ficar dentro do circulo central, somente ficarão os jogadores que iniciarão a partida
              if (not iniciaPartida[1]) and (not iniciaPartida[2]) and (abs(posFinal.y) < sim.getCampo.getRaioCirculoCentral) then
                  begin
                    if (posFinal.x < 0) and (posFinal.x > (sim.getCampo.getRaioCirculoCentral * (-1))) then
                       posFinal.x := posFinal.x - sim.getCampo.getRaioCirculoCentral
                    else
                       if (posFinal.x >= 0) and (posFinal.x < (sim.getCampo.getRaioCirculoCentral)) then
                          posFinal.x := posFinal.x + sim.getCampo.getRaioCirculoCentral
                  end;

            end;

       end;
```

Quadro 9 – Parte do método `posicionaJogadorPosicaoBase` do `TJogadorPartida` que posiciona o jogador no caso do jogo não ter iniciado ou se ocorreu um gol.

Enquanto o estado da partida está `emAndamento`, o `TJogadorPartida` faz algumas verificações para depois tomar uma atitude. A primeira verificação é do goleiro (Quadro 10). Se o jogador goleiro estiver com a bola, ele aguarda dois (2) segundos e passa a bola, senão é executada o método `acaoGoleiro` (Quadro 11), que determina a ação do goleiro enquanto ele não está com a bola.

```
// se o jogador é o goleiro
if goleiro then
   begin
     // se o goleiro esta com a bola, delay de 2 segundos
     // e passa a bola
     if comBola then
        begin
          sleep(2000);
          passa(false);
        end
     else
        begin
          // senão executa o acaoGoleiro
          acaoGoleiro;
        end;
   end
```

Quadro 10 – Parte do método `execute` do `TJogadorPartida` que se refere ao goleiro

No método `acaoGoleiro` (Quadro 11), o goleiro verifica se a bola está perto, ou seja, se pelo menos um dos pontos ocupados pela bola entre em colisão com um ou mais pontos do goleiro. Caso esteja, é determinado um número aleatório de zero (0) a nove (9). Se o número sorteado for maior que o seu atributo de defesa a gol naquele momento da partida, o goleiro não defenderá a bola. Se o goleiro não está perto da bola, ele tenta se posicionar no mesmo "eixo x" da bola.

```
procedure TJogadorPartida.acaoGoleiro;
var novaPos : TLinCol;
begin

  // se o goleiro conseguiu chegar perto da bola
  if jogadorAoLadoDaBola then
     begin

       // é sorteado um numero (0 a 9) para ver se o goleiro vai fazer a defesa
       setNumAleatorio;

       // se a defesaGol for maior que o numero sorteado, o goleiro vai defender a bola
       // portanto quanto maior o atributo defesa maior a chance da defesa ser feita
       if (numAleatorio <= getDefesaGolAtual) then
          begin
            // agarra a bola
            sim.setJogadorComBola(self);
            exit;
          end;

     end;

  // delay de locomoção do goleiro, quanto mais velocidade menor é o delay
  sleep(120 - (getVelocidadeAtual * 4));

  // mesmo longe da bola o goleiro fica acompanhando a bola
  if (sim.getBola.getPosicaoAtualX < sim.getCampo.getXMinGol) then
     novaPos.x := sim.getCampo.getXMinGol
  else
  if (sim.getBola.getPosicaoAtualX > sim.getCampo.getXMaxGol) then
     novaPos.x := sim.getCampo.getXMaxGol
  else
     novaPos.x := sim.getBola.getPosicaoAtualX;

  // o valor de Y sempre será o base
  novaPos.y := self.getPosicaoBaseY;

  // altera a posicao
  alteraPosicao(novaPos);

end;
```

Quadro 11 – Método `acaoGoleiro` da classe `TJogadorPartida`

Quando o goleiro está com a bola, os demais jogadores movimentam-se para as suas posições bases (Quadro 12).

```
// enquanto o goleiro esta com a bola o jogador se move para a posicao base
if (jogComBola <> nil) and
    jogComBola.getGoleiro then
   begin
     movimentaJogador(posicaoBase);
   end
```

Quadro 12 – Parte do método `execute` que se refere a ação dos demais jogadores no momento que um goleiro esta com a bola

O jogador que está com a bola possui três opções de ação: passar para um companheiro, chutar a bola para o gol ou dar mais um passo com a bola (Quadro 13).

```
// se o jogador esta com a bola
if comBola then
   begin
     // se o jogador não vai chutar e nem passar ele se movimenta.
     if (not chuta) and (not passa(false)) then
        begin
          // quando o jogador está com a bola o método novaPosicaoJogadorComBola
          // da classe TMovimentaJogador é que determina o próximo passo do jogador
          linColMov := movimentoJogador.novaPosicaoJogadorComBola;
          movimentaJogador(linColMov);
          dominaBola;
        end;
   end
```

Quadro 13 - Parte do método `execute` que se refere ao momento que o próprio jogador está com a bola

No caso de nenhum jogador das duas equipes estiver com a bola (Quadro 14), o jogador verifica se ele está ao lado da bola, ou seja, se um dos pontos ocupados pela bola entra em colisão com algum dos seus pontos. Se estiver, é feito o domínio da mesma. Se não estiver, é verificado se existe um jogador da sua equipe mais perto da bola. Caso existir, ele move-se para a sua posição base. Se não existir, ele mesmo vai em busca da bola.

```
// se nenhum jogador esta com a bola
if jogComBola = nil then
   begin
     if (sim.getUltimoJogadorComBola = nil) or
        (sim.getUltimoJogadorComBola.getId <> self.getId) then
        begin
          // se a bola esta ao lado, o jogador domina a bola
          if jogadorAoLadoDaBola then
              begin
               sim.getBola.paraBola;
               if jogComBola = nil then
                  sim.setJogadorComBola(self);
              end
          else
              begin
                // se existe um jogador do time dele mais proximo,
                // ele vai para a sua posicao base, senão,
                // ele vai de encontro com a bola
                if existeJogadorMaisProximo then
                   movimentaJogador(posicaoBase)
                else
                   movimentaJogador(sim.getBola.getPosicaoAtual);
              end;
        end;
   end
```

Quadro 14 - Parte do método `execute` que se refere ao momento que nenhum jogador está com a bola

Se um companheiro estiver com a bola (Quadro 15), o jogador acompanha a bola se ele não for zagueiro. A distância de passe (cem (100) pontos do campo) é respeitada para evitar que vários jogadores fiquem ao redor do jogador com a bola.

```
// outro jogador da sua equipe está com a bola
if jogComBola.getEquipe = self.getEquipe then
   begin
     // se o jogador não for zagueiro e se ele esta próximo a bola, mas mais distante que a distância de passe, entao,
     // o jogador se move em direção a bola, senão,
     // o jogador se movimenta para a sua posicao base.
     if (not posicaoZagueiro) and
        (sim.getCampo.calculaDiscancia(sim.getBola.getPosicaoAtual,self.posicaoAtual) > (sim.getCampo.getDistPasse * 1.3)) then
        begin
          movimentaJogador(sim.getBola.getPosicaoAtual);
        end
     else
        begin
          movimentaJogador(self.getPosicaoBase);
        end;
   end
```

Quadro 15 - Parte do método `execute` que se refere ao momento que um companheiro de equipe está com a bola

Se um adversário estiver com a bola, primeiramente o jogador verifica a proximidade dele com a mesma ou ao do jogador que está com a ela. Se estiver próximo dele, vai tentar fazer o desarme. Caso contrário, ele vai verificar se é o jogador da sua equipe mais próximo da bola. Se for, ele vai de encontro a bola e tentar tomá-la. Se ele for zagueiro e a distância dele para a bola for menor que a distância de marcação, ele vai combater o adversário. Em qualquer outra situação o jogador irá permanecer na sua posição base (Quadro 16).

```
// outro jogador da equipe adversária está com a bola
else
   begin
     // se a bola, ou jogador com a bola estiver ao lado, então ele tenta desarmar
     if jogadorAoLadoDaBola or jogadorAoLadoJogadorComBola then
        desarma
     else
        begin
          // se ele é o jogador mais próximo da sua equipe do jogador que esta com a bola, então
          // ele vai ao encontro da bola
          if (not existeJogadorMaisProximo) then
             movimentaJogador(sim.getBola.getPosicaoAtual)
          else
          // se ele for zagueiro, e se ele estiver dentro da posição de marcação então
          // ele vai ao encontro da bola
          if posicaoZagueiro then
             begin
               if sim.getCampo.calculaDiscancia(self.posicaoAtual,sim.getBola.getPosicaoAtual) <
                  sim.getCampo.getDistMarcacao then
                  movimentaJogador(sim.getBola.getPosicaoAtual)
               else
                  movimentaJogador(self.getPosicaoBase);
             end
          else
             // senão vai para a posicao base
             movimentaJogador(self.getPosicaoBase);
        end;

   end;
```

Quadro 16 - Parte do método `execute` que se refere ao momento que um jogador adversário está com a bola

Independentemente da situação da partida, o jogador possui quatro (4) ações: passar, chutar, desarmar e movimentar. No caso do goleiro as ações são três (3): passar, movimentar e defender a bola.

Todas estas ações sofrem interferência direta dos atributos dos jogadores, os quais são: velocidade, resistência, chute, controle de bola, passe, desarme e defesa a gol. Sabe-se que quanto mais cansado o jogador estiver, mais lento ele vai correr. Essa relação entre os atributos pode ser visto a partir do Quadro 17 ao Quadro 23.

A resistência atual do jogador durante a partida somente sofre interferência do tempo de jogo. A cada 18 minutos jogados é decrementado em um (1) o atributo resistência.

```
function TJogadorPartida.getResistenciaAtual: byte;
begin
 //  Resistencia - diminui 5 bolas a resistência até o final da partida
 result := valPositivo(getResistencia - Floor(getMinutosJogados / 18));
end;
```

Quadro 17 – Valor atual da resistência do jogador durante a partida

A velocidade atual do jogador na partida é composta por sessenta por cento (60%) do atributo velocidade e quarenta por cento (40%) pela resistência atual (Quadro 18).

```
function TJogadorPartida.getVelocidadeAtual: byte;
begin
 // Velocidade - 60% depende da velocidade e 40% da resistência
 result := valPositivo((getVelocidade * 0.6) + (getResistenciaAtual * 0.4));
end;
```

Quadro 18 - Valor atual da velocidade do jogador durante a partida

O chute atual do jogador na partida é composto em sessenta por cento (60%) pelo valor nominal do atributo chute, vinte por cento (20%) pelo controle de bola atual e em vinte por cento (20%) pela resistência atual (Quadro 19).

```
function TJogadorPartida.getChuteAtual: byte;
begin

 // Chute Atual - 60% depende do chute, 20% do controle e 20% da resistência
 result := valPositivo((getChute * 0.6) + (getContrBolaAtual * 0.2) + (getResistenciaAtual * 0.2));

 if result = 0 then
    result := 1;

end;
```

Quadro 19 - Valor atual do chute do jogador durante a partida

O controle de bola atual do jogador na partida é composto em oitenta por cento (80%) pelo valor nominal do atributo controle de bola e em vinte por cento (20%) pela resistência atual (Quadro 20).

```
function TJogadorPartida.getContrBolaAtual: byte;
begin

 // Controde de Bola - 80% depende do controle e 20% da resistência
 result := valPositivo((getContrBola * 0.8) + (getResistenciaAtual * 0.2));

 if result = 0 then
    result := 1;

end;
```

Quadro 20 – Valor atual do controle de bola do jogador durante a partida

O passe atual do jogador durante a partida é composto em sessenta por cento (60%) pelo valor nominal do atributo passe, vinte por cento (20%) pelo controle de bola atual e em vinte por cento (20%) pela resistência atual (Quadro 21).

```
function TJogadorPartida.getPasseAtual: byte;
begin

 // Passe Atual - 60% depende do passe, 20% do controle e 20% da resistência
 result := valPositivo((getPasse * 0.6) + (getContrBolaAtual * 0.2) + (getResistenciaAtual * 0.2));

 if result = 0 then
    result := 1;

end;
```

Quadro 21 - Valor atual do passe real do jogador durante a partida

O desarme atual do jogador durante a partida é composto em oitenta por cento (80%)

pelo valor nominal do atributo desarme e em vinte por cento (20) % pela resistência atual (Quadro 22).

```
function TJogadorPartida.getDesarmeAtual: byte;
begin

 // Desarme Atual - 80% depende do desarme e 20% da resistência
 result := valPositivo((getDesarme * 0.8) + (getResistenciaAtual * 0.2));

 if result = 0 then
    result := 1;

end;
```

Quadro 22 - Valor atual do desarme real do jogador durante a partida

A defesa a gol real do jogador é composta em oitenta por cento (80%) pelo valor nominal do atributo defesa a gol e em vinte por cento (20%) pela resistência atual (Quadro 23). Porém, a defesa a gol real não pode passar de sete (7), evitando-se que o goleiro tenha um percentual de êxito de defesa maior que setenta por cento (70%).

```
function TJogadorPartida.getDefesaGolAtual: byte;
begin

 // no maximo a defesa será de 7, porque provavelmente todos os usuários criarão o
 // goleiro com 10 de defesa a gol

 // Defesa gol -  80% depende da defesa a gol e 20% da resistência
 result := valPositivo( ((getDefesaGol * 0.8) + (getResistenciaAtual * 0.2)) * 0.7);

 if result = 0 then
    result := 1;

end;
```

Quadro 23 – Valor atual da defesa a gol do jogador durante a partida

No Quadro 25, Quadro 28 e Quadro 29 são mostradas as interferências que os atributos anteriormente descritos influenciam perante as ações dos jogadores.

No chute, primeiramente o jogador deve decidir se deve ou não chutar. Isso é determinado através do método `deveChutar` (Quadro 24). Se `deveChutar` retornar `true`, o jogador chuta a bola.

```
function devechutar : boolean;
var faixa : integer;
    posAtu : Currency;
begin

  posAtu := self.posicaoAtual.y * multLadoCampo;

  {
    existe uma área de chute nos dois campos(aproximadamente 1/4 do campo de jogo),
    e quando o jogador entrar nesta área de chute é que ele vai poder chutar a bola ao gol.

    está área chute foi dividida em 10 partes iguais, a primeira parte é a
    faixa mais distante do gol, a decima é a mais proxima ao gol.

    quando o jogador entra na primeira faixa de chute ele tem 1 chance em 10
    no sorteio que vai determinar se vai chutar a bola ou não. Quando ele entra na
    segunda faixa ele tem 2 chances em 10, e assim sucessivamente.

  }

  // se o jogador esta dentro da área do chute
  if (posAtu > sim.getCampo.getDistChuteAoGol) then
     begin

       // verifica em qual faixa ele esta, a faixa estará na variável
       for faixa := 1 to 10 do
         begin
           if sim.getCampo.getDistChuteAoGol + (sim.getCampo.getDistChuteAoGolDiv10 * faixa) > posAtu then
              break;
         end;

       // sorteia um número
       setNumAleatorio;

       // chuta se a faixa for maior que o numero sorteado
       result := faixa >= numAleatorio+1;

     end
  else
     result := false; // não chuta

end;
```

Quadro 24 – Método `deveChutar` da classe `TJogadorPartida`

No método `chuta` (Quadro 25), inicialmente e realizado o sorteio para determinar se o chute vai ou não em direção ao gol. É sorteado um número de zero (0) a nove (9). Se este número for maior ou igual ao chute atual do jogador, a bola vai para fora do gol. Quanto maior o atributo chute do jogador, maior é a chance da bola ir para o gol.

Após é feito mais um sorteio para saber-se o destino certo da bola (se ela irá para o canto do gol ou para o meio). No caso da bola ir para fora também será feito este sorteio para saber a direção da mesma. Estes sorteios podem ser vistos no Quadro 25.

```
function TJogadorPartida.chuta : boolean;
var destinoBola : TLinCol;
begin

  // verifica se deve chutar
  result := devechutar;

  if result then
     begin

       // o Y de destino da bola sempre vai ser a linha de fundo
       destinoBola.y := sim.getCampo.getYMinGol * (-1);
       destinoBola.y := (destinoBola.y * multLadoCampo) - (sim.GetCampo.getTamanhoBolaDiv2 * multLadoCampo);

       // é sorteado um número para saber se a bola vai em direção a baliza (X)
       setNumAleatorio;

       // se o chute atual for maior/igual que o numero sorteado a bola vai para o gol.
       // então quanto maior o chute mais chance a bola tem de ir para o gol
       if getChuteAtual >= numAleatorio then
          begin

            // a bola vai em direção ao gol, mas no canto ?  no meio ? onde ?
            // então é sorteado mais um número para definir onde vai a bola
            setNumAleatorio;

            case numAleatorio of
              // 0 vai no canto extremo esquerdo (de quem vizualiza a partida)
              0 : destinoBola.x := sim.GetCampo.getXMinGol + sim.GetCampo.getTamanhoBolaDiv2;

              // 1,2,3 vai no canto esquerdo(de quem vizualiza a partida), entre o meio do gol e a trave
              1,2,3 : destinoBola.x := sim.GetCampo.getXMinGol / 2;

              // 4,5 vai no centro do gol
              4,5 : destinoBola.x := 0;

              // 1,2,3 vai no canto direito(de quem vizualiza a partida), entre o meio do gol e a trave
              6,7,8 : destinoBola.x := sim.GetCampo.getXMaxGol / 2;

              // 9 vai no canto extremo direito (de quem vizualiza a partida)
              9 : destinoBola.x := sim.GetCampo.getXMaxGol - sim.GetCampo.getTamanhoBolaDiv2;
            end;
          end
       else
          begin

            // a bola vai para fora do gol, mas onde ?
            // raspando a trave ? ou bem longe da trave.
            setNumAleatorio;

            case numAleatorio of
              // raspa a trave esquerda (de quem vizualiza a partida)
              0 : destinoBola.x := sim.GetCampo.getXMinGol - sim.GetCampo.getTamanhoBolaDiv2;

              // sai ao lado da trave esquerda (de quem vizualiza a partida)
              1 : destinoBola.x := sim.GetCampo.getXMinGol * 1.2;
              2 : destinoBola.x := sim.GetCampo.getXMinGol * 1.5;
              3 : destinoBola.x := sim.GetCampo.getXMinGol * 1.7;
              4 : destinoBola.x := sim.GetCampo.getXMinGol * 2.0;

              // raspa a trave direita (de quem vizualiza a partida)
              5 : destinoBola.x := sim.GetCampo.getXMaxGol + sim.GetCampo.getTamanhoBolaDiv2;

              // sai ao lado da trave direita (de quem vizualiza a partida)
              6 : destinoBola.x := sim.GetCampo.getXMaxGol * 1.2;
              7 : destinoBola.x := sim.GetCampo.getXMaxGol * 1.5;
              8 : destinoBola.x := sim.GetCampo.getXMaxGol * 1.7;
              9 : destinoBola.x := sim.GetCampo.getXMaxGol * 2.0;
            end;

          end;

        // novo sorteio para definir a força do chute
        setNumAleatorio;

        // manda a bola para a posicao destino
        sim.getBola.novaPosicao(destinoBola,numAleatorio);

        // o jogador atual nao esta mais com a bola
        sim.setJogadorComBola(nil);

     end;

end;
```

Quadro 25 – Método `chuta` da classe `TJogadorPartida`

No método `passa`, primeiramente é definido se o jogador deve passar a bola (Quadro 26). Após o jogador escolhe qual o jogador que vai receber a bola (Quadro 27) e efetua o passe (Quadro 28).

Toda vez que o jogador recebe a bola (método `setJogadorComABola`) é feito um sorteio para saber o tempo mínimo (`segundosComABola`) que o jogador vai ficar com a mesma. O jogador que esta com a bola, pode passa-la após este tempo. Se for o goleiro que está com a bola ou for uma cobrança de lateral, esta função sempre retorna `true`, desconsiderando os segundos de domínio de bola.

```
function podePassar : boolean;
begin
  // se for o goleiro que esta com a bola ou for um arremesso lateral sempre
  // será feito o passe, independente do tempo que o jogador com a bola
  if goleiro or prLateral then
     result := true
  else
     // o jogador só vai passar a bola após o jogador ficar "segundosComABola" com a bola no pé
     // se isto não fosse feito o jogador logo iria passa a bola
     result := ((GetTickCount - tempoDominioBola) / 1000) > segundosComABola;
end;
```

Quadro 26 – Método `devePassar` da classe `TJogadorPartida`

O jogador com a bola procura outro para passar. A bola é passada para o jogador que está dentro da área de passe mais próximo (Quadro 27). Se este jogador estiver mais recuado que ele, é feito um sorteio para saber se este jogador deverá ser escolhido. É sorteado um número, e se este número for maior que sete (7) este jogador é escolhido, caso não existir um jogador mais próximo.

```
// define qual é o jogador que vai receber a bola
function carregaJogPasse : boolean;
var I : integer;
    listaJogadoresEsc : TObjectList;
    jogAtu : TJogadorPartida;
begin

  // carrage a lista de jogadores escalados
  if equipe = e1 then
     listaJogadoresEsc := sim.getTaticaAtualE1.getTatica.getListaJogadoresEscalados
  else
     listaJogadoresEsc := sim.getTaticaAtualE2.getTatica.getListaJogadoresEscalados;

  for I := 0 to listaJogadoresEsc.Count-1 do
    begin

      jogAtu := TJogadorEscalado(listaJogadoresEsc.Items[I]).getJogadorPartida;

      // não vai passar para ele mesmo, e nem para o goleiro.
      // e so vai passar pas os jogadores que estão dentro da distância do passe
      if (self.getId <> jogAtu.getId) and
         (not jogAtu.getGoleiro) and
         (goleiro or prLateral or (sim.getCampo.calculaDiscancia(self.posicaoAtual,jogAtu.posicaoAtual) <= sim.getCampo.getDistPasse)) then
        begin

          if prLateral then
             jogPasse := jogAtu
          else
             begin
               // verifica se o jogador esta mais perto do que o jogador escolhido até o momento
               if (jogPasse = nil) or
                  (sim.getCampo.calculaDiscancia(jogPasse.getPosicaoAtual,self.posicaoAtual) >
                   sim.getCampo.calculaDiscancia(jogAtu.getPosicaoAtual,self.posicaoAtual)) then
                  begin
                    // só passa para tras se o numero aleatorio for 7
                    setNumAleatorio;
                    if (numAleatorio > 7) or ((jogAtu.getPosicaoBaseY * multLadoCampo) >= (self.getPosicaoBaseY * multLadoCampo)) then
                       jogPasse := jogAtu;
                  end;
             end;

        end;

    end;

  result := jogPasse <> nil;
```

Quadro 27 – Método `carregaJogadorPasse` da classe `TJogadorPartida`

Com o jogador que vai receber a bola já escolhido, o jogador com a bola efetua o passe. No método `passa` (Quadro 28) primeiramente é feito o sorteio para determinar se o passe vai ou não em direção ao jogador destino. É sorteado um número de zero (0) a nove (9). Se este número for maior ao passe atual do jogador, a bola vai na direção certa. Quanto maior o atributo passe do jogador, maior é a chance da bola ir para a direção correta.

Se o passe não for executado da maneira correta, é feito mais um sorteio para determinar o destino da bola (Quadro 28).

```
result := false;
jogPasse := nil;

if podePassar and carregaJogPasse then // vai passar  para o jogador jogPasse
   begin

     setNumAleatorio;

     // se o atributo passe for maior que o numero aleatório, então passa certo
     // pontanto, quanto maior o atributo passe, maior é a chance de acertar o destino
     if getPasseAtual > numAleatorio then
        begin
          destinoBola := jogPasse.posicaoAtual;
        end
     else
        begin

          // se o numero sorteado for igual ao atributo passe, então,
          // erro o alvo pela distância do corpo de um jogador
          if numAleatorio = getPasseAtual then
             begin
               destinoBola.x := jogPasse.posicaoAtual.x + sim.GetCampo.getTamJogGrid;
               destinoBola.y := jogPasse.posicaoAtual.y + sim.GetCampo.getTamJogGrid;
             end
          else
          // se o numero sorteado - passe for igual a 1, então,
          // erro o alvo pela distância do corpo de um jogador,
          // mas para o lado oposto da opção anterior
          if numAleatorio - getPasseAtual = 1 then
             begin
               destinoBola.x := jogPasse.posicaoAtual.x - sim.GetCampo.getTamJogGrid;
               destinoBola.y := jogPasse.posicaoAtual.y - sim.GetCampo.getTamJogGrid;
             end
          else
          // se o numero sorteado - passe for igual a 2, então,
          // erro o alvo pela distância de dois corpos
          if numAleatorio - getPasseAtual = 2 then
             begin
               destinoBola.x := jogPasse.posicaoAtual.x + sim.GetCampo.getTamJogGrid * 2;
               destinoBola.y := jogPasse.posicaoAtual.y + sim.GetCampo.getTamJogGrid * 2;
             end
          else
          // se o numero sorteado - passe for igual a 3, então,
          // erro o alvo pela distância de dois corpos
          // mas para o lado oposto da opção anterior
          if numAleatorio - getPasseAtual = 3 then
             begin
               destinoBola.x := jogPasse.posicaoAtual.x - sim.GetCampo.getTamJogGrid * 2;
               destinoBola.y := jogPasse.posicaoAtual.y - sim.GetCampo.getTamJogGrid * 2;
             end
          else
          // se o numero sorteado - passe for igual a 4, então,
          // erro o alvo pela distância de quatro corpos
          if numAleatorio - getPasseAtual = 4 then
             begin
               destinoBola.x := jogPasse.posicaoAtual.x + sim.GetCampo.getTamJogGrid * 4;
               destinoBola.y := jogPasse.posicaoAtual.y + sim.GetCampo.getTamJogGrid * 4;
             end
          else
          // se o numero sorteado - passe for igual a 5, então,
          // erro o alvo pela distância de quatro corpos
          // mas para o lado oposto da opção anterior
          if numAleatorio - getPasseAtual = 5 then
             begin
               destinoBola.x := jogPasse.posicaoAtual.x - sim.GetCampo.getTamJogGrid * 4;
               destinoBola.y := jogPasse.posicaoAtual.y - sim.GetCampo.getTamJogGrid * 4;
             end
          else
             begin
               // se o numero sorteado for par, então,
               // erro o alvo pela distância de seis corpos
               if numAleatorio mod 2 = 0 then
                  begin
                    destinoBola.x := jogPasse.posicaoAtual.x - sim.GetCampo.getTamJogGrid * 6;
                    destinoBola.y := jogPasse.posicaoAtual.y - sim.GetCampo.getTamJogGrid * 6;
                  end
               else
                  // se o numero sorteado for impar, então,
                  // erro o alvo pela distância de seis corpos
                  // mas para o lado oposto da opção anterior
                  begin
                    destinoBola.x := jogPasse.posicaoAtual.x - sim.GetCampo.getTamJogGrid * 6;
                    destinoBola.y := jogPasse.posicaoAtual.y - sim.GetCampo.getTamJogGrid * 6;
                  end
             end;

        end;

     // sorteia um novo numero
     setNumAleatorio;

     // passa a bola
     sim.getBola.novaPosicao(destinoBola,numAleatorio);

     // o jogador atual nao tem mais a bola

     sim.setJogadorComBola(nil);

     result := true;

   end;
```

Quadro 28 – Método `passa` do `TJogadorPartida`

No método `desarma` (Quadro 29) é confrontado desarme atual do marcador com o controle de bola atual do jogador com a posse de bola. É feita uma regra de três com estes dois elementos para que no sorteio seja decidido qual jogador terá sucesso. Se o desarme atual do marcador e maior que o controle do adversário, ele terá mais chances. Se o controle for superior ao desarme, então o jogador que está com a bola terá mais chance de êxito.

Se o desarme ocorre, o árbitro decidirá se a bola foi ou não roubada com falta. A forma como o árbitro decidirá é apresentada no Quadro 40.

O jogador que perder a disputa sofrerá uma pausa de seiscentos (600) milisegundos.

```
procedure TJogadorPartida.desarma;
var des : byte;
    jogComBola : TJogadorPartida;
begin

  // carrega o jogador aom a bola
  jogComBola := TJogadorPartida(sim.getJogadorComBola);
  if jogComBola = nil then
     exit;

  // regra de 3 para saber das 10 possibilidades quem tem mais chance se sucesso, que esta com a bola ou que quer desarmar
  // Exemplo: desarme 4 controle 1
  // 4 * 10 / (4 + 1) = 8
  // neste caso em 10 possibilidades temos 8 chances de desarme contra 2 de o jogador se manter com a bola

  des := floor(getDesarmeAtual * 10 / (getDesarmeAtual + jogComBola.getContrBolaAtual));

  if des = 0 then // mesmo com o desarme 0 deve haver a possibilidade de desarme
     des := 1
  else
  if des = 10 then // mesmo com o desarme 10 deve haver a possibilidade do jogador ficar com a bola
     des := 9;

  setNumAleatorio;

  // o desarme acontece se o numero sorteado for menor que o desarme
  // entao quanto maior o desarme mais chances do desarme ser efetuado
  // se o controle de bola do jogador for alto é menos provável que seja feito o desarme
  if numAleatorio <= des then
     begin
       // mesmo sendo feito o desarme o arbitro vai decidir se houve falta no lance
       if not sim.getArbitro.falta(self,TJogadorPartida(sim.getJogadorComBola)) then
          begin

            // se não houve falta o jogador que desarmou fica com a bola
            sim.setJogadorComBola(self);

            jogComBola.perdeuBola;

          end;
     end
  else
     begin
       // é dada uma pausa no marcador se ele não conseguir roubar a bola
       self.perdeuBola;
     end;

end;
```

Quadro 29 – Método `desarma` da classe `TJogadorPartida`

O jogador não pode sobrepor outro em campo. Para fazer este controle a classe `TCampo` possui um semáforo para cada ponto do campo.

Quando o jogador deseja trocar de posição em campo, ele executa o método `alteraPosicaoJogador` (Quadro 30) da classe `TCampo`. Neste método é convertida a posição atual e nova do tipo `TLinCol` para `TPosicao`. `TLinCol` só possui o ponto central do jogador. `TPosicao` possui os quatro pontos que o jogador poderá ocupar. Na Figura 28 é representada a conversão do ponto central (`TLinCol`) para os quatro pontos (`TPosicao`) p1, p2, p3 e p4.

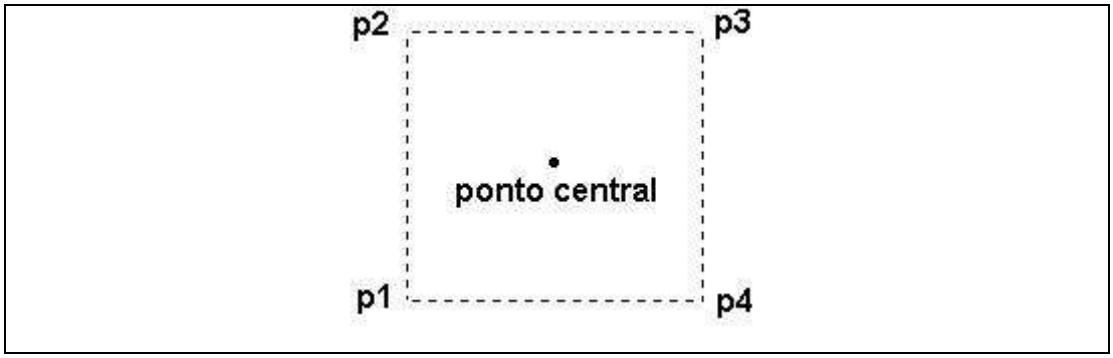


Figura 28 – Conversão de `TLinCol` para `TPosicao`

A execução do método `reservaPosicao` (Quadro 31 e Quadro 32) serve para verificar se a posição desejada está disponível. Este método percorre todos os pontos das arestas p1-p2, p2-p3, p3-p4 e p4-p1 para verificar se pelo menos um destes pontos está ocupado por outro jogador. Se não houver colisão retorna `coNenhuma`, caso contrário retorna o lado em que ocorreu.

```
begin

  jogador := TJogadorPartida(prJogador);

  // carrega a posicao que o jogador realmente esta ocupando em campo
  posAtu := converte_TColLin_TPosicao(jogador.getPosicaoAtual);

  // carrega a posicao que o jogador irá realmente ocupar em campo
  posNova := converte_TColLin_TPosicao(prPosNova);

  // verifica se a nova posicao esta livre
  result := reservaPosicao;

  // se o result for igual "coNenhuma" quer dizer que a nova posicao está disponível
  if result = coNenhuma then
     begin

       // a lista de pontos da posicao antiga é setada com a posicao atual
       listaPontosPosicaoAntiga := jogador.getPontosPosicaoAtual;

       // seta a nova lista de pontos
       jogador.setPontosPosicaoAtual(listaPontosPosicaoNova);

       // libera a lista de pontos antiga
       liberaPontosAltJog(listaPontosPosicaoAntiga,true);

       // limpa listaPontosPosicaoAntiga
       SetLength(listaPontosPosicaoAntiga,0);

     end;

end;
```

Quadro 30 – Método `alteraPosicaoJogador` da classe `TCampo`

```
function reservaPosicao : TColisao; // F
var x,y : Currency;
    proxPt : byte;
    direcaoPt : Currency;

begin

  // lista de pontos da nova posicao
  SetLength(listaPontosPosicaoNova,0);

  // lista de pontos requiridos da nova posicao
  SetLength(pontosRequiridos,0);

  // variável utilizada para ir de ponto a ponto
  direcaoPt := 0.001;

  // inicializa o result
  result := coNenhuma;

  // proximo ponto
  proxPt := 2;

  // posição nova é igual ao ponto1
  x := posNova.p1.x;
  y := posNova.p1.y + direcaoPt;

  // percorre cada ponto das 4 arestas do jogador
  while true do
    begin

      // garante que o ponto está dentro do campo
      if (x >= xMinCampo) and (x <= xMaxCampo) and
         (y >= yMinCampo) and (y <= yMaxCampo) then
        begin

          // inclui o ponto na lista de pontos da posicao nova
          SetLength(listaPontosPosicaoNova,length(listaPontosPosicaoNova)+1);
          listaPontosPosicaoNova[length(listaPontosPosicaoNova)-1].x := x;
          listaPontosPosicaoNova[length(listaPontosPosicaoNova)-1].y := y;

          if not posOcup(x,y) then // não requisita um ponto que o próprio jogador ja esteja ocupando
             begin
               // se o ponto ja esta com outro jogador
               if WAIT_TIMEOUT = WaitForSingleObject(getSemaforo(x,y),0) then
                  begin

                    // seta a colisao
                    case proxPt of
                      2 : result := coEsquerda;
                      3 : result := coCima;
                      4 : result := coDireita;
                      1 : result := coBaixo;
                    end;

                    // libera os pontos requiridos
                    liberaPontosAltJog(pontosRequiridos);

                    // sai da rotina
                    break;

                  end
               else
                  begin
                    // se o ponto esta livre ele é adicionado a lista de pontos requiridos
                    SetLength(pontosRequiridos,length(pontosRequiridos)+1);
                    pontosRequiridos[length(pontosRequiridos)-1].x := x;
                    pontosRequiridos[length(pontosRequiridos)-1].y := y;
                  end;
             end;

        end;
```

Quadro 31 – Método `reservaPosicao` da classe `TCampo` (Parte 1/2)

```
      case proxPt of
        2 : // aresta p1p2
          begin  // 1 para 2
            // se chegou no ponto p2
            if (y = posNova.p2.y) then
               begin
                 x := posNova.p2.x + direcaoPt;
                 y := posNova.p2.y;
                 inc(proxPt);
               end
            else
              y := y + direcaoPt;// senão, vai para o proximo ponto da aresta
          end;
        3 : // aresta p2p3
          begin
            // se chegou no ponto p3
            if (x = posNova.p3.x) then
               begin
                 x := posNova.p3.x;
                 y := posNova.p3.y - direcaoPt;
                 inc(proxPt);
               end
            else
               x := x + direcaoPt;// senão, vai para o proximo ponto da aresta
          end;
        4 : // aresta p3p4
          begin
            // se chegou no ponto p4
            if (y = posNova.p4.y) then
               begin
                 x := posNova.p4.x - direcaoPt;
                 y := posNova.p4.y;
                 proxPt := 1;
               end
            else
               y := y - direcaoPt; // senão, vai para o proximo ponto da aresta
          end;
        1 : // aresta p4p1
          begin  // 4 para 1
            // se chegou no ponto p1
            if (x = posNova.p1.x) then
               begin
                 break;
               end
            else
               x := x - direcaoPt; // senão, vai para o proximo ponto da aresta
          end;
      end;

    end;
end;
```

Quadro 32 - Método `reservaPosicao` da classe `TCampo` (Parte 2/2)

A bola (`TBola`) possui um objeto `TMovimento` e o jogador (`TJogadorPartida`) possui um objeto do tipo `TMovimentoJogador`. Ambos são responsáveis por fornecer a coordenada da próxima movimentação conforme o destino que é passado. O método `movimenta` (Quadro 33) é que retorna a nova coordenada. Dentro do método movimenta estão os métodos `deslocX` e `deslocY` (Quadro 34). Estes métodos determinam se a coordenada X ou Y deve ser alterada na iteração atual. Esta decisão é tomada com base na distância do X origem para

X destino e Y origem para Y destino e nas últimas movimentações do objeto. Um exemplo da proporcionalidade de deslocamento pode ser visto no Quadro 35.

```
function TMovimento.movimenta(const prPosAtual : TLinCol; const prPosDest : TLinCol;
             const prUltColisao : TColisao) : TLinCol;
var wDifX,
    wDifY : Currency;
begin
  // verifica a diferencia do x atual para o x destino
  wDifX := distPontos(prPosDest.x,prPosAtual.x);
  // verifica a diferencia do y atual para o y destino
  wDifY := distPontos(prPosDest.y,prPosAtual.y);

  // verifica se deve ir para a direita (aumentar o X)
  if ((prPosAtual.x < prPosDest.x) and (prUltColisao <> coDireita)) or (prUltColisao = coEsquerda) then
     begin
       // verifica de deve-se alterar o valor de x na iteração atual
       result.x := prPosAtual.x + deslocX;

       // verifica se não foi ultrapassado o X destino com o deslocamento feito
       if (result.x > prPosDest.x) and (result.x - prPosDest.x <= desloc) then
          result.x := prPosDest.x;

     end
  else
  // verifica se deve ir para a esquerda (diminuir o X)
  if (prPosAtual.x > prPosDest.x) or (prUltColisao = coDireita) then
     begin
       // verifica de deve-se alterar o valor de x na iteração atual
       result.x := prPosAtual.x - deslocX;

       // verifica se não foi ultrapassado o X destino com o deslocamento feito
       if (result.x < prPosDest.x) and (prPosDest.x - result.x <= desloc) then
          result.x := prPosDest.x;

     end
  else
  // ja esta no X destino
     result.x := prPosDest.x;

  // verifica se deve ir para cima (aumentar o Y)
  if ((prPosAtual.y < prPosDest.y) and (prUltColisao <> coCima)) or (prUltColisao = coBaixo) then
     begin
       // verifica de deve-se alterar o valor de y na iteração atual
       result.y := prPosAtual.y + deslocY;

       // verifica se não foi ultrapassado o Y destino com o deslocamento feito
       if (result.y > prPosDest.y) and (result.y - prPosDest.y <= desloc) then
          result.y := prPosDest.y;

     end
  else
  // verifica se deve ir para baixo (diminuir o Y)
  if (prPosAtual.y > prPosDest.y) or (prUltColisao = coCima) then
     begin
       // verifica de deve-se alterar o valor de y na iteração atual
       result.y := prPosAtual.y - deslocY;

       // verifica se não foi ultrapassado o Y destino com o deslocamento feito
       if (result.y < prPosDest.y) and (prPosDest.y - result.y <= desloc) then
          result.y := prPosDest.y;

     end
  else
  // ja esta no Y destino
     result.y := prPosDest.y;
end;
```

Quadro 33 – Método `movimenta` da classe `TMovimento`

```
function deslocX : Currency; // verifica se o X deve ser alterado na iteração atual
var I : integer;
begin
  // se a difX for menor que a difY altera-se o X
  // agora se o X for maior que o Y, deve-se verificar as ultimas movimentações
  if wDifX / wDifY < 1 then
     begin
       // vai fazer o laço o número de vezes que o Y deve ser alterado para um X ser alterado
       for I := 1 to floor(wDifY / wDifX) do
         begin
           // se um dos valores de x na lista for "posIniPadrao" então não
           // deve-se alterar x nesta iteração.
           if listaMov[I].x = posIniPadrao then
              begin
                  break;
                result := 0;
                exit;
              end
           else
           // se houve uma alteração de X então não deve-se alterar x nesta iteração
           if prPosAtual.x <> listaMov[I].x then
              begin
                result := 0;
                exit;
              end;
         end;
     end;

  result := desloc;
end;

function deslocY : Currency; // verifica se o Y deve ser alterado na iteração atual
var I : integer;
begin
  // se a difY for menor que a difX altera-se o Y
  // agora se o Y for maior que o X, deve-se verificar as ultimas movimentações
  if wDifY / wDifX < 1 then
     begin
       // vai fazer o laço o número de vezes que o X deve ser alterado para um Y ser alterado
       for I := 1 to floor(wDifX / wDifY) do
         begin
           // se um dos valores de Y na lista for "posIniPadrao" então não
           // deve-se alterar Y nesta iteração.
           if listaMov[I].y = posIniPadrao then
              begin
                result := 0;
                exit;
              end
           else
           // se houve uma alteração de Y então não deve-se alterar x nesta iteração
           if prPosAtual.y <> listaMov[I].y then
              begin
                result := 0;
                exit;
              end;
         end;
     end;

  result := desloc;
end;
```

Quadro 34 – Métodos `deslocX` e `deslocY` da classe `TMovimento`

```
Ex: ptOrigem.X = 0  ptDestino.X = 30
    ptOrigem.Y = 20  ptDestino.Y = 35

    diferença no eixo X = 30
    diferença no eixo Y = 15

Para fazer o objeto chegar no XDestino ao mesmo tempo do YDestino ele deve se
locomover na seguinte proporção. A cada 2 movimentações no eixo X, 1 no
eixo Y.
```

Quadro 35 – Exemplo de proporção de deslocamento

Quando o jogador está com a bola é o método `novaPosicaoJogadorComBola` (Quadro

36) da classe `TMovimentoJogador` que decide a coordenada final do movimento através de sorteio, e também decide quantas iterações este movimento vai durar. Depois é usado o método movimenta para locomover o jogador.

```
function TMovimentoJogador.novaPosicaoJogadorComBola : TLinCol;
var jog : TJogadorPartida;
begin

  jog := TJogadorPartida(jogador);

  // duracaoDirecaoMov é o numero de iterções do movimento "direcaoMov".
  // se a "duracaoDirecaoMov" dor igual a -1 então sorteia-se uma nova direção
  if (duracaoDirecaoMov = -1) then
     begin
       // nova direção
       direcaoMov := jog.getSim.getNumAleatorio;

       // se a direção for maior que 7 significa que é para tras ou para o lado.
       // O sorteioé feino novamente para que seja mais difícil disto acontecer
       if direcaoMov > 7 then
          direcaoMov := jog.getSim.getNumAleatorio;
     end;

  case direcaoMov of
      0,1,2: begin
           // 0,1 e 2 significa que o jogador vai se deslocar em direrão a área
           // adversária, com a sua coluna(X) da posição base como referencia
           result.x := jog.getPosicaoBase.x;
           result.y := jog.getSim.getCampo.getYMaxCampo * jog.multLadoCampo;
         end;
      3,4  : begin
           // 3 e 4 significa que o jogador vai se deslocar em direrão a área
           // adversária no centro de campo (x = 0)
           result.x := 0;
           result.y := jog.getSim.getCampo.getYMaxCampo * jog.multLadoCampo;
         end;
      5,6,7: begin
           // 5,6 e 7 significa que o jogador vai se deslocar em direrão a área
           // adversária com a sua coluna(X) da posição base invertida como referencia
           result.x := jog.getPosicaoBase.x * (-1);
           result.y := jog.getSim.getCampo.getYMaxCampo * jog.multLadoCampo;
         end;
      8  : begin
           // 8 quer dizer que o jogador vai para um dos lados (não muda o Y)
           result.y := jog.getPosicaoAtualY;

           // sorteia-se novamente mais um número para saber qual dos lados ele vai
           lado := jog.getSim.getNumAleatorio;

           // se o numero do lado for par dele incrementa o X
           if lado mod 2 = 0 then
              result.x := jog.getSim.getCampo.getXMaxCampo
           else
              result.x := jog.getSim.getCampo.getXMinCampo;
         end;
      9  : begin
           // 9 significa que ele vai para tràs, em direção ao seu goleiro
           result.y := jog.getSim.getCampo.getYMinCampo * jog.multLadoCampo;

           // sorteia-se novamente mais um número para saber qual dos lados ele vai (eixo X)
           // o lado da sua posioção base, ou para o outro lado

           lado := jog.getSim.getNumAleatorio;

           // se o numero do lado for par dele vai para o lado da sua posicao base
           if lado mod 2 = 0 then
              result.x := jog.getPosicaoBase.x
           else
              result.x := jog.getPosicaoBase.x * (-1);
         end;
    end;

  // se a duração do movimento for igual a -1 sorteia-se uma nova duração de tempo do movimento,
  // senão decrementa-se
  if (duracaoDirecaoMov = -1) then
     duracaoDirecaoMov := jog.getSim.getNumAleatorio * 2
  else
     dec(duracaoDirecaoMov);

end;
```

Quadro 36 – Método `novaPosicaoJogadorComBola` da classe `TMovimentoJogador`

A bola está implementada na classe `TBola`, no método `execute` (Quadro 37) se encontra o laço principal da classe. É neste laço que é verificado se a bola está em movimento, e se estiver faz a mesma se locomover. O jogador quando chuta ou passa a bola, ele coloca a bola em movimento através do método `novaPosicao` (Quadro 38)

```
procedure TBola.Execute;
var proxPos : TLinCol;
begin

  inherited;

  // se fim da partida sai do Execute da bola
  while TSimulador(sim).getEstadoPartida <> epFinal do
    begin

      if TSimulador(sim).getEstadoPartida <> epEmAndamento then
         begin
           // se a partida não estiver em andamento da uma pausa maior na bola
           sleep(100)
         end
      else
      if not emMovimento then
         begin
           // também é dada uma pausa maior na bola se a partida está em andamento
           // mas a bola não estiver em movimento
           sleep(50);
         end
      else
         begin

           // incrementa o número de iterações
           inc(iteracoes);

           // a cada multiplo de 5 é diminuída a força da bola
           if iteracoes mod 5 = 0 then
              diminuiForca;

           // delay na bola, quanto maior a força menor vai ser esta pausa
           sleep(15);// - (forcaAtual));

           // movimentaBola retornará qual a próxima posição da bola
           proxPos := movimentoBola.movimenta(self.posicaoAtual,destinoAtual,coNenhuma);

           // se a bola chegou no destino final ela é tirada do modo de movimento
           if (proxPos.x = destinoAtual.x) and (proxPos.y = destinoAtual.y) then
              begin
                paraBola;
              end;

           // atribui nova posicao
           posicaoAtual := proxPos;

         end;

    end;

end;
```

Quadro 37 – Método `execute` da classe `TBola`

```
procedure TBola.novaPosicao(const prDestino: TLinCol;
  const prForca: byte);
begin

  if (prForca > 0) and (prForca < 11) then
      begin

       // novo destino da bola
       destinoAtual := prDestino;

       // forca do chute ou passe
       forcaInicial := prForca;
       forcaAtual := prForca;

       // zera o numero de iterações
       iteracoes := 0;

       // cria um novo objeto TMovimento para "gerenciar" o deslocamento da bola
       pFreeAndNil(movimentoBola);
       movimentoBola := TMovimento.Create(TSimulador(sim).getCampo.getDeslocBola);

       // coloca a bola em movimento
       emMovimento := true;

      end;

end;
```

Quadro 38 – Método `novaPosicao` da classe `TBola`

O árbitro está implementado na classe `TArbitro` que é uma *thread*. O método `execute` (Quadro 39) do árbitro contém o laço em que é verificado constantemente possíveis situações da partida, tais como:

a) intervalo da partida;
b) final da partida;
c) bola pela linha de fundo;
d) bola dentro do gol;
e) bola pela linha lateral.

```
procedure TArbitro.Execute;
var s : TSimulador;
begin
  inherited;

  s := TSimulador(sim);

  // enquanto a partida nao acabar
  while s.getEstadoPartida <> epFinal do
    begin

      if s.getEstadoPartida <> epEmAndamento then
         sleep(100)
      else
         begin

           sleep(500);

           // verifica se é para fazer o intervalo
           if (s.getTempoPlacar.getMinutos >= 45) and (s.getTempoPlacar.getTempo = 1) then
              intervalo
           else
           // final da partida
           if (s.getTempoPlacar.getMinutos >= 90) then
              alteraEstadoPartida(epFinal);
           else
           // verifica se a bola saiu pela linha de fundo
           if bolaPelalinhaDeFundo then
              begin
                // verifica se foi gol
                if bolaDentroDoGol then
                   assinalaGol // se foi marca o gol
                else
                   tiroDeMeta; // senão o goleiro deve cobrar o tiro de meta
              end
           else
           // verifica se a bola saiu pela linha alteral
           if bolaPelalinhalateral then
              begin
                // se saiu determina a cobrança de lateral
                arremessoLateral;
              end;

         end;

    end;

end;
```

Quadro 39 – Método `execute` da classe `TArbitro`

O método `falta` (Quadro 40) é executado toda a vez em que é feito um desarme por um jogador. Se o mesmo retornar `true`, significa que houve falta.

```
function TArbitro.falta(const prDes, prContr: TJogador): boolean;
var jogDes,
    jogContr : TJogadorPartida;
begin

  jogContr := TJogadorPartida(prContr); // jogador com a bola
  result := false;

  if jogContr <> nil then
     begin

       jogDes := TJogadorPartida(prDes); // jogador que esta desarmando (marcador)

       // se o desarmeAtual do marcador for maior que o Controle do que está com a bola, então
       // o desarme do marcador é que irá determinar de foi ou não falta, senão
       // o controle do jogador com a bola é que irá determinar se foi ou não falta.

       if jogDes.getDesarmeAtual > jogContr.getContrBolaAtual then
          begin
            { sorteia um número para determinar se foi falta,
              se este numero for maior que o desarme do marcador, então é falta
              quanto mais alto o desarme mais difícil que seja falta }
            result := TSimulador(sim).getNumAleatorio > jogDes.getDesarmeAtual
          end
       else
          begin
            { sorteia um número para determinar se foi falta,
              se este numero for menor que o controle do jog. com a bola, então é falta
              quanto mais alto o controle mais fácil que seja falta }
            result := TSimulador(sim).getNumAleatorio < jogContr.getContrBolaAtual;
          end;

       // se result = true então é falta
       if result then
          begin
            // altera o estado da partida para falta
            alteraEstadoPartida(epFalta);
            // diz ao simulador que ocorreu uma falta em cima do jogador jogContr
            TSimulador(sim).cobrancaFalta(jogContr);
          end;
     end;

end;
```

Quadro 40 – Método `falta` da classe `TArbitro`

3.4.3 Operacionalidade da implementação

O software possui dois módulos: o módulo de cadastro e o de simulação. No módulo de cadastro é feita a criação das equipes, jogadores, estratégias e táticas. No módulo de simulação o usuário define o confronto e as estratégias. Neste módulo é feita a visualização da partida em 2D.

Na tela inicial do simulador (Figura 29) o usuário determina o módulo que deseja explorar.

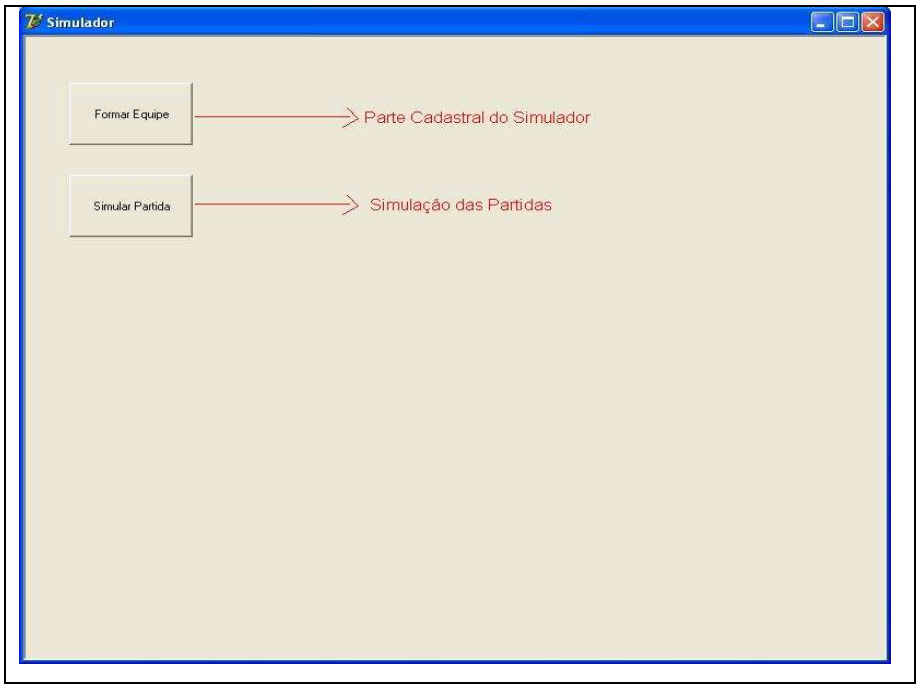


Figura 29 – Tela inicial do simulador

Na parte cadastral do simulador tem-se a tela de cadastro das equipes (Figura 30). Neste local o usuário poderá criar a sua equipe. Esta tela possui três (3) campos a serem informados:

a) `Apelido Equipe`: sigla ou um nome curto que identifique a equipe;

b) `Nome Equipe`: nome completo da equipe;

c) `Nome Técnico`: nome do técnico da equipe.

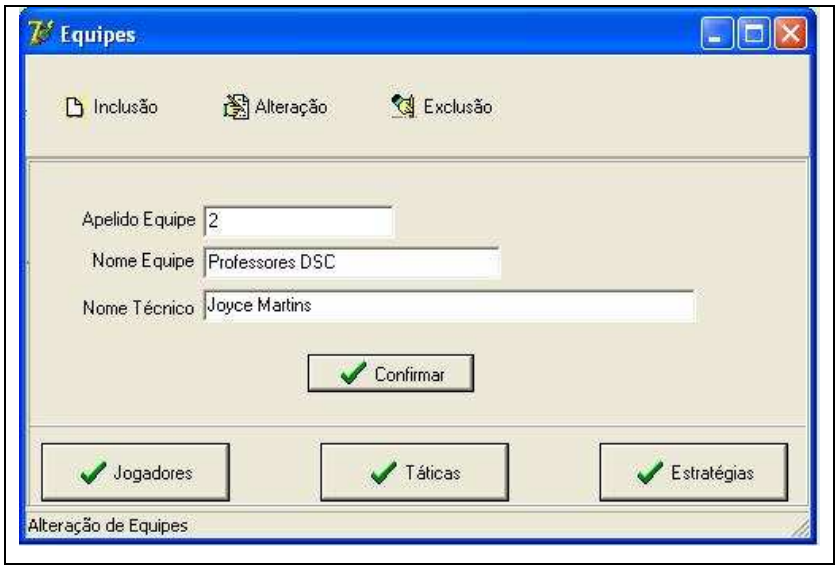


Figura 30 - Cadastro de Equipes

Observa-se que na Figura 30 existe três (3) botões na parte inferior, `Jogadores`, `Táticas` e `Estratégias`.

O botão `Jogadores` acessa a tela na qual o usuário poderá criar os jogadores da equipe (Figura 31). Os atributos são informados pelo usuário. Cada atributo vai de zero (0) a dez (10) e a soma de todos os atributos do jogador não pode passar de trinta (30). No cadastro de jogadores (Figura 31) tem-se os seguintes campos:

a) `ID Jogador`: código do jogador;
b) `Nome`: nome do Jogador;
c) `Nro Camisa`: numero da camisa do jogador;
d) `Velocidade`: índice de velocidade do jogador;
e) `Resistência`: índice de resistência do jogador;
f) `Passe`: índice de precisão dos passes do jogador;
g) `Controle de bola`: índice de controle de bola do jogador;
h) `Desarme`: índice de eficiência de marcação do jogador;
i) `Defesa a Gol`: índice de eficiência nas defesas dos chutes (goleiro).

Figura 31 – Cadastro de jogadores

Com os jogadores já cadastrados, o usuário poderá criar as táticas da equipe. Esta operação é acessada através do botão tática no cadastro de equipes (Figura 30). Na tática é escolhido os jogadores que farão parte da mesma. Juntamente com a escolha dos jogadores é definida a posição base de cada um dentro de campo. A posição é escolhida como se o campo

fosse um plano cartesiano. A coordenada “y” pode variar de -700 a + 700 e a coordenada “x” de -470 a +470 (Figura 32). A representação do centro do campo fica nas coordenadas “x” e “y” igual a zero (0). A coordenada “x” com valor negativo indica que o posicionamento do jogador será na esquerda do campo e positivo na direita. A coordenada “y’ com valor negativo indica que o posicionamento do jogador será no campo de ataque e negativo no campo de defesa.

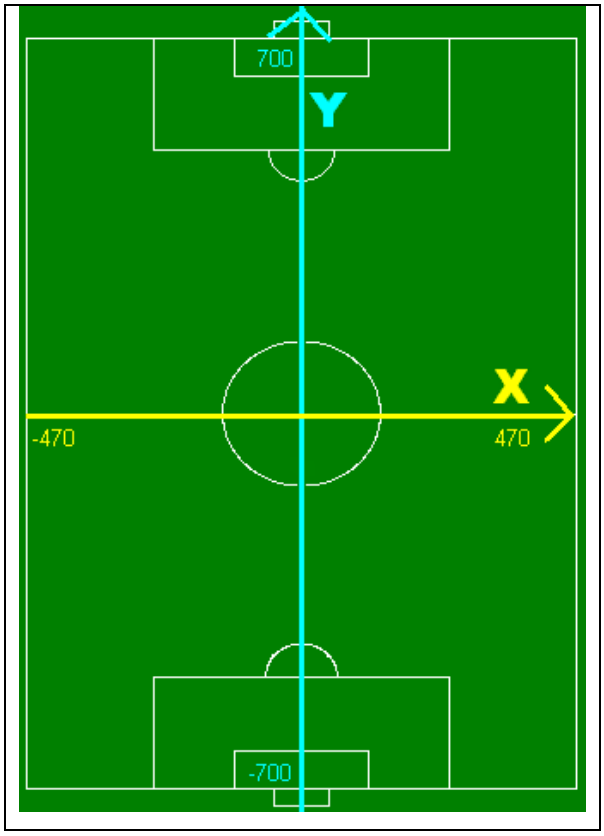


Figura 32 - Posições no campo

A tela do cadastro de tática (Figura 33) possui os seguintes campos:

a) `Id Tática`: código da tática;

b) `Jogadores`: lista com todos os jogadores do plantel;

c) `Posição`: coordenada cartesiana do jogador selecionado no momento. Se o jogador é o goleiro, um assinalamento no `checkbox Goleiro` deve ser feito.

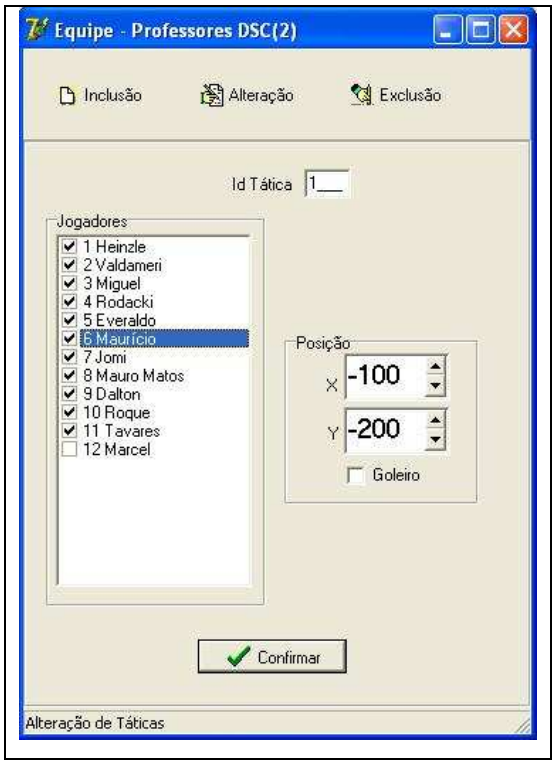


Figura 33 – Cadastro de táticas

O botão `Estratégia` no cadastro de equipe (Figura 30) tem como propósito acessar a tela de cadastro de estratégias (Figura 34). Uma estratégia contém uma ou mais táticas para a partida. O usuário pode prever uma situação de derrota parcial, por exemplo, e colocar o time mais ofensivo no segundo tempo.

Os campos do cadastro de estratégias são (Figura 34):

a) `Id Estratégia`: código da estratégia;

b) `Lista de Táticas`: lista de táticas da estratégia, mostrando o minuto que ela deve ser ativada;

c) `Dados Tática`: inclui, altera, exclui (conformes os botões no centro da tela) uma tática da lista de táticas da estratégia.

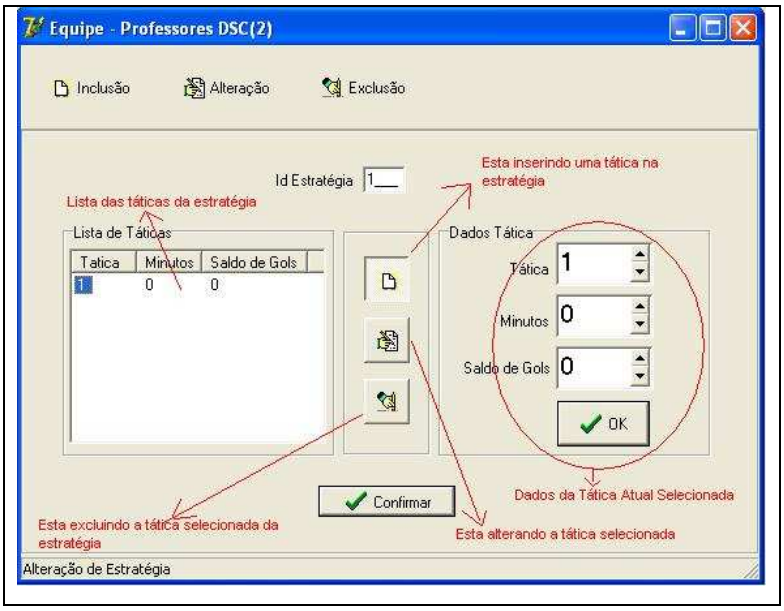


Figura 34 – Cadastro de estratégias

No módulo de simulação encontra-se a tela de definição da partida (Figura 35). Nesta tela o usuário seleciona as duas equipes que irão disputar a partida, juntamente com a estratégia que cada uma delas vai usar.

Figura 35 – Tela de definição da partida

Após a escolha das equipes e das estratégias utiliza-se o botão `Jogar` para visualizar a simulação (Figura 36).

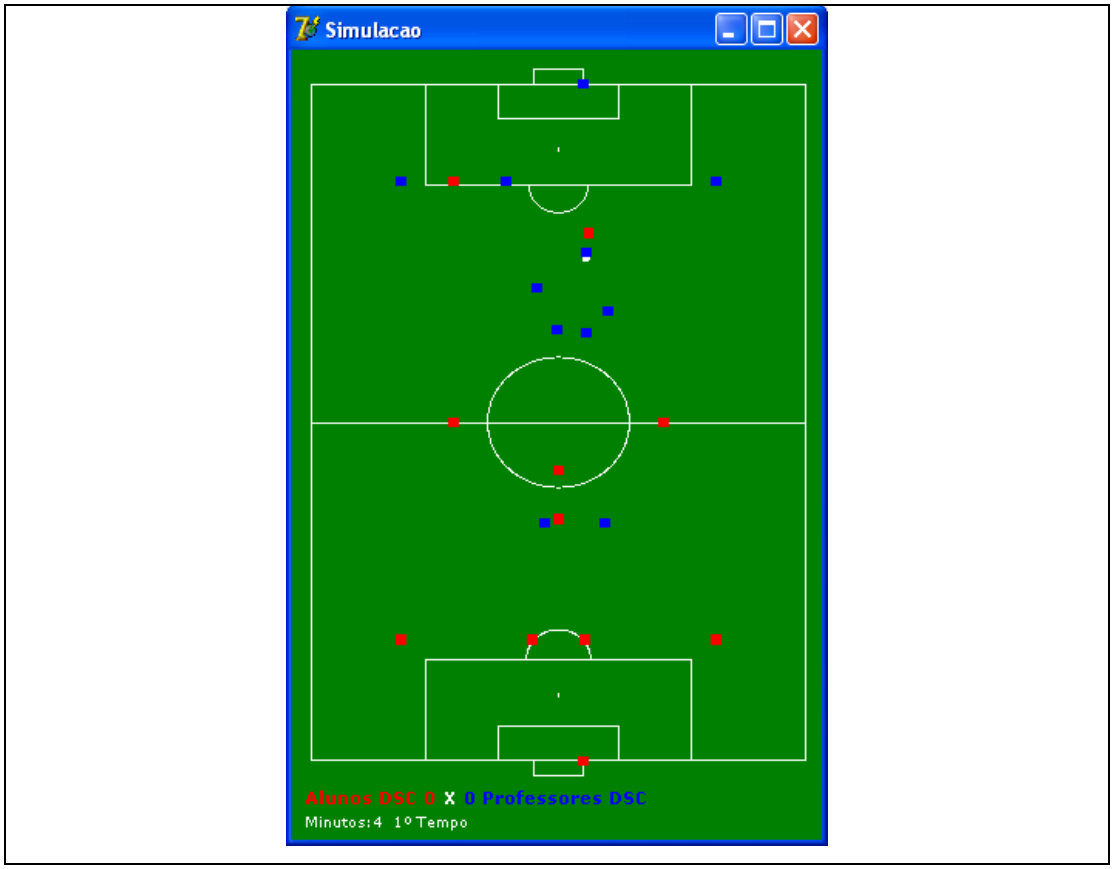


Figura 36 – Tela de visualização da simulação

3.5 RESULTADOS E DISCUSSÃO

No Quadro 41 tem-se uma comparação entre os simuladores estudados e o simulador desenvolvido.

	Managerzone	**Hattrick**	**Gamegol**	**Simulador Desenvolvido**
Aplicação	Web	Web	Web	Desktop
Operacionalidade	Simples	Complexa	Simples	Muito Simples
Nível aproximação com as atribuições de um clube no mundo real (gerenciamento do time).	Muito próximo	Próximo	Muito próximo	Distante
Nível de aproximação da simulação com uma partida real	Muito próximo	Próximo	Muito próximo	Distante
Substituições durante a partida	Possui	Não possui	Possui	Possui
Troca de tática durante a partida	Não possui	Não possui	Não Possui	Possui
Visualização da partida	2D e 3D	Não possui	2D	2D

Quadro 41 – Comparação entre os simuladores estudados e o simulador implementado

Como visto no Quadro 41 o software está em nível inferior em alguns aspectos. Em contrapartida, permite a visualização da partida, visto que Hattrick (2007) não possui. Possibilita também a troca de tática durante a partida, o que não foi visto em nenhum simulador estudado.

4 CONCLUSÕES

O objetivo principal do trabalho, construir um software simulador de uma partida de futebol, foi atingido. Todas as ações dos jogadores, êxitos nas disputas estão relacionados com os seus atributos.

O simulador permite que a equipe troque de tática e jogadores durante a partida. Verifica-se que a troca de tática não foi vista em nenhum outro simulador estudado.

É permitido ao usuário visualizar em tempo real a partida.

Para a implementação do simulador da partida foi necessário um estudo detalhado do funcionamento de processos concorrentes e da biblioteca OpenGl.

As limitações do software de simulação são:

a) não implementação do eixo Z, impossibilitando jogadas aéreas como lançamentos e cabeceios;
b) as regras de impedimento, pênalti e escanteio não foram implementadas;
c) a bola não perde força nos chutes e passes.

As principais dificuldades na implementação do simulador foram:

a) simular novamente um erro ou lance ocorrido devido a aleatoriedade nos lances;
b) depuração do software devido ao uso de *threads*;
c) fazer a interação entre os jogadores, principalmente nos passes e no posicionamento sem a bola.

4.1 EXTENSÕES

Como possíveis extensões para o trabalho, assinalam-se:

a) implementar jogadas áreas, como cabeceios, lançamentos (eixo Z);
b) sofisticar com algoritimos de IA o comportamento e as ações dos jogadores;
c) incluir novos atributos aos jogadores, como peso, altura, destro ou canhoto, experiência, inteligência, cabeceio e forma física;
d) considerar fatores físicos como aceleração da bola e jogadores;
e) desenvolver o visualizador do jogo em 3D;

f) possibilitar que um usuário enfrente outro usuário sem que um conheça a estratégia, táticas e os atributos dos jogadores adversários. Agendar a partida pela *web* seria uma forma. Cada usuário teria o *login* e senha, criaria o seu time e desafiaria outro usuário.

REFERÊNCIAS BIBLIOGRÁFICAS

ALEGRETTI, Francisco J. P. **Sistema distribuído para futebol de robôs.** 2004. 74 f. Trabalho de Diplomação (Bacharelado em Ciências da Computação) – Universidade Federal do Rio Grande do Sul, Porto Alegre.

AZEVEDO, Guilherme M. et al. **Dissuasão de entrada, teoria dos jogos e Michel Porter – convergências teóricas, diferenças e aplicações a administração estratégica**. Cadernos de pesquisa em administração, São Paulo, 2002. Disponível em: <http://www.ead.fea.usp.br/cad-pesq/arquivos/v9n3art6.pdf>. Acesso em: 24 ago. 2006.

BOTELHO, Silvia C. (Coord.). **Futebol de robôs**. Rio Grande, 2006. Disponível em: <http://www.ee.furg.br/~furgbol/futebol.html>. Acesso em: 24 ago. 2006.

CORNÉLIO FILHO, Plínio. **O modelo de simulação do GPCP-1:** jogo do planejamento e controle da produção**.** 1998. Não paginado. Dissertação (Mestrado em Engenharia de Produção) – Programa de Pós-Graduação em Engenharia de Produção e Sistemas, Universidade Federal de Santa Catarina, Florianópolis. Disponível em: <http://www.eps.ufsc.br/disserta98/plinio/index.htm>. Acesso em: 30 abr. 2007.

ESTRATÉGIA. In: INDG, Instituto de Desenvolvimento Gerencial. [S.l.]: INDG, 2007. Disponível em: <http://www.indg.com.br/info/glossario/glossario.asp?e>. Acesso em: 29 maio 2007.

FERREIRA, Aurélio Buarque de Holanda. **Novo Aurélio século XXI**: dicionário de língua portuguesa. Rio de Janeiro: Nova Fronteira, 1999.

FIFA, Federation Internationale de Football Association. **Regras do jogo.** Tradução Confederação Brasileira de Futebol - CBF. Rio de Janeiro, 2006. Disponível em: < http://www2.uol.com.br/cbf/regras/livroderegras.pdf>. Acesso em: 30 abr. 2007.

FUTEBOL. In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2007. Disponível em: < http://pt.wikipedia.org/wiki/Futebol>. Acesso em: 30 abr. 2007.

GAMEGOL. S**imulador de futebol.** [S.1.], 2007. Disponível em: <http://www.gamegol.com.br>. Acesso em: 29 maio 2007.

GOMES, Jonas; VELHO, Luiz. **Fundamentos da computação gráfica**: computação gráfica. Rio de Janeiro: IMPA, 2003.

HATTRICK. S**imulador de futebol.** [S.1.], 2007. Disponível em: <http://www.hattrick.org>. Acesso em: 20 abr. 2007.

HISTÓRIA DO FUTEBOL. In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2007. Disponível em: <http://pt.wikipedia.org/wiki/Hist%C3%B3ria_do_Futebol>. Acesso em: 25 maio 2007.

JACOBSEN, Francis M. **Extensão da linguagem LARF para a comunicação entre robôs jogadores de futebol.** 2003. 75 f. Trabalho de Diplomação (Bacharelado em Ciências da Computação) – Universidade Regional de Blumenau, Blumenau.

LOGOX. **Simulador de corrida de carros F1 Race Brasil.** [S.l.], 2006. Disponível em: <http://www. f1racebrasil.com.br>. Acesso em: 13 abr. 2006.

MANAGERZONE. **Game multiplayer online.** Suécia, 2007. Disponível em: <http://www.managerzone.com>. Acesso em: 20 abr. 2007.

NOGUEIRA, Fernando. **Teoria dos jogos**. Juiz de Fora, 2006. Disponível em: <http://www.engprod.ufjf.br/fernando/epd042/jogos.pdf>. Acesso em: 7 jun. 2006.

ROBOCUP. **RoboCup Federation**. [S.1.], 2007. Disponível em: <http://www.robocup.org>. Acesso em: 01 jun. 2007.

SCHLEI, Edson E. **Uma linguagem para definição de estratégias de controle de times de robôs jogadores de futebol em um ambiente simulado.** 2002. 77 f. Trabalho de Diplomação (Bacharelado em Ciências da Computação) – Universidade Regional de Blumenau, Blumenau.

SEBESTA, Robert W. **Conceitos de linguagens de programação:** programação de computadores. 4. ed. Tradução José Carlos Barbosa dos Santos. Porto Alegre: Bookman, 2000.

SIMULAÇÃO. In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2006. Disponível em: <http://pt.wikipedia.org/wiki/Simula%C3%A7%C3%A3o>. Acesso em: 14 set. 2006.

TÁTICA. In: INDG, Instituto de Desenvolvimento Gerencial. [S.l.]: INDG, 2007. Disponível em: < http://www.indg.com.br/info/glossario/glossario.asp?t>. Acesso em: 29 maio 2007.

WANGENEHIM, Aldo V. **OpenGL**. Florianópolis, 2006. Disponível em: <http://www.inf.ufsc.br/~awangenh/CG/opengl.html>. Acesso em: 19 abr. 2006.