CENTRO UNIVERSITÁRIO DA FEI

FELIPE FARIAS FERRARI
JOÃO AUGUSTO TEIXEIRA MAROTTI
JULIANA BOIN CABRAL
SILAS SHEDERSON DE OLIVEIRA

SIMULAÇÃO DE TRÁFEGO DE VEÍCULOS POR RACIOCÍNIO DE AÇÕES E MUDANÇAS

São Bernardo do Campo
2008

FELIPE FARIAS FERRARI
JOÃO AUGUSTO TEIXEIRA MAROTTI
JULIANA BOIN CABRAL
SILAS SHEDERSON DE OLIVEIRA

SIMULAÇÃO DE TRÁFEGO DE VEÍCULOS POR RACIOCÍNIO DE AÇÕES E MUDANÇAS:

Projeto de Formatura, apresentado ao Centro Universitário da FEI, como parte dos requisitos necessários para obtenção do título de Bacharel em Ciência da Computação.

Orientado pelo Prof. Dr. Paulo Eduardo Santos.

Felipe Farias Ferrari
João Augusto Teixeira Marotti
Juliana Boin Cabral
Silas Shederson de Oliveira

Simulação de Tráfego de Veículos por Raciocínio de Ações e Mudanças

Projeto de Formatura II – Centro Universitário da FEI

Comissão Julgadora

_____

Orientador e Presidente

_____

Examinador (1)

_____

Examinador (2)

_____

Examinador (3)

_____

São Bernardo do Campo
12/12/2008

AGRADECIMENTOS

Primeiramente agradecemos a Deus, por ter feito com que estivéssemos aqui para a conclusão de mais uma etapa de nossas vidas.

A nossos pais, pela paciência, apoio e que na hora do cansaço nos deram motivação para continuarmos com o trabalho.

A nosso orientador Paulo Eduardo Santos, por ter nos transmitido seu conhecimento, nos ajudando e estando sempre presente. E ao grupo responsável pela criação do Simulador de Raciocínio Especial, que encontrou tempo e estava sempre à disposição para esclarecer nossas dúvidas.

A todos os professores da FEI que passaram por nosso caminho durante esses anos e que, de alguma forma, nos ajudaram a amadurecer e nos tornar profissionais.

A todos os nossos amigos, pelo incentivo e paciência. Não os esquecemos, apenas fizemos uso de nosso tempo livre, inclusive finais de semana e madrugadas, para concluírmos essa fase.

Enfim, agradecemos a todos que, de alguma forma, contribuíram para o desenvolvimento de nosso projeto. Nossos sinceros agradecimentos.

"É muito melhor arriscar coisas grandiosas,
alcançar triunfos e glórias,
mesmo expondo-se a derrota,
do que formar fila com os pobres de espírito
que nem gozam muito nem sofrem muito,
porque vivem nessa penumbra cinzenta
que não conhece vitória nem derrota."

Theodore Roosevelt

RESUMO

Com o desenvolvimento de sistemas voltados ao Raciocínio Espacial (representação de relacionamentos entre objetos no espaço através de suas informações) e Cálculo de Situações (representação de ações e seus efeitos, e a partir deles tomar decisões), passa a ser possível compreender e analisar os relacionamentos espaciais entre os objetos físicos inseridos em um domínio dinâmico, resultando na possibilidade de utilizar uma inteligência artificial, capaz de simular um comportamento semelhante a um carro dirigido por um ser humano.

Este projeto foi dividido em duas etapas. A primeira delas teve como objetivo, aperfeiçoar a interface gráfica de um Simulador de Raciocínio Espacial (SRE) desenvolvido em um Trabalho de Conclusão de Curso no ano de 2007, o qual permitia a visualização de relações entre os objetos num ambiente dinâmico, capaz de gerar arquivos a serem analisados com uso de formalismos de raciocínio espacial. Nesta etapa, estendemos o simulador para situações envolvendo tráfego de veículos. Foram criados desde objetos estáticos como ruas, até objetos em movimento como carros e pedestres, resultando num cenário no qual a trajetória de um veículo poderia ser descrita.

A segunda etapa foi caracterizada pela aplicação de Raciocínio de Ações em Inteligência Artificial, elemento que estava fora do escopo do trabalho anterior, sendo caracterizado como uma possível continuidade do mesmo. Nesta etapa, pudemos investigar o apoio à tomada de decisões dos agentes no domínio, utilizando uma base de conhecimentos pré-definida. A percepção do ambiente foi obtida através de Raciocínio Espacial Qualitativo.

ABSTRACT

With the development of systems aimed at Spatial Reasoning (the representation of relations in space) and Situation Calculus (representation of actions and their effects) is now possible to understand and analyze the spatial relationships between the physical objects inserted into a dynamic domain, resulting in the possibility of creating an artificial intelligence system capable of simulating a behavior similar to a car driven by a human being.

This project is divided into two stages. The first of them had the objective to improve the graphic interface of the Spatial Reasoning Simulator (SRE) developed in a Final Year Project at FEI in 2007, which allows the viewing of relationships between objects in a dynamic environment, capable of generating files to be analyzed using spatial reasoning formalisms. In this step, we extended the simulator to situations involving traffic of vehicles. New objects were created in the simulator representing, for instance, streets and moving objects such as cars and pedestrians, resulting in a scenario in which the path of a vehicle could be defined.

The second stage was characterized by the application of Reasoning about Actions on Artificial Intelligence, an element that was outside the scope of the previous work. At this stage we could investigate the construction of a decision support system for the agents in the domain, using a pre-defined knowledge base. The perception of the environment was implemented within the definitions of a well found Qualitative Spatial Reasoning formalism.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1. INTRODUÇÃO

Atualmente, uma das situações mais comuns e corriqueiras de nossa sociedade é o tráfego de veículos. Nas grandes cidades, o intenso aumento da frota normalmente gera altos níveis de congestionamento e estresse, quando podemos levantar a hipótese de como seria interessante se o automóvel pudesse, por exemplo, auxiliar o motorista prevendo situações durante o trajeto percorrido. Fatos como esse abrem inúmeras possibilidades à computação, no seu papel de solucionar ou ajudar na solução de problemas pelos quais o homem se depara no seu dia-a-dia.

Através do poder computacional que a humanidade hoje possui, uma grande utilidade que se pode atribuir aos computadores é a simulação. Ao simular situações da vida real, o homem pode fazer previsões e conclusões futuras mais seguras. Utilizando-se de Computação Gráfica, podemos visualizar o resultado da simulação, e através de interações físicas de alta qualidade entre os objetos, podemos tornar essa simulação mais próxima da realidade, abrangendo diversas situações do cotidiano.

Levando isso em conta, simuladores como o Simulador de Raciocínio Espacial (SRE) são construídos para que essas situações possam ser representadas. Possibilitando a inclusão de objetos e definição de suas trajetórias num cenário e, dessa forma, disponibilizando para análise, através de Raciocínio Espacial e Cálculo de Situações, dados que contém desde a percepção e reconhecimento dos objetos, até as decisões que esses poderiam tomar durante a simulação.

Através de simulações gráficas, já seria possível desenvolver interfaces naturais, capazes de se transformarem em excelentes ferramentas para o estudo de tráfegos em grandes cidades. Entretanto, o software ainda teria uma limitação muito incômoda: os veículos só poderiam agir de maneiras pré-determinadas. No cenário real, a verdade é que, cada veículo

direcionado por um ser humano, pode gerar infinitas situações. Visando suprimir essa limitação, pode-se envolver ainda, outra ferramenta: a Inteligência Artificial.

A Inteligência Artificial (IA), campo de estudo que busca entender a mente humana e imitar seu comportamento, nasceu após a Segunda Guerra Mundial, impulsionada pelas primeiras tentativas de sucesso, como o *Logic Theorist* de Newell e Simon. Ao longo das décadas, ela foi se desenvolvendo e se expandindo, com áreas que têm o propósito de dotar a máquina de raciocínio e capacidade de aprendizado como, por exemplo, a programação de jogos envolvendo o raciocínio, o aprendizado por meio da experiência e conhecimento, Visão Computacional (bidimensional ou tridimensional), Raciocínio Baseado em Casos, o qual visa a resolução de problemas com consulta e recuperação de casos já conhecidos, entre outras.

O Raciocínio Espacial é um importante aspecto humano que possibilita o entendimento do ambiente no qual está e a tomada de decisões a partir desse. A Inteligência Artificial procura levar à máquina esse mesmo aspecto. Utilizando-se de mais esta ferramenta, torna-se possível o desenvolvimento de softwares que não somente simulam situações de tráfego, mas também procuram simular os comportamentos dos veículos nesse ambiente.

1.1. Objetivo

Esse trabalho propõe a continuidade do SRE, com melhorias em seu ambiente gráfico, voltadas a situações que envolvem o tráfego de veículos, além da aplicação de Raciocínio de Ações, visando o apoio à tomada de decisões pelo agente com relação aos objetos incluídos no cenário.

Essas melhorias contemplam a inclusão de novos botões no Editor, referentes aos objetos veículo, agente e pedestre, além de tornar possível a escolha do usuário entre os

cenários disponibilizados para a simulação. Além disso, o campo de visão do agente será alterado para que se aproxime do campo visualizado por um motorista em seu veículo.

Já no Simulador, teremos a inclusão dos alertas a serem exibidos ao usuário durante uma situação de risco.

1.2. Problema a ser resolvido e Justificativa

A partir dos objetivos descritos no item anterior, foram propostas algumas situações, as quais o simulador deve ser capaz de prever e evitar. As situações escolhidas, ultrapassagem indevida e cruzamento entre vias, estão entre as maiores causas de acidentes.

A primeira ocorre principalmente em estradas de vias duplas, onde um motorista necessita invadir a pista contrária para realizar uma ultrapassagem sobre outro e, devido à falta de espaço para realizar a ultrapassagem, um grave acidente pode ser causado.

Acidentes no cruzamento entre vias ocorrem tanto em estradas como nas ruas das cidades, onde na maioria das vezes o motorista não pára antes de avançá-lo, acarretando em uma colisão lateral.

Devido a estas situações, esse trabalho se propõe a estudá-las através de simulação e, a partir de seus resultados, possibilitar a exibição de alertas ao usuário, em tempo hábil para que uma possível tomada de decisões venha a ser realizada. No entanto, o tratamento dessas reações está fora do escopo desse projeto.

2. TRABALHOS RELACIONADOS

Nesse capítulo descrevemos os trabalhos que serviram como base para o nosso projeto.

No trabalho de VAZ, Thiago G. et al (2007), foi desenvolvido o SRE. Através do desenvolvimento de um simulador gráfico, uma situação espacial com interações entre as entidades poderia ser representada. O sistema foi desenvolvido com o uso de uma Interface de Programação de Aplicativos (API) JAVA2D (movimentação de objetos gráficos, detecção e resposta à colisão entre objetos), a qual será detalhada na seção 3.4.1 desse documento, e criação de curvas *splines* e curvas de Bézier para a definição da trajetória de movimentação dos objetos. Ao término da simulação, o sistema foi capaz de gerar dados qualitativos para serem analisados por formalismos de raciocínio espacial.

Uma das extensões possíveis a esse trabalho é a aplicação de Raciocínio de Ações, ramificação da Inteligência Artificial, nos objetos do ambiente simulado. Assim, podemos dizer que este trabalho se baseia fundamentalmente na continuidade do trabalho de VAZ, Thiago G. et al. (2007).

Sabendo que a Robótica Cognitiva é caracterizada pelos problemas enfrentados por um robô autônomo (ou pelo agente) com o raciocínio de conhecimento em um mundo dinâmico e não conhecido completamente, Levesque H. et al. (2007) apresentou um estudo sobre isso. Nele são representadas situações que geram mudanças e ações determinísticas ou não-determinísticas, podendo realizar, nesse caso, uma recuperação de raciocínio para mapeamento de falhas. Esse estudo é útil para mostrar que, mais do que programar os controladores do robô, de forma que ele possa resolver uma classe de problemas dentro de um

domínio específico, o maior desafio é o controle de alto nível da robótica, possibilitando ações no mundo, sempre em mudança.

Em Santos, M. V. dos, et al. (2007) apresenta-se uma teoria de aperfeiçoamento para o estudo e análise de mudanças observáveis geometricamente, as quais ocorrem em informações obtidas através de uma seqüência de imagens de uma cena dinâmica. Para isso, é definido um formalismo para designar o conteúdo de uma cena, assim como as mudanças geométricas ocorridas e, por fim, um algoritmo para construir a descrição dessas mudanças a partir de deduções lógicas.

Ainda de acordo com Santos, M. V. dos, et al. (2007), os experimentos realizados se mostraram satisfatórios para elementos geométricos de cores distintas, analisando corretamente em 100% dos casos a ação de rotação realizada. Porém, para elementos de mesma cor, apenas em 60% dos casos onde os elementos se encontravam separados por uma distância grande foram analisados corretamente, e em nenhum caso, onde os elementos se encontravam separados por uma distância pequena, foi analisado corretamente.

Uma provável solução do problema, proposta pelo próprio autor do trabalho Santos, M. V. dos, et al. (2007), seria obtida através do aperfeiçoamento do algoritmo, que analisa as informações capturadas pelo sensor antes de passá-las adiante para os cálculos relacionados às transições das cenas, fazendo com que o mesmo seja mais eficiente em distinguir elementos geométricos de mesma cor.

No trabalho de Rahul Sukthankar (1997), foi criado um simulador de trânsito onde o agente (um carro) toma decisões em tempo real, através da prevenção de situação de risco, mesmo tendo informações incompletas sobre o cenário. Esse controle é feito utilizando-se objetivos de direção em alto nível (como traçado de rotas), ações táticas de nível

intermediário (como ultrapassar um carro lento) e comportamento reativo de baixo nível (como brecar ou virar à esquerda).

Duas abordagens de um sistema de raciocínio tático baseado em prevenção de situação de risco foram implementadas no trabalho de Rahul Sukthankar (1997), a primeira, MonoSAPIENT, se resume ao uso de regras explicitamente codificadas e algoritmos especiais, e a segunda, PolySAPIENT, uma inteligência distribuída que se utiliza de objetos de raciocínio independentes focados na observação e análise de apenas um aspecto do trânsito, associado a entidades do cenário (como um carro ao lado ou a próxima saída da estrada). Também, para o PolySAPIENT, foi implementada uma estratégia de aprendizado chamada PBIL, que ajuda a otimizar os parâmetros de decisão para múltiplos objetos de raciocínio.

Para diversos cenários testados, de acordo com Rahul Sukthankar (1997), os resultados foram satisfatórios, porém, se o PolySAPIENT for submetido a uma situação que não foi corretamente analisada em seu treinamento de aprendizado pelo PBIL, situações críticas, como não frear o carro quando houver uma necessidade de parada emergencial, podem ocorrer. Independente do treinamento, outra situação – a de desviar de um veículo parado até momentos antes escondido por outro em movimento – não foi bem sucedida.

Uma possível solução, proposta pelo autor Rahul Sukthankar, para esse problema, seria fazer com que ambas as abordagens tenham conhecimento sobre essas situações de risco e estejam preparadas para elas, mesmo que a situação seja descoberta demasiadamente tarde.

Em Santos, M. V. dos, et al. (2008) é possível encontrar uma teoria clara e satisfatória de como obter informações relevantes de um espaço onde se encontram objetos, em movimento ou não, e as relações entre os mesmos. O que se deseja é que um robô em movimento seja capaz de absorver as informações do meio em que está (espaço) através de

sensores e, conseqüentemente, calcular situações. Os sensores, posicionados a uma altura média, fornecem a forma horizontal e a profundidade dos objetos à frente do robô. A partir destes dados, constrói-se o Perfil de Profundidade, que será a base para o cálculo de situações. Por Perfil de Profundidade entende-se o mapeamento em gráficos dos dados capturados pelos sensores.

Com o Perfil de Profundidade em mãos, o artigo citado (Santos, M. V. dos, et al. (2008)) passa a descrever estratégias de Cálculo de Perfil de Profundidade (DPC) para a formação de axiomas e sentenças lógicas que permitirão ao robô, no papel de agente, calcular e prever situações. Além disso, no Perfil de Profundidade são encontradas algumas situações diferenciadas quanto ao relacionamento de objetos capturados pelos sensores, tais como o aparecimento ou desaparecimento repentino de objetos e a oclusão desses, que também são discutidos no artigo. Todas as situações do Perfil de Profundidade são reunidas e formalizadas logicamente, culminando na Teoria de Profundidade e Movimento. Essa teoria é a descrição, por sentença lógica, das conclusões tiradas a partir das informações no Perfil de Profundidade, permitindo assim que o robô, agente, possa raciocinar em função do que seus sensores estão constantemente capturando.

O trabalho de Santos P. (2007) tem como objetivo discutir alguns aspectos de seu trabalho anterior (Santos P. (2003)) e demonstrar através de axiomas que é possível obtermos informações do mundo real, o qual é dinâmico e sujeito a mudanças, extraindo informações sobre os objetos e seus movimentos no espaço, a partir do ponto de vista de um observador. Para isso, utilizou Cálculo de Perfil de Profundidade (DPC) e Cálculo de Perfil de Profundidade Dinâmico (DDPC), possibilitando prever e gerar hipóteses sobre os movimentos de um objeto.

Através do uso do DPC é possível obter, de acordo com Santos P. (2007), em um determinado tempo *t*, características da imagem analisada, como disparidade, tamanho e distância dos objetos físicos contidos nessa imagem. Complementando o DPC, o DDPC interpreta as transições entre cada par de imagens analisada sobre os objetos, os quais devem estar dentro de três premissas básicas: um objeto não pode simplesmente aparecer ou desaparecer em seu domínio, qualquer mudança na sua posição é por causa de seu contínuo movimento e que também o mesmo não pode transpassar outro objeto.

Para isso, toda a análise, cálculo e identificação das transições sucessivas devem ser feitas em espaços de tempo muito pequenos, possibilitando a percepção das reais mudanças do ambiente.

Ao desenvolver o objetivo proposto o autor delimita o escopo de seu trabalho (Santos P. (2007)) mostrando alguns pontos que infringem os paradigmas utilizados, como objetos que podem transpassar outros (ex: cortina de fumaça), uso de comparações entre o observador e o objeto para obter o movimento do objeto e utilização de cálculos para regiões 3D, ao invés de Perfil de Profundidade, permitindo assim o estudo do raciocínio espacial sob outra perspectiva.

Em Herzog, O. et al. (2005) os autores demonstram que é possível simular veículos inteligentes criando um sistema de assistência ao motorista ou inteiramente autônomo, dirigido sem a intervenção humana, em rodovias. Através de relações entre espaço e tempo, sensores e Cálculo de Conexão de Região (RCC), permitem a extração de dados do ambiente onde está sendo feita a simulação, que juntamente com uma base de conhecimento pré-definida (constituída por dados sobre o tráfego, a rede de ruas e segmentos relevantes, a movimentação de objetos dinâmicos, as relações entre objetos e entre objetos e regiões do terreno) e padrões comparados em tempo real automaticamente usando inferência baseada em

Prolog, possibilitando que todos os cálculos sejam feitos em tempo real, deixando a simulação próxima da realidade.

Ainda de acordo com Herzog, O. et al. (2005), na vida real, a representação das cenas de tráfego assumida pela simulação, poderia ser fornecida por sensores que devem ser mapeados e transformados para uma série de representações qualitativas espaciais e temporais existentes. Apesar disso, muitos desafios do mundo real ficaram de fora da simulação, como o processamento de imagem em tempo real, o monitoramento dos objetos e a manipulação de dados com ruído. Também existe uma limitação na simulação sobre a eficácia do sistema para realizar toda a análise entre cada ciclo (tempo para realizar as comparações e decisões) em até 150 milisegundos para que seja em tempo real. Por isso é limitado a sete o número de objetos dinâmicos na cena, e assim, a cada novo objeto adicionado aumenta quadraticamente o tempo de análise, como representado na Figura 1.

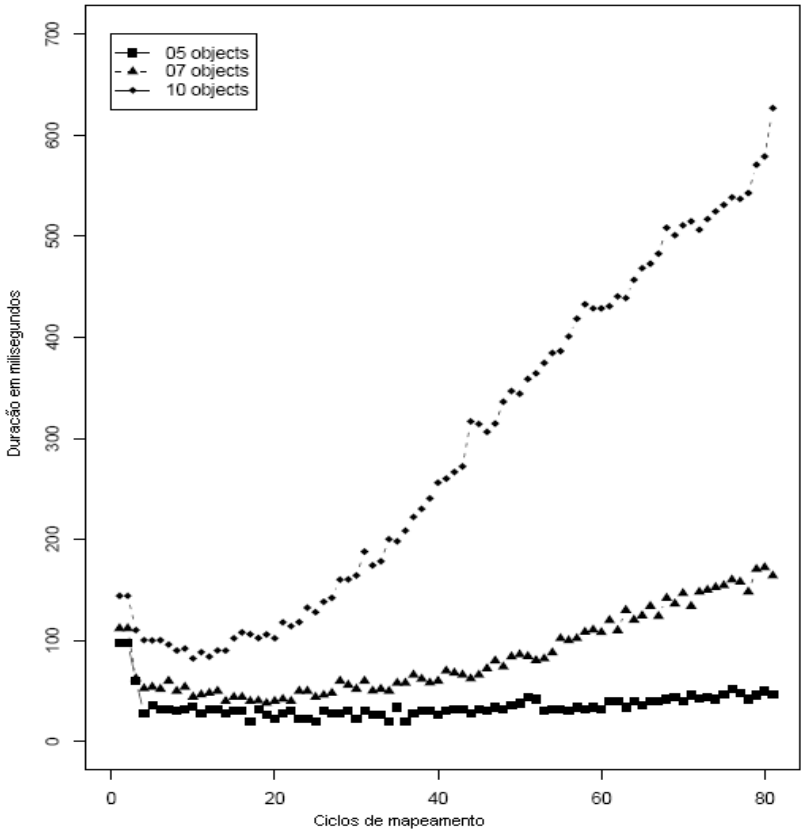


Figura 1: Demonstração do tempo de análise

Fonte: Autores “adaptado de” Herzog, O. et. al, 2005

O tema pode ainda, ser abordado sobre diferentes perspectivas também propostas no trabalho de Herzog, O. et al. (2005), sendo uma delas a realização do controle da velocidade do veículo inteligente, não somente no módulo de decisão do comportamento, e criação de novos padrões para a avaliação da situação, implementando um comportamento mais complexo. Outra perspectiva seria implantar a previsão do comportamento em outros participantes do tráfego, baseado na representação qualitativa já existente.

3. REVISÃO BIBLIOGRÁFICA

Neste capítulo apresentaremos os conceitos básicos sobre as técnicas e teorias que são utilizadas nesse trabalho. São abordados pontos como ambiente simulado, raciocínio de ações e técnicas de raciocínio espacial, além de ferramentas como Eclipse e Prolog. Esses itens são necessários para garantir o correto entendimento da solução adotada e, ao término, os resultados gerados.

3.1. Simulador de Raciocínio Espacial (SRE)

O SRE consiste num simulador capaz de representar condições espaciais, além de permitir a análise do conhecimento espacial, obtido através do uso de perfis de profundidade. Essas e outras técnicas usadas serão explicadas a seguir e durante todo o projeto aqui apresentado, uma vez que, conforme dito anteriormente, teremos esse simulador como base para a aplicação de Inteligência Artificial e adaptação à situações de tráfego de veículos.

Na Figura 2, vemos a interface do simulador contendo o Observador e um objeto, além da definição de suas trajetórias.

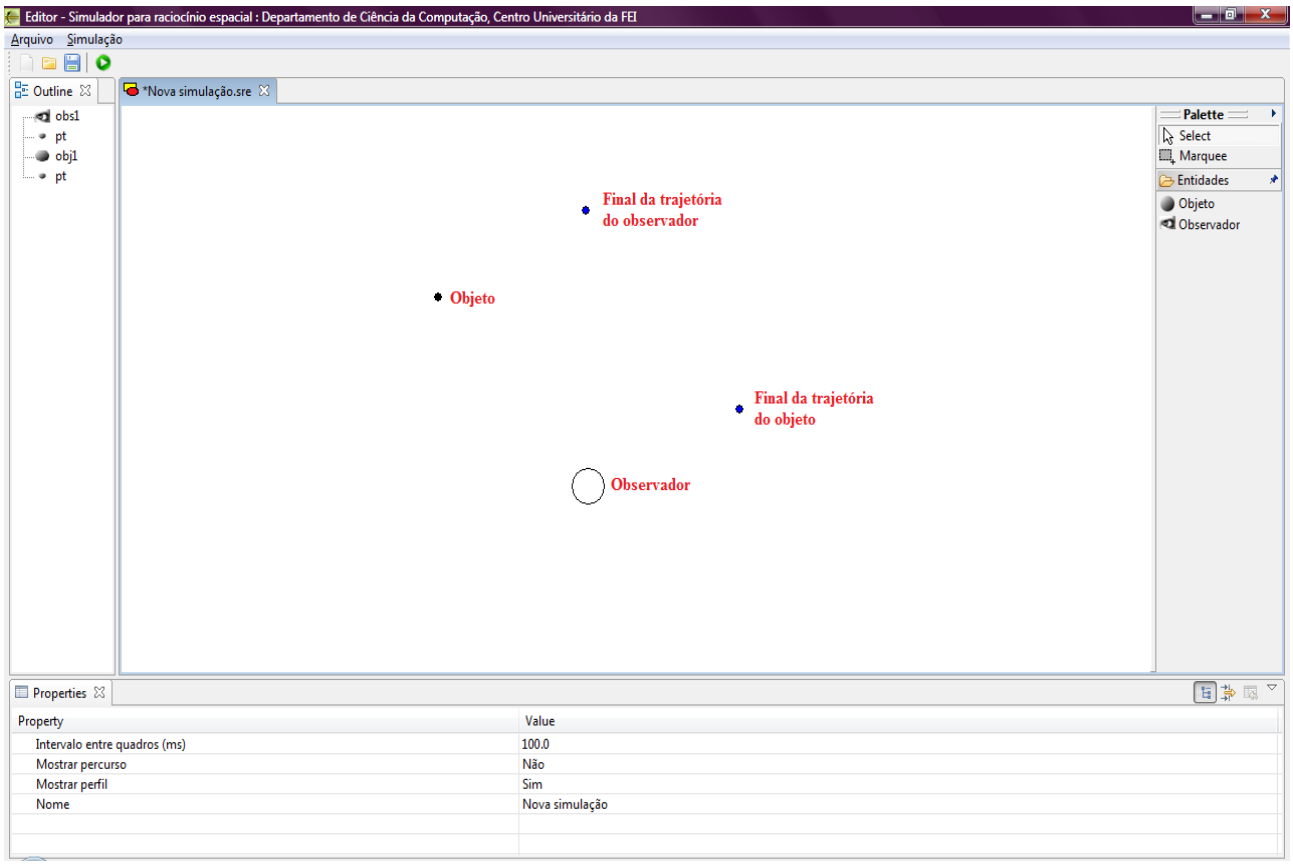


Figura 2: Simulador de Raciocínio Espacial

Fonte: Autores

3.1.1. Solução adotada

Através da representação do raciocínio espacial num ambiente simulado bidimensional, esperava-se que o sistema fosse capaz de gerar hipóteses sobre a movimentação dos objetos, conforme Figura 3.

Figura 3: Diagrama de blocos do SRE

Fonte: VAZ, Thiago G. et al, 2007, p. 45

A situação representada na Figura 2 tem seu ambiente simulado representado pela Figura 4 em dois instantes de tempo.

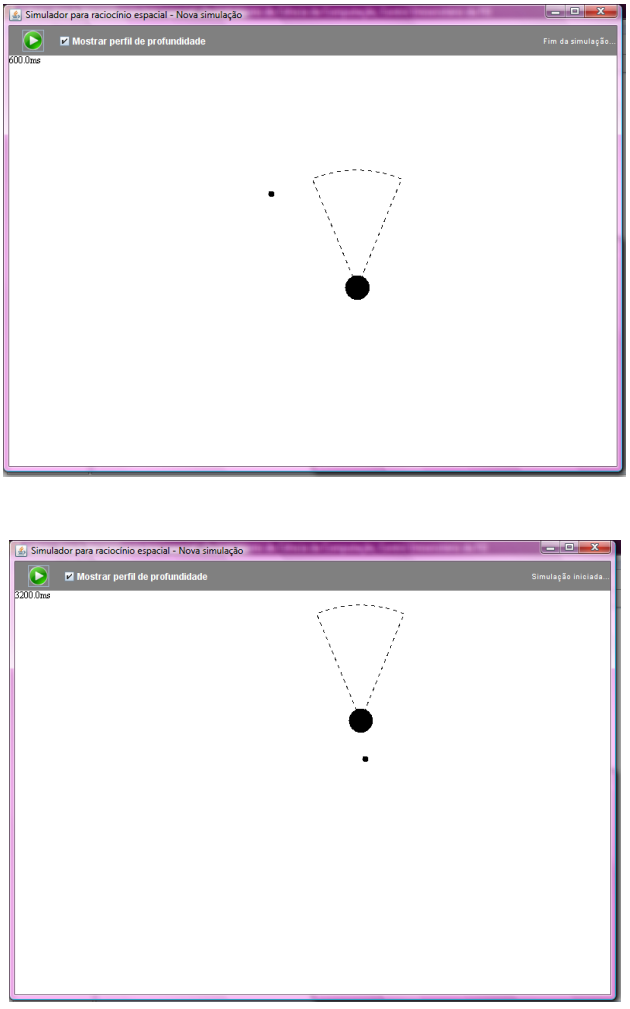


Figura 4: Ambiente Simulado

Fonte: Autores

A visão do observador foi limitada em um ângulo de 45 graus, campo referente a linha tracejada no Ambiente Simulado. Na Figura 5, podemos definir que os objetos X1 e X2 estão dentro de sua visão, e é baseado nessa relação entre o observador e esses que as hipóteses são geradas, já o objeto X3, está fora do campo de visão.

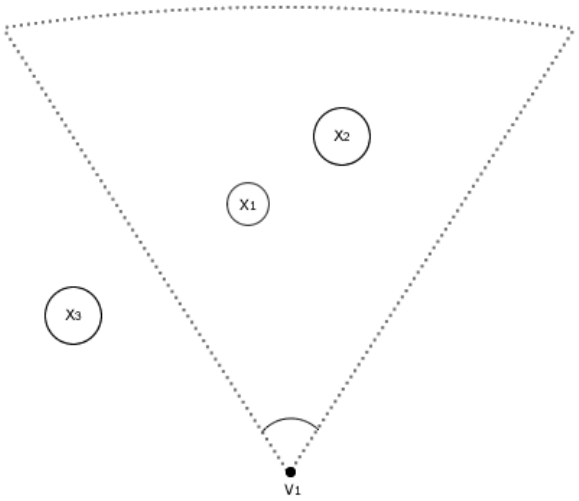


Figura 5: Definição do ambiente

Fonte: VAZ, Thiago G. et al, 2007, p. 46

Dentro dessa visão, o reconhecimento dos objetos é feito através da visão linear, onde são traçadas retas que têm como origem o observador e, como fim, o limite de visão e o próprio objeto. É através dessas informações que é possível gerar o perfil de profundidade, gráfico que representa o ponto de vista do observador e demonstra a distância que os objetos se encontram dele, e que será explicado com mais detalhes ainda nessa seção.

3.1.2. Conclusões apresentadas através do SRE

Por ter sido possível gerar dados referentes à simulação, descrevendo os relacionamentos entre os objetos e observador, dentro do ambiente, podemos dizer que os objetivos desse trabalho foram atingidos.

Além disso, com a definição dos requisitos, pesquisas sobre as possíveis técnicas a serem utilizadas puderam ser realizadas pelos autores, tendo esses escolhido as que apresentariam melhores resultados para atingir o objetivo desejado.

3.1.3. Proposta de Continuidade para o SRE

Foram propostas três continuações ao SRE, a primeira delas consiste na aplicação de Raciocínio Espacial e, a segunda, na análise dos acontecimentos ocorridos durante a simulação. Com isso, é possível prever as percepções futuras do observador. Essas propostas estarão incluídas em nosso projeto.

Além disso, há outra continuação possível, a qual está fora do escopo desse projeto e que envolve o desenvolvimento de um simulador tridimensional.

3.2. Ambiente simulado

O ambiente simulado criado teve como base o SRE, com alterações necessárias para que seja possível simular uma situação de trânsito de veículos. Para isso, entidades móveis já

existentes foram modificadas (o observador é o veículo inteligente e os objetos são os carros e pedestres) e uma nova entidade estática (rua) foi criada.

A movimentação das entidades deve ser obrigatoriamente definida pelo usuário. Tanto o veículo inteligente quanto as outras entidades móveis, devem seguir fielmente o caminho identificado pelos pontos determinados pelo usuário. Durante o percurso, o veículo inteligente estará sempre recebendo estímulos do ambiente simulado em forma de Perfil de Profundidade e exibirá alertas com a situação atual, de acordo com a análise dos dados obtidos através desses estímulos.

Mais informações sobre o SRE podem ser encontradas na Seção de Trabalhos Relacionados.

3.3. Raciocínio Espacial

O objetivo do raciocínio espacial qualitativo é possibilitar a descoberta de conhecimento, a percepção do ambiente como um todo, com o reconhecimento e a análise de relacionamentos implícitos entre os objetos de um espaço através de suas informações. Esses relacionamentos podem ser manipulados e sempre têm suas representações feitas em alto nível, o que diferencia o raciocínio espacial de outras técnicas.

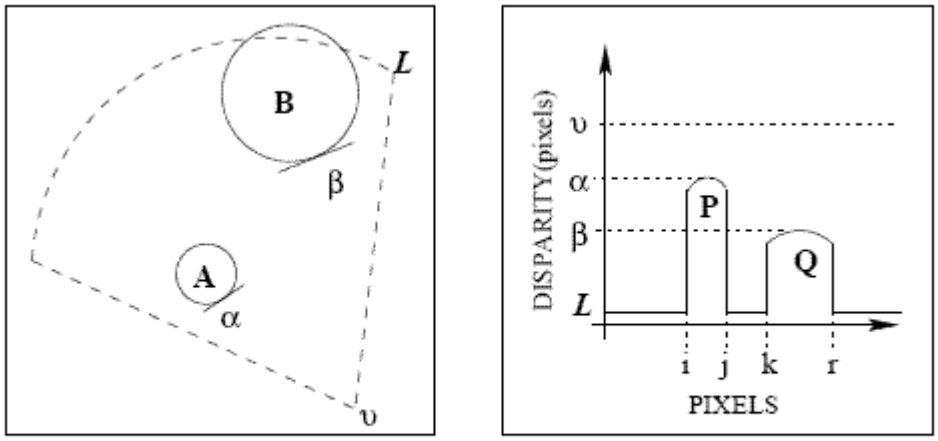
Essas representações devem possibilitar a análise dos dados para exibição de alertas de prevenção mesmo nos casos em que não for possível obter todas as informações, as quais podem estar incompletas ou imprecisas.

Os conceitos utilizados para atingir esse objetivo serão abordados nessa seção.

3.3.1. Perfis de profundidade

Perfil de profundidade pode ser descrito como uma representação gráfica de dados capturados por sensores, transformando-se em uma visão linear de uma dimensão e meia. Essa representação gráfica é capaz de exibir a forma horizontal e a profundidade dos objetos capturados pelos sensores, além de exibir o limite do campo de visão do observador, vide Figura 6.

É a partir dessa representação que será possível retirar informações valiosas do ambiente mapeado através dos sensores.



(a) Dois objetos A, B e um observador v com seu campo de visão visto de uma perpectiva aérea

(b) Uma representação do perfil de profundidade em relação ao ponto de vista do observador v da Figura 6(a)

Figura 6: Perfil de Profundidade

Fonte: Autores “adaptado de” Santos P., 2007

Também nessa figura, a visão do observador é representada por *v* e limitada por *L*. Os picos *P* e *Q* representam, respectivamente, os objetos *A* e *B* contidos dentro do campo de visão do observador em um instante t_1. A profundidade dos objetos do ambiente em relação ao observador é dada por α e β. As relações $|j - i|$ e $|r - k|$ representam o tamanho dos picos

P e *Q*, podendo ser assumidos também como o tamanho aparente dos objetos *A* e *B*. A distância entre *P* e *Q* é definida por $| k - j |$.

3.3.2. Cálculo de Conexão de Região (RCC)

O Cálculo de Conexão de Região (RCC – Region Conection Calculus) visa tratar o relacionamento entre áreas ou regiões que, no caso deste projeto, serão fornecidas pelos Perfis de Profundidade. É através dele que será possível obter as primeiras conclusões em relação ao espaço e gerar axiomas de base a serem utilizados no Raciocínio Espacial.

São encontradas 6 relações espaciais básicas no Cálculo de Conexão de Região:

a) DC (x, y): (*Disconnected*) Uma região está desconectada da outra;

b) EC (x, y): (*Externally Connected*) Uma região está externamente conectada à outra, isto é, as regiões esbarram-se;

c) PO (x, y): (*Partially overlaps*) Uma região está sobrepondo parte da outra região;

d) TPP(x, y): (*Tangential proper part*) Uma região pertence quase que totalmente à outra, à exceção de que suas bordas tangenciam-se;

e) NTPP(x, y): (*Non-tangential proper part*) Uma região pertence totalmente à outra;

f) EQ(x, y): (*Equal*) Ambas as regiões têm o mesmo tamanho e ocupam a mesma área, isto é, uma está sobrepondo completamente a outra de mesmo tamanho.

3.3.3. Cálculo de perfil de profundidade (DPC)

O Cálculo de Perfil de Profundidade (DPC – Depth Profile Calculus) é a formalização lógica, através de axiomas, de conhecimentos adquiridos através do Perfil de Profundidade. É a partir dele que o Raciocínio Espacial começa a criar forma.

A seguir, alguns dos axiomas propostos por (Souchanski, M. e Santos, P., 2008) e (Santos, P., 2007) para o cálculo de perfil de profundidade:

a) pk(b, u, z, d): Simboliza o pico de um corpo b localizado a uma profundidade u que, do ponto de vista atual, tem tamanho z e distância angular d da borda esquerda;

b) loc(x, y): Simboliza a localização de coordenadas x e y no plano cartesiano;

c) startMove(b, l_1, l_2, t): Simboliza o início de movimento de um objeto b, movendo-se entre l_1 e l_2 no instante t;

d) endMove(b, l_1, l_2, t): Simboliza o término de movimento de um objeto b, movendo-se entre l_1 e l_2 no instante t;

e) sense(p, loc(x, y), t): Simboliza a captura de um perfil de profundidade p, quando localizado em x e y no instante t;

f) disC(a, b, t): Simboliza que o objeto a está desconectado do objeto b no instante t;

g) extC(a, b, t): Simboliza que o objeto a está externamente conectado a b no instante t;

h) co(a, b, t): Simboliza que o objeto a está agregado a b no instante t;

i) FT(a, b, t): Simboliza que a está mais distante do observador que b no instante t;

j) CT(a, b, t): Simboliza que a está mais próximo do observador que b no instante t;

k) DEq(a, b, t): Simboliza que a está tão distante do observador quanto b no instante t;

l) LT(a, b, t): Simboliza que o objeto a é maior que b no instante t;

m) ST(a, b, t): Simboliza que a é menor que b no instante t;

n) SEq(a, b, t): Simboliza que a é do mesmo tamanho que b no instante t;

Todos esses axiomas simbolizam relações e situações capturadas pelos sensores para o perfil de profundidade e serão utilizadas para o Raciocínio Espacial.

3.3.4. Cálculo de perfil de profundidade dinâmico (DDPC)

De acordo com Souchanski, M. e Santos, P. (2008) são assumidas três restrições principais para as correspondências entre objetos e picos ao longo do tempo:

a) Persistência de Objeto: Os objetos não podem aparecer ou desaparecer instantaneamente;

b) Suavidade de movimento: Os objetos não podem pular de um lugar para o outro abruptadamente;

c) Não Interpenetração: Objetos não atravessam uns aos outros.

Assumidas as três restrições apresentadas acima, pode-se efetivamente enumerar as situações comuns encontradas no cálculo de perfil de profundidade dinâmico (DDPC – Dynamic Depth Profile Calculus). Conforme Figura 7, são elas:

a) Aproximação (approaching): Os objetos estão aproximando-se um do outro;

b) Distanciamento (receding): Os objetos estão distanciando-se um do outro;

c) Aglutinação (coalescing): Os objetos estão aglutinando-se um ao outro;

d) Separação (splitting): Os objetos estão separando-se um do outro;

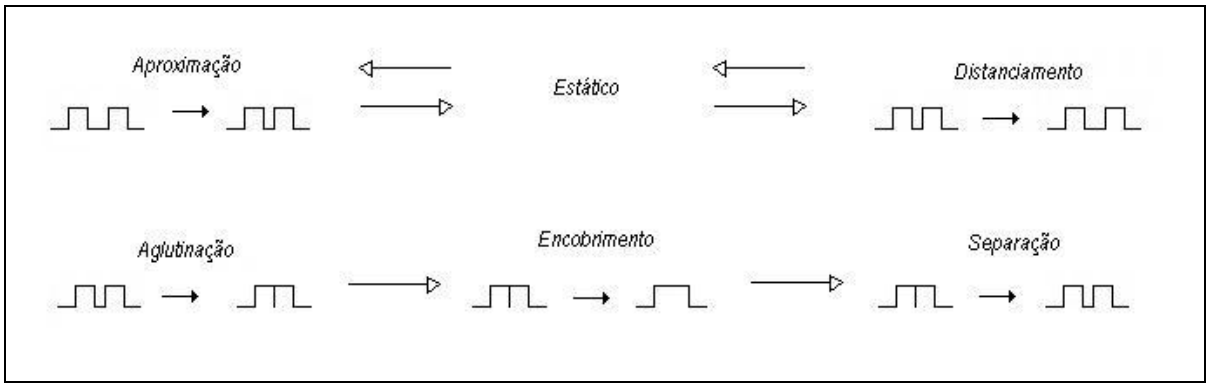


Figura 7: Cálculo de Perfil de Profundidade Dinâmico

Fonte: Autores “adaptado de” Santos, P. et al, 2008

No cálculo de perfil de profundidade dinâmico podem ser encontradas também situações que se referem à dinâmica de um único objeto, tais como:

a) Ampliação/Expansão (extending): O objeto está aumentando seu tamanho conforme a variação do tempo;

b) Compactação (shrinking): O objeto está diminuindo seu tamanho conforme a variação do tempo;

c) Aparecimento repentino (appearing): O objeto aparece repentinamente em um determinado instante. Isso se dá ao fato de estar muito perto do observador;

d) Desaparecimento repentino (vanishing): O objeto desaparece repentinamente em um determinado instante, também devido à alta proximidade em relação ao observador.

Portanto, os cálculos de perfil de profundidade dinâmico mapeiam e transformam em axiomas as diferentes situações encontradas nas relações entre objetos ou, até mesmo, entre um objeto e o observador.

3.4. Ferramentas Auxiliares

Aqui apresentaremos os principais conceitos e características das tecnologias de apoio que foram usadas, juntamente com as teorias descritas anteriormente nessa seção, para atigirmos nosso objetivo.

3.4.1. Java 2D

Em nosso trabalho, a interface e a representação gráfica do que ocorre na simulação é renderizada pelo Java 2D.

Java 2D é uma Interface de Programação de Aplicativos (API), em português, ou seja, um conjunto de pacotes com chamadas de serviços (não sendo necessário o conhecimento de detalhes de sua implementação) da linguagem de programação Java que provê uma série de rotinas para criar e manipular elementos gráficos em duas dimensões.

Ela oferece os seguintes pacotes (conjunto de classes já implementadas que executam ações quando são criadas ou seus métodos são chamados):

a) **java.awt -** É o principal pacote. Com ele, é possível criar janelas, containers (onde as imagens serão renderizadas), botões, caixas de seleção entre outros elementos gráficos de interface com o usuário. Ele também possui listeners para esses elementos, ou seja, objetos que respondem com alguma ação quando o elemento que ele está "ouvindo" sofre alguma interação, como, por exemplo, clicar em algum botão ou mudar o foco para alguma janela. Além disso, tem funções para renderização, preenchimento, transformação, suavizamento, entre outras operações para serem realizadas com os elementos gráficos;

b) **java.awt.color** - Usado para manipular e gerenciar as cores e as diversas formas de representá-la;

c) **java.awt.font** - Esse pacote é utilizado para manipular como um conjunto de caracteres será exibido (tamanho, fonte, negrito, etc);

d) **java.awt.geom** - Provê rotinas e funções para operações com objetos em duas dimensões, como retângulos, elipses, retas, pontos e curvas e suas variantes;

e) **java.awt.images** - Usado para manipular imagens, criadas ou já existentes no sistema. Provê funções para o processamento de imagens e diversos filtros;

f) **java.awt.print -** Esse pacote auxilia na configuração de dados para impressão, como jobs de impressoras, paginação, formato, etc.

3.4.2. Eclipse

Trata-se de uma plataforma *open source* para desenvolvimento de ferramentas e Ambientes de Desenvolvimento Integrado (IDEs), com arquitetura extensível e baseada em *plug-ins* e que visa a otimização do processo de desenvolvimento de software. Uma das suas principais características é a portabilidade para sistemas operacionais, já que faz uso das APIs portáveis da Máquina Virtual Java (JVM). "O principal objetivo da plataforma Eclipse é oferecer aos desenvolvedores de ferramentas, mecanismos e regras que permitam a implementação de funcionalidades para um ambiente em comum." (VASCONCELOS, A. T., 2005, p. 10)

Um *plug-in* é a mínima unidade funcional do Eclipse, o qual pode conter bibliotecas Java, arquivos de ajuda, imagens e etc. Eles compõem a maior parte do Eclipse e são conectados uns aos outros através de um arquivo *manifest* do tipo XML, o qual contém as interfaces de interconexões. Trata-se de um número de pontos de extensão e a quantidade de

extensões de algum ponto de extensão de outro *plug-in*. É através de um *manifest* que a plataforma consegue identificar as ligações entre os *plug-ins*.

Os *plug-ins* são dependentes entre si e contribuem com um ou mais pontos de extensão (usados para agrupamento de contribuições), podendo ou não, declarar novos.

Eles podem acrescentar documentação ao sistema e definir onde os tópicos de ajuda serão inseridos na estrutura, além de pertencerem ao seu próprio diretório de *plug-ins*. É importante lembrar que com apenas um, já seria possível gerar uma ferramenta, embora as mais complexas sejam compostas por vários deles.

3.4.2.1. Plataforma Eclipse

É através da execução da *Plataform Runtime*, o núcleo, que os *plug-ins* são gerenciados e as informações sobre eles são atualizadas em um registro da plataforma.

O gerenciador de recursos *Workspace*, possibilita a manutenção e criação de recursos paralelamente, por usuário ou por diretórios no sistema de arquivos.

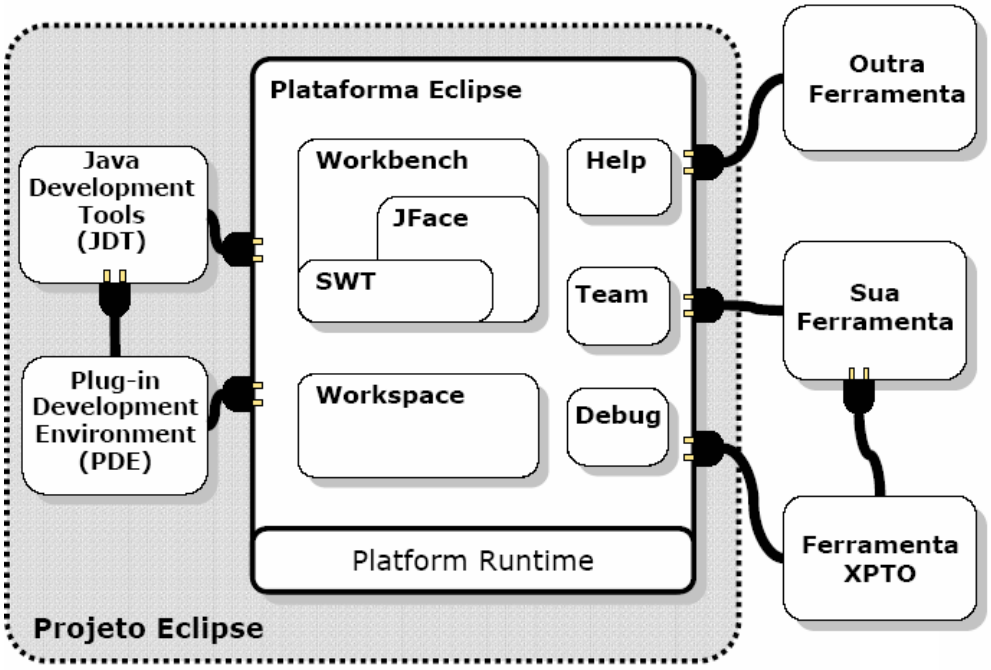


Figura 8: Arquitetura da plataforma Eclipse

Fonte: Daflon, [200-]

Já o *workbench* representa a interface gráfica para o usuário (Figura 9). Ele faz uso dos conjuntos de bibliotecas gráficas abaixo:

a) Jface – É composta por frameworks de Interface Usuário (UI), facilitando o desenolvimento de interfaces;
b) SWT (Standart Widget Toolkit) – Providencia eficiência e é independente de sistema operacional.

Além disso, é composto por visões, editores e perspectivas, os quais permitem que o usuário obtenha informações sobre os objetos do *workbench*, abra, edite e salve os recursos e organize editores e visões, respectivamente.

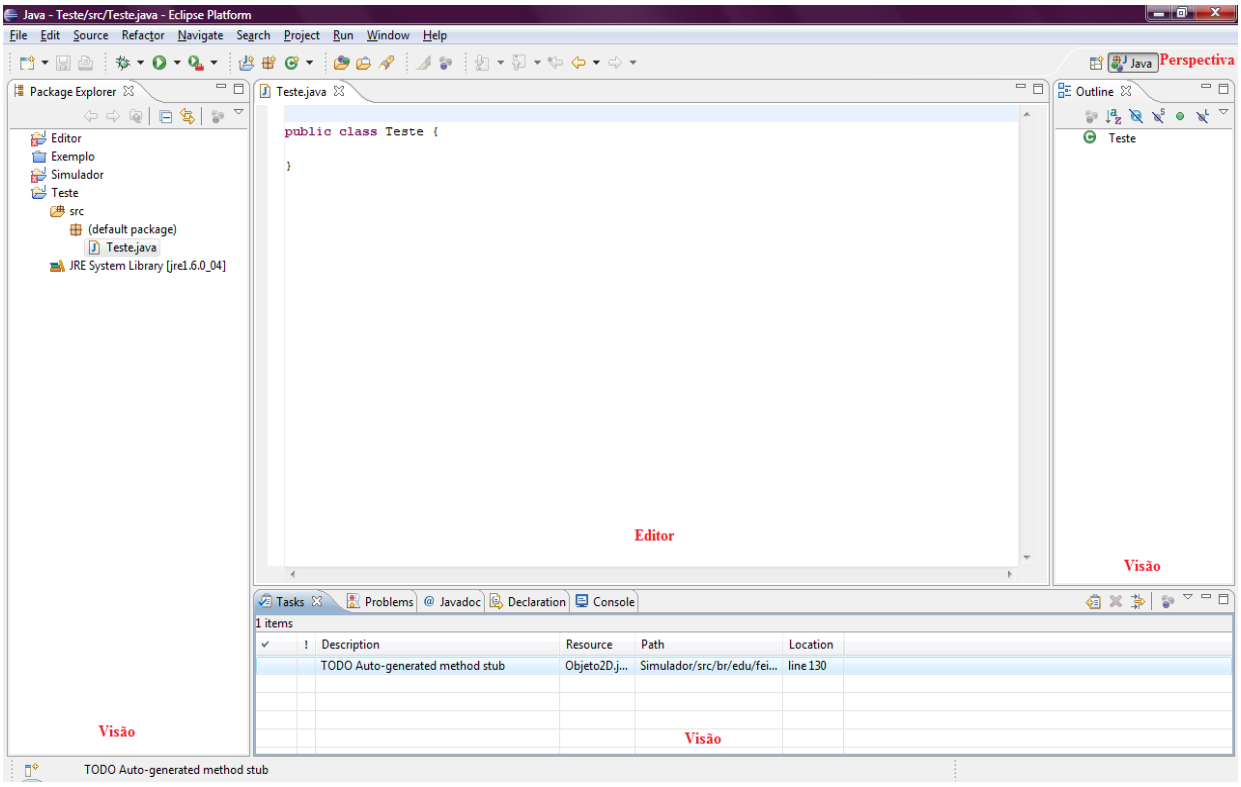


Figura 9: Workbench da ferramenta Eclipse

Fonte: Autores

Ainda dentro do *workbench*, temos o Ambiente de Desenvolvimento de *Plug-ins* (PDE). O próprio PDE também é um *plug-in*, criado a partir dos recursos da plataforma em conjunto com a Ferramenta de Desenvolvimento Java (JDT). É através dele que o arquivo *manifest*, citado anteriormente, é editado e a criação, manipulação e instalação dos *plug-ins* desenvolvidos são automatizadas.

O JDT, por sua vez, é composto por um conjunto de *plug-ins* e faz uso das APIs e pontos de extensão, podendo dessa forma, concentrar as funcionalidades necessárias a um desenvolvedor, como visualização, compilação, execução e teste de códigos Java.

Para finalizar, temos ainda o *Help* e o *Team*. O primeiro provê mecanismos de suporte e documentação à plataforma e o segundo possui ferramentas a serem usadas pela equipe, como controle de versões e acesso de repositório, por exemplo.

3.4.2.2. *Rich Client Plataform* (RCP)

O RCP pode ser definido como uma plataforma rica em interfaces, layouts e menus, composta pelo conjunto mínimo de *plug-ins* necessários para a construção de aplicações *rich client*. Esse conjunto é composto por apenas dois deles: **org.eclipse.ui** e **org.eclipse.core.runtime**.

A construção dessas aplicações é realizada em cima do *framework* do Eclipse, o que faz com que essa plataforma se beneficie de todas as vantagens desse.

Para construir uma aplicação RCP, o desenvolvedor define os pontos de extensão e cria perspectivas com visões, menus e editores como os da própria ferramenta, utilizando recursos nativos ou ferramentas auxiliares (VAZ, Thiago G. et al, 2007).

Com o uso de Eclipse RCP, não é necessário que os programadores reescrevam as classes do *framework* desde o início para o desenvolvimento de aplicações. O RCP disponibiliza todo o procedimento necessário para permitir que outros *plug-ins* sejam inseridos na aplicação ou para atualizar os que já fazem parte dela. O Eclipse permite que essas ações sejam realizadas sem que a aplicação seja reiniciada, por isso o *framework* é caracterizado como *plug-in-play*.

3.4.2.3. *Graphical Editing Framework* (GEF)

O GEF é um *framework* que permite, a partir de um modelo/tipo de aplicação existente, a criação de um rico editor gráfico. Ele é composto por dois *plug-ins*, org.eclipse.gef e org.eclipse.draw2d, sendo esse último responsável pela exibição da parte gráfica do editor, disponibilizando bordas, cursores, implementações de layouts, entre outros.

Com isso, passa ser possível ao GEF oferecer funcionalidades como barras de ferramentas e paletas para exibição dessas, além da redimensão de objetos e identificação de requisições feitas através de mouse e teclado.

O GEF faz uso do padrão de arquitetura *model-view-controller* (MVC), que tem como característica principal a implementação de funcionalidades diferentes em classes diferentes, buscando maior modularidade e, consequentemente, tornando a manutenção mais fácil e independente, evitando modificações com grande impacto, além de maior isolamento entre parte gráfica (*View*) e modelo lógico (*Model*), sendo o *Controller*, o responsável pela comunicação entre as duas partes.

A figura 10 representa a arquitetura citada acima.

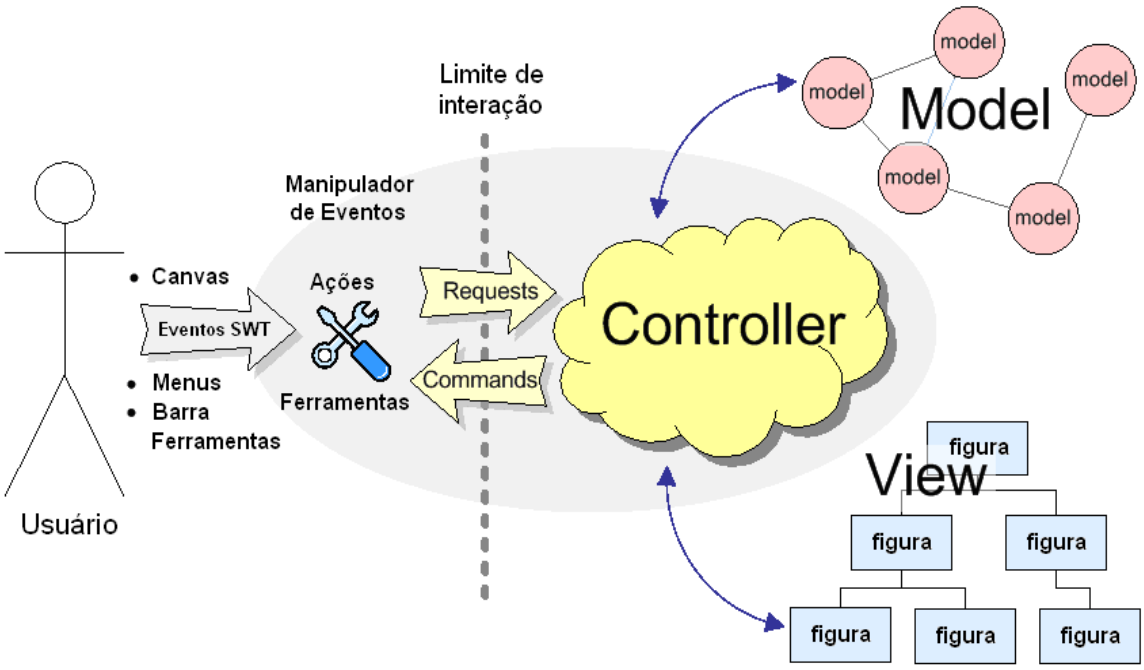


Figura 10: Arquitetura do GEF

Fonte: VAZ, Thiago G. et al, 2007, p. 41 “adaptado de” Eclipse.org, [2006]

3.4.3. XML (*EXtensible Markup Language*)

A XML é uma linguagem estruturada de dados, criada pela *Word Wide Web Consortium* (W3C), devido às limitações apresentadas por outras linguagens, e que oferece simplicidade e facilidade de edição.

Diferente do HTML, o qual especifica como o documento deve ser apresentado na tela por um navegador, o XML (que também faz uso de *tags*) define o conteúdo do documento, se concentrando em sua estrutura, e não na aparência. Outro diferencial dessa linguagem em comparação com o HTML, é que novas *tags* podem ser criadas pelo usuário. Elas serão armazenadas no *Document Type Definition* (DTD), o qual funciona como um glossário.

Os arquivos XML são arquivos texto e têm as regras de formatação mais rígidas que no HTML. Dessa forma, caso o arquivo apresente uma *tag* que tiver sido apenas aberta, por exemplo, ele não poderá ser interpretado e um erro será reportado.

A seguir, podemos ver um exemplo básico criado através dessa linguagem:

```
<PROJETO>
      <TITULO> Simulador de Tráfego de Veículos </TITULO>
      <ORIENTADOR> Paulo Eduardo Santos </ORIENTADOR>
      <AUTORES>
            <AUTOR>
                  <NOME> Felipe Farias Ferrari </NOME>
                  <MATRICULA> 22105026-3 </MATRICULA>
            </AUTOR>
            <AUTOR>
                  <NOME> João Augusto Teixeira Marotti </NOME>
                  <MATRICULA> 22105035-4 </MATRICULA>
            </AUTOR>
            <AUTOR>
                  <NOME> Juliana Boin </NOME>
                  <MATRICULA> 22105048-7 </MATRICULA>
            </AUTOR>
            <AUTOR>
                  <NOME> Silas Shederson de Oliveira </NOME>
                  <MATRICULA> 22105040-0 </MATRICULA>
            </AUTOR>
      </AUTORES>
</PROJETO>
```

3.4.4. XStream

O XStream é uma biblioteca *open source* que permite a geração de arquivos XML a partir de objetos Java, processo chamado serialização. O inverso também é possível e útil em diversos casos, e a restauração dos mesmos pode ser realizada sem qualquer modificação e perda de dados.

Com relação a outras ferramentas, o XStream possui diversas vantagens, como por exemplo, a facilidade de manipulação do XML, já que ele próprio se encarrega de desserializar o arquivo e carregar o objeto, tarefa que poderia ser atribuída ao programador. Com isso, temos vantagem também com a diminuição de erros de programação e aumento da produtividade. Além disso, a integração com outras APIs é permitida.

No caso do projeto aqui proposto, o XStream será aplicado na interpretação de arquivos XML, contendo cenários pré-definidos para o usuário iniciar uma nova simulação. Além disso, ao salvar um perfil de simulação, um arquivo, também com estrutura XML, será gerado para posterior uso, se assim o usuário desejar.

A seguir, podemos acompanhar um exemplo de serialização e deserialização:

1) Definição da classe Projeto, a qual terá sua estrutura transformada em XML:

```
class Projeto{
String titulo = " Simulador de Tráfego de Veículos ";
String orientador = " Paulo Eduardo Santos ";
}
```

2) Código que definirá a serialização, com instância de novo objeto do tipo XStream e definição do nome da *tag* com o uso de alias:

```
XStream xstream = new XStream();

Projeto projeto = new Projeto();

xstream alias("projeto", Projeto.class);

String xml = xstream.toXML(projeto);
```

3) O resultado obtido seria esse:

```
<projeto>

      <titulo> Simulador de Tráfego de Veículos </titulo>

      <orientador> Paulo Eduardo Santos </orientador>

</projeto>
```

4) E em caso de deserialização, podemos usar o seguinte comando:

```
Projeto novoProjeto = (Projeto)xstream.fromXML(xml);
```

Mais informações sobre o XStream podem ser encontradas em sua página oficial, citada nas Referências desse documento.

3.4.5. Prolog

O Prolog, criado na década de 70, é uma linguagem de programação lógica declarativa, onde apenas o problema a ser resolvido é fornecido através de sua estrutura lógica, sem uma maneira específica de chegar à solução, com uso normalmente associado a Inteligência Artificial e Linguística Computacional e que se diferencia das linguagens procedimentais por diversos fatores.

Um deles é o fato de que essa linguagem não possui estruturas de controle, como IF-ELSE, por exemplo, e sim métodos lógicos. Além disso, todos os dados são tratados como sendo de um único tipo.

Para conhecer a solução, o usuário pode realizar consultas a base de dados, o resultado é obtido com a aplicação da lógica de predicados. A estrutura da linguagem é composta por cláusulas (consultas, fatos e regras).

Fatos são compostos por um predicado e seus objetos, e para conclusão da instrução usamos um ponto (.), semelhante ao ponto-vírgula (;) de outras linguagens, conforme Figura 11.

```
disciplina(fisica, engenharia).
```

Figura 11: Exemplo de fato em Prolog

Fonte: Autores

Dessa forma, definimos que física é uma disciplina do curso de Engenharia.

Para variáveis, é importante dizer que, uma vez que um objeto é instanciado a ela, a mesma não será mais modificada. No exemplo da Figura 12, o objeto quimica será atribuído à variável X.

```
disciplina(quimica, engenharia).
disciplina(fisica, engenharia).
disciplina(fisica, informatica).

?- disciplina(X, engenharia).
```

Figura 12: Exemplo de instância em Prolog

Fonte: Autores

O uso de operadores relacionais e aritméticos também é possível e, quando queremos especificar que uma questão é verdadeira se alguma condição for satisfeita, fazemos uso de regras, através do símbolo “ := ” . Para que uma variável receba o valor de uma operação aritmética, existe o operador “is”.

Caso exista mais de uma resposta à consulta, todas elas serão fornecidas se assim o usuário desejar.

Além do Prolog, a programação em lógica de uma forma geral, vem sendo aplicada em diversas áreas, como Sistemas Baseados em Casos (SBCs), Sistemas Especialistas (SEs), Processamento da Linguagem Natural (PLN), entre outros. Essa grande utilização se deve aos diferenciais da linguagem, como soluções heurísticas, estruturas de conhecimento e controle separadas, fácil modificação e inclusão de todas as soluções possíveis, além dos já citados anteriormente. Possuem também capacidade dedutiva, são naturalmente recursivos e possibilitam a obtenção de respostas alternativas.

Com essa base, podemos então, descrever uma das ferramentas que possibilitam a utilização dessa linguagem.

3.4.6. Socket

Em o nosso projeto, usaremos a comunicação via *socket* entre Java e Prolog. O *socket* é o ponto final de um elo de comunicação entre dois (ou mais) processos de programas, os quais acessam uma rede através do protocolo TCP/IP. Ele funciona como uma *interface* entre um processo e a porta TCP de uma máquina, que por sua vez se comunica com outra porta TCP (da mesma ou de outra máquina).

Para que haja conexão utilizando *socket*, cada processo tem uma função inicial estabelecida como servidor ou cliente.

Inicialmente, algum processo que faz o papel do servidor cria um *socket* "passivo" que escolhe uma porta TCP e começa a "escutá-la", à espera de um pedido de conexão de um cliente.

O processo que faz o papel de cliente, por sua vez, tendo a disposição o *hostname* (nome único como um dispositivo é conhecido na rede) em que a aplicação com que deseja se comunicar está rodando e a porta TCP que o servidor está escutando, envia um pedido de conexão.

Nesse pedido, o cliente envia sua identificação (hostname e porta) para que um *socket* "ativo" relacionado à nova conexão seja estabelecido no servidor.

Quando um cliente realiza um pedido de conexão e o mesmo é aceito pelo servidor, uma resposta é retornada ao cliente e um *socket* "ativo" também é estabelecido no mesmo. A conexão está feita.

A partir desse ponto, tanto o cliente quanto o servidor podem escrever e ler a partir de seus sockets.

Em nosso projeto, o processo servidor é o Java. Ele inicia um novo socket que começa a escutar a porta 22105. Em seguida, logo que o processo do Prolog é iniciado (também pelo Java), um pedido de conexão é feito à porta 22105. O pedido é aceito e a conexão será estabelecida.

4. SOLUÇÃO ADOTADA

Através da metodologia proposta para nosso projeto (Figura 13), será possível definir um ambiente com elementos móveis e seus trajetos e, a partir desse ambiente, executar a simulação.

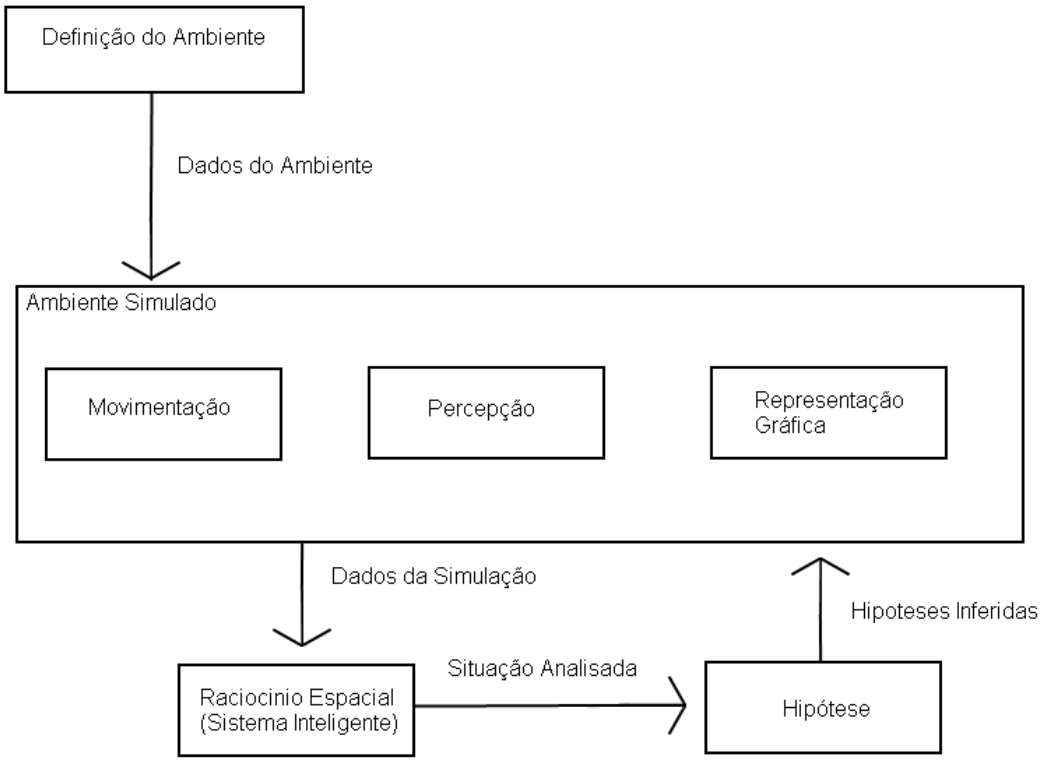


Figura 13: Diagrama da metodologia do sistema

Fonte: Autores

Antes de apresentarmos mais detalhes sobre a solução proposta, é interessante citar as principais diferenças entre nosso projeto e o SRE, as quais podem ser resumidas pela exclusão do uso de Curvas de Bèzier e retirada do Tratamento de Colisão antes implementados. Além disso, fizemos uso de Prolog com transmissão de dados ao Java via Socket.

4.1. Definição do ambiente

Teremos sua definição através de uma interface gráfica semelhante à apresentada na seção 3.1. Um arquivo com a configuração das ruas deve ser carregado e, a partir dele, o usuário poderá definir os elementos móveis do ambiente simulado (veículo observador, pedestre, veículo não inteligente), suas posições (somente dentro das ruas), atributos e trajetos.

Com o início da simulação, esses dados são utilizados para determinar o estado inicial do ambiente e o comportamento dos elementos.

4.2. Ambiente Simulado

O ambiente simulado é um espaço bidimensional representado de um ponto de vista aéreo. Ele guarda as informações dos elementos durante a simulação e é dividido em três outros módulos menores: definição do ambiente, percepção e representação gráfica.

4.2.1. Definição do ambiente

Responsável pelo cálculo da movimentação dos elementos móveis e atualização de suas posições em relação ao próximo ponto de seu trajeto.

Também é responsável por alterar a direção de cada elemento. O veículo observador, em especial, terá o seu ângulo de visão alterado, relacionado à sua direção.

4.2.2. Percepção

Etapa na qual o perfil de profundidade do observador, em um determinado instante, é gerado. Isso é feito através da percepção de visão do veículo observador, sendo ela uma visão linear do ambiente, restringida por seu campo e ângulo de visão.

Ao contrário do SRE, o qual traçava diversas retas radiais a partir do observador, com o objetivo de escanear o que está dentro de seu campo de visão, foi adotada a utilização de relações trigonométricas entre o observador e os obstáculos.

Após separar apenas os objetos inclusos no campo de visão do observador, para cada um deles são definidas equações da reta compreendidas entre o ponto central de cada objeto e o ponto de origem do campo de visão. Tendo essas equações, é possível estabelecer o coeficiente angular para cada uma e atribuir a cada objeto um ângulo de referência em relação ao campo de visão, como ilustra a parte A da figura 14.

Representando-se os objetos por círculos é possível estabelecer também o diâmetro médio de cada um deles. A partir do conhecimento do diâmetro e da distância em relação à origem do campo de visão, utilizamos a equação do diâmetro angular (figura 15) para obter em termos de graus o quanto o objeto está ocupando da visão do observador, como ilustra a parte B da figura 14.

Por fim, para montar a descrição de todo o perfil de profundidade, os objetos são ordenados em relação ao ângulo de referência que possuem. O perfil de profundidade é

gerado varrendo-se todos os objetos tratados e obtendo-se suas distâncias (profundidade) e diâmetros angulares. Entre um objeto e outro pode ser gerado ainda uma descrição de *background*, isto é, a ausência de objeto numa parte do perfil.

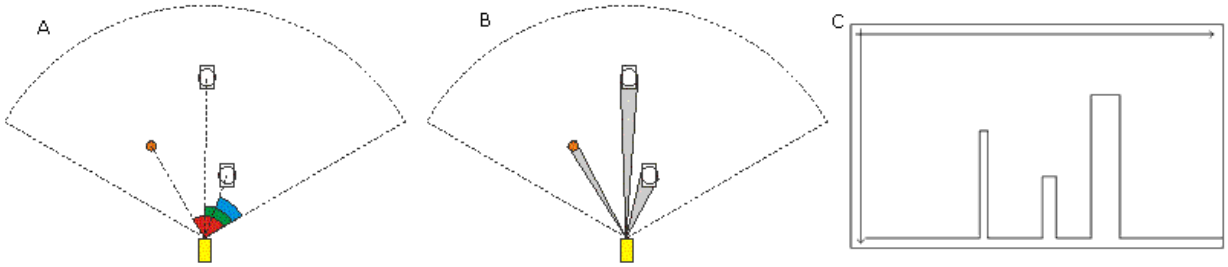


Figura 14: Esquema de Geração do Perfil de Profundidade

Fonte: Autores

Diâmetro Angular = Arco-Seno(Raio / Distância)

Figura 15: Equação do diâmetro angular utilizada

Fonte: Autores

Após todos os passos é possível montar um único axioma descritivo, o qual é fornecido ao Prolog e transformado também em um gráfico, no qual pode ser identificada a distância e posição dos objetos do ponto de vista do observador. Para cada pico representado, a distância pode ser observada por sua altura e o ângulo de abertura, por sua largura.

A figura abaixo representa o Perfil de Profundidade em um instante, onde os picos p e q representam a identificação de dois objetos X1 e X2. Maiores informações sobre o perfil de profundidade podem ser encontradas na seção 3.3.1.

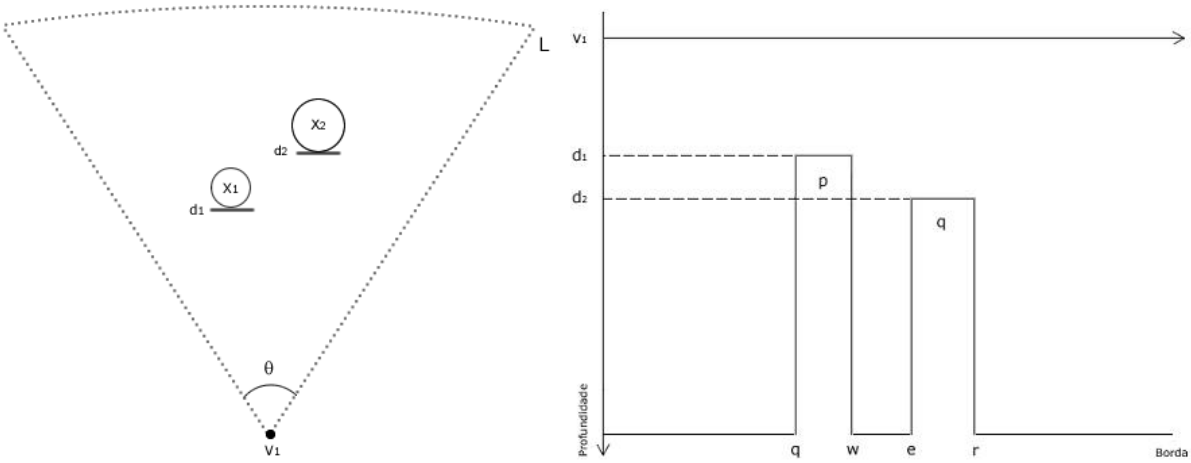


Figura 16: Perfil de Profundidade

Fonte Autores “adaptado de” VAZ, Thiago G. et al, 2007, p. 29

4.2.3. Representação Gráfica

A cada iteração da simulação, esse módulo será responsável por atualizar as imagens e informações de saída exibidas para o usuário. As saídas podem ser divididas em três partes distintas, Visão Aérea, Perfil de Profundidade e Mensagens de Saída, compondo a tela apresentada na Figura 17:

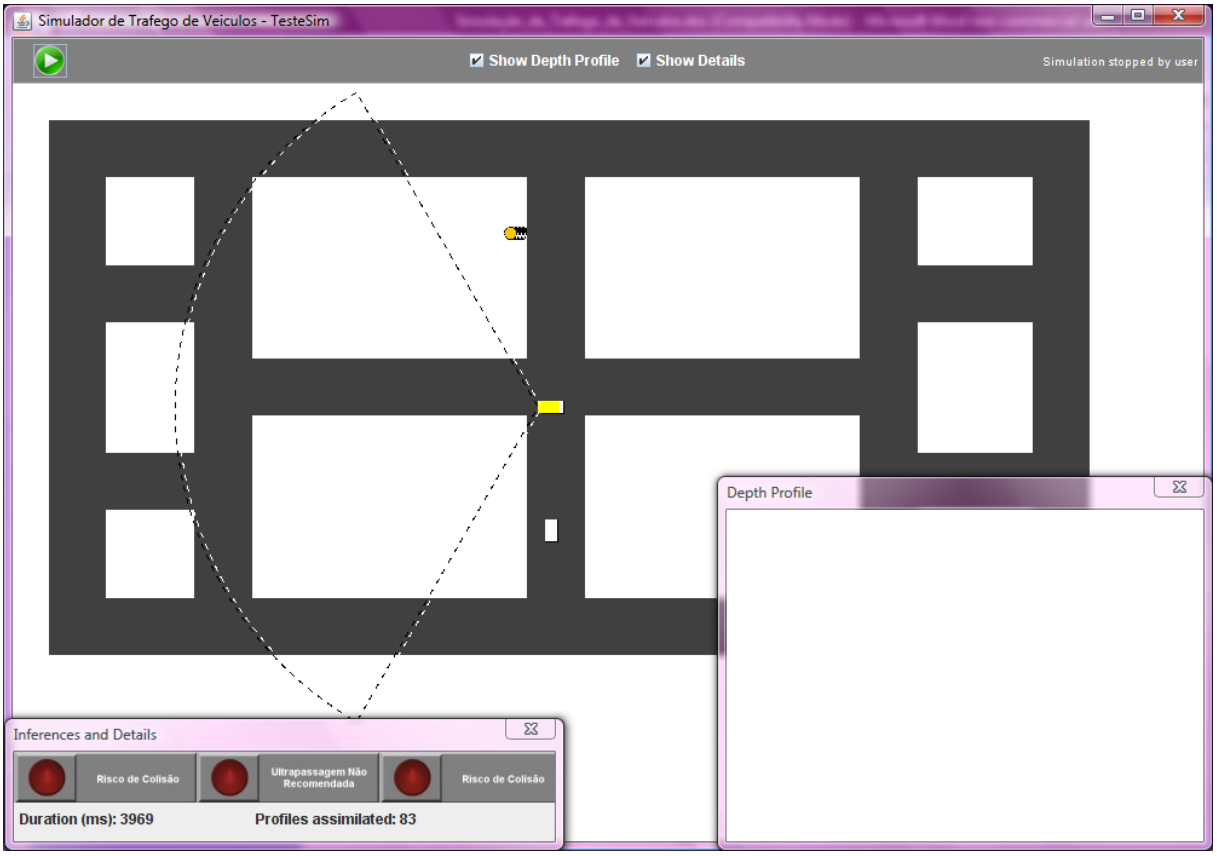


Figura 17: Representação Gráfica

Fonte: Autores

4.2.3.1. Visão Aérea

Exibirá a representação do ambiente simulado de um ponto de vista aéreo, semelhante ao usado na Definição do Ambiente (conforme seção 4.2.1).

4.2.3.2. Perfil de Profundidade

A representação do perfil de profundidade é exibida no instante atual. Ela é semelhante à representação na Figura 18.

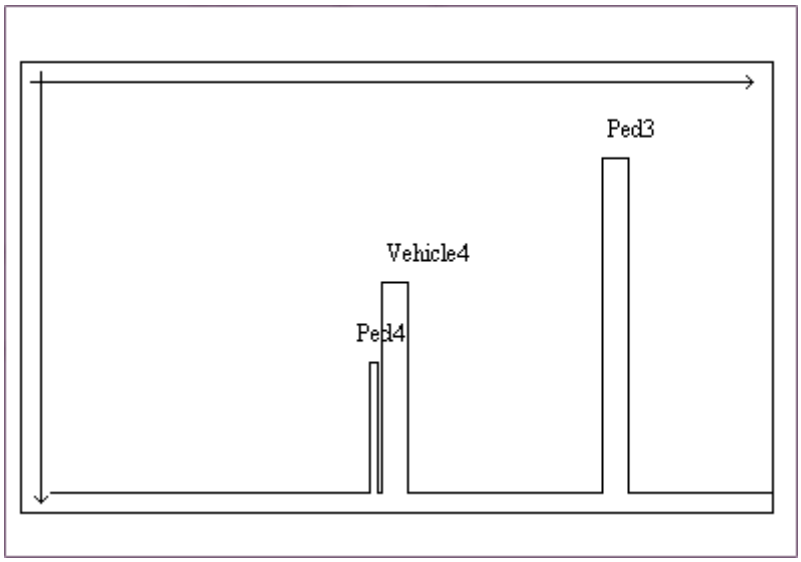


Figura 18: Perfil de Profundidade

Fonte: Autores

4.2.3.3. Mensagens de Saída

Exibirá alertas para as mensagens, de acordo com os dados gerados nos módulos Raciocínio Espacial e Hipóteses. Essas mensagens são referentes à interpretação do cenário no instante atual, como “Frenagem Recomendada”, “Risco de Colisão” e “Ultrapassagem não Recomendada”.

4.3. Raciocínio Espacial

Tendo recebido pares de perfil de profundidade (do instante atual e do instante anterior), esse módulo é capaz de inferir relações espaciais dinâmicas entre as entidades presentes no ambiente. Para isso, o sistema utiliza teorias de raciocínio espacial, como o RCC, o DPC e o DDPC, encontrados na seção 3.3.

4.4. Hipótese

Tendo a saída do modulo anterior, esse módulo será responsável por formular possíveis hipóteses sobre o que foi observado pela visão linear do veículo observador.

Para isso, o sistema realizará várias perguntas à base de conhecimento do Prolog através de axiomas de alto nível. A resposta de cada pergunta será passada de volta para o módulo do Ambiente Simulado para ser exibida ao usuário.

A seção a seguir, referente a Inferências do Prolog, apresenta mais detalhes do processo aqui descrito.

4.5. Inferências do Prolog

Etapa na qual o Prolog, com todos os seus axiomas, é capaz de inferir a respeito do que ocorrerá com o agente.

4.5.1. Interpretação do Perfil de Profundidade

Após o recebimento das informações instantâneas da cena, providas da conexão com o socket, o Prolog interpreta essas informações e realiza uma aproximação da cena em um plano cartesiano.

Nesse plano, o "ponto de visão" do veículo inteligente se situa na origem (0,0). A borda direita da abertura do ângulo de visão se inicia na abscissa direita e termina na borda esquerda, a qual está localizada a 120 graus do início da borda direita.

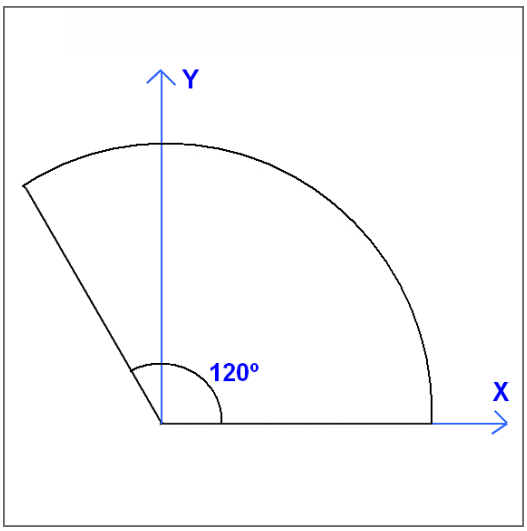


Figura 19: Campo de Visão

Fonte: Autores

Esta situação não se altera durante toda a simulação, ou seja, independente de o veículo inteligente estar se movimentando e defrontando diversas direções, essas referências continuam as mesmas.

Os perfis de profundidade recebidos pelo Prolog são semelhantes a descrição a seguir:

description(t**140**,[*background*((*leftborder*,ped1),-1.0,99.28777), *foreground*(ped1,30.326029,3.6586456), *background*((ped1,*rightborder*),-1.0,17.053581)])

E o Prolog os interpreta da seguinte forma:

O tempo da simulação é definido no primeiro parâmetro, sempre precedido da letra t. No exemplo acima, o tempo é **140**, o que significa que se trata do 140º perfil gerado.

Logo em seguida, é definida uma lista do que é visualizado pelo sensor do veículo inteligente. Seus itens podem ter duas características, *background* e *foreground*.

O *foreground* é recebido quando o sensor do veículo detecta um obstáculo entre ele e o limite de seu raio de visão. Ele traz as seguintes informações: Nome do objeto detectado, distância desse objeto e diâmetro angular do objeto. Em *foreground*(ped1,30.326029,3.6586456), o nome do objeto é ped1, sua distância é 30.326029 metros e sua abertura angular é de 3.6586456 graus. Como as informações recebidas pelo Prolog são escassas, e por motivo de simplificação, ele assume que todos os objetos possuem formas cilíndricas.

O *background* é o oposto do *foreground*, quando o sensor do veículo não detecta nenhum obstáculo até o fim do limite de seu raio de visão. Ele é delimitado por outros objetos e/ou pelas bordas de visão do veículo. Em *background*((*leftBorder*, ped1),-1.0,99.28777), o

intervalo sem obstáculos vai da borda esquerda (*leftBorder*) até o objeto 1 (ped1), sua distância, por definição, é -1.0 e sua abertura angular é de 99.28777 graus.

A partir das informações combinadas do *foreground* e *background*, é possível saber a posição relativa dos objetos detectados (X e Y no plano cartesiano delimitado pelo Prolog, explicado anteriormente) e o seus diâmetros em metros.

4.5.2. Previsões

Tendo a posição relativa e o tamanho dos objetos no instante atual e em alguns instantes no passado, é possível saber a situação atual do objeto (*extending*, *shrinking*, *vanishing*, *appearing*, explicados na seção 3.3.4 - Cálculo de perfil de profundidade dinâmico (DDPC) - e *stationary*, quando o objeto se mantêm na mesma posição) em relação ao veículo inteligente, e com isso, realizar uma previsão referente a localização do objeto em alguns instantes futuros. Para poupar processamento desnecessário, apenas os objetos que apresentarem a situação "*extending*" terão sua previsão feita, já que são esses os que realmente importam para os cálculo de colisão, ultrapassagem e frenagem.

A previsão é feita da seguinte forma: selecionam-se todos os objetos com situação *extending*, no instante em que se deseja fazer a previsão. De cada objeto selecionado, é analisado um número determinado de instantes no passado e, a partir daí, se obtém um delta X e um delta Y, ou seja, a média de quanto o objeto se movimenta nos eixos X e Y (no plano cartesiano criado pelo Prolog) a cada detecção do sensor do veículo.

Assumindo que o objeto manterá a direção e a velocidade que apresentou nos instantes anteriores, basta aplicar essa mesma direção e velocidade no objeto em questão a partir do instante em que se quer fazer a previsão e, dessa forma, obtereremos a provável posição do objeto em instantes futuros. Vale ressaltar que essa previsão precisa ser calculada a cada nova detecção do sensor do veículo, pois os objetos podem mudar sua rota a qualquer momento, invalidando as previsões feitas anteriormente.

4.5.3. Detecção de Colisão, Ultrapassagem Insegura e Frenagem Recomendada

Para realizar a detecção de "colisão", "ultrapassagem insegura" e "frenagem recomendada", é necessário ter passado pelas duas etapas anteriores (Interpretação do Perfil de Profundidade e Previsão). Analisam-se então todas as previsões dos objetos.

Para isso, utilizamos as "retas virtuais" no Prolog. Elas funcionam de modo a delimitar o formato do veículo inteligente no plano cartesiano, assim como a área de ultrapassagem insegura nas duas laterais do veículo inteligente. As retas podem ser interpretadas da seguinte forma:

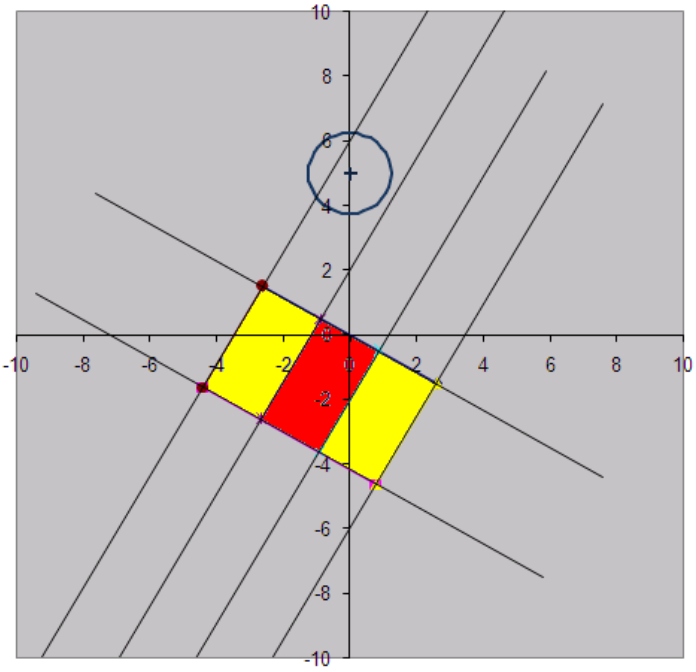


Figura 20: Retas Virtuais

Fonte: Autores

Na imagem acima, podemos visualizar as retas virtuais (em preto) e um objeto próximo (circunferência). A área de colisão é a área em vermelho. Ela tem as exatas medidas do veículo inteligente (agente). As áreas laterais em amarelo são as áreas de ultrapassagem insegura.

Para saber se haverá colisão, basta analisar a posição dos objetos em instantes futuros. Se em alguns desses instantes próximos, o objeto tiver uma parte de si dentro da área de colisão, essa estará detectada, conforme Figura 21:

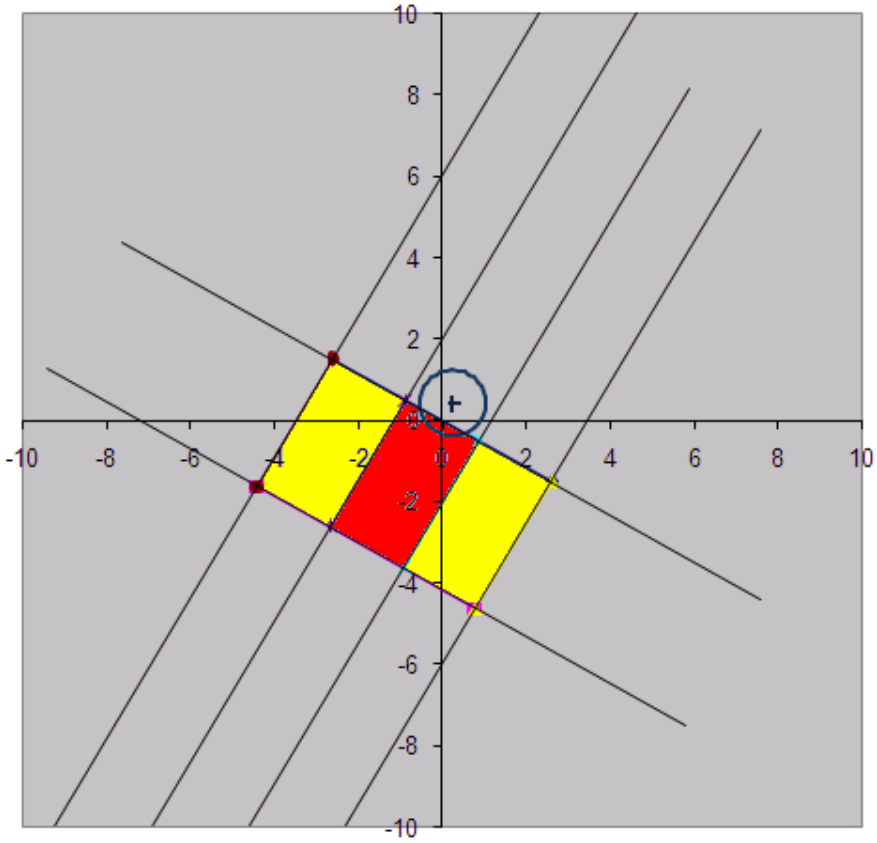


Figura 21: Detecção de Colisão

Fonte: Autores

O mesmo vale para a situação de ultrapassagem, porém levando-se em conta a área de ultrapassagem insegura. Para a situação de frenagem necessária, também se usa a área de colisão, porém, levando em consideração instantes mais distantes no futuro do que os da situação de colisão.

4.5.4. Principais Axiomas

Aqui são apresentados os axiomas que serão usados em nosso projeto para dedução e inferências de outras verdades. Mais detalhes podem ser obtidos no Apêndice D desse documento.

Constantes:

1) forecastFrames(Number)

Obtém o número de instantes que serão usados na previsão, ou seja, caso *Number* seja 30, obteremos a previsão da situação em até 30 instantes no futuro.

2) previousFrames(CurrentTime, NumberOfFrames)

Obtém o número de instantes no passado que serão analisados na previsão.

3) collisionFrames(InitialFrame, EndingFrame)

Obtém o instante inicial e o instante final em que será feito o reconhecimento da colisão.

4) overtakingFrames(InitialFrame, EndingFrame)

Obtém o instante inicial e o instante final em que será feito o reconhecimento da ultrapassagem insegura.

5) breakingFrames(InitialFrame, EndingFrame)

Obtém o instante inicial e o instante final em que será feito o reconhecimento da frenagem nessessária.

6) horizFrontLine(A, B, C)

Obtém A, B e C da equação geral da reta referente a linha virtual horizontal frontal (HFL).

7) horizBackLine(A, B, C)

Obtém A, B e C da equação geral da reta referente a linha virtual horizontal traseira (HBL).

8) vertOuterRightLine(A, B, C)

Obtém A, B e C da equação geral da reta referente a linha virtual vertical direita externa (VOR).

9) vertInnerRightLine(A, B, C)

Obtém A, B e C da equação geral da reta referente a linha virtual vertical direita interna (VIR).

10) vertInnerLeftLine(A, B, C)

Obtém A, B e C da equação geral da reta referente a linha virtual vertical esquerda interna (VIL).

11) vertOuterLeftLine(A, B, C)

Obtém A, B e C da equação geral da reta referente a linha virtual vertical esquerda externa (VOL).

12) horizDist(Distance)

Obtém a distância entre as duas retas virtuais horizontais.

13) vertOuterDist(Distance)

Obtém a distância entre a reta virtual vertical externa e a reta virtual vertical interna.

14) vertInnerDist(Distance)

Obtém a distância entre as reta virtuais verticais internas.

Base de Conhecimento:

1) peak(Body, Depth, Size, Ang, Time)

Representa um pico do objeto *Body*, de Distância *Depth*, diâmetro *Size*, Ângulo relativo Ang e instante *Time*.

2) situation(Body, Time, Situation)

Representa a situação do objeto *Body* no instante *Time*.

Inferências:

1) holdsAt(Body, CurrentTime, Situation)

Tenta descobrir qual é a situação (*extending*, *shrinking*, *appearing*, *vanishing* ou *stationary*) do objeto *Body* no instante *CurrentTime*.

2) extending(Body, CurrentTime)

Verifica se o objeto *Body* está expandindo no instante *CurrentTime*.

3) shrinking(Body, CurrentTime)

Verifica se o objeto *Body* está diminuindo no instante *CurrentTime*.

4) appearing(Body, CurrentTime)

Verifica se o objeto *Body* aparece repentinamente no instante *CurrentTime*.

5) vanishing(Body, CurrentTime)

Verifica se o objeto *Body* desaparece repentinamente no instante *CurrentTime*.

6) stationary(Body, CurrentTime)

Verifica se o objeto *Body* não está se aproximando nem se distanciando (andando na mesma velocidade) no instante *CurrentTime*.

7) forecast(CurrentTime)

prepareAssertForecast([Peak|PeakList])

assertForecast(peak(Body, Depth, Size, AngDist, CurrentTime), DeltaX, DeltaY, DeltaSize, Frames)

Esses três axiomas inferem as posições futuras de todos os picos detectados no instante *CurrentTime*, assim como insere na base de conhecimento a posição futura desses picos.

8) retriveDeltas(peak(Body, Depth, Size, Ang, Time), PreviousFrames, DeltaX, DeltaY, DeltaSize, TotalX, TotalY, TotalSize, TotalFrames)

Obtém o delta de X e Y da movimentação do objeto *Body*, analisando sua posição em instantes passados.

9) queryObjectInColisionRoute(Time)

queryObjectInColisionRouteLoopTime(SituationList, InitTime, EndTime)

queryObjectInColisionRouteLoopPeak([situation(Body, _, _)|Tail], Time)

verifyCollision(HFL, HBL, VIR, VIL)

Esses quatro axiomas inferem se o instante *Time* apresenta a situação de risco de colisão.

10) queryUnsafeOvertaking(Time)

queryUnsafeOvertakingLoopTime(SituationList, InitTime, EndTime)

queryUnsafeOvertakingLoopPeak([situation(Body, _, _)|Tail], Time)

verifyUnsafeOvertaking(HFL, HBL, VIR, VIL, VOR, VOL)

Esses quatro axiomas inferem se o instante *Time* apresenta a situação de ultrapassagem insegura.

11) queryVelocityReduceNeeded(Time)

Esse axiomas inferem se o instante *Time* apresenta a situação de frenagem necessária.

Obs: Não existe um axioma queryVelocityReduceNeededLoopTime, queryVelocityReduceNeededLoopPeak ou verifyReductionNeeded, pois o queryObjectInColisionRouteLoopTime, queryObjectInColisionRouteLoopPeak e verifyCollision têm a mesma função, e por isso são reutilizados dentro do queryVelocityReduceNeeded.

Auxiliares:

1) straightLineEquation((X1, Y1), (X2,Y2), A, B, C)

Obtém A, B e C da equação geral da reta que passa pelos pontos (X1, Y1) e (X2,Y2).

2) distancePointToLine((X,Y), A, B, C, Dist)

Obtém a distância Dist do ponto (X, Y) à uma reta, definida por sua equação geral de A, B e C.

3) relativeLocation(Ang, Dist, (X, Y))

Obtém a posição relativa de um ponto (X,Y) dado o seu ângulo Ang em relação à abscissa e a distância Dist à origem (0,0).

4) quarter(X,Y, NumberQuarter)

Dado X e Y de um ponto, é possível saber em qual quadrante do círculo trigonométrico *NumberQuarter* ele está localizado.

5) angularLocation((X, Y), Ang, Dist)

Obtém o ângulo Ang em relação à abcissa e a distância Dist à origem (0,0) do ponto (X, Y).

6) getSizeAndCenterDepth(Depth, AngDiam, Size, CenterDepth)

Obtém o diâmetro Size e a distância de seu ponto central *CenterDepth*, em relação à origem (0,0) de uma circunferência, cuja distância de seu exterior seja *Depth* e seu diâmetro angular seja AngDiam.

7) distancesPeakToDefinedLines(peak(Body, Depth, Size, Ang, Time), HFL, HBL, VOR, VIR, VIL, VOL)

Obtém as distâncias inversas do objeto *Body* em relação às retas virtuais. Ou seja, se a reta horizontal frontal (HFL) passar dentro do objeto *Body*, essa distância será positiva, do contrário, ela será negativa.

8) oneGreaterThanZero(A, B)

Verifica se ou A ou B é maior que zero.

9) description(Time, [DepthProfileList])

Dada a lista *DepthProfileList* das características de um perfil de profundidade do instante Time, obtém todos os objetos peak (*Body*, *Depth*, *Size*, Ang, *Time*) e os adiciona na base de conhecimento.

Interfaces com o Java:

1) performFunction(Function, FunctionArguments, Response)

Realiza a função *Function* requisitada pelo Java (onde 0 = Inserção de um perfil de profundidade na base de dados, 1 = Verificar situação de colisão, 2 = verificar situação de ultrapassagem insegura, 3 = Verificar situação de frenagem necessária, 4 = Fechar conexão) usando os argumentos *FunctionArguments* e retornando a resposta *Response* (do tipo boolean).

2) connect(Port)

Se conecta à porta definida em Port para se comunicar com o Java.

5. PLANO DE TESTE

Os cenários de teste, os quais estão descritos no Apêndice E, foram definidos visando obter qualidade para o aplicativo desenvolvido, com a identificação de problemas e definição de taxa de erros e acertos, localizados durante as simulações executadas. As situações citadas foram construídas com base no escopo definido para o projeto.

5.1. Resultados

Com relação às funcionalidades dos módulos Editor e Simulador e levando em conta o que foi proposto no escopo desse projeto, pudemos observar que os testes apresentaram alta taxa de sucesso. Ações como execução de simulação sem a existência de um veículo observador, como também alterações nas ruas contidas nos cenários, não estão sendo liberadas ao usuário. Isso para que as situações geradas representem um nível próximo a realidade e estejam de acordo com o que o simulador espera para suas execuções.

Além disso, com a execução dos cenários citados, foi possível identificar as taxas de erros e acertos referentes aos alertas emitidos pelo simulador. Para que fosse possível confirmar a reação do aplicativo, assumimos que as simulações criadas deveriam ou não representar situações de risco em determinados momentos, e pudemos verificar o comportamento e a saída gerados pelo sistema.

Para os cenários com representação de riscos, o Simulador apresentou correto comportamento, ou seja, exibiu todos os alertas necessários durante as execuções. Com relação a situações onde não deveriam ser identificados riscos, obtivemos taxa de 99% de

acerto, através da qual constatamos que para uma situação específica de colisão, o simulador identificou risco inadequadamente.

Para medição de performance, o aplicativo teve sua execução realizada com alto número de objetos e também situações de risco, onde seria necessária uma eficiente troca de mensagens entre Prolog e Java, através do uso de Socket. Para essas situações, em nenhum momento o Simulador apresentou falhas ou mensagens de erro, mesmo naquelas em que ele foi sobrecarregado pela quantidade de objetos inseridos. Todos os retornos do Prolog foram realizados em tempo hábil para que o alerta fosse exibido pelo Simulador e que uma possível tomada de decisões fosse realizada.

Ainda com relação à performance, pudemos verificar que, embora a quantidade de perfis passados ao Prolog seja semelhante em todos os casos, o tempo de simulação é proporcional a quantidade de objetos incluídos no editor.

# Simulação	Qtde de Objetos	Qtde de Perfis	Tempo de Simulação (ms)
1	10	84	12947
2	20	84	16086
3	30	82	17157
4	40	82	18802
5	50	82	20818

Deve ser levado em conta, o fato de que a trajetória percorrida pelo agente foi a mesma para todos os casos, caso contrário poderíamos ter tempos maiores independentemente da quantidade de objetos, já que a simulação é interrompida quando o veículo observador alcança o fim de seu trajeto.

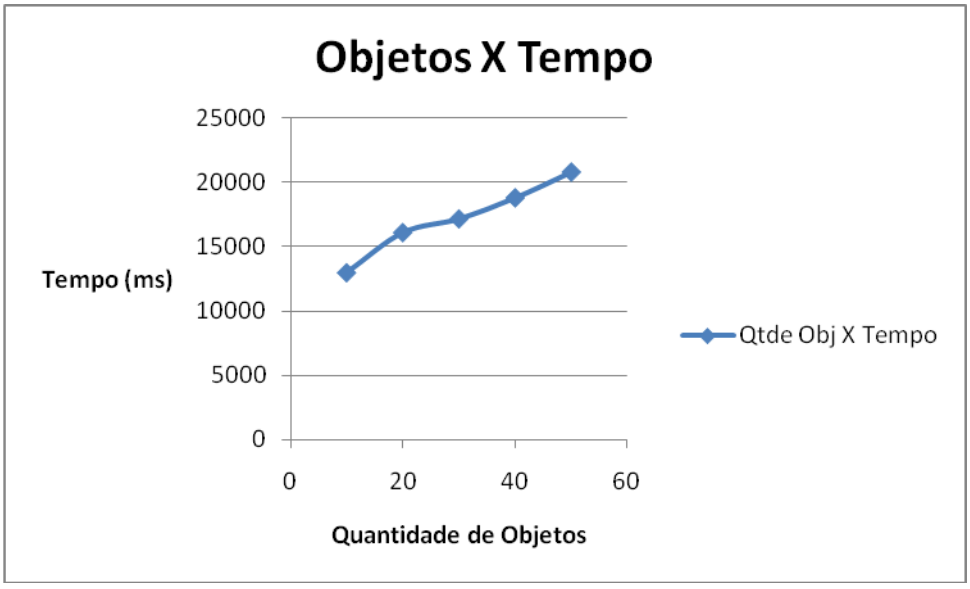


Figura 22: Teste de Performance

Fonte: Autores

Através do gráfico acima podemos visualizar o aumento de tempo de simulação, ocorrido com a inclusão de mais objetos na simulação.

6. CONSIDERAÇÕES FINAIS

Nessa seção apresentaremos uma visão geral sobre o desenvolvimento do projeto, além de possíveis continuidades para o trabalho.

6.1. Trabalhos Futuros

Assim como demos continuidade ao projeto Simulador de Raciocínio Espacial, propomos aqui algumas extensões para nosso trabalho.

É interessante que, posteriormente, seja desenvolvida a tomada de decisões por parte do Agente, o qual apenas identifica situações de erro e exibe alertas ao usuário no Simulador de Tráfego de Veículos. O tratamento de Colisões e outras possíveis situações, pode ser implementado para que seja possível observar o comportamento do Agente nesses casos, verificando quais decisões seriam tomadas e com o objetivo de evitar a ocorrência de acidentes durante as execuções.

Além disso, propomos a criação de um Simulador 3D e a possibilidade de personalização de cenários por parte do usuário, que atualmente escolhe entre os três disponibilizados para a simulação.

6.2. Conclusão

De acordo com a realização dos testes realizados, em comparação com os resultados que foram definidos como esperados desde o início do projeto, podemos dizer que foi possível atingir nosso objetivo.

Com a continuação do SRE, o qual já possuía um ambiente capaz de definir uma situação espacial e executá-la, além de gerar dados qualitativos para serem analisados pelas teorias de raciocínio espacial, pudemos aplicar a funcionalidade ao tráfego de veículos e fazer com que esses dados fossem analisados em tempo real, através da troca de mensagens entre Java e Prolog e, exibindo assim, alertas ao usuário com a indicação de situações de risco nas quais o agente se encontrava.

Podemos dizer que uma das etapas mais trabalhosas do projeto, foi o próprio estudo das tecnologias e técnicas a serem utilizadas. Isso porque, por se tratar de um aplicativo com tratamento das situações em tempo real, teríamos a performance como foco, já que a exibição dos alertas deveria ocorrer antes mesmo da situação se realizar.

Dessa forma, esperamos que o projeto seja um incentivo para novos estudos nessa área de pesquisa e que extensões para o Simulador de Tráfego de Veículos sejam criadas.

REFERÊNCIAS

BYSTRONSKI, M. **Um ambiente para integração de ferramentas de tolerância a falhas.** 2004. 35 f. Monografia (Graduação em Engenharia da Computação) – Universidade Federal do Rio Grande do Sul, Porto Alegre.

ECLIPSE.ORG. **GEF**. [200-]. Disponível em: < http://www.eclipse.org/gef/>
Acesso em: 07 set 2008

ECLIPSE.ORG. **PDE**. [200-]. Disponível em: <http:// www.eclipse.org/pde>. Acesso em: 13 abr 2008, 19:06.

ECLIPSE.ORG. **Rich Client Plataform**. [200-]. Disponível em: <http://www.eclipse.org/home/categories/rcp.php>. Acesso em: 04 jun 2008.

HERZOG, O. et al. **Qualitative Mapping of Sensory Data for Intelligent Vehicles**. Workshop on Agents in Real-Time and Dynamic Environments at the 19th International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, p. 51-60, jul. 2005.

JOHN FISHER. **Prolog Tutorial**. Disponível em: <http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/8_4.html>. Acesso em: 16 ago 2008, 14:47.

LEANDRO DAFLON. **Introdução à Plataforma Eclipse**. Disponível em: <http://web.teccomm.les.inf.puc- rio.br:8080/eclipse/files/IntroducaoAPlataformaEclipse%5BDaflon03%5D.pdf>. Acesso em: 11 mai 2008, 11:28.

Levesque Hector et al. Cognitive Robotics. In:_____. **Handbook of knowledge representation**. [S.I.] Elsevier, 2007. cap. 24, p. 1-13

PEDRO CARVALHO. **Curiosidades da Matemática**. Disponível em: <http://pedrofariacarvalho.blogspot.com/2008/08/equao-de-reta.html>. Acesso em: 28 ago 2008, 18:21.

REITER Raymond. **Knowledge in action**. Logical Foundations for Specifying and Implementing Dynamical Systems. [S.I.] Mit Pr, 2001

RUSSELL, S. NORVIG, P. Agentes Lógicos. In_____: **Inteligência Artificial**. 2. ed. [S.I.] Campus, 2003. cap. 7, p. 189-231.

RUSSELL, S. NORVIG, P. Lógica de Primeira Ordem. In_____: **Inteligência Artificial**. 2. ed. [S.I.] Campus, 2003. cap. 8, p. 232-262.

RUSSELL, S. NORVIG, P. Representação de Conhecimento. In_____: **Inteligência Artificial**. 2. ed. [S.I.] Campus, 2003. cap. 10, p. 263-356.

SANTOS, M. V. dos, et al. **Logic-Based Interpretation of Geometrically Observable Changes Occuring in Dynamic Scenes**; 2007 (to appear)

SANTOS, P. Reasoning about Depth and Motion from an Observer´s Viewpoint. In: _____: **Spatial Cognition & Computation**, [S.I.] Taylor & Francis, 2007, vol. 7, p. 133-178.

SANTOS, P. **Spatial Reasoning and abductive interpretation of sensor data obtained by a mobile robot in a dynamic environment**. 2003. 150 f. Tese (Doutorado em Inteligência Artificial). Imperial College of London, Londres.

SOUCHANSKI, M., SANTOS, P. **Reasoning about Dynamic Depth Profiles**; 2008 (manuscript)

SUKTHANKAR, R. **Situation Awareness For Tactical Driving.** 1997**.** PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh.

VASCONCELOS, A.T. **Ferramenta para construção de linha de produtos no Eclipse.** 2005. 65 f. Monografia (Graduação em Ciência da Computação) – Universidade Federal de Pernambuco, Recife.

VAZ, Thiago G. et al. **Simulador para raciocínio espacial.** 2007. 118 f. Monografia (Graduação em Ciência da Computação) – Centro Universitário da FEI, São Bernardo do Campo.

XSTREAM.CODEHAUS.ORG. **XStream**. Disponível em: <http://xstream.codehaus.org>. Acesso em: 26 mai 2008, 15:55.

APÊNDICE A – ANÁLISE DE REQUISITOS

Requisitos de Negócio:

Identificador:	RN01
Requisito:	Permitir o uso de cenários, isto é, composições de ruas, disponibilizados em arquivos.
Tipo:	Funcional
Classificação:	Volátil

Identificador:	RN02
Requisito:	Calcular perfis de profundidade entre o ponto de vista do veículo observador e as demais entidades presentes em seu campo de visão.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RN03
Requisito:	Realizar os cálculos de situações através dos perfis de profundidade, gerando inferências para os axiomas.
Tipo:	Não Funcional
Classificação:	Estável

Identificador:	RN04
Requisito:	Utilizar os axiomas tradicionais do ramo de cálculo de situações.
Tipo:	Não Funcional
Classificação:	Estável

Identificador:	RN05
Requisito:	Permitir salvar os perfis de simulação em arquivos para posterior utilização.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RN06
Requisito:	O campo de visão do veículo observador deve ser de 120 graus.
Tipo:	Não Funcional
Classificação:	Volátil

Identificador:	RN07
Requisito:	A velocidade de Pedestres e Veículos será limitada entre 0 e 4.
Tipo:	Não Funcional
Classificação:	Volátil

Identificador:	RN08
Requisito:	Devem ser exibidos alertas em tempo real, a fim de prevenir situações de risco.
Tipo:	Funcional
Classificação:	Volátil

Requisitos de Usuário:

Identificador:	RU01
Requisito:	O usuário pode incluir entidades, tais como pedestres e veículos, no cenário.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RU02
Requisito:	O usuário pode mover as entidades existentes no cenário.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RU03
Requisito:	O usuário pode excluir entidades existentes no cenário.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RU04
Requisito:	O usuário pode incluir apenas um veículo observador no cenário.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RU05
Requisito:	O usuário não poderá rotacionar os objetos.
Tipo:	Funcional
Classificação:	Volátil

Identificador:	RU06
Requisito:	O usuário pode alterar as propriedades das entidades e do perfil de simulação.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RU07
Requisito:	O usuário pode desfazer / refazer alterações no editor.
Tipo:	Funcional
Classificação:	Volátil

Identificador:	RU08
Requisito:	O usuário pode definir os pontos do trajeto a ser realizado pelas entidades.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RU09
Requisito:	O usuário pode abrir um perfil de simulação já salvo.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RU10
Requisito:	O usuário pode salvar o perfil de simulação criado.
Tipo:	Funcional
Classificação:	Estável

Requisitos de Sistema:

Identificador:	RS01
Requisito:	Permitir que seja possível o uso do sistema em qualquer plataforma (portabilidade).
Tipo:	Não Funcional
Classificação:	Volátil

Identificador:	RS02
Requisito:	Utilizar o padrão XML para os arquivos e outras formas de interface do sistema.
Tipo:	Não Funcional
Classificação:	Estável

Identificador:	RS03
Requisito:	Utilizar o plugin SWI-Prolog para o cálculo de situações através dos axiomas gerados.
Tipo:	Não Funcional
Classificação:	Volátil

Identificador:	RS04
Requisito:	Utilizar a tecnologia de sockets para a comunicação entre as aplicações Java e o SWI-Prolog.
Tipo:	Não Funcional
Classificação:	Volátil

Identificador:	RS05
Requisito:	O sistema deve exibir a representação da simulação a partir do perfil de simulação criado pelo usuário.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RS06
Requisito:	O sistema deve exibir o perfil de profundidade em tempo real.
Tipo:	Não Funcional
Classificação:	Volátil

Identificador:	RS07
Requisito:	Os veículos devem respeitar os caminhos definidos pelo usuário em tempo de criação.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RS08
Requisito:	O sistema valida a localização dos veículos e de seus pontos de trajeto no momento da inclusão. A inclusão não será permitida fora dos limites da rua.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RS09
Requisito:	Os veículos não podem sair dos limites da rua em tempo de simulação.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RS10
Requisito:	Os veículos devem seguir sua trajetória passando por todos os pontos definidos no perfil de simulação até o ponto final.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RS11
Requisito:	A simulação deve terminar quando o veículo observador alcançar o último ponto de trajeto definido.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RS12
Requisito:	A resolução mínima de vídeo deve ser configurada como 1024x768.
Tipo:	Funcional
Classificação:	Estável

Requisitos Inversos (não serão atendidos):

Identificador:	RI01
Requisito:	O sistema não toma decisões a partir das hipóteses levantadas ou reações válidas encontradas.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RI02
Requisito:	O usuário não tem a possibilidade de editar ou criar cenários. Eles já estarão prontos e disponibilizados em arquivos.
Tipo:	Funcional
Classificação:	Estável

Identificador:	RI03
Requisito:	O sistema não calcula ou aperfeiçoa os movimentos dos veículos e pedestres através das leis de física.
Tipo:	Não Funcional
Classificação:	Estável

APÊNDICE B – ESPECIFICAÇÕES DE CASOS DE USO

INFORMAÇÕES GERAIS DO CASO DE USO			
Identificação:	**UC - 01**	Nome:	**Manter Simulação**
Descrição:	**O sistema deve permitir a criação e edição de uma simulação.**		

ESPECIFICAÇÃO DO CASO DE USO		
Atores:	Identificador	Nome/Descrição
	AT - 01	**Usuário**

Fluxo normal	Ação do ator	Ação do sistema
	1. Selecionar a opção para criar nova simulação	
		2. Simulação criada e carregada na tela
	3. Alterar o que deseja	
	4. Selecionar a opção para salvar simulação	
		5. Simulação gravada

Fluxo alternativo 1: Editar Simulação	Ação do ator	Ação do sistema
	1. Selecionar a opção para abrir simulação	
		2. Simulação carregada na tela
	3. Alterar o que deseja	
	4. Selecionar a opção para salvar simulação	
		5. Simulação gravada

Fluxo alternativo 2: Erro Abertura	Ação do ator	Ação do sistema
	1. Selecionar a opção para abrir simulação	
		2. Erro na abertura do arquivo

Fluxo alternativo 3: Erro gravação	Ação do ator	Ação do sistema
	1. Selecionar a opção para abrir simulação	
		2. Erro na gravação do arquivo

INFORMAÇÕES GERAIS DO CASO DE USO			
Identificação:	**UC - 02**	Nome:	**Manter entidades**
Descrição:	**O sistema deve permitir a inclusão, edição e exclusão de entidades (pedestres, veículos e observador) em uma simulação.**		
ESPECIFICAÇÃO DO CASO DE USO			
Atores:	Identificador	Nome/Descrição	
	AT - 01	**Usuário**	
Pré-condições:	Nome/Descrição		
	É necessário que uma simulação seja criada		
Fluxo normal	Ação do ator		Ação do sistema
	1. Selecionar a opção de adicionar entidade		
			2. Entidade inserida na simulação
Fluxo alternativo 1: Alteração	Ação do ator		Ação do sistema
	1. Selecionar uma entidade da simulação		
	2. Alterar as propriedades da entidade		
			3. Observador alterado na simulação
Fluxo alternativo 2: Exclusão	Ação do ator		Ação do sistema
	1. Selecionar a entidade da simulação		
	2. Selecionar a opção de excluir a entidade		
			3. Entidade excluída da simulação
Fluxo alternativo 3: Mais de um observador	Ação do ator		Ação do sistema
	1. Selecionar a opção de adicionar observador		
			2. Sistema não permite, pois já existe um na simulação
Fluxo alternativo 4: Dados Inválidos	Ação do ator		Ação do sistema
	1. Selecionar um observador da simulação		
	2. Alterar as propriedades do observador		
			3. Sistema rejeita pois o dado informado está fora do domínio da propriedade

INFORMAÇÕES GERAIS DO CASO DE USO			
Identificação:	**UC - 03**	Nome:	**Definir Percurso**
Descrição:	**O sistema deve permitir a definição dos pontos de cada trajetória, a ser seguida por uma entidade (objetos e observador)**		

ESPECIFICAÇÃO DO CASO DE USO		
Atores:	Identificador	Nome/Descrição
	AT - 01	**Usuário**

Pré-condições:	Nome/Descrição
	É necessário ter uma entidade criada.

Fluxo normal	Ação do ator	Ação do sistema
	1. Selecionar um objeto ou observador	
	2. Definir os pontos do percurso	
		3. Percurso definido

Fluxo alternativo 1: Alteração	Ação do ator	Ação do sistema
	1. Selecionar um ponto do percurso	
	2. Alterar a posição do ponto	
		3. Posição alterada

Fluxo alternativo 2: Exclusão	Ação do ator	Ação do sistema
	1. Selecionar um ponto do percurso	
	2. Selecionar a opção de excluir o ponto	
		3. Ponto excluído

INFORMAÇÕES GERAIS DO CASO DE USO			
Identificação:	**UC - 04**	Nome:	**Carregar Cenário**
Descrição:	**O sistema deve permitir que os cenários pré-definidos sejam carregados.**		

ESPECIFICAÇÃO DO CASO DE USO		
Atores:	Identificador	Nome/Descrição
	AT - 01	**Usuário**

Pré-condições:	Nome/Descrição
	É necessário que os cenários (mapas de ruas) tenham sido disponibilizados em xml ou mesmo que uma simulação tenha sido anteriormente salva pelo usuário.
Pós-condições:	Nome/Descrição
	Em caso de simulação salva anteriormente, além de mapa de rua, serão carregadas entidades e suas trajetórias (se tiverem sido definidas). Nesse caso, o usuário tomará ações de exclusão e edição com os objetos já existentes, além de poder definir novos.

Fluxo normal	Ação do ator	Ação do sistema
	1. Selecionar a opção para iniciar nova simulação	
	2. Selecionar uma das opções de cenário	
		3. Cenário carregado na tela
Fluxo alternativo 1: Simulação salva	Ação do ator	Ação do sistema
	1. Selecionar a opção para abrir simulação	
	2. Apontar caminho onde arquivo foi salvo	
		3. Cenário carregado na tela
Fluxo alternativo 2: Erro abertura	Ação do ator	Ação do sistema
	1. Selecionar a opção para abrir simulação	
	2. Apontar caminho onde arquivo foi salvo	
		3. Erro ao tentar carregar cenário

INFORMAÇÕES GERAIS DO CASO DE USO			
Identificação:	**UC - 05**	Nome:	**Executar Simulação**
Descrição:	**O sistema deve permitir a execução da simulação**		

ESPECIFICAÇÃO DO CASO DE USO		
Atores:	Identificador	Nome/Descrição
	AT - 01	**Usuário**

Pré-condições:	Nome/Descrição
	É necessário que uma simulação tenha sido criada. Além disso, entidades devem ter sido incluídas e suas trajetórias, definidas.

Fluxo normal	Ação do ator	Ação do sistema
	1. Selecionar opção de executar simulação	
		2. Uma janela será aberta para execução da simulação
	3. Selecionar opção de iniciar simulação	
		4. Incluir caso de uso Calcular Movimentação

Fluxo alternativo 1: Pré-condições	Ação do ator	Ação do sistema
	1. Selecionar opção de executar simulação	
		2. Simulação não será executada de forma correta pois nem todas as précondições foram atendidas

INFORMAÇÕES GERAIS DO CASO DE USO			
Identificação:	**UC - 06**	Nome:	**Detectar Riscos**
Descrição:	**O sistema deve identificar as situações em que as entidades se encontrem em risco de colisão, necessidade de frenagem e ultrapassagem não recomendada**		

ESPECIFICAÇÃO DO CASO DE USO		
Atores:	Identificador	Nome/Descrição
	AT - 01	**Usuário**

Pré-condições:	Nome/Descrição
	É necessário que exista uma simulação em andamento, com movimentação de entidades

Pós-condições:	Nome/Descrição
	Execução do UC - 07 (Exibir Alertas)

Fluxo normal	Ação do ator	Ação do sistema
		1. Sistema calcula se o agente se encontra em alguma das situações de risco

INFORMAÇÕES GERAIS DO CASO DE USO			
Identificação:	**UC - 07**	Nome:	**Exibir Alertas**
Descrição:	**De acordo com os cálculos realizados no UC – 06 (Detectar Riscos) o sistema deve exibir alertas ao usuário, indicando em qual das situações de risco o agente se encontra**		

ESPECIFICAÇÃO DO CASO DE USO		
Atores:	Identificador	Nome/Descrição
	AT - 01	**Usuário**

Pré-condições:	Nome/Descrição
	É necessário que o caso de uso UC - 06 (Detectar Riscos) tenha sido executado.

Fluxo normal	Ação do ator	Ação do sistema
		1. Obter, para cada elemento, os potenciais riscos de acordo com o agente
		2. Exibir alertas ao usuário, indicando qual situação está ocorrendo naquele momento

INFORMAÇÕES GERAIS DO CASO DE USO			
Identificação:	**UC - 08**	Nome:	**Calcular Movimentação**
Descrição:	**O sistema deve calcular a movimentação das entidades durante a execução da simulação**		

ESPECIFICAÇÃO DO CASO DE USO		
Atores:	Identificador	Nome/Descrição
	AT - 01	**Usuário**

Pré-condições:	Nome/Descrição
	É necessário que o caso de uso UC - 05 (Executar Simulação) tenha sido iniciado.

Pós-condições:	Nome/Descrição
	Execução do UC - 06 (Detectar Riscos)

Fluxo normal	Ação do ator	Ação do sistema
		1. Sistema calcula movimentação com relação ao próximo ponto do percurso da entidade
		2. Entidades mudam suas posições para os pontos calculados

INFORMAÇÕES GERAIS DO CASO DE USO			
Identificação:	**UC - 09**	Nome:	**Gerar Perfil de Profundidade**
Descrição:	**O sistema deve gerar perfis de profundidade de acordo com a visão do observador durante a simulação**		

ESPECIFICAÇÃO DO CASO DE USO		
Atores:	Identificador	Nome/Descrição
	AT - 01	**Usuário**

Pré-condições:	Nome/Descrição
	É necessário que exista uma simulação em execução

Pós-condições:	Nome/Descrição
	Deve ser gerado um arquivo com as descrições dos perfis

Fluxo normal	Ação do ator	Ação do sistema
		1. Sistema obtêm dados da simulação
		2. Sistema calcula perfil de profundidade
		3. Executa Caso de Uso Exibir Perfil de Profundidade

INFORMAÇÕES GERAIS DO CASO DE USO			
Identificação:	**UC - 10**	Nome:	**Exibir Perfil de Profundidade**
Descrição:	**O sistema deve permitir que o usuário defina se quer ou não a exibição do perfil de profundidade**		

ESPECIFICAÇÃO DO CASO DE USO		
Atores:	Identificador	Nome/Descrição
	AT - 01	**Usuário**

Pré-condições:	Nome/Descrição
	É necessário que exista uma simulação em execução. Aém disso, o caso de uso Gerar Perfil de Profundidade deve ter sido executado.

Fluxo normal	Ação do ator	Ação do sistema
	1. Iniciar simulação	
	2. Assinalar o checkbox "Exibir Perfil de Profundidade"	
		3. Gerar o gráfico do Perfil de Profundidade
		4. Exibir o gráfico do Perfil de Profundidade para o ususário

INFORMAÇÕES GERAIS DO CASO DE USO			
Identificação:	**UC - 11**	Nome:	**Gerenciar Hipóteses**
Descrição:	**O sistema deve se comunicar com o Prolog para obter respostas quanto ao estado das entidades**		

ESPECIFICAÇÃO DO CASO DE USO		
Atores:	Identificador	Nome/Descrição
	AT - 02	**Prolog**

Pré-condições:	Nome/Descrição
	É necessário que exista uma simulação em execução

Pós-condições:	Nome/Descrição
	As informações são armazenas para exibição ao usuário

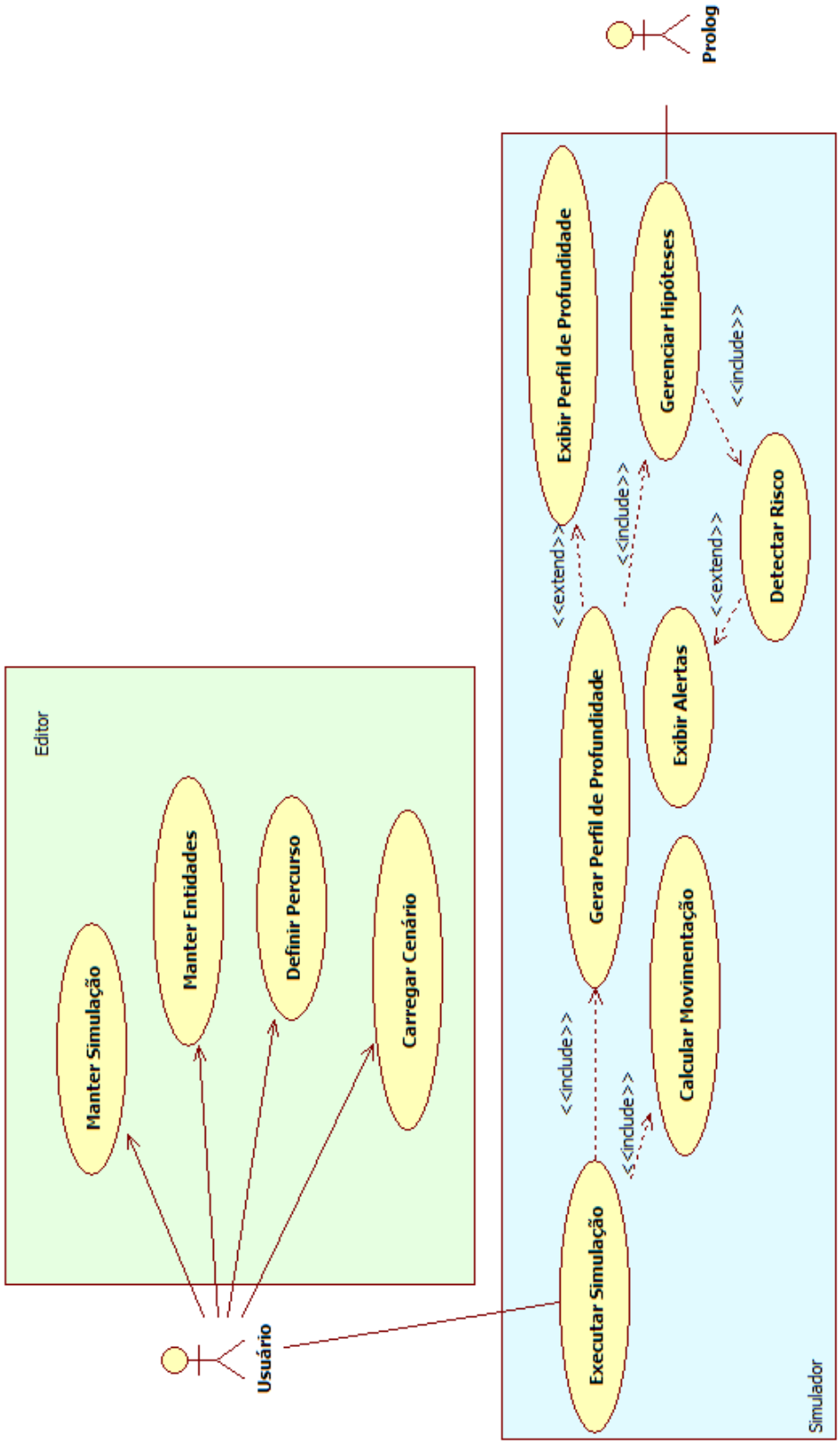
Fluxo normal	Ação do ator	Ação do sistema
	1. Iniciar simulação	
		2. Executar Casos de Uso Calcular Movimentação e Detectar Riscos
		3. Realizar perguntas ao Prolog
		4. Armazenar respostas

O diagrama de Casos de Uso pode ser verificado no Apêndice C, a seguir.

APÊNDICE C – DIAGRAMAS UML

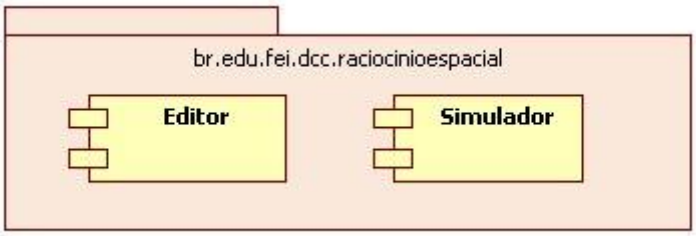
1) Diagrama de Casos de Uso

As descrições dos Casos de Uso a seguir podem ser verificadas no Apêndice B.



2) Diagrama de Componentes

O Simulador de Tráfego de Veículos está dividido em dois componentes. Um deles é o Editor, responsável pela parte de edição da simulação. O Editor é a porta de entrada para o uso do Simulador de Tráfego de Veículos como um todo. O segundo componente é o Simulador, que é responsável por exibir a execução da simulação. Embora os dois componentes interajam e sejam utilizados normalmente juntos, eles podem ser executados distintamente.



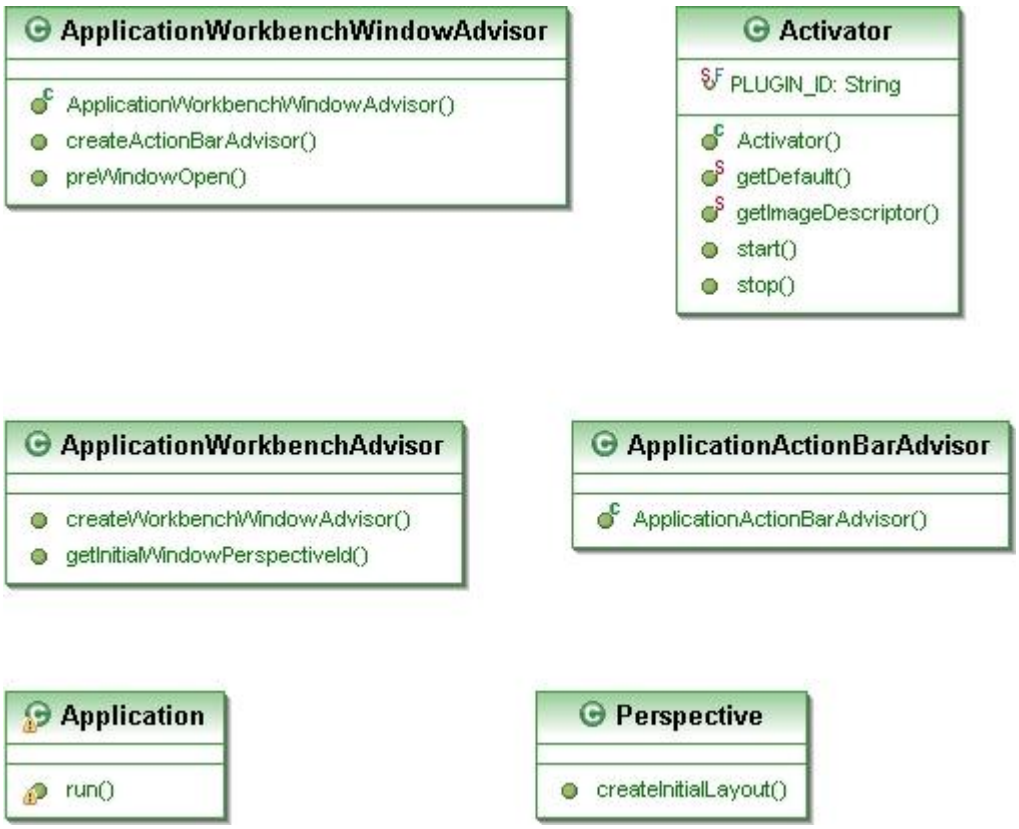
3) Diagrama de Classes

Para os diagramas de classes foram adotados recursos de cores que auxiliem e facilitem a visualização do que será alterado ou implementado em relação projeto anterior. Classes em cor verde não serão alteradas a princípio, classes em azul serão alteradas e em vermelho estão aquelas que serão totalmente implementadas.

a. Diagramas do Componente Editor

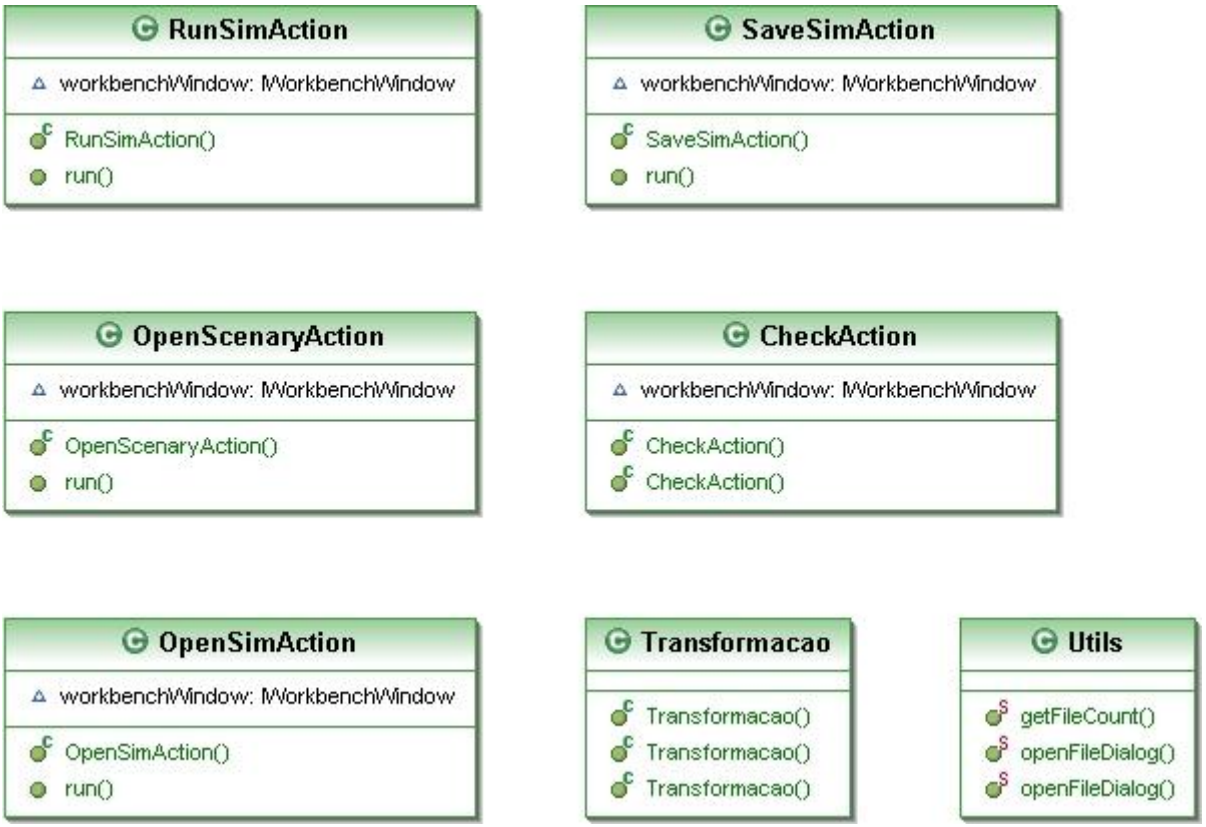
Pacote: `br.edu.fei.dcc.raciocinioespacial.editor.application`

As classes representadas abaixo são responsáveis por tornar o Editor de Simulação um ambiente de desenvolvimento semelhante ao Eclipse. Elas fazem parte de uma estrutura mínima para uma aplicação RCP. Destas nenhuma terá a necessidade de ser alterada para concretizar os resultados esperados deste projeto.



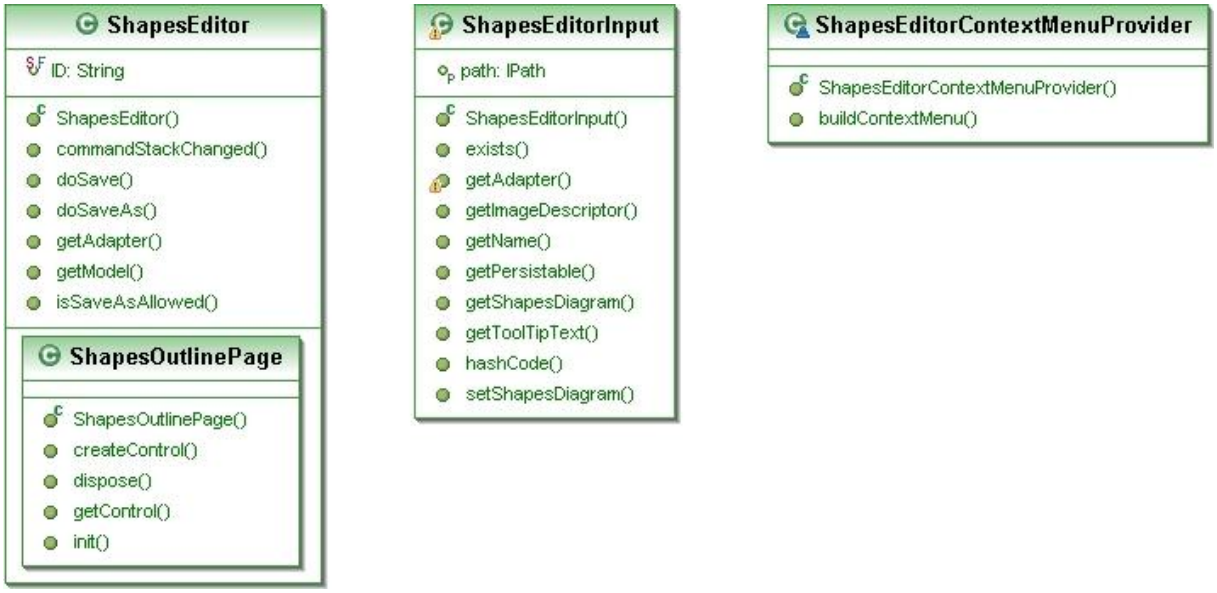
Pacote: `br.edu.fei.dcc.raciocinioespacial.editor.commands`

As classes representadas abaixo são responsáveis por implementar as ações possíveis no Editor de Simulação, tais como abrir e salvar. Algumas dessas classes sofrerão alterações para os novos conceitos da simulação, tais como a existência de cenário.



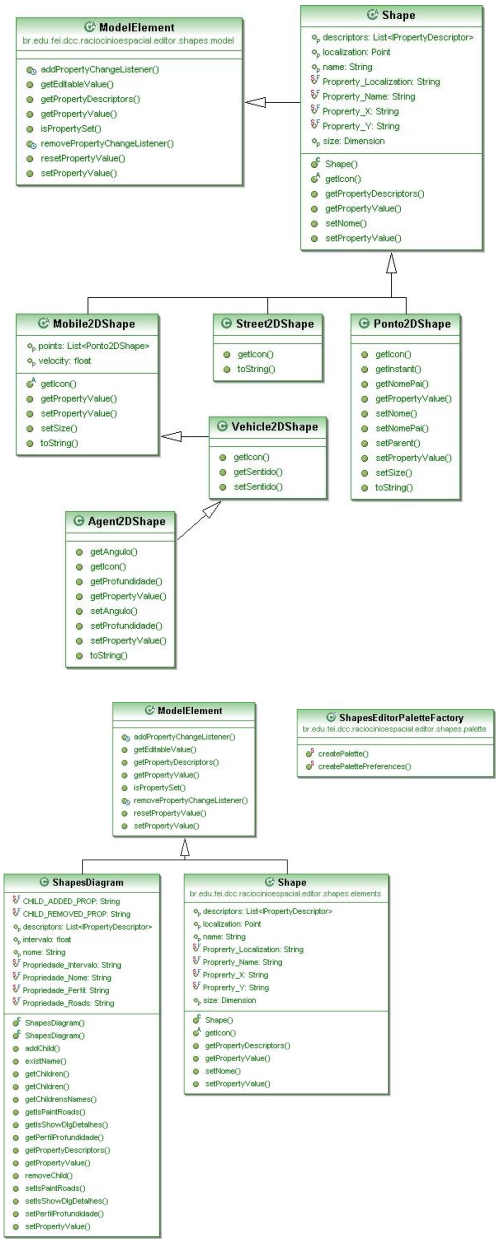
Pacote: `br.edu.fei.dcc.raciocinioespacial.editor.shapes`

As classes representadas abaixo são responsáveis pela interface e opções do editor gráfico existente no Editor de Simulação. É a partir dessas classes que é possível manipular as entidades, como arrastá-las de um lado para o outro e alterá-las clicando com o botão direito. Poucas alterações serão feitas nessas classes a fim de comportar as novas opções e objetos.



Pacotes: `br.edu.fei.dcc.raciocinioespacial.editor.shapes.model`
`br.edu.fei.dcc.raciocinioespacial.editor.shapes.elements`

Os dois pacotes, cujas classes são representadas abaixo, são responsáveis pelos objetos que podem ser adicionados ao editor gráfico e suas propriedades. Serão acrescentadas algumas classes para tornar possível o uso do SRE como um Simulador de Tráfego de Veículos da forma como se deseja que fique implementado.

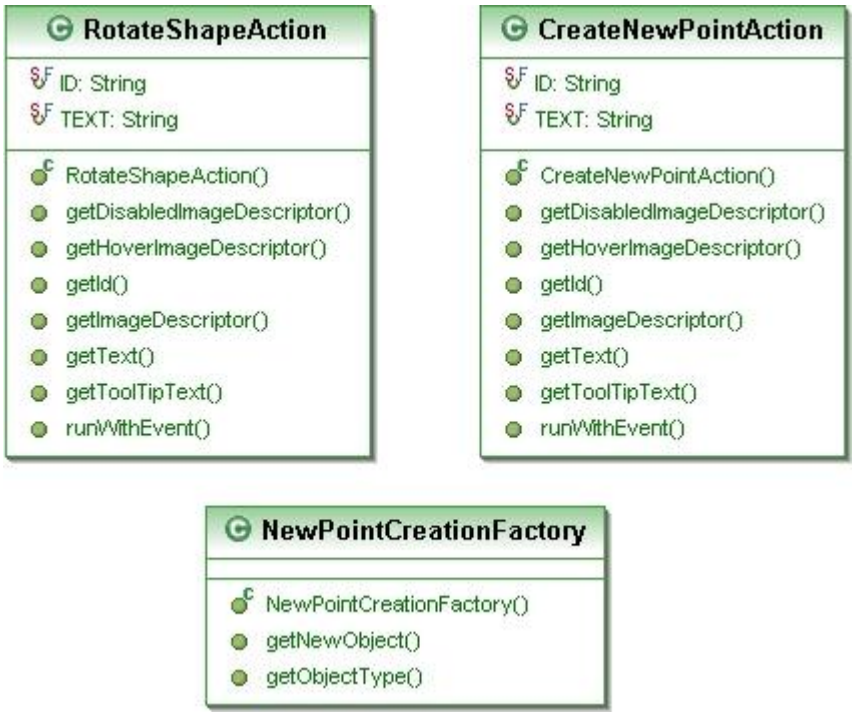


Pacote: `br.edu.fei.dcc.raciocinioespacial.editor.shapes.model.command`

As classes desse pacote são responsáveis pelos comandos que podem ser aplicados aos objetos existentes no editor gráfico.

Pacotes: `br.edu.fei.dcc.raciocinioespacial.editor.shapes.actions`

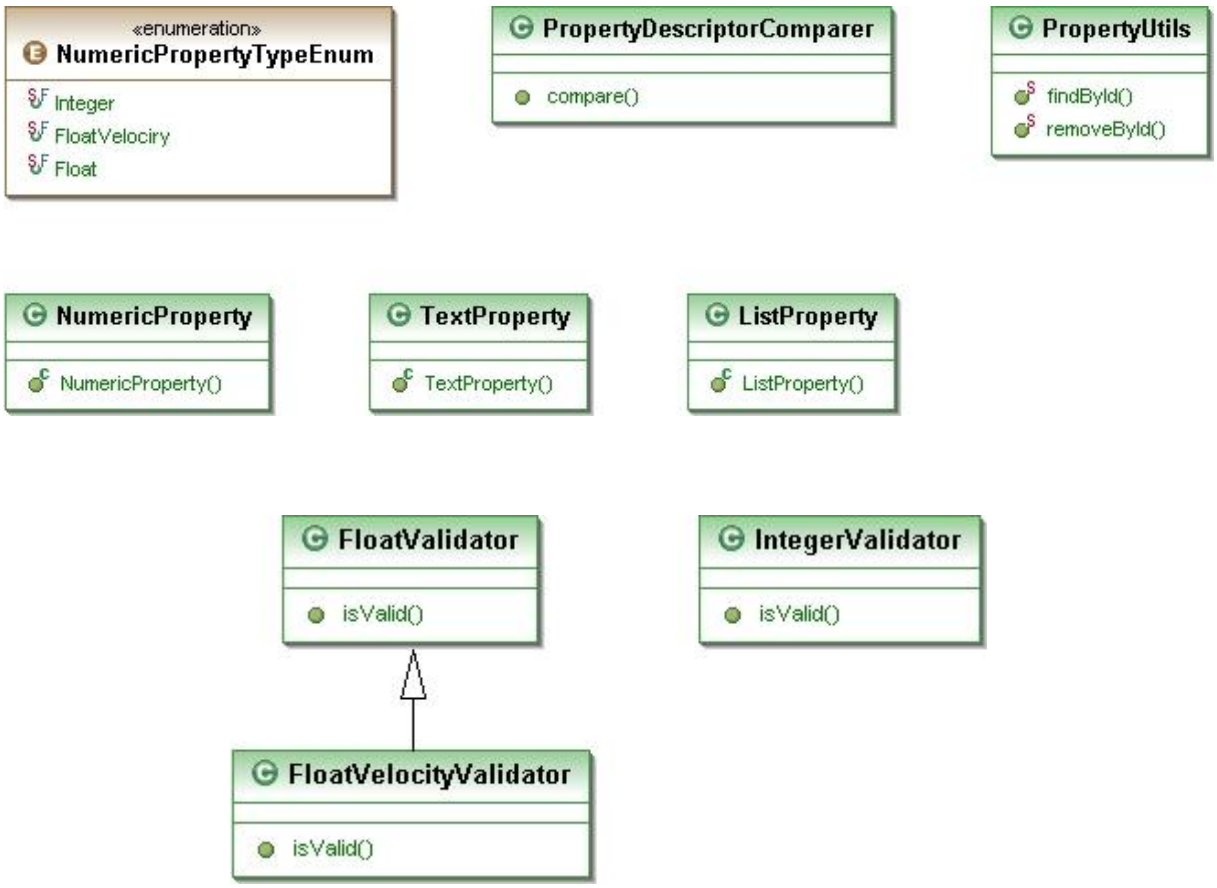
As classes representadas abaixo são responsáveis por implementar as ações que podem ser praticadas com os objetos do editor gráfico. Também está representada abaixo a classe responsável por montar a paleta de ferramentas em que estão representadas as entidades que podem ser adicionadas ao editor gráfico.



Pacotes: `br.edu.fei.dcc.raciocinioespacial.editor.shapes.properties`

`br.edu.fei.dcc.raciocinioespacial.editor.shapes.properties.validator`

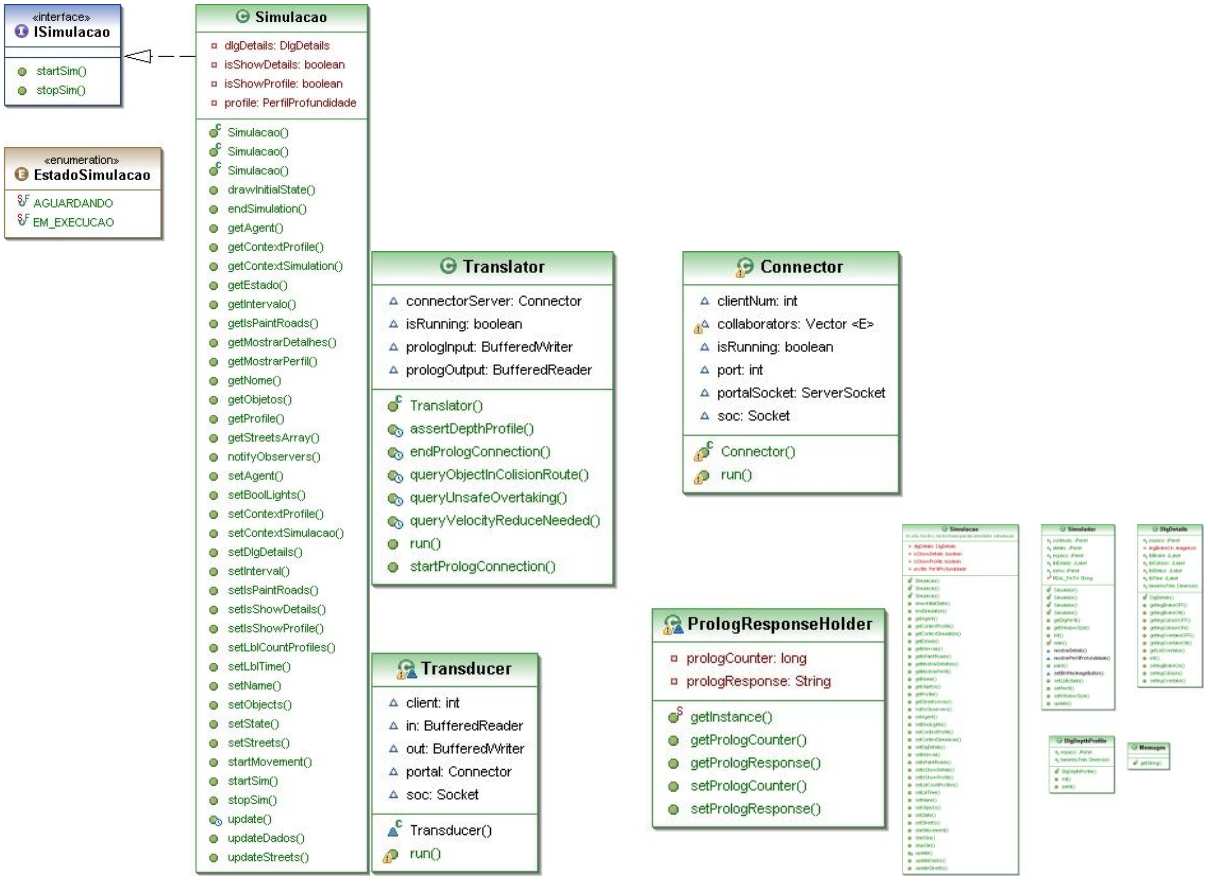
As classes representadas abaixo são responsáveis por implementar controles e validações para os tipos de propriedades possíveis aos objetos do editor gráfico.



b. Diagramas do Componente Simulador

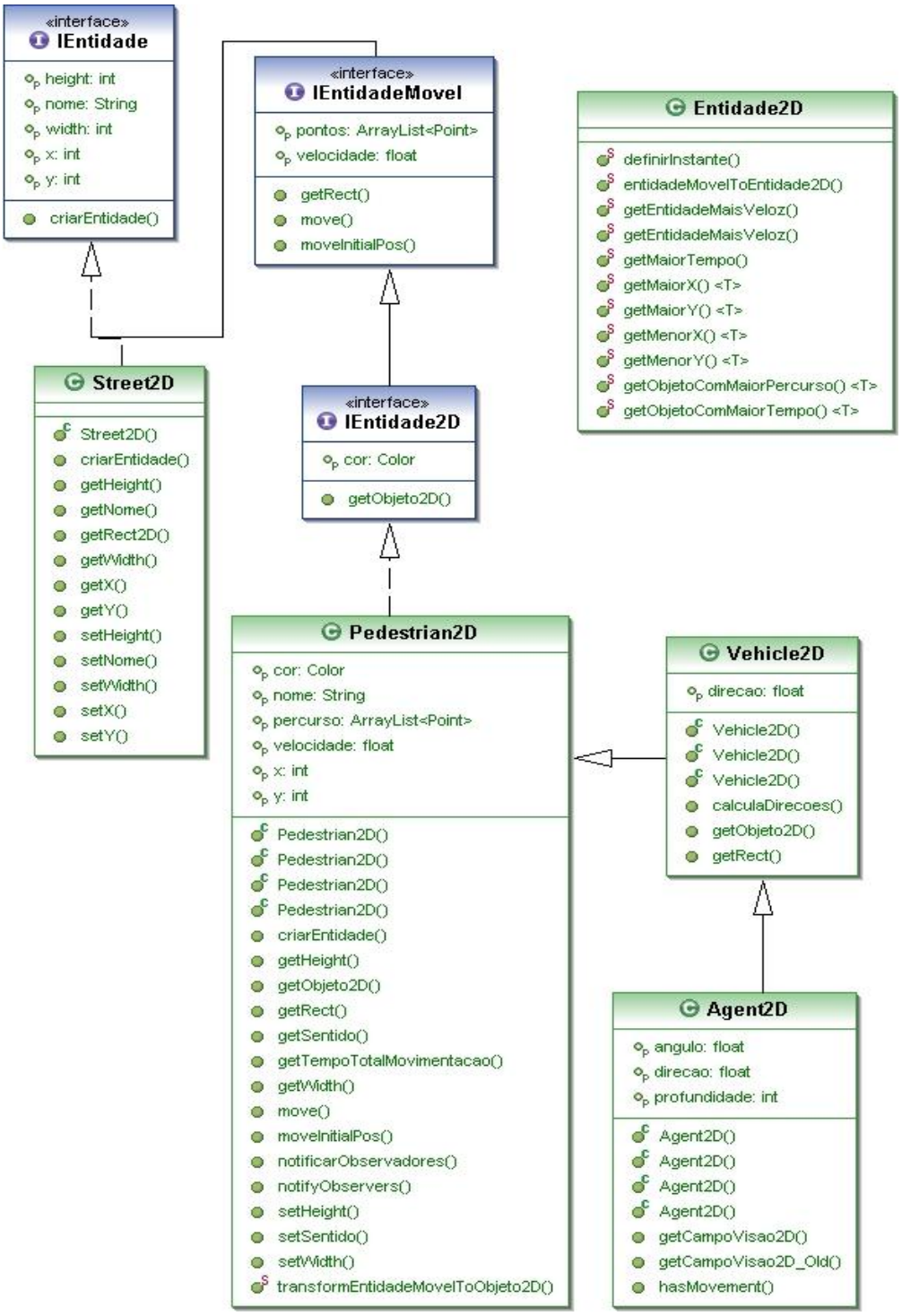
Pacotes: `br.edu.fei.dcc.raciocinioespacial.simulador.execucao`
`br.edu.fei.dcc.raciocinioespacial.simulador.prologCommunication`
`br.edu.fei.dcc.raciocinioespacial.simulador.simulacao`

Os pacotes abaixo representados são os principais. São eles que implementam, dentre outras coisas, a execução da simulação e seus controles, a geração dos axiomas e a comunicação com o Prolog.



Pacote: `br.edu.fei.dcc.raciocinioespacial.simulador.entidades`

As classes desse pacote são responsáveis por representar as entidades existentes na simulação e seus atributos em tempo de simulação. Foram acrescentadas classes e atributos para a representação de veículos e para que o próprio agente seja um veículo.

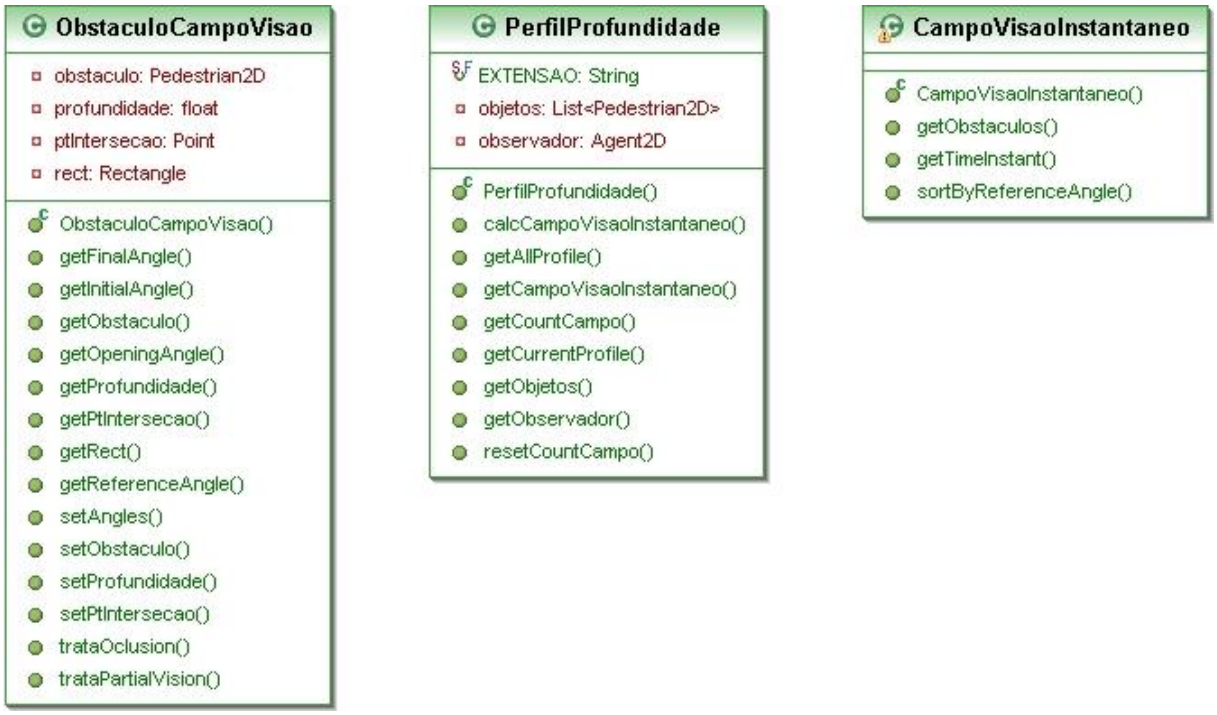


Pacote: `br.edu.fei.dcc.raciocinioespacial.simulador.movimentacao`

A classe desse pacote é responsável pela geração dos movimentos das entidades em tempo de simulação.

Pacote: `br.edu.fei.dcc.raciocinioespacial.simulador.perfilprofundidade`

As classes representadas abaixo são responsáveis pela geração do perfil de profundidade em tempo de simulação. Não serão necessárias alterações para a geração e manipulação do perfil de profundidade.

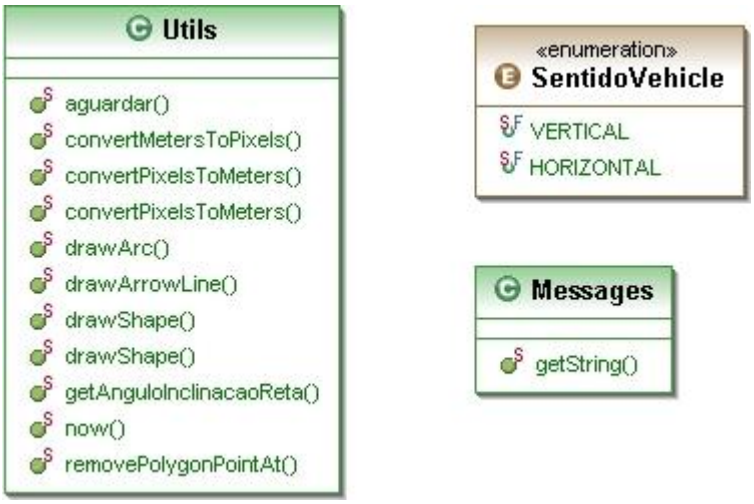


Pacote: `br.edu.fei.dcc.raciocinioespacial.simulador.persistencia`

A classe desse pacote é responsável por gerar o arquivo de saída que contém todos os picos enviados para o Prolog.

Pacote: `br.edu.fei.dcc.raciocinioespacial.simulador.util`

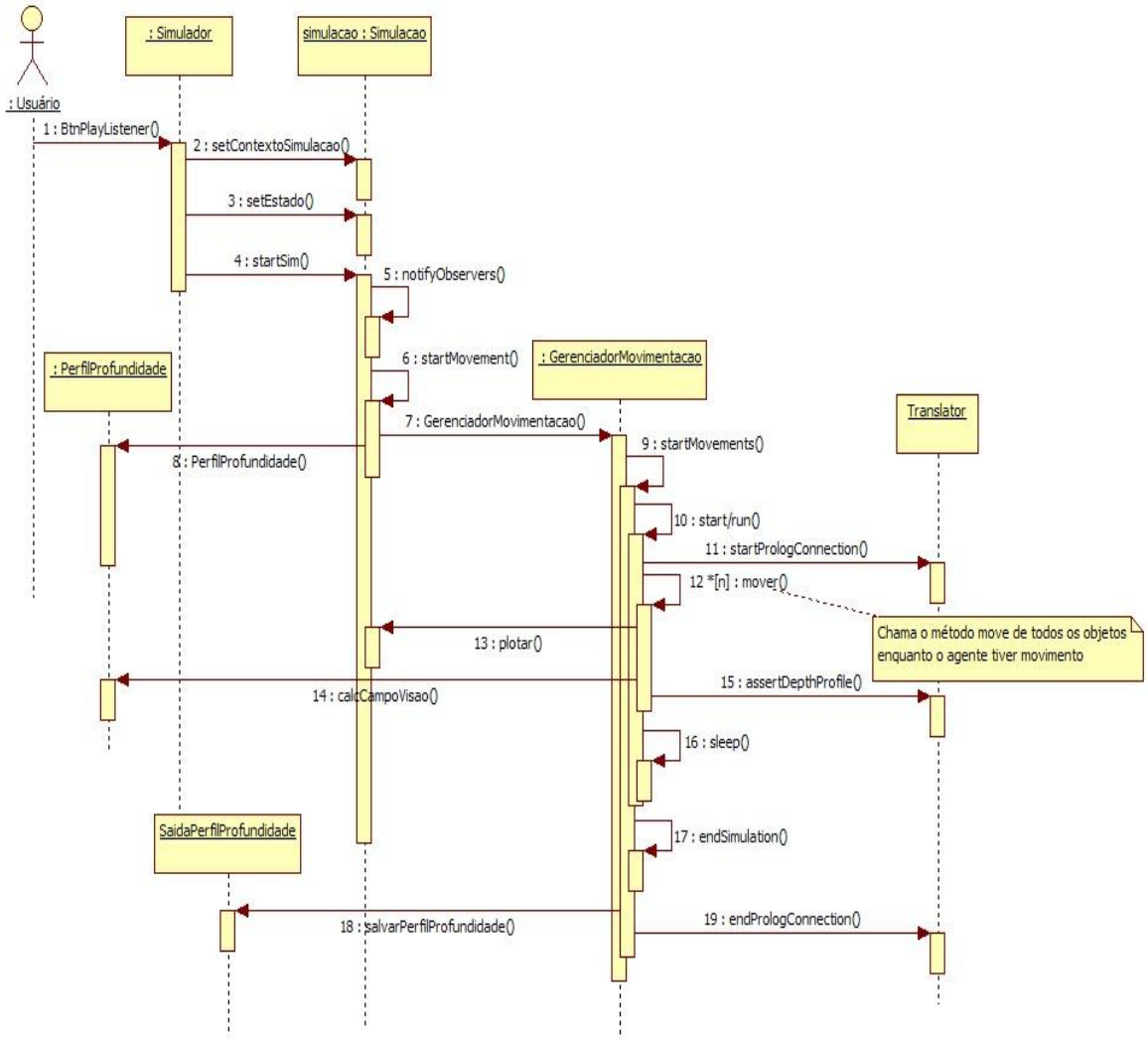
Pacote que recebe classes com funcionalidades gerais.



4) Diagrama de Seqüência

a. Fluxo principal de execução da Simulação

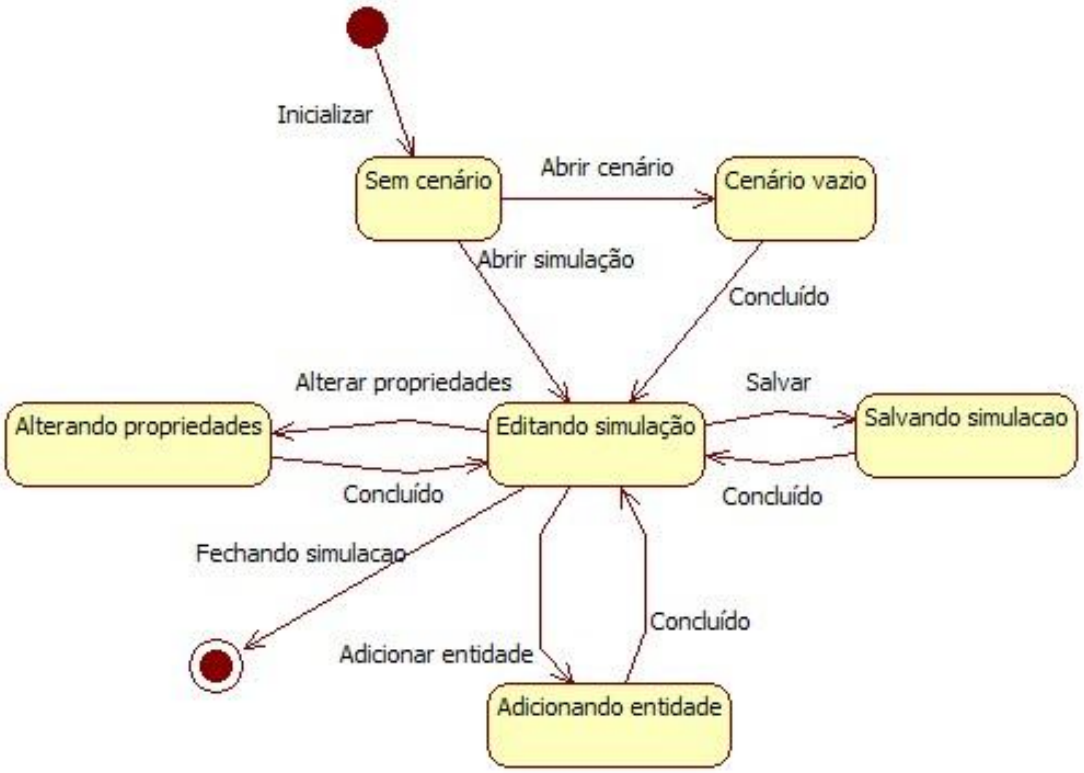
O diagrama abaixo apresenta a seqüência de execução da simulação a partir do momento em que é pressionado o botão de início. Ele foi alterado para que os cálculos de posição e movimento sejam feitos em tempo real de execução.



5) Diagrama de Estados

Classe: ShapeEditor

O diagrama abaixo representa os possíveis estados do Editor enquanto não pressionado o botão para inicializar a simulação. Foi alterado para possibilitar a abertura de um cenário no programa, ou carregar a simulação salva anteriormente.



Sem cenário: Na inicialização do programa, o sistema está sem cenário

Cenário vazio: Estado quando o programa é inicializado e posteriormente aberto um dos cenários para a construção da cena

Editando simulação: Estado quando é possível editar os objetos que compõem a simulação, como a posição dos mesmos. Tanto a partir de uma simulação previamente salva como de um cenário carregado

Alterando propriedades: Estado quando o usuário altera alguma propriedade da simulação

Adicionando entidade: Estado quando é adicionada uma entidade na simulação

Salvando simulação: Estado quando a simulação é salva

Classe: Objeto2D

Ativo: Após inclusão de um objeto, a classe está ativa

Alterando propriedades: Estado quando o usuário altera alguma propriedade do objeto

Definindo percurso: Estado quando são incluídos pontos para definição do percurso do objeto

Classe: Observador2D

Ativo: Após inclusão de um objeto, a classe está ativa

Alterando propriedades: Estado quando o usuário altera alguma propriedade do observador

Definindo percurso: Estado quando são incluídos pontos para definição do percurso do observador

Classe: IEntidadeMovel

Movimentando: Estado quando a entidades está em movimento.

Obtendo percurso: Estado quando se obtem o percurso e é calculado o próximo ponto

Alterando Posição: Estado quando já foi analisada e tratada à possibilidade de colisão e a entidade se movimenta para o ponto calculado

Classe: PerfilProfundidade

Gerando Perfil: Estado quando está sendo gerado o perfil de profundidade de acordo com a visão do observador.

Calculando campo: Estado quando o campo de visão do observador está sendo calculado

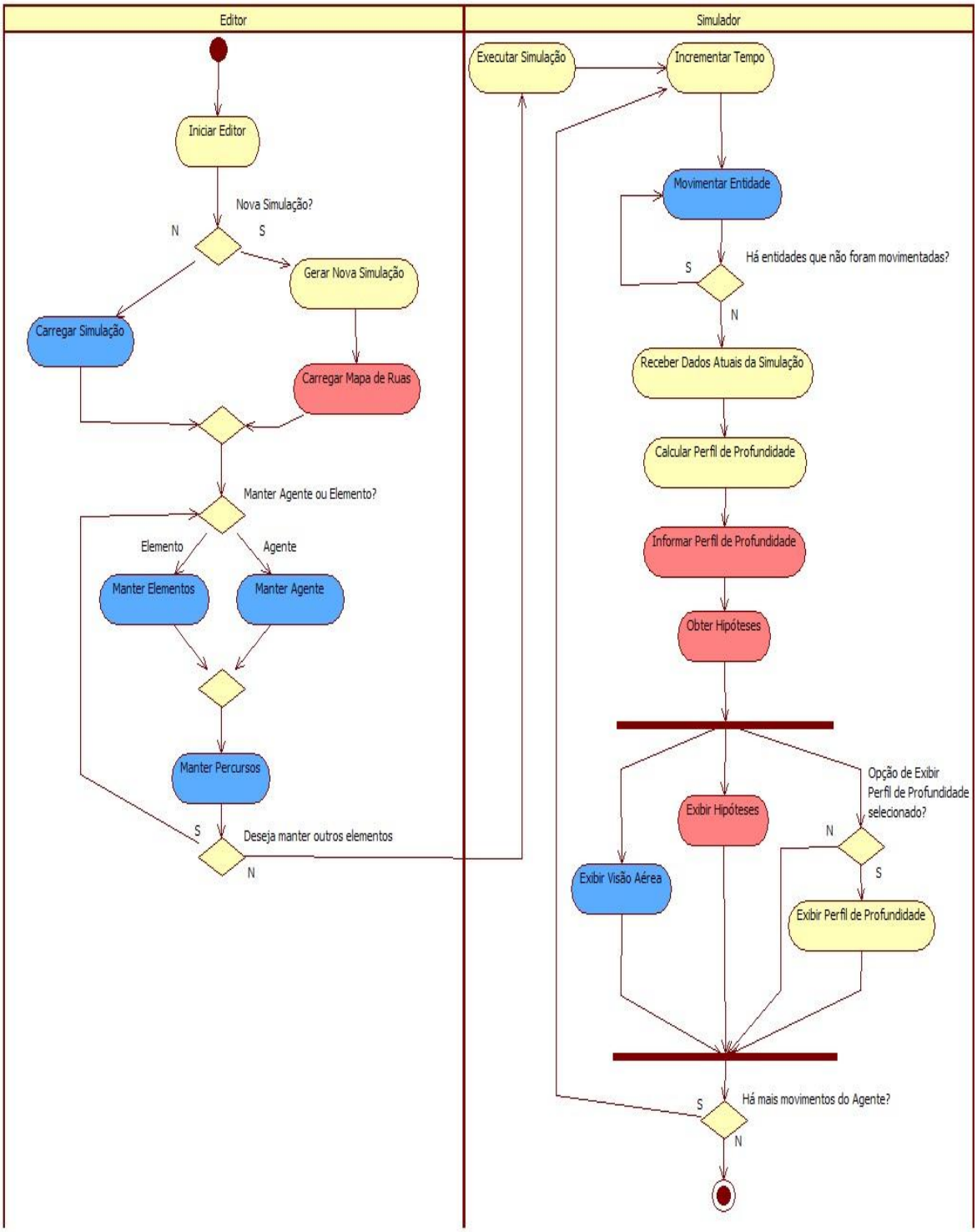
Verificando conteúdo: Estado quando se verifica se existe algum objeto no campo de visão do observador

Calculando gráfico: Estado quando está sendo calculado o gráfico do perfil, de acordo com o campo de visão

Exibindo gráfico: Estado quando o gráfico calculado é exibido na tela

6) Diagrama de Atividade

O diagrama criado abaixo mostra o fluxo das atividades realizadas, desde a configuração do ambiente até o fim da simulação.



As atividades em amarelo não serão alteradas ou terão alterações mínimas com relação ao SRE. As atividades em azul terão alterações significativas em relação ao SRE. As atividades em vermelho serão criadas em nosso trabalho.

a) **Iniciar Editor:** Ação do usuário que iniciar o editor gráfico.

b) **Carregar Simulação:** Ação do usuário que inicializa os objetos obtidos de uma simulação já salva anteriormente em um arquivo xml.

c) **Gerar Nova Simulação:** Ação do usuário que cria uma nova simulação.

d) **Carregar Mapa de Rua:** Ação do usuário que carrega o mapa de rua escolhido de um arquivo xml e o insere na simulação.

e) **Manter Objeto Móvel:** Ação do usuário que cria, altera ou exclui um Objeto Móvel (Carro ou Pedestre) e suas propriedades.

f) **Manter Agente:** Ação do usuário que cria, altera ou exclui o agente e suas propriedades.

g) **Manter Percurso:** Ação do usuário que cria, altera, ou exclui os pontos de percurso de uma Entidade (Agente ou Objeto Móvel).

h) **Executar Simulação:** Ação do usuário de iniciar a simulação

i) **Incrementar Tempo:** Aumenta o tempo da simulação.

j) **Movimentar Entidade:** Realiza o cálculo da movimentação da entidade para o tempo atual.

k) **Receber Dados do Perfil:** Os dados obtidos da simulação no tempo atual são enviados para o cálculo de perfil de profundidade.

l) **Calcular Perfil de Profundidade:** Com os dados recebidos, o perfil de profundidade é calculado para o tempo atual.

m) **Informar Perfil de Profundidade:** O perfil de profundidade do tempo atual é adicionado à base de conhecimento.

n) **Obter Hipóteses:** As hipóteses sobre a simulação no tempo atual são deduzidas a partir da base de conhecimento que são gerenciadas no Prolog.

o) **Exibir Visão Aérea:** A visão aérea da simulação para o tempo atual é renderizada.

p) **Exibir Hipóteses:** As hipóteses obtidas para o tempo atual são renderizadas.

q) **Exibir Perfil de Profundidade:** O perfil de profundidade do tempo atual é renderizado.

APÊNDICE D – AXIOMAS

Um axioma trata a identificação de um fato primário, o qual não pode ser analisado, e assim é reduzido a outros fatos ou decomposto em componentes menores. Ele não requer prova, mas é dele que dependem todas as provas e explicações.

Constantes:

1) forecastFrames(Number)

   forecastFrames(30).

O número de instantes no futuro em que serão criado estimativas das posições dos objetos é 30.

2) previousFrames(CurrentTime, NumberOfFrames)

   previousFrames(CurrentTime, NumberOfFrames):-

   CurrentTime >= 5,

   NumberOfFrames is 5 .

   previousFrames(CurrentTime, NumberOfFrames):-

   CurrentTime =< 5,

   NumberOfFrames is CurrentTime.

O número de instantes passados considerados nos cálculos é 5. Caso o número de instantes no passado seja menor que 5 (instantes iniciais), é considerado o número de instantes que já se passaram.

3) collisionFrames(InitialFrame, EndingFrame)

collisionFrames(0, 15).

O número de instantes que serão utilizados para os cálculos da situação de colisão, sendo que 0 = instante atual e 15 = 15 instantes no futuro.

4) overtakingFrames(InitialFrame, EndingFrame)

overtakingFrames(0, 15).

O número de instantes que serão utilizados para os cálculos da situação de ultrapassagem insegura, sendo que 0 = instante atual e 15 = 15 instantes no futuro.

5) breakingFrames(InitialFrame, EndingFrame)

breakingFrames(15, 30).

O número de instantes que serão utilizados para os cálculos da situação de frenagem recomendada, sendo que 15 = 15 instantes no futuro e 30 = 30 instantes no futuro. .

6) horizFrontLine(A, B, C)

horizFrontLine(3, 5.19615242270663, 0).

A linha definida acima é a linha horizontal frontal (usada tanto nos cálculos de colisão como de ultrapassagem), sendo *horizFrontLine*(A, B, C), onde A, B e C são os termos de uma equação geral da reta no formato Ax + By + C = 0.

7) horizBackLine(A, B, C)

horizBackLine(3, 5.19615242270663, 21.6).

A linha definida acima é a linha horizontal traseira (usada tanto nos cálculos de colisão como de ultrapassagem), sendo *horizFrontLine*(A, B, C), onde A, B e C são os termos de uma equação geral da reta no formato Ax + By + C = 0.

8) vertOuterRightLine(A, B, C)

vertOuterRightLine(3.11769145362398, -1.8, -10.8).

A linha definida acima é a linha vertical direita externa (usada nos cálculos de ultrapassagem), sendo *horizFrontLine*(A, B, C), onde A, B e C são os termos de uma equação geral da reta no formato Ax + By + C = 0.

9) vertInnerRightLine(A, B, C)

vertInnerRightLine(3.11769145362398, -1.8, -3.6).

A linha definida acima é a linha vertical direita interna (usada tanto nos cálculos de colisão como de ultrapassagem), sendo *horizFrontLine*(A, B, C), onde A, B e C são os termos de uma equação geral da reta no formato Ax + By + C = 0).

10) vertInnerLeftLine(A, B, C)

vertInnerLeftLine(3.11769145362398, -1.8, 3.6).

A linha definida acima é a linha vertical esquerda interna (usada tanto nos cálculos de colisão como de ultrapassagem), sendo *horizFrontLine*(A, B, C), onde A, B e C são os termos de uma equação geral da reta no formato Ax + By + C = 0.

11) vertOuterLeftLine(A, B, C)

vertOuterLeftLine(3.11769145362398, -1.8, 10.8).

A linha definida acima é a linha vertical esquerda externa (usada nos cálculos de ultrapassagem), sendo *horizFrontLine*(A, B, C), onde A, B e C são os termos de uma equação geral da reta no formato Ax + By + C = 0.

12) horizDist(Distance)

horizDist(3.6).

Esta é a distância entre as linhas virtuais horizontais (3,6 metros).

13) vertOuterDist(Distance)

vertOuterDist(2).

Esta é a distância entre as linhas virtuais verticais externas e verticais internas (2 metros).

14) vertInnerDist(Distance)

vertInnerDist(2).

Esta é a distância entre as linhas virtuais verticais internas (2 metros).

Base de Conhecimento:

1) peak(Body, Depth, Size, Ang, Time) :- dynamic peak/5.

Definição de um pico. Representado nos outros axiomas por *peak*(Obj, Distância, Tamanho, Ângulo, Tempo), onde, por exemplo, um *peak*(Pedestre1, 10, 1.5, 60, 100) significa que o objeto de nome Pedestre1 está a distância de 10 metros, possui o diâmetro de 1.5 metros, está localizado a 60º graus da borda direita da visão do observador, tendo sido detectado no instante 100.

2) situation(Body, Time, Situation) :- dynamic situation/3.

Definição de uma situação. Representado nos outros axiomas por *situation*(Obj, Tempo, Situação). Onde, por exemplo, *situation*(Pedestre1, 100, extending) significa que o objeto de nome Pedestre1, no tempo 100, tem sua situação reconhecida como expandindo (aproximando).

Inferências:

1) holdsAt(Body, CurrentTime, Situation)

holdsAt(Body, CurrentTime, extending):-

situation(Body, CurrentTime, extending).

holdsAt(Body, CurrentTime, shrinking):-

situation(Body, CurrentTime, shrinking).

holdsAt(Body, CurrentTime, appearing):-

situation(Body, CurrentTime, appearing).

holdsAt(Body, CurrentTime, vanishing):-

situation(Body, CurrentTime, vanishing).

holdsAt(Body, CurrentTime, stationary):-

situation(Body, CurrentTime, stationary).

holdsAt(Body, CurrentTime, extending):-

```
extending(Body, CurrentTime).

holdsAt(Body, CurrentTime, shrinking):-

shrinking(Body, CurrentTime).

holdsAt(Body, CurrentTime, appearing):-

appearing(Body, CurrentTime).

holdsAt(Body, CurrentTime, vanishing):-

vanishing(Body, CurrentTime).

holdsAt(Body, CurrentTime, stationary):-

stationary(Body, CurrentTime).
```

É utilizado para descobrir a Situação de um objeto em um dado tempo *CurrentTime*. Isso a partir dos axiomas *extending*, *shrinking*, *appearing*, *vanishing* e *stationary* que estão abaixo.

2) extending(Body, CurrentTime)

```
extending(Body, CurrentTime) :- peak(Body, Depth1, _, _, CurrentTime),

PreviousTime is CurrentTime-1,

peak(Body, Depth2, _, _, PreviousTime),

Depth1 < Depth2,

not(situation(Body, PreviousTime, vanishing)),
```

```
retractall(situation(Body, CurrentTime, _)),

assert(situation(Body, CurrentTime, extending)).
```

Um objeto está em estado *extending* caso a distância do mesmo esteja menor em um instante *CurrentTime* do que no *PreviousTime*, em relação ao veículo inteligente. Desde que ele também não não esteje em uma situação de desaperecimento no tempo *PreviousTime*.

3) shrinking(Body, CurrentTime)

```
shrinking(Body, CurrentTime):- peak(Body, Depth1, _, _, CurrentTime),

PreviousTime is CurrentTime-1,

peak(Body,Depth2, _, _, PreviousTime),

Depth1 > Depth2,

not(situation(Body, PreviousTime, vanishing)),

retractall(situation(Body, CurrentTime, _)),

assert(situation(Body, CurrentTime, shrinking)).
```

Um objeto está em estado *shrinking* caso a distância do mesmo esteja maior em um instante *CurrentTime* do que no *PreviousTime*, em relação ao veículo inteligente. E desde que também não esteje em uma situação de desaparecimento no tempo *PreviousTime*.

4) appearing(Body, CurrentTime)

```
appearing(Body, CurrentTime):- peak(Body, _, _, _, CurrentTime),
```

```
PreviousTime is CurrentTime-1,
not(peak(Body, _, _, _, PreviousTime)),
not(situation(Body, PreviousTime, extending)),
not(situation(Body, PreviousTime, shrinking)),
not(situation(Body, PreviousTime, appearing)),
retractall(situation(Body, CurrentTime, _)),
assert(situation(Body, CurrentTime, appearing)).
```

Um objeto está em estado appearing caso o objeto não exista no tempo *PreviousTime* dentro da Base de Conhecimento (o mesmo esteja fora do semi-círculo da visão do veículo inteligente). E desde que também não esteje em uma situação de desaparecimento, como de aparecimento e aproximação no tempo *PreviousTime*.

5) vanishing(Body, CurrentTime)

```
vanishing(Body, CurrentTime) :- not(peak(Body, _, _, _, CurrentTime)),
PreviousTime is CurrentTime-1,
peak(Body, _, _, _, PreviousTime),
not(situation(Body, PreviousTime, vanishing)),
retractall(situation(Body, CurrentTime, _)),
assert(situation(Body, CurrentTime, vanishing)).
```

Um objeto está em estado *vanishing* caso o objeto exista no tempo *PreviousTime* dentro da Base de Conhecimento (o mesmo esteja fora do semi-círculo da visão do veículo inteligente). E desde que também não esteje em uma situação de desaparecimento no tempo *PreviousTime*.

6) stationary(Body, CurrentTime)

```
stationary(Body, CurrentTime):-
peak(Body, Depth1, _, _, CurrentTime),
PreviousTime is CurrentTime-1,
peak(Body, Depth2, _, _, PreviousTime),
Depth1 = Depth2,
not(situation(Body, PreviousTime, vanishing)),
retractall(situation(Body, CurrentTime, _)),
assert(situation(Body, CurrentTime, stationary)).
```

Um objeto está em estado stationary caso o objeto exista no tempo *PreviousTime* dentro da Base de Conhecimento (o mesmo esteja fora do semi-círculo da visão do veículo inteligente). E a distância *Depth1* do tempo *CurrentTime* seja igual a *Depth2* do tempo *PreviousTime*, além disso, o mesmo não deve estar em uma situação de desaparecimento no tempo *PreviousTime*.

7) forecast(CurrentTime)

```
forecast(CurrentTime):- indall((peak(Body,Depth,Size,AngDist,CurrentTime)),
peak(Body,Depth,Size,AngDist,CurrentTime), PeakList),
```

É o axioma inicial sobre a previsão das situações no futuro, onde a partir do *CurrentTime* atreavés do *findall*(...), retorna todos os picos (objetos) que estão inseridos dentro do campo de visão do veículo inteligente em *Peaklist*, em seguida o proximo passo é o *prepareAssertForecast*

8)
```
prepareAssertForecast(PeakList).

prepareAssertForecast([]).

prepareAssertForecast([Peak|PeakList]) :- prepareAssertForecast(PeakList),

peak(Body, _, _, _, CurrentTime) = Peak,

previousFrames(CurrentTime, PreviousFrames),

retractall(situation(Body, CurrentTime, _)),

(holdsAt(Body, CurrentTime, extending) ->

retriveDeltas(Peak, PreviousFrames, DeltaX, DeltaY, DeltaSize, _, _, _, _),

forecastFrames(Frames),

assertForecast(Peak, DeltaX, DeltaY, DeltaSize, Frames); true)

.prepareAssertForecast([Peak|PeakList])
```

O axioma *prepareAssetForecast* é chamado n vezes até a que a *PeakList* (lista que contém todos os objetos dentro do semi-círculo) esteja vazia. Quando vazia, começa o

processo de desempilhamento, onde cada *peak* de cada objeto é analisado através das instruções posteriores. Assim é verificado quantos frames serão analisados através do *PreviousFrames* e em seguida retirado através do retractall qualquer "sujeira" de previsões anteriores para aquele objeto. Deste modo, caso o este objeto esteja em estado *extending*, continuam as verificações/inferências através do *retriveDeltas*, *forecastFrames* e *assetForecast*, caso contrário retorna-se simplesmente *true* para aquele objeto.

9) assertForecast(peak(Body, Depth, Size, AngDist, CurrentTime), DeltaX, DeltaY, DeltaSize, Frames)

   assertForecast(peak(_, _, _, _, _), _, _, _, 0).

   assertForecast(peak(Body, Depth, Size, AngDist, CurrentTime), DeltaX, DeltaY, DeltaSize, Frames) :- forecastFrames(AuxTime),

   FrameTime is AuxTime - Frames + 1,

   relativeLocation(AngDist, Depth, (X, Y)),

   FrameX is DeltaX * FrameTime + X,

   FrameY is DeltaY * FrameTime + Y,

   FrameSize is Size,

   angularLocation((FrameX, FrameY), FrameAng, FrameDist),

   ForecastTime is FrameTime + CurrentTime,

   retractall(peak(Body, _, _, _, ForecastTime)),

   assert(peak(Body, FrameDist, FrameSize, FrameAng, ForecastTime)),

FollowingFrames is Frames-1,

assertForecast(peak(Body, Depth, Size, AngDist, CurrentTime), DeltaX, DeltaY, DeltaSize, FollowingFrames).

Este axioma é responsável por realizar os asserts (inserções na base de conhecimento) de um objeto no futuro, indicando qual será a sua distância. Este iniciando a previsão no instante *Frames* igual ao que foi determinado em *forecastFrames*(*Frames*) até chegar a 0, quando é terminado o processo de inserção na base de conhecimento para um determinado objeto. Porém, a cada decremento do *Frames*, é calculada a posição do objeto naquele instante de tempo *FrameTime* com as passagens *relativeLocation*(...), *angularLocation*(...), além de retirar alguma "sujeira" no instante *ForecastTime* que possa ter de previsões antigas através do *retractall*(...).

10) retriveDeltas(peak(Body, Depth, Size, Ang, Time), PreviousFrames, DeltaX, DeltaY, DeltaSize, TotalX, TotalY, TotalSize, TotalFrames)

retriveDeltas(_, 0, DeltaX, DeltaY, DeltaSize, TotalX, TotalY, TotalSize,TotalFrames):- DeltaX is 0,

DeltaY is 0, DeltaSize is 0, TotalX is 0, TotalY is 0,

TotalSize is 0, TotalFrames is 0.

retriveDeltas(Peak, PreviousFrames, DeltaX, DeltaY, DeltaSize, TotalX, TotalY, TotalSize, TotalFrames):- peak(Body, Depth, Size, AngDist, CurrentTime) = Peak,

PreviousTime is CurrentTime-1,

```
(peak(Body, PrevDepth, PrevSize, PrevAngDist, PreviousTime)->
PrevPeak = peak(Body, PrevDepth, PrevSize, PrevAngDist, PreviousTime),
AuxPreviousFrames is PreviousFrames - 1 ,
retriveDeltas(PrevPeak, AuxPreviousFrames, PrevDeltaX, PrevDeltaY,
PrevDeltaSize, PrevTotalX, PrevTotalY, PrevTotalSize, PrevTotalFrames);
peak(Body, PrevDepth, PrevSize, PrevAngDist, CurrentTime),
PrevPeak = peak(Body, PrevDepth, PrevSize, PrevAngDist, CurrentTime),
retriveDeltas(PrevPeak, 0, PrevDeltaX, PrevDeltaY, PrevDeltaSize,
PrevTotalX, PrevTotalY, PrevTotalSize, PrevTotalFramesAux),
PrevTotalFrames is PrevTotalFramesAux - 1),
relativeLocation(AngDist, Depth, (X,Y)),
relativeLocation(PrevAngDist, PrevDepth, (PrevX,PrevY)),
TotalX is X-PrevX+PrevTotalX,
TotalY is Y-PrevY+PrevTotalY,
TotalSize is Size-PrevSize+PrevTotalSize,
TotalFrames is PrevTotalFrames + 1,
(TotalFrames is 0 -> DeltaX is TotalX,
DeltaY is TotalY, DeltaSize is TotalSize;
DeltaX is TotalX / TotalFrames,
```

```
DeltaY is TotalY / TotalFrames,

DeltaSize is TotalSize / TotalFrames).
```

Basicamente, obtenção do Delta X e Delta Y da movimentação do objeto, analisando sua posição nos instantes passados. A partir da existência de um pico no tempo *PreviousTime* é calculado recursivamente a média dos Delta X e Delta Y de cada instante até o limite de picos anteriores estipulado pelo *PreviousFrames*(...). E, caso não exista pico anterior e somente exista o atual, o Delta X e Delta Y são atribuídos como sendo a própria distância percorrida até o pico atual.

11) queryObjectInColisionRoute(Time)

```
queryObjectInColisionRoute(Time) :- findall((situation(Body, Time,
extending)), situation(Body, Time, extending), SituationList),

collisionFrames(InitTime1, EndTime1),

InitTime is Time + InitTime1,

EndTime is Time + EndTime1,

not(queryObjectInColisionRouteLoopTime(SituationList, InitTime,
EndTime)).
```

Este axioma é o mais externo na verificação da situação de colisão. Inicialmente, ele acha todos os *situation* dos objetos que estão *extending* no instante dado (*findall*(...)), armazenados em *SituationList*. Então ele obtém o intervalo de instantes no futuro que será feita a verificação (*collisionFrames*(...)) e os adiciona ao instante atual. Por exemplo, se o instante atual (*Time*) for 100 e o intervalo e os instantes obtidos em *collisionFrames*

(InitTime1 e EndTime1) são 0 e 15, então o intervalo de instantes que serão analisados vão de 100 (100+0) à 115 (100+15). Inicia-se, dessa forma, a verificação de fato de objetos em rota de colisão (*queryObjectInColisionRouteLoopTime*(...)).

Caso *queryObjectInColisionRouteLoopTime* seja falso, é identificada uma situação de colisão.

12) queryObjectInColisionRouteLoopTime(SituationList, InitTime, EndTime)

```
queryObjectInColisionRouteLoopTime(_, InitTime, EndTime):-
InitTime > EndTime.

queryObjectInColisionRouteLoopTime(SituationList, InitTime, EndTime):-
EndTime >= InitTime,
(
        queryObjectInColisionRouteLoopPeak(SituationList, InitTime)->
        InitTime1 is InitTime + 1,
        queryObjectInColisionRouteLoopTime(SituationList, InitTime1,
EndTime);
        fail
).
```

Este axioma tem como finalidade interar entre todos os instantes do intervalo definido entre InitTime e EndTime, obtendo todos os *peak*(...) dos objetos pertencentes à *SituationList*.

Inicialmente, ele verifica se InitTime é maior que EndTime. Caso essa afirmação seja verdadeira, não é realizado mais interações (todos os instantes do intervalo foram analisados). Do contrário, inicia-se a verificação de colisão em todos os picos (*queryObjectInColisionRouteLoopPeak*(...)). Se ela for verdadeira (nenhuma colisão foi detectada para aquele instante), o axioma entra em recursão, passando agora no InitTime1 o InitTime + 1. Caso contrário (uma colisão foi detectada), o axioma pára de fazer recursões.

13) queryObjectInColisionRouteLoopPeak([situation(Body, _, _)|Tail], Time)

queryObjectInColisionRouteLoopPeak([], _).

queryObjectInColisionRouteLoopPeak([situation(Body, _, _)|Tail], Time):-

peak(Body, Depth, Size, AngDist, Time),

Peak = peak(Body, Depth, Size, AngDist, Time),

distancesPeakToDefinedLines(Peak, HFL, HBL, _, VIR, VIL, _),

not(verifyCollision(HFL, HBL, VIR, VIL)),

queryObjectInColisionRouteLoopPeak(Tail, Time).

Este axioma tem como finalidade interar entre todos os picos (peak(...)) de um dado instante, verificando se ele caracteriza uma situação de colisão.

Inicialmente, ele verifica se a lista de *situation*(...) recebida está vazia. Caso afirmativo, a verificação é concluída. Do contrátio, obtêm-se o *peak* do objeto *Body* no instante *Time* (Peak= peak(...)). Em seguida, verifica a distância do objeto em relação as linhas virtuais (*distancesPeakToDefinedLines*(...)). Estas distâncias (HFL, HBL, VIR, VIL) serão repassadas para *verifyCollision*(...), que verificará a situação de colisão em si.

Caso a verificação seja falsa (o objeto não se encontra em situação de colisão no instante atual), é realizada a recursão, porém agora com o resto da lista de *situations*. Caso a verificação seja verdadeira (o objeto se encontra em situação de colisão no instante atual), o axioma pára de fazer recursões.

14) verifyCollision(HFL, HBL, VIR, VIL)

```
verifyCollision(HFL, HBL, VIR, VIL) :- oneGreaterThanZero(HFL,HBL),
oneGreaterThanZero(VIR,VIL).
verifyCollision(HFL, HBL, VIR, VIL) :- horizDist(Dist),
- Dist < HFL + HBL,
oneGreaterThanZero(VIR, VIL).
verifyCollision(HFL, HBL, VIR, VIL) :- vertInnerDist(Dist),
- Dist < VIR + VIL,
oneGreaterThanZero(HFL, HBL).
verifyCollision(HFL, HBL, VIR, VIL) :- horizDist(HDist),
vertInnerDist(VDist),
```

- HDist < HFL + HBL,

- VDist < VIR + VIL.

Esse axioma verifica se o objeto, cujas distâncias às retas virtuais são conhecidas, está ou não na área de colisão. Para isso, são observadas 4 situações distintas:

a. Quando o objeto é interseccionado por uma reta horizontal e uma vertical interna (colisão no canto);

b. Quando o objeto é interseccionado por uma reta vertical interna e está entre as duas retas horizontais (colisão lateral);

c. Quando o objeto é interseccionado por uma reta horizontal e está entre as duas retas verticais internas (colisão frontal);

d. Quando o objeto não é interseccionado por nenhuma reta, mas está dentro das duas retas horizontais e dentro das duas retas verticais internas (objeto completamente contido dentro do veículo observador).

15) queryUnsafeOvertaking(Time)

queryUnsafeOvertaking(Time) :- findall((situation(Body, Time, extending)), situation(Body, Time, extending), SituationList),

overtakingFrames(InitTime1, EndTime1),

InitTime is Time + InitTime1,

EndTime is Time + EndTime1,

not(queryUnsafeOvertakingLoopTime(SituationList, InitTime, EndTime)).

Este axioma é o mais externo na verificação da situação de ultrapassagem insegura.

Inicialmente, ele acha todos os *situation* dos objetos que estão *extending* no instante dado (findall(...)), armazenados em *SituationList*. Então ele obtém o intervalo de instantes no futuro, quando será feita a verificação (*overtakingFrames*(...)) e os adiciona ao instante atual. Por exemplo, se o instante atual (*Time*) for 100 e o intervalo e os instantes obtidos em *collisionFrames* (InitTime1 e EndTime1) são 0 e 15, então o intervalo de instantes que serão analisados vão de 100 (100+0) à 115 (100+15). Inicia-se então, a verificação de fato de objetos em situação de ultrapassagem insegura (queryUnsafeOvertakingLoopTime(...)). Caso *queryUnsafeOvertakingLoopTime* seja falso, é identificada uma situação de ultrapassagem insegura.

16) queryUnsafeOvertakingLoopTime(SituationList, InitTime, EndTime)

```
queryUnsafeOvertakingLoopTime(_, InitTime, EndTime):-
InitTime > EndTime.
queryUnsafeOvertakingLoopTime(SituationList, InitTime, EndTime):-
EndTime >= InitTime,
(
        queryUnsafeOvertakingLoopPeak(SituationList, InitTime)->
        InitTime1 is InitTime + 1,
        queryUnsafeOvertakingLoopTime(SituationList, InitTime1, EndTime);
        fail
```

).

Este axioma tem como finalidade interar entre todos os instantes do intervalo definido entre InitTime e EndTime, obtendo todos os *peak*(...) dos objetos pertencentes à *SituationList*.

Inicialmente, ele verifica se InitTime é maior que EndTime. Caso essa afirmação seja verdadeira, não são realizadas mais interações (todos os instantes do intervalo foram analisados). Do contrário, inicia-se a verificação de ultrapassagem insegura em todos os picos (*queryUnsafeOvertakingLoopPeak*(...)). Se ela for verdadeira (não há situação de ultrapassagem insegura para aquele instante), o axioma entra em recursão, passando agora no InitTime1 o InitTime + 1. Caso contrário (uma situação de ultrapassagem insegura foi detectada), o axioma pára de fazer recursões.

17) queryUnsafeOvertakingLoopPeak([situation(Body, _, _)|Tail], Time)

queryUnsafeOvertakingLoopPeak([], _).

queryUnsafeOvertakingLoopPeak([situation(Body, _, _)|Tail], Time):-

peak(Body, Depth, Size, AngDist, Time),

Peak = peak(Body, Depth, Size, AngDist, Time),

distancesPeakToDefinedLines(Peak, HFL, HBL, VOR, VIR, VIL, VOL),

not(verifyUnsafeOvertaking(HFL, HBL, VIR, VIL, VOR, VOL)),

queryUnsafeOvertakingLoopPeak(Tail, Time).

Este axioma tem como finalidade, interar entre todos os picos (*peak*(...)) de um dado instante verificando se ele caracteriza uma situação ultrapassagem insegura.

Inicialmente, verifica se a lista de *situation*(...) recebida está vazia. Caso afirmativo, finaliza-se a verificação. Do contrátio, obtêm-se o *peak* do objeto *Body* no instante *Time* (Peak= peak(...)). Em seguida, verifica a distância do objeto em relação as linhas virtuais (*distancesPeakToDefinedLines*(...)). Estas distâncias (HFL, HBL, VOR, VIR, VIL, VOL) serão repassadas para *verifyUnsafeOvertaking*(...), que verificará a situação de colisão em si. Caso a verificação seja falsa (o objeto não se encontra em situação de colisão no instante atual), é realizada a recursão, porém agora com o resto da lista de *situations*. Caso a verificação seja verdadeira (o objeto se encontra em situação de colisão no instante atual), o axioma pára de fazer recursões.

18) verifyUnsafeOvertaking(HFL, HBL, VIR, VIL, VOR, VOL)

verifyUnsafeOvertaking(HFL, HBL, _, _, VOR, VOL):-

oneGreaterThanZero(HFL,HBL),

oneGreaterThanZero(VOR,VOL).

verifyUnsafeOvertaking(HFL, HBL, _, _, VOR, VOL) :- horizDist(Dist),

- Dist < HFL + HBL,

oneGreaterThanZero(VOR, VOL).

verifyUnsafeOvertaking(HFL, HBL, VIR, _, VOR, _) :- horizDist(Dist),

- Dist < HFL + HBL,

oneGreaterThanZero(VOR, VIR).

```
verifyUnsafeOvertaking(HFL, HBL, _, VIL, _, VOL) :- horizDist(Dist),
- Dist < HFL + HBL,
oneGreaterThanZero(VOL, VIL).
verifyUnsafeOvertaking(HFL, HBL, _, _, VOR, VOL) :- vertInnerDist(Dist),
- Dist < VOR + VOL,
oneGreaterThanZero(HFL, HBL).
verifyUnsafeOvertaking(HFL, HBL, _, VIL, _, VOL) :- horizDist(HDist),
vertInnerDist(VDist),
- HDist < HFL + HBL,
- VDist < VOL + VIL.
verifyUnsafeOvertaking(HFL, HBL, VIR, _, VOR, _) :- horizDist(HDist),
vertInnerDist(VDist),
- HDist < HFL + HBL,
- VDist < VOR + VIR.
```

Esse axioma verifica se o objeto, cuja as distâncias às retas virtuais são conhecidas, está ou não na área de ultrapassagem insegura. Para isso, são observadas sete situações distintas:

a. Quando o objeto é interseccionado por uma reta horizontal e uma vertical interna (ultrapassagem no canto danto direito quanto esquerdo);

b. Quando o objeto é interseccionado por uma reta vertical externa e está entre as duas retas horizontais (ultrapassagem lateral externa);

c. Quando o objeto é interseccionado por uma reta vertical direita e está entre as duas retas horizontais (ultrapassagem lateral direita);

d. Quando o objeto é interseccionado por uma reta vertical esquerda e está entre as duas retas horizontais (ultrapassagem lateral esquerda);

e. Quando o objeto é interseccionado por uma reta horizontal e está entre as duas retas verticais externas (ultrapassagem frontal esquerda e direita);

f. Quando o objeto não é interseccionado por nenhuma reta, mas está dentro das duas retas horizontais e dentro das duas retas verticais esquerda (objeto completamente contido dentro do área de ultrapassagem esquerda);

g. Quando o objeto não é interseccionado por nenhuma reta, mas está dentro das duas retas horizontais e dentro das duas retas verticais direita (objeto completamente contido dentro do área de ultrapassagem direita).

19) queryVelocityReduceNeeded(Time)

```
queryVelocityReduceNeeded(Time) :- findall((situation(Body, Time,
extending)), situation(Body, Time, extending), SituationList),

breakingFrames(InitTime1, EndTime1),

InitTime is Time + InitTime1,

EndTime is Time + EndTime1,
```

not(queryObjectInColisionRouteLoopTime(SituationList, InitTime,

EndTime)).

Este axioma é o mais externo na verificação da situação de redução de velocidade necessária.

Inicialmente, ele acha todos os situation dos objetos que estão *extending* no instante dado (*findall*(...)), armazenados em *SituationList*. Então ele obtém o intervalo de instantes no futuro que será feita a verificação (*collisionFrames*(...)) e os adiciona ao instante atual. Por exemplo, se o instante atual (*Time*) for 100 e o intervalo e os instantes obtidos em *collisionFrames* (InitTime1 e EndTime1) são 15 e 30, então o intervalo de instantes que serão analisados vão de 115 (100+15) à 130 (100+30). Inicia-se então a verificação de fato de objetos em situação de redução de velocidade necessária (*queryObjectInColisionRouteLoopTime*(...)). Caso *queryObjectInColisionRouteLoopTime* seja falso, é identificado uma situação de redução de velocidade necessária.

Auxiliares:

1) straightLineEquation((X1, Y1), (X2,Y2), A, B, C)

   straightLineEquation((X1, Y1), (X2,Y2), A, B, C):-

   A is Y1 - Y2,

   B is X2 - X1,

   C is X1 * Y2 - Y1 * X2.

Esse axioma obtém os termos A, B e C da equação geral da reta no formato Ax+By+C = 0, que passa pelos dois pontos definidos em (X1, Y1) e (X2,Y2). As fórmulas para calcular os valores estão de acordo com a relação encontrada por CARVALHO, Pedro (2008).

2) distancePointToLine((X,Y), A, B, C, Dist)

```
distancePointToLine((X,Y), A, B, C, Dist) :- AuxAbs is A * X + B * Y + C,
abs(AuxAbs, Abs),
AuxSqr is A * A + B * B,
sqrt(AuxSqr, Sqrt),
Dist is Abs / Sqrt.
```

Esse axioma obtém a distância (Dist) de um ponto (X,Y) à uma linha definida por A, B e C com equação geral da reta no formato Ax+By+C = 0. A fórmula utilizada é a seguinte:

$$Dist = \frac{|A*X + B*Y + C|}{\sqrt{(A^2 + B^2)}}$$

3) relativeLocation(Ang, Dist, (X, Y))

```
relativeLocation(Ang, Dist, (X, Y)) :- RadAng is pi * Ang / 180,
sin(RadAng, Sin),
cos(RadAng, Cos),
X is Dist * Cos,
```

Y is Dist * Sin.

Esse axioma obtém a posição relativa de um ponto em um plano cartesiano, sendo que inicialmente se conhece apenas o ângulo (Ang) formado do ponto em relação à origem e a abscissa e a distância (Dist) do ponto à origem.

Resolvemos o problema por trigonometria, onde Dist = Hipotenusa e Ang = ângulo formado pelo cateto adjacente e hipotenusa.

Inicialmente, calcula-se o valor radiano do ângulo em RadAng é pi * Ang / 180. A partir de RadAng, achamos os valores de seno (Sin) e de cosseno(Cos) desse ângulo. Multiplicamos então, Dist por Cos, para achar X e Dist por Sin, para achar Y.

4) quarter(X,Y, NumberQuarter)

quarter(X,Y, 1) :- X >= 0 , Y >= 0 .

quarter(X, Y, 2) :- X < 0, Y >= 0 .

quarter(X, Y, 3) :- X =< 0, Y < 0 .

quarter(X, Y, 4) :- X > 0, Y < 0 .

Esse axioma verifica em qual quadrante do círculo trigonométrico um ponto se encontra. Sendo X e Y positivos, então se encontra no primeiro quadrante. Se X é negativo e Y positivo, se encontra no segundo quadrante. Se X e Y são negativos, se encontra no terceiro quadrante. Por fim, se X é positivo e Y negativo, se encontra no quarto quadrante.

5) angularLocation((X, Y), Ang, Dist)

```
angularLocation((X, Y), Ang, Dist):- Square is X**2 + Y**2,
sqrt(Square, Dist),
AuxSin is Y / Dist,
asin(AuxSin, RadAng),
AuxAng is 180 * RadAng / pi,
(
        quarter(X, Y, 1) -> Ang is AuxAng;
        (
                quarter(X, Y, 2) -> Ang is 180 - AuxAng;
                        (
                                quarter(X, Y, 3)-> Ang is 180 - AuxAng;
                                quarter(X, Y, 4),
                                Ang is 360 + AuxAng
                        )
        )
).
```

Esse axioma obtém o angulo em relação a abscissa e a distância de um ponto à origem, também por trigonometria. A distância é obtida através de pitágoras (é a hipotenusa). O ângulo é obtido através do arco-seno de Y por Dist. Como o resultado dessa conta vai de -

90 a 90 graus, é necessário saber em qual quadrante o ponto está localizado, para descobrir o ângulo real.

6) getSizeAndCenterDepth(Depth, AngDiam, Size, CenterDepth)

```
getSizeAndCenterDepth(Depth, AngDiam, Size, CenterDepth):-
SinAngDiamDiv2Rad = sin((AngDiam/2) * (pi/180)),
Num is 2 * Depth * SinAngDiamDiv2Rad ,
Denom is 1 - SinAngDiamDiv2Rad,
Size is (Num / Denom),
CenterDepth is Depth + (Size / 2).
```

Esse axioma obtém o diâmetro físico em metros (Size) e a distância ao centro do objeto(CenterDepth), recebendo apenas a distância aparente à "borda" do objeto (Depth) e seu diâmetro angular (AngDiam). Como assumimos que todos os objetos são circunferências, a distância da origem ao centro do objeto é o mesmo que a distância aparente mais o raio do objeto. A distância da origem ao centro do objeto, à tangente da circunferência que passa pela origem e o raio do objeto formam um triangulo retângulo. Manipulando as fórmulas de trigonometria, chegamos à seguinte relação para calcular o diâmetro do objeto:

$$\text{Diametro} = \frac{2 * \text{Distância Aparente} * (\text{sen}(\text{Diâmetro Angular}/2))}{(1 - (\text{sen}(\text{Diâmetro Angular}/2)))}$$

Tendo o diâmetro, a distância ao centro do objeto é apenas a distância aparente mais o raio da circunferência (Size/2).

7) distancesPeakToDefinedLines(peak(Body, Depth, Size, Ang, Time), HFL, HBL, VOR, VIR, VIL, VOL)

```
distancesPeakToDefinedLines(Peak, HFL, HBL, VOR, VIR, VIL, VOL):-
peak(_, Depth, Size, Ang, _) = Peak,
Radius is Size/ 2,
relativeLocation(Ang, Depth, (X, Y)),
horizFrontLine(A_HFL, B_HFL, C_HFL),
distancePointToLine((X,Y), A_HFL, B_HFL, C_HFL, DistHFL),
HFL is Radius-DistHFL,
horizBackLine(A_HBL, B_HBL, C_HBL),
distancePointToLine((X,Y), A_HBL, B_HBL, C_HBL, DistHBL),
HBL is Radius-DistHBL,
vertOuterRightLine(A_VOR, B_VOR, C_VOR),
distancePointToLine((X,Y), A_VOR, B_VOR, C_VOR, DistVOR),
VOR is Radius-DistVOR,
vertInnerRightLine(A_VIR, B_VIR, C_VIR),
```

```
distancePointToLine((X,Y), A_VIR, B_VIR, C_VIR, DistVIR),
VIR is Radius-DistVIR,
vertInnerLeftLine(A_VIL, B_VIL, C_VIL),
distancePointToLine((X,Y), A_VIL, B_VIL, C_VIL, DistVIL),
VIL is Radius-DistVIL,
vertOuterLeftLine(A_VOL, B_VOL, C_VOL),
distancePointToLine((X,Y), A_VOL, B_VOL, C_VOL, DistVOL),
VOL is Radius-DistVOL.
```

Este axioma define a distância inversa de um objeto à todas as linhas virtuais. Inicialmente, obtemos o raio do objeto (*Size*/2). Depois, achamos a posição de seu ponto central no plano cartesiano (*relativeLocation*(...)). Tendo essas duas informações, basta subtrair o raio do objeto pela distância de seu centro às linhas virtuais. Ou seja, se uma linha virtual passar dentro do objeto, sua distância inversa será positiva (Raio maior que a distância). Do contrário, ela será negativa (Raio menor que a distância). Sendo que:

- HFL é a distância inversa do ponto à linha horizontal frontal;
- HBL é a distância inversa do ponto à linha horizontal traseira;
- VOR é a distância inversa do ponto à linha vertical direita externa;
- VIR é a distância inversa do ponto à linha vertical direita interna;
- VIL é a distância inversa do ponto à linha vertical esquerda interna;
- VOL é a distância inversa do ponto à linha vertical esquerda externa.

8) oneGreaterThanZero(A, B)

```
oneGreaterThanZero(A, _):- A > 0 .

oneGreaterThanZero(_, B):- B > 0 .
```

Axioma utilizado para fazer uma comparação se os valores A ou B são maiores que zero, deste modo retornando true caso um deles for maior que zero.

9) description(Time, [DepthProfileList])

```
description(Time, [foreground(Body,Depth,AngDiam)|Tail]):-

description(Time, Tail),

getAngDist([foreground(Body,Depth,AngDiam)|Tail], AngDist1),

AngDist is AngDist1 - AngDiam/2,

getTime(Time, NumberTime), % remove the 't' from the beginning of the time

getSizeAndCenterDepth(Depth, AngDiam, Size, CenterDepth),

assert(peak(Body,CenterDepth,Size,AngDist, NumberTime)),

retractall(situation(Body, NumberTime, _)),

holdsAt(Body, NumberTime, _).

description(Time, [background(_,_,_)|Tail]):- description(Time, Tail).
```

Este axioma é responsável por adicionar os picos atuais provenientes da conexao Java via socket, assim, para ser incluído na base de conhecimento, deve existir no campo de visão do veículo, caso contrário o mesmo é considerado apenas background do instante em análise.

Portanto, com a existência de picos são efetuadas as seguintes passagens, cálculo da distância angular em relação ao veículo inteligente em getAngDist(...), cálculo da distância ao centro do objeto em *getSizeAndCenterDepth*(...). Por fim é retirada qualquer "sujeira"da base de conhecimento sobre a situação neste determinado instante de tempo que está sendo inserido e verificação/inserção do estado do objeto neste instante em comparação com o anterior.

Interfaces com o Java:

1) performFunction(Function, FunctionArguments, Response)

```
performFunction(0, FunctionArguments, Response) :-
description(Time, _) = FunctionArguments,
getTime(Time, TimeNumber),
retractall(peak(_, _, _, _, TimeNumber)),
(
        FunctionArguments-> forecast(TimeNumber),
        Response = 'true';
        Response = 'false'
),
previousFrames(TimeNumber, NumberOfFrames),
CleanUpTime is TimeNumber - NumberOfFrames - 1,
```

```
 retractall(situation(_, CleanUpTime, _)),
 retractall(peak(_, _, _, _, CleanUpTime)).
 performFunction(1, FunctionArguments, Response) :-
(
 queryObjectInColisionRoute(FunctionArguments)-> Response = 'true';
 Response = 'false'
 ).
 performFunction(2, FunctionArguments, Response) :-
(
 queryUnsafeOvertaking(FunctionArguments)-> Response = 'true';
 Response = 'false'
 ).
 performFunction(3, FunctionArguments, Response) :-
(
 queryVelocityReduceNeeded(FunctionArguments)-> Response = 'true';
 Response = 'false'
 ).
 performFunction(4, _, Response) :- Response = end_prolog.
```

Realiza a função *Function* requisitada pelo Java, repassando os parâmetros recebidos, com exceção da *Function* = 0, onde há um processamento extra que, incialmente, limpa todos os *peaks* do instante atual (*retractall*(...)) para, então, chamar o axioma *description* (contido em *FunctionArguments*), o axioma *forecast*, e, por fim, realizar uma limpeza de *peaks* e *situations* de instantes que não serão mais utilizados para melhorar a performance.

2) connect(Port)

```
connect(Port) :- tcp_socket(Socket),
gethostname(Host),  % local host
tcp_connect(Socket,Host:Port),
tcp_open_socket(Socket,INs,OUTs),
 assert(connectedReadStream(INs)),
assert(connectedWriteStream(OUTs)).
:- connect(22105).
spatialReasoning :- connectedReadStream(IStream),
read(IStream,(Function, RequestNumber, FunctionArguments)),
performFunction(Function, FunctionArguments, Response),
connectedWriteStream(OStream),
write(OStream, (RequestNumber, Response)),
nl(OStream), flush_output(OStream),
```

spatialReasoning.

:- spatialReasoning.

Se conecta à porta definida em Port para se comunicar com o Java. Utiliza funções prórpias do prolog para realizar a conexão. Axioma criado com base no modelo apresentado por FISHER, John (200-).

APÊNDICE E – CENÁRIOS DE TESTE

A seguir, estão descritos os cenários de teste criados de acordo com o escopo proposto para o projeto. Baseado na execução desses cenários, pudemos observar o comportamento do Simulador com relação às diversas situações apresentadas.

O resultado das execuções é apresentado com mais detalhes na Seção 5.1 desse documento. As figuras citadas nos cenários, embora não incluídas nesse documento, foram utilizadas apenas para nossa orientação e tratam as situações apresentadas na coluna Descrição de cada cenário.

#	Descrição	Dados de Entrada	Resultados Esperados
Funcionalidades do Editor			
1	Iniciar nova Simulação	Acessar menu File -> Open Scenary Escolher os itens 'Cenário_Baixa.sce', 'Cenário_Média.sce' e 'Cenário_Alta.sce' individualmente	Verificar se os três cenários foram exibidos corretamente
2	Simulação sem veículo Observador	Executar simulação sem que tenha sido incluído um Agente	A mensagem 'The simulation must have agent' deve ser exibida
3	Salvar uma Simulação	Incluir objetos e seus pontos de trajetória e acessar o menu File -> Save Simulation Definir nome do arquivo e localização	A mensagem 'Simulation saved sucessfully' deve ser exibida Verificar no destino especificado se o arquivo definido foi salvo
4	Utilizar uma Simulação salva anteriormente	Acessar menu File -> Open Simulation Definir diretório e localizar arquivo salvo no Cenário 3	O cenário deve ser visualizado, juntamente com os objetos e suas trajetórias, exatamente como haviam sido definidos antes de ser salvo
Funcionalidades de Objetos			
5	Inclusão de Objetos e Definição de Pontos de Trajetória	Através da *Pallette*, realizar a inclusão de um Pedestre, de um Veículo e de um Agente. Após isso, definir pontos de trajetória para todos os objetos Realizar a tentativa de inclusão de um segundo Agente	Os objetos devem ser incluídos e os pontos de trajetória devem ter ter suas localizações validadas, com a permissão de inclusão apenas dentro das ruas O sistema não deve permitir a inclusão de mais de um Agente

6	Velocidade dos Objetos na simulação	Utilizar os objetos e trajetórias definidos no Cenário 5 para definir as Velocidades de simulação	Devem ser permitidos apenas Valores Numéricos entre 0 e 4 para as velocidades dos objetos Ao incluir valores diferentes dos permitidos, o editor deve descartar a alteração e manter o valor anterior
7	Exclusão de Objetos	Utilizar os objetos e pontos de trajetória definidos no Cenário 5	O sistema deve permitir a exclusão de objetos e deve, automaticamente, excluir os pontos de trajetórias referentes aos objeto selecionado O sistema não deve permitir a exclusão de ruas
Identificação de Situações de Risco			
8	Cenário Baixa Complexidade sem situações de risco	Inclusão de 3 veículos Inclusão de 2 pedestres Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme figura 01_Cen8.jpg)	O agente deve percorrer toda a trajetória definida pelo usuário sem que sejam exibidos alertas de situações de risco
9	Cenário Baixa Complexidade com situações de risco	Inclusão de 3 veículos Inclusão de 2 pedestres Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme fig. 02_Cen9.jpg)	O agente deve percorrer toda a trajetória definida pelo usuário com a exibição de alertas para as situações de risco detectadas
10	Cenário Média Complexidade sem situações de risco	Inclusão de 6 veículos Inclusão de 4 pedestres Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme fig. 03_Cen10.jpg)	O agente deve percorrer toda a trajetória definida pelo usuário sem que sejam exibidos alertas de situações de risco
11	Cenário Média Complexidade com situações de risco	Inclusão de 6 veículos Inclusão de 4 pedestres Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme fig. 04_Cen11.jpg)	O agente deve percorrer toda a trajetória definida pelo usuário com a exibição de alertas para as situações de risco detectadas
12	Cenário Alta Complexidade sem situações de risco	Inclusão de 9 veículos Inclusão de 6 pedestres Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme fig. 05_Cen12.jpg)	O agente deve percorrer toda a trajetória definida pelo usuário sem que sejam exibidos alertas de situações de risco

13	Cenário Alta Complexidade com situações de risco	Inclusão de 9 veículos Inclusão de 6 pedestres Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme fig. 06_Cen13.jpg)	O agente deve percorrer toda a trajetória definida pelo usuário com a exibição de alertas para as situações de risco detectadas
Visualização de Objetos no Perfil de Profundidade			
14	Quantidade Baixa de Objetos	Inclusão de 1 veículo Inclusão de 1 pedestre Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme fig. 07_Cen14.jpg)	O perfil de profundidade deve exibir os dois objetos incluídos na simulação (com exceção do próprio agente), sendo possível observar o período em que os mesmos permanecem no campo de visão do agente, como também a distância em que eles apresentam um do outro
15	Quantidade Média de Objetos	Inclusão de 2 veículos Inclusão de 2 pedestres Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme fig. 08_Cen15.jpg)	O perfil de profundidade deve exibir todos objetos incluídos na simulação (com exceção do próprio agente), sendo possível observar o período em que os mesmos permanecem no campo de visão do agente, como também a distância em que eles apresentam um do outro
16	Quantidade Alta de Objetos	Inclusão de 3 veículos Inclusão de 3 pedestres Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme fig. 09_Cen16.jpg)	O perfil de profundidade deve exibir todos objetos incluídos na simulação (com exceção do próprio agente), sendo possível observar o período em que os mesmos permanecem no campo de visão do agente, como também a distância em que eles apresentam um do outro
Performance da Aplicação			
17	Quantidade Baixa de Objetos	Inclusão de 6 veículos Inclusão de 4 pedestres Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme fig. 10_Cen17.jpg)	A aplicação deve apresentar a simulação em andamento e não são toleráveis delays entre o comando para início da simulação e a execução de fato. Além disso, a aplicação não deverá apresentar delays para exibição de alertas nem travar.
18	Quantidade Média de Objetos	Inclusão de 9 veículos Inclusão de 6 pedestres Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme fig. 11_Cen18.jpg)	A aplicação deve apresentar a simulação em andamento e não são toleráveis delays entre o comando para início da simulação e a execução de fato. Além disso, a aplicação não deverá apresentar delays para exibição de alertas nem travar.

19	Quantidade Alta de Objetos	Inclusão de 12 veículos Inclusão de 8 pedestres Inclusão de 1 agente Definição de pontos de trajetória para todos os objetos envolvidos na simulação (conforme fig. 12_Cen19.jpg)	A aplicação deve apresentar a simulação em andamento e não são toleráveis delays entre o comando para início da simulação e a execução de fato. Além disso, a aplicação não deverá apresentar delays para exibição de alertas nem travar.