Universidade Federal de Santa Catarina

Departamento de Informática e Estatística

Simulador de Robôs Móveis Terrestres

Florianópolis

2019

Ricardo Ademar Bezerra de Almeida

Simulador de Robôs Móveis Terrestres

Trabalho de conclusão de curso submetido ao curso de Ciência da Computação como parte dos requisitos para a obtenção de Grau de Bacharel em Ciências da Computação.

Orientadora: Prof.ª Patricia Della Méa Plentz

Florianópolis

2019

Ricardo Ademar Bezerra de Almeida

Simulador de Robôs Móveis Terrestres

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de “Bacharel em Ciências da Computação”, e aprovado em sua forma final pelo Curso de Ciências da Computação.

Florianópolis, 1 de abril 2019

_____

Prof.ª Patricia Della Méa Plentz

Coordenadora

Banca examinadora:

_____

Prof. Leandro José Komosinski

_____

Prof.ª Luciana de Oliveira Rech

Dedico este trabalho a minha família que sempre acreditou em mim e priorizou a minha educação acima de tudo.

Agradecimentos

Agradeço a Prof.ª Patricia Plentz pela ajuda e oportunidade de trabalhar neste projeto. Também agradeço meus amigos cujo suporte foi essencial para o término desse trabalho.

Resumo

Atualmente a robótica móvel é uma área bastante abrangente, com vários tipos de robôs realizando uma diversidade de tarefas, de forma a automatizar trabalhos que eram previamente realizadas por pessoas. No entanto, para testar diferentes casos e situações em que esses robôs devem atuar, pode ser muito caro e demorado utilizando os mesmos. Por isso a utilização de simuladores para robôs móveis é uma decisão comum para definir e solucionar problemas de modo mais barato e rápido. O objetivo deste projeto é desenvolver um simulador para robôs terrestres autônomos em um ambiente parcialmente conhecido, onde os elementos estáticos já estão mapeados mas os móveis, como pessoas e outros robôs não, como por exemplo um galpão de estoque da Amazon, onde múltiplos robôs vão compartilhar o mesmo ambiente com o objetivo de carregar e transportar itens evitando colisões e buscando o melhor caminho aos seus objetivos, obedecendo as deadlines estipuladas. Este simulador foi desenvolvido utilizando o framework Cocos2d-x, comumente utilizado no desenvolvimento de jogos, e testado em um caso proposto demonstrando sua utilização, no qual foram obtidos resultados dos tempos demonstrando que um algoritmo de busca de produtos por distancia se mostrou mais eficiente que os demais testados.

Palavras-Chaves: simulador, robótica móvel, swarm robots, busca de caminho, prevenção de colisão.

Abstract

Currently mobile robotics is an area of expertise very broad, with many different types of robots performing a variety of tasks to automate work that was generally done by people. However, to test so many different cases in which robots can find themselves can be very expensive, unless we utilize a simulator to do so. With the growth of robotic came the increase need for simulators to perform different tasks and adapt to a variety of needs, such as different robots, environments and ways of navigating. The objective of this thesis is to develop of a simulator for terrain robots on an already known environment mimicking a transport robot, similar to the ones seem on Amazon's warehouses, allowing a swarm of robot to share the same environment without colliding between themselves while looking for optimal paths as a way to reduce the cost and time of the tasks being performed, respecting deadlines that were previously determined. On this project a simulator was developed using the framework Cocos2d-x, usually used for the development of games, and a test case was proposed to demonstrate his functionalities, on these tests the results acquired show that an algorithm with focus on search of products by distance being the most efficient on completing the task that was specified.

Key-words: simulator, mobile-robotic, swarm robots, pathfinding, collision prevention.

Lista de Tabelas

Lista de Figuras

Lista de Abreviaturas e Siglas

GPS	Global Positioning System
RTS	Real Time System
2D	Duas dimensões
3D	Três dimensões
ROS	Robot Operating System
OpenCV	Open Computer Vision
GUI	Graphical User Interface
API	Application Programming Interface
ODE	Open Dynamics System

Sumário

Capítulo 1

1. Introdução

Atualmente o uso de robôs móveis para realizar diversas tarefas tem crescido, uma das razões para esse aumento é a redução do custo de hardware incentivando o mercado a utilizar robôs como um método para uma melhor eficiência na realização de determinadas tarefas antes realizadas por pessoas [MARKS, 2019].

Robôs móveis podem ser utilizados em uma variedade de funções a fim de completar certos objetivos, como por exemplo na exploração de ambientes [JIA, 2004], até mesmo fora do planeta [NILSON, 2018], no deslocamento de cargas de um ponto a outro, como nos armazéns da Amazon ou na perseguição de alvos móveis [MARKS, 2019].

Uma grande parte dessas tarefas é limitada por restrições temporais. Por exemplo, um robô da Amazon precisa entregar pacotes em um ponto de entrega em um tempo limite [MARKS, 2019], ou então um robô em Marte deve explorar o máximo do ambiente em que se encontra para adquirir uma maior quantidade de informações antes da sua capacidade energética se esgotar [NILSON, 2018]. Como essas tarefas devem ser realizadas em um tempo limite, robôs móveis são considerados Sistemas de Tempo Real (RTS, *Real-Time Systems*) [BREGA, 2000] e a falta de eficiência nesses casos pode causar grande prejuízo financeiro aos indivíduos ou organizações responsáveis por esses robôs [MARKS, 2019].

No entanto o problema não termina por aí, pois nem sempre esses robôs agem individualmente, mas sim em conjunto com vários outros robôs, humanos ou qualquer outra entidade que possa afetar o movimento dos robôs. Com isso é necessário a utilização de sensores ou conhecimento prévio do ambiente, e por vezes ambos, para que o robô realize suas tarefas com sucesso e sem colisão, evitando causar danos aos mesmos ou em determinados casos à pessoas, animais, ao ambiente ou outros equipamentos que compartilham o local onde o robô se movimenta.

Em diversos domínios como aviação, automação e robótica são utilizados sistemas físicos, que são sistemas que simulam um espaço físico onde os elementos dentro dele possuem propriedades físicas como gravidade, aceleração, etc. Neste espaço físico, diferentes objetos interagem entre si, podendo possivelmente colidir uns com os outros. Sistemas de controle de robôs evitam a colisão entre eles quando um robô tenta percorrer de um ponto a outro e é interceptado por alguma entidade.

A criação de um sistema de simulação de robôs torna-se extremamente útil para evitar danificar robôs em diferentes ambientes físicos e também contribui para situações nas quais visa-se verificar a usabilidade de certos algoritmos que consigam atingir os requisitos de navegação, tempo e energia estimados. Além disso existem casos onde há necessidade de testar uma grande quantidade de robôs simultaneamente, para analisar se um possível projeto é viável, mas essa grande quantia de robôs não se tem a disposição de imediato. Para esses casos, dentre outros, a utilização de um simulador se torna extremamente necessária e muito mais barata em comparação com a alternativa.

Este trabalho propõe o desenvolvimento de um simulador 2D de robôs de carga em um ambiente parcialmente conhecido e de criticalidade *soft*, altamente modular utilizando o framework Cocos2d-x, permitindo ao usuário uma grande capacidade de modificar componentes de cada robô, adicionar ou desabilitar módulos e modificar o comportamento de cada robô de modo a realizar seus objetivos sejam eles para pesquisa ou para o mercado.

1.1 Motivação

Primeiramente um dos motivos para a implementação de um simulador robótico deve-se a necessidade de simuladores com capacidade de modularização para a solução de problemas que utilizam robôs móveis, reduzindo o custo e tempo necessário para testar possíveis cenários e verificar quais são as melhores soluções.

Além disso existem vantagens na criação de um simulador utilizando o framework Cocos2d-x, escolhido para este projeto e comumente utilizado no desenvolvimento de jogos. Ele disponibiliza um sistema físico pronto de fácil utilização que permite ao usuário escolher entre dois motores de física. Também há o fato dele possuir um código-aberto o que permite a modificação e adição de novos módulos contribuindo com a flexibilidade do Simulador para futuros desenvolvimentos.

Este projeto não é apenas útil como adição de um novo simulador para uso, mas também como via de pesquisa no desenvolvimento e arquitetura de um software de

simulação de robôs móveis, contribuindo para um melhor entendimento neste tipo de aplicativo para futuros desenvolvedores e pesquisadores interessados no assunto.

1.2 Objetivos

A implementação de um simulador para robôs móveis autônomos exige o desenvolvimento de algumas tarefas. Estas tarefas refletem o objetivo geral e se desdobram em objetivos específicos, os quais são descritos a seguir.

1.2.1 Objetivo geral

Desenvolver um simulador 2D para robôs móveis autônomos em um ambiente parcialmente conhecido com o intuito de simular o transporte de carga de um ponto a outro. Nele um robô percorre um caminho de um ponto inicial para um ponto destino e então volta ao seu ponto de origem quando não houver mais tarefas. Este simulador deve ser altamente modular e será desenvolvido utilizando o framework Cocos2D-x. A partir dele o usuário pode modificar o comportamento dos robôs de transporte, criar diferentes ambientes de simulação e adquirir os dados dos tempos decorridos e caminhos pelos quais os robôs tomaram para completar suas tarefas para fins de análise.

1.2.2 Objetivos específicos

- Implementar um Simulador em C++ para robôs autônomos 2D;

- Utilizar o framework Cocos2d-x no desenvolvimento deste simulador;
- Manter um nível de acoplamento baixo para fácil adição e remoção de módulos;
- Criar atributos que expressam os requisitos temporais dos robôs (tempo máximo permitido para finalizar uma tarefa);

1.3 Metodologia

Inicialmente foram levantadas pesquisas relacionadas ao tema do trabalho, como modos de navegação, localização e movimento de robôs para definir uma base teórica no desenvolvimento do protótipo do simulador. Após essa etapa foi definido a linguagem e o framework a ser utilizado para o desenvolvimento do simulador que se encaixam com os requisitos de um software desse tipo.

Com as ferramentas definidas foi iniciado o desenvolvimento do simulador com as características propostas a fim de cumprir os objetivos acima descritos. Por fim foram criados testes para garantir sua correta funcionalidade e então descrito a sua arquitetura e funcionamento neste documento.

1.4 Estrutura

No capítulo 2, são apresentados os conceitos básicos relacionados ao domínio do projeto quanto a robótica móvel e sistemas de tempo real. No capítulo 3 é discutido o framework Cocos2d-x que foi utilizado no desenvolvimento do simulador. Uma comparação e descrição de simuladores relevantes já existentes é realizada no capítulo

4. A estrutura do simulador desenvolvido, os módulos que o compõe e pseudocódigos mais relevantes serão descritos no capítulo 5. No capítulo 6 uma proposta de cenário é apresentada e o simulador desenvolvido neste trabalho de conclusão de curso (TCC) é utilizado a fim de verificar diferentes modos de resolver o problema apresentado, comparando os dados coletados a fim de definir qual das soluções é a mais apropriada. Por fim, no capítulo 7, são apresentadas as considerações finais quanto ao trabalho apresentando e sugerindo trabalhos futuros relacionados ao que foi desenvolvido neste TCC.

Capítulo 2

2. Fundamentação teórica

Este capítulo é reservado ao embasamento teórico para a compreensão do domínio deste trabalho. Aqui são descritos apenas os conceitos fundamentais relevantes a robótica que foram utilizados no desenvolvimento do simulador deste projeto. Inicialmente será discutido Sistemas de Tempo Real (STR) e as classificações desses sistemas, então é descrita a área da robótica móvel e os tipos de robôs e sensores que são comumente utilizados. Por fim foram apresentados os tipos de navegação que robôs podem ser categorizados dependendo de seu conhecimento do ambiente que se encontram.

2.1 Tempo real

Sistemas computacionais onde há uma restrição temporal que deve ser respeitada são chamados de sistemas de tempo real. Nesses sistemas o tempo de resposta para um evento deve ser respeitado e, caso ele ultrapasse um limite pré-definido, mesmo que a tarefa tenha sido executada ela será caracterizada como uma falha [FARINES, 2000]. Além disso, uma tarefa pode ser dividida em subtarefas que também possuem um tempo estipulado, com isso, mesmo que no final a tarefa tenha sido feita dentro do tempo limite, caso uma subtarefa tenha ultrapassado a *deadline*, isso também é caracterizado como uma falha em um sistema de tempo real.

Tarefas executadas em sistemas de tempo real podem ser classificadas de duas formas, periódicas, como o nome já diz, executam dentro de um determinado período; já as aperiódicas podem ocorrer em diferentes momentos, muitas vezes sendo executadas dependendo de uma reação de um sensor que foi ativado. A perda de um deadline podes ser classificada em diferentes níveis de criticalidade que serão discutidos a seguir.

2.1.1 Criticalidade

Um sistema de tempo real também pode ser classificado pelo impacto que pode ser causado pela perda de um deadline, definido o quão crítico é a necessidade de que esse sistema cumpra as suas tarefas em um tempo predefinido. As classificações de criticalidade são:

- *Hard*: são sistemas onde a perda de um *deadline* possui o risco de danificar equipamento ou até mesmo colocar vidas em risco. Um exemplo desse tipo de sistema seria um marca-passo que regula os batimentos cardíacos de um paciente.
- *Soft*: onde a perda de deadline não corresponde a uma falha completa, mas que mesmo assim, prevê consequências na performance do sistema. Isso pode afetar o seu funcionamento à medida que muitos deadlines são perdidos, um exemplo disso seria o sistema de som de um computador, a

perda de alguns bits pode ser difícil de detectar, mas se muitos são perdidos começa a se perceber a degradação no som emitido.

- *Firm*: similar ao soft, a perda de deadline não corresponde a falha do sistema, mas diferente do soft, múltiplas perdas podem fazer com que o sistema falhe por completo, por exemplo: em uma fábrica que utiliza robôs em uma linha de montagem, repetidas perdas de *deadline* podem eventualmente tornar impossível o cumprimento da tarefa, sendo necessário a parada completa do sistema.

2.2 Robótica Móvel

A robótica é uma área relativamente nova, ela se tornou inicialmente bastante popular graças a sua utilização na área de fabricação industrial com o uso de braços robóticos para realizar tarefas repetitivas rapidamente, como por exemplo a solda e pintura de peças para automóveis [SIEGWART; NOURBAKHSH, 2011].

No entanto ainda existem várias tarefas que se enquadram como repetitivas e que poderiam ser realizadas por robôs para uma maior eficiência e rapidez que necessitam que estes robôs se movam. A adição de locomoção para robótica trás vários benefícios à indústria, no entanto adiciona muitos problemas pois em geral esses robôs estão se locomovendo juntamente com outros robôs e pessoas, em locais desconhecidos ou parcialmente conhecidos para realizar tarefas que podem ser consideradas inseguras.

Para se mover com segurança e compreensão do ambiente, robôs móveis utilizam sensores que permitem evitar acidentes. A seguir, será discutido os tipos de robôs móveis e os sensores comumente utilizados por eles para compreensão dos ambientes em que se encontram.

2.2.1 Tipos de Robôs

Robôs móveis podem ser divididos entre dois tipos: os tele operados, robôs controlados por uma pessoa a todo momento durante seu funcionamento e os autônomos, robôs que se movem por conta própria seguindo um algoritmo interno que define seu comportamento e o uso de sensores que permitem que ele compreenda e responda adequadamente ao ambiente que se encontra.

Além disso robôs móveis podem se enquadrar em diferentes subcategorias, sendo elas os robôs terrestres, aquáticos e aéreos.

2.2.2 Sensores

Como foi estabelecido anteriormente, robôs autônomos necessitam de sensores para compreender o ambiente no qual eles se encontram, especialmente se esse ambiente se encontra em constante mudança, como um ambiente que contenha objetos com mobilidade. Sensores desse tipo são classificados como sensores externos pois seu objetivo é obter informações sobre o ambiente no qual o robô se encontra. Graças a

esses sensores o robô pode reagir a essas mudanças e tomar diferentes decisões. Dependendo do ambiente, pode ser absolutamente necessário para que ele, no mínimo, complete suas tarefas, mas também pode contribuir para sua eficiência e rapidez nisso.

Além de sensores externos também há os sensores internos, responsáveis por informações do próprio robô. Como, por exemplo, o ângulo de uma junta em um robô com patas ou a quantidade de rotações que uma roda deu em um intervalo de tempo.

Existem diversos tipos de sensores que podem contribuir para que um robô móvel tenha uma melhor compreensão do que está acontecendo ao seu redor, dentre eles se destacam:

- Contato: permite que o robô consiga detectar uma colisão.
- GPS: disponibiliza ao robô uma posição global para que ele possa se localizar.
- Câmeras: com a utilização de algoritmos da área de Visão Computacional, permite que eles interpretem uma quantidade de informação através de das imagens produzidas por câmeras.
- *Lasers*: capaz de medir o tempo que um feixe de luz emitido leva para alcançar um objeto, estimando a distância do robô até este objeto com grande precisão.
- Odômetro: estima a distância percorrida por um robô através da quantidade de rotações de uma roda.
- Giroscópio: sensor que avalia a orientação de um objeto em relação a um eixo de referência.

2.2.3 Navegação

Além de distinguir os tipos de robôs e os sensores que eles utilizam que são características dos robôs em si, também é necessário diferenciar os diferentes tipos de ambientes nos quais eles se locomovem, os quais podem ser divididos em duas categorias, ambiente conhecidos, não-conhecidos e os parcialmente conhecidos.

2.2.3.1 Ambientes conhecidos

São considerado ambientes conhecidos aqueles que já foram previamente mapeados e os quais o robô possui este mapa internamente, não necessitando de sensores para determinar qual o caminho deve ser tomando, considerando que não haja entidades se movimentando e que o mapa esteja atualizado, ou seja, o ambiente não foi modificado, o diferenciando do mapa que o robô possui. Este tipo de ambiente facilita a navegação do robô quando consideramos a necessidade de achar um caminho ótimo que permita que o robô realiza suas tarefas no menor tempo possível. No entanto, em uma grande parte das situações e tarefas que se necessitam de robôs móveis não é possível um ambiente deste tipo.

2.2.3.2 Ambiente não-conhecidos

Quando um robô não tem um mapa do ambiente no qual se locomove dizemos que o ambiente é não-conhecido, para navegar em um ambiente não-conhecido é

necessário que o robô disponha de sensores, como um dos mencionados anteriormente, para que ele compreenda obstáculos e percorra o local a fim de cumprir seu objetivo com sucesso. Além disso também é necessário ressaltar a necessidade, muitas vezes comum, de que o robô não só cumpra a sua tarefa, mas que faça o mesmo sem causar danos a si mesmo.

Também é comum que à medida que um robô caminhe em um ambiente não conhecido, com a ajuda de seus sensores ele pode mapear este ambiente, permitindo que ele aprenda o seu *layout* e caso haja uma segunda tarefa no mesmo ambiente, esta tarefa pode ser executada em um menor tempo, é bastante comum que robôs de limpeza façam esse tipo mapeamento.

2.2.3.3 Ambientes híbridos ou parcialmente conhecidos

Existem diversas formas nas quais um ambiente pode ser híbrido, é possível que se conheça todos os obstáculos estáticos, sem movimento, mas que haja entidades se movendo, havendo a necessidade de algum tipo de sensor externo. Também é possível que se conheça apenas parte do ambiente, talvez ele não tenha sido completamente mapeado e com isso também haveria necessidade de sensores para corretamente realizar tarefas.

Neste projeto são utilizados ambientes híbridos onde o usuário insere inserir os objetos estáticos antes da simulação começar e o robô utiliza apenas os sensores para detectar entidades móveis como outros robôs.

2.3 Considerações finais

Com tudo isso que foi visto, os diferentes ambientes, sensores e tipos de robôs que existem, pode-se notar que há uma grande quantidade de sistemas que interagem entre si de modo que dependendo das tarefas que os robôs devem realizar, eles podem ser classificados em diversos modos, seja pelo ambiente em que navegam, os sensores que utilizam ou a criticalidade dos requisitos temporais que devem cumprir.

Capítulo 3

3. Framework

Neste capítulo é discutido o *framework* utilizado no desenvolvimento do simulador proposto neste projeto. Como a ferramenta principal para este projeto foi o Cocos2d-x, este capítulo será dedicado a explicação de seu funcionamento, os principais módulos utilizados neste projeto que fazem parte dele, incluindo os motores físicos e a criação de uma GUI com o uso de *Sprites*, *Scenes*, *Layers* e outros componentes.

3.1 Cocos2d-x

A principal ferramenta para o desenvolvimento deste simulador foi o framework Cocos2d-x. Em geral este *framework* é voltado para o desenvolvimento de jogos e apresenta, portanto, muitas características semelhantes àquelas necessárias para a criação de simuladores para robótica móvel como, por exemplo, ambientes físicos, interfaces gráficas e facilidade para interação do usuário com o sistema. Estes fatores foram decisivos para a escolha da sua utilização neste projeto. Além disso, o fato dele ser um *framework* de código-aberto, permitindo a sua modificação interna caso necessário, o torna bastante interessante para a comunidade científica.

Além da facilidade na criação da interface gráfica, ele fornece dois motores físicos a disposição do desenvolvedor, que são necessários no desenvolvimento deste projeto

já que prevemos que múltiplos robôs irão compartilhar o mesmo espaço físico e com isso podem colidir um com o outro.

Por fim, ele fornece um sistema de troca de mensagens que permite que dois módulos que não se conhecem comuniquem entre si. Isso contribui para que o design do protótipo possa manter alta modularidade que permita desativar ou desabilitar módulos sem quebrar o sistema e sem impedir o funcionamento do simulador.

Ele suporta múltiplas plataformas, incluindo iOS, Android, Tizen, Windows, Windows Phone 8, Linux e Mac OS X. O desenvolvimento utilizando sua API é suportado para as linguagens *Javascript*, *C*++ e Lua e a versão utilizada neste projeto foi a 3.17.1.

Na Figura 1 podemos ver a arquitetura geral do *Cocos2d-X* e seus diferentes módulos, *Cocos2d-x Games* representa o aplicativo desenvolvido utilizando uma de suas APIs, sendo o *Cocos2d C*++ *Engine* e o *C*++ *APIs* utilizados no desenvolvimento do simulador criado neste projeto. Por fim, na parte inferior da figura é possível distinguir as plataformas suportadas, que dependem da API sendo utilizada.

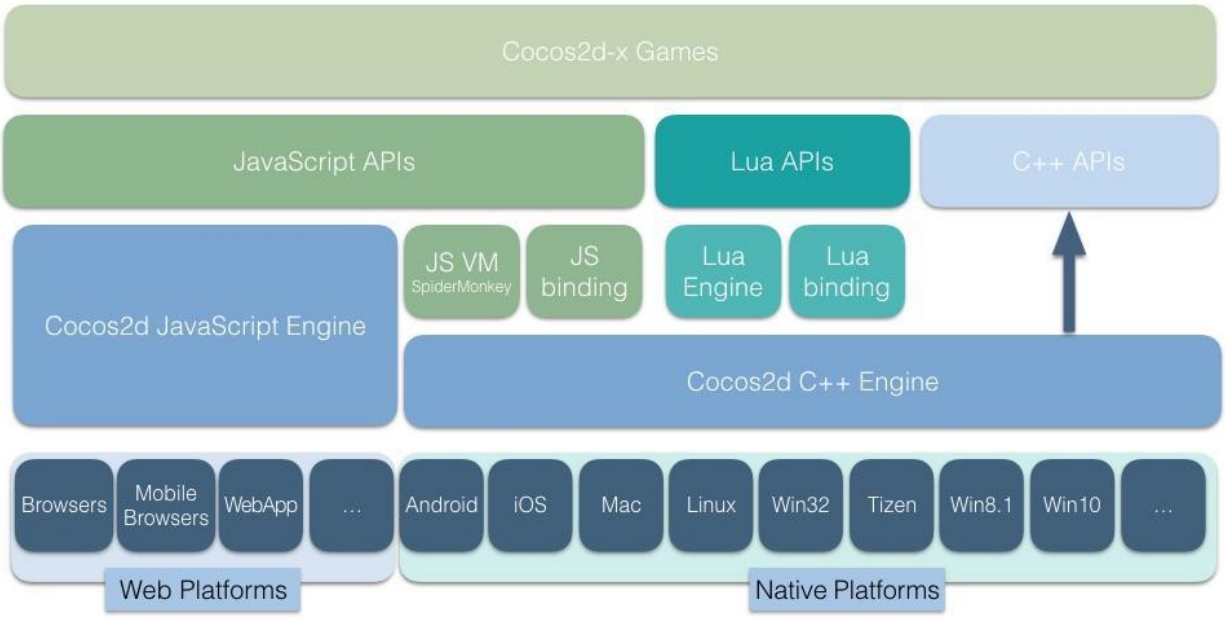

Figura 1 Arquitetura do framework Cocos2d-x

A seguir será descrito os principais componentes que serão utilizados no desenvolvimento do simulador, incluindo Sprites que permitem a visualização dos robôs na GUI e motores físicos. Esses componentes serão explicados de modo a disponibilizar ao leitor uma melhor compreensão de como eles interagem entre si e contribuem para o desenvolvimento do simulador.

3.1.1 Principais componentes

Para compreender e utilizar o Cocos2D-x é necessário primeiramente o entendimento de alguns componentes básicos que são comumente utilizados para o desenvolvimento de jogos usando este *framework*, eles são brevemente descritos a

seguir:

- *Node:* elemento básico com propriedades como posição, escala e ponto ancora.
- *Sprite*: elemento visual 2D de um aplicativo.
- *Action*: ação realizada com o intuito de modificar as propriedades de um nodo.
- *Layer*: contêiner de nodos.
- *Scheduler*: responsável por ativar *callbacks* em determinado tempo.
- *Scene*: elemento abstrato que define uma cena de um jogo que contem *layers*.
- *Director:* responsável pela mudança de cenas.

3.1.2 Sprites

Sprites são primitivas básicas do framework que podem representar um elemento 2D. Todas as sprite podem apresentar uma textura que representa o seu elemento visual que será disponibilizado na tela. Elas são processadas pelo motor gráfico desde que estejam incluídas na *Scene Graph*.

O *Scene Graph* consiste em uma árvore que determina a ordem na qual esses nodos serão processados pelo motor gráfico. Na Figura 2 é demonstrado um exemplo de uma dessas árvores, nela, cada nodo é uma sprite. Seguindo a linha de processamento, vista em pontilhado, é possível determinar que, primeiramente a Sprite “A” é processada, seguida pela “B” e assim por diante até a Sprite “I”.

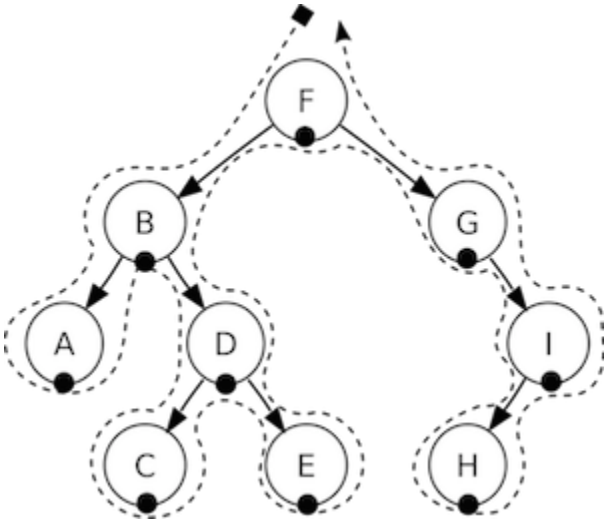

Figura 2 Ordem no qual elementos do Scene Graph são processados pelo OpenGL.

Cada objeto na tela pode ser representado como uma *Sprite*, que possui os dados de sua localização, dimensões e textura, estes que podem ser facilmente traduzidos como características de um robô móvel neste projeto e por isso cada robô será uma subclasse da classe *Sprite*.

No Cocos2D uma *Sprite* possui a seguintes propriedades:

1. *Position*: posição relativa ao nodo pai.
2. *Anchor Point*: ponto referência para movimentos e rotações, sendo um *Anchor Point* de (0.5, 0.5) o centro da *Sprite*.
3. *Scale*: valor de razão que aumenta ou reduz o tamanho da imagem sendo visualizada, por exemplo: uma *Scale* de 2.0 mostra uma imagem duas vezes maior que a suas dimensões iniciais.
4. *Color*: modifica a coloração da imagem original.
5. *Opacity*: define a transparência da *sprite.*

3.1.3 Motores físicos

Além da facilidade na criação da interface gráfica, Cocos2d-x já disponibiliza dois motores físicos para simular objetos em um espaço físico, o *Chipmunk* e *Box2D* que podem ser permutados sem necessidade de modificar a estrutura ou código do projeto.

Através desses motores físicos pode se atribuir um corpo físico para qualquer Sprite, lhes dando propriedades físicas que permite a colisão de corpos, aceleração e gravidade a essas Sprites, o que neste projeto se traduz em adicionar essas propriedades aos robôs móveis.

Para fazer isso é necessário a criação de um *PhysicsBody*, e o adicionando como componente da *Sprite*. Feito isso quaisquer outras *Sprite* que também possuam um *PhysicsBody* poderão interagir no mesmo espaço físico. Essa interação é determinada por um dos motores físicos incluídos no Cocos2d-x, a seguir será feita uma descrição de ambos e seus principais componentes.

3.1.3.1 Box2D

O Box2D é um motor de física gratuito e de código aberto, escrito em C++, mas com suporte a outras linguagens como Javascript, Lua, Java, etc. Ele possui um conjunto de conceitos básicos que são necessários compreender para sua utilização, que são

brevemente definidos a seguir:

- *Shape*: formato 2D de um corpo, como um círculo ou polígono.
- *Rigid Body*: um objeto material que possui massa e não é deformável.
- *Fixture*: conecta uma forma com um corpo e retêm propriedades como densidade, fricção e restituição.
- *World*: um conjunto que mantém elementos como corpos e fixações.
- *Constraint*: é uma conexão física que impede certos movimentos de um corpo.
- *Contact Constraint*: um tipo especial de conexão que é designada a prevenir penetrações de corpos rígidos e restituição.
- *Joint*: é um tipo de *constraint* utilizado para conectar dois corpos.
- *Joint Limit*: é utilizado para restringir o alcance ou variedade de um movimento.
- *Joint Motor*: aciona um conjunto de movimentos de acordo com seu grau de liberdade.
- *Solver*: utilizado para avançar no tempo e resolver o contato.

O Box2D é dividido em três módulos, cada um responsável por um aspecto do motor:

- *Common*: possui o código para alocação de memória, fórmulas matemáticas e configurações gerais.
- *Collision*: define os formatos e funções de colisão.
- *Dynamics*: responsável por simular corpos, fixações e articulações.

3.1.3.2 Chipmunk

Assim como o Box2D, o Chipmunk é uma biblioteca para simulação de física 2D distribuído sobre a licença MIT. Existem quatro tipos de objetos básicos que são utilizados nele descritos abaixo:

Rigid Bodies: um corpo rígido possui propriedades físicas de um objeto (massa, posição, velocidade, etc.) no entanto inicialmente ele não possui um formato(shape) a não ser que esta seja adicionado a ele.

Collision Shapes: são formatos que podem ser conectados a um corpo e que será usado para determinar o seu corpo quando uma colisão ocorrer. Mais de um formato pode ser adicionado ao mesmo corpo para criar formatos mais complexos.

Constrains/Joints: definem como dois corpos são interligados um com o outro.

Spaces: são ambientes para a simulação de objetos no Chipmunk. *Bodies*, *Shapes* e *Joints* são adicionados a um mesmo espaço e simulados nele como um todo. Eles controlam como todos os corpos, formatos e juntas interagem uns com os outros.

3.1.4 Troca de mensagens

Para a criação de um sistema altamente modular é necessário a utilização de um modo de comunicação que permita que módulos distintos se comuniquem entre si sem a necessidade de conhecimento um do outro. Cocos2d-x possui um módulo que permite

uma comunicação por via de troca de mensagens, este que será utilizado neste projeto para a maior parte das comunicações entre os módulos do simulador.

O seu funcionamento segue o padrão *publish-subscribe*, onde um processo cria um evento e os objetos inscritos aguardam este evento, quando ele ocorre, todos os objetos inscritos são avisados do ocorrido. A Figura 4 ilustra o padrão *publish-subscribe* de comunicação, nela pode-se perceber que dois *Subscribers* estão aguardando eventos, um aguarda o *Event 1* e o outro o *Event 2*, mas nenhum aguarda o *Event 3* e, portanto, nenhum deles é notificado do ocorrido.

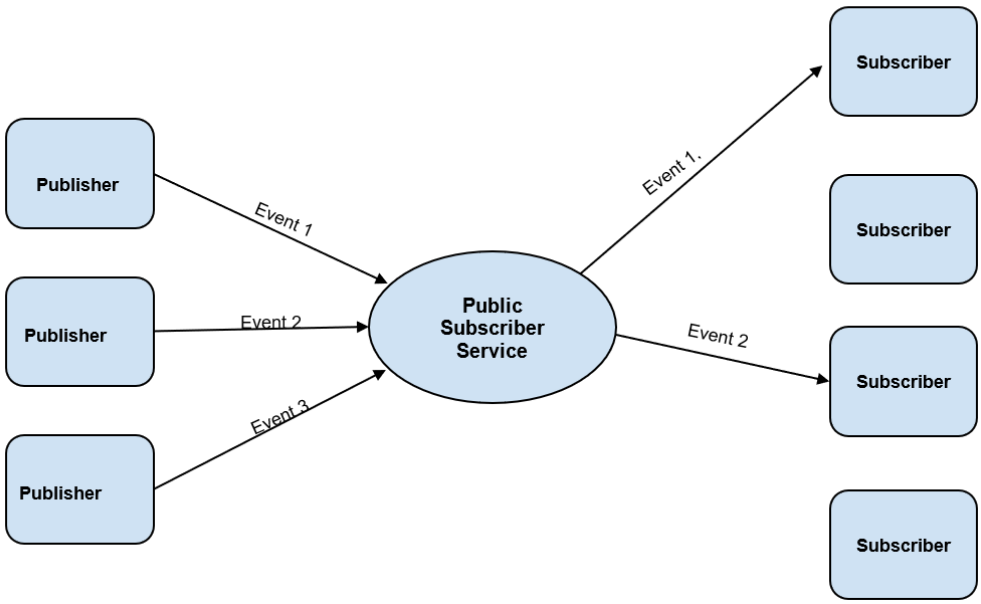

Figura 3 Exemplo de comunicação publish-subscribe.

No Cocos2d-x os objetos que aguardam que um evento ocorra são os *event listeners*, e os que criam eventos os *event dispatchers*. Os *event listeners* definem o

identificador do evento que aguardam e especificam a função que deve ser executada quando tal evento ocorrer. Já os *event dispatchers* criam os eventos definindo os dados relevantes a ele junto com o seu identificador e enviam estes dados para todos os que aguardam o evento.

Este modo de comunicação é útil pois permite um baixo acoplamento de classes, facilitando a adição de funcionalidades ao aplicativo desenvolvido assim como a remoção de certas funcionalidades, como os sensores dos robôs por exemplo.

3.2 Considerações finais

Neste capitulo foi possível ver algumas da funcionalidade que o framework escolhido dispõe, em principal o sistema de mensagem permitirá que os requisitos especificados no início do trabalho com relação ao mantimento de um acoplamento baixo para fácil modificação e adição de funcionalidade no futuro seja cumprida. Além disso os componentes de *Sprite*, *Scenes* e *Layers* permitirão a criação da interface gráfica que foi criada no simulador. Por fim, ambos os Motores físicos permitem a simulação de ambiente nos quais os robôs vão compartilhar e a escolha entre ambos garante uma maior flexibilidade caso um não satisfaça algum requisito no futuro.

Entre os principais motivos para a utilização deste framework podemos citar as funcionalidades descritas acima que foram necessárias para os desenvolvimentos do simulador, o fato dele possuir código aberto, já que o objetivo deste trabalho é desenvolvimento de um sistema para a comunidade acadêmica que possa ser

modificado no futuro. Por fim o conhecimento prévio da ferramenta pelo autor do trabalho também foi um dos pontos que levou ao seu uso.

Capítulo 4

4. Outros simuladores

Neste capítulo serão discutidos outros simuladores de robôs móveis, brevemente introduzindo os considerados mais relevantes, explicitando as suas funcionalidades e por fim será feita uma comparação entre eles e o simulador desenvolvido neste projeto.

4.1 Webots

Desenvolvido pela empresa Cyberbotics o simulador 3D Webots foi inicialmente concebido como um projeto de código aberto chamado “Khepera Simulator” e em principio apenas podia simular robôs da plataforma Khepera. O nome do simulador foi modificado em 1998 para Webots quando a quantidade de robôs que ele era capaz de simular foi expandida para 15 diferentes plataformas.

Ele utiliza o motor físico ODE (Open Dynamics Engine) e possui uma renderização realística de ambos os robôs e os ambientes. Ele permita a utilização de múltiplos robôs no mesmo ambiente e a modificação de suas plataformas, permitindo adicionar ou modificar sensores, simular diferentes comportamentos e ate mesmo criar o *design* do seu próprio veículo, permitido a criação de uma variedade de tipos de robôs.

4.2 Player/Stage

O projeto Player consistia em um sistema formado por três componentes, uma rede de servidores hardware (Player), um simulador 2D para múltiplos robôs, com sensores em um ambiente bit-mapped (Stage) e um simulador 3D para simples ambientes abertos (Gazebo). Eventualmente o Gazebo se tornou independente e atualmente não faz mais parte do sistema.

Stage é um simulador 2D voltado a espaços fechados. Ele pode ser usado como uma aplicação separada, uma biblioteca de C++, ou um plug-in para o Player. O foco dele é na simulação de uma alta quantidade de robôs, sacrificando precisão, portanto, suas capacidades gráficas são bastante limitadas.

4.3 Gazebo

Gazebo é um simulador 3D feito para uma população menor de robôs, ele tem como objetivo uma maior precisão na simulação dos robôs do que o Stage. Ele pode ser usado tanto para ambientes internos quanto externos. Além disso ele se utiliza de *plug-ins* para expandir suas capacidades, como carregar modelos de robôs customizados ou usar sensores de câmera estéreo. Ele também utiliza ODE na simulação física.

4.4 OpenRAVE

OpenRAVE (Open Robotics and Animation Virtual Envirotment) é um aplicativo de código aberto desenvolvido pela Carnegie Mellon University. Ele é focado no

planejamento e simulação de robôs que tem como objetivo agarrar ou segurar assim como robôs humanoides. Ele é usado em conjunto com *frameworks* como o ROS e Player para realizar essas simulações. O suporte ao OpenRAVE foi um dos objetivos da equipe responsável pelo desenvolvimento do ROS devido as suas capacidades de planejamento e ao código aberto.

Uma das grandes vantagens do OpenRAVE é o seu sistema de *plug-ins*. Basicamente tudo é conectado ao OpenRAVE por meio de *plug-ins*, seja o controlador, o planejador, ou motores de simulação ou até mesmo o *hardware* do robô. *Plug-ins* são carregados dinamicamente. Ele suporta diversas linguagens de script como Python e MATLAB/Octave.

4.5 Comparação entre simuladores

Nesta seção serão comparados os simuladores acima mencionados, junto com o simulador desenvolvido neste projeto, na tabela abaixo contribui em destacar alguma das características dos simuladores discutidos. É possível notar que todos eles são de código aberto, sendo que o WeBots, que até 2018 tinha seu código fechado recentemente mudou sua política. Todos eles foram desenvolvidos em C++ assim como este projeto. O motor de física ODE foi o mais comum dentre os simuladores analisados.

Simulador	Código aberto	Plataformas	2D/3D	Motor de Física	Linguagens
WeBots	Sim	Linux, macOS, Windows	3D	ODE	C++
Stage	Sim	Linux, Solaris, macOS	2D	-	C++
Gazebo	Sim	Linux, macOS, Windows	3D	ODE, Bullet, Simbody, DART	C++
OpenRAVE	Sim	Linux, macOS, Windows	3D	ODE, Bullet	C++, Python
TCC	Sim	Windows, iOS, macOS, Linux, Android	2D	Chipmunk, Box2D	C++

Tabela 1 Tabela de Simuladores

Com exceção do Stage, a maior parte destes simuladores tende a uma alta precisão de simulação, com modelos 3D e poucos robôs em seus ambientes. Este foi um dos diferenciais que este TCC atacou, focando em uma alta quantidade de robôs em um mesmo ambiente e uma menor precisão de simulação, ou seja, neste trabalho não estamos preocupados com especificidades dos robôs e sim na definição das tarefas que eles devem cumprir e nos tempos que devem ser respeitados. No contexto de pesquisa deste trabalho, as especificidades dos robôs em relação ao hardware usado podem ser consideradas uma preocupação secundária já que eles serão usados em ambientes fechados, parcialmente conhecidos e toleram a perda de alguns deadlines. Estamos tratando de sistemas de tempo real não críticos e, portanto, o detalhamento do hardware

dos robôs se mostra uma questão menos relevante do que a definição das tarefas e seus tempos de conclusão.

O objetivo principal foi a criação de um simulador de fácil modificação e adição de novos módulos, com foco em algoritmos de navegação e cumprimento de requisitos temporais.

Capítulo 5

5. Solução Proposta

Neste capítulo será discutido a arquitetura do sistema, seus módulos e os principais algoritmos que definem seu funcionamento. Este simulador foi desenvolvido com o objetivo de fácil modificação e alta coesão para que no futuro a adição de novos módulos seja simples para outros desenvolvedores.

Para começar a descrever a arquitetura do simulador começaremos discutindo o sistema como um todo para uma melhor compreensão de como os módulos interagem entre si e depois discutiremos cada módulo individualmente, especificando a função de cada um no simulador.

5.1 Arquitetura Geral

O simulador proposto neste TCC é divido em 6 módulos, que se comunicam entre si através do sistema de mensagens disponibilizado pelo Cocos2d-x. Os módulos *Simulator, Robot*, GUI, *Sensor* e *Controller* foram desenvolvidos e implementados neste trabalho, enquanto o modulo Cocos2d-x apenas foi utilizado. Estes módulos são iniciados pelo módulo principal, chamado *Simulator* que constitui a cena principal do aplicativo.

O módulo *Simulator* interage com o GUI de forma a responder eventos iniciados pelo usuário, em contrapartida o *Simulator* responde a estes eventos, modificando os dados que definem os elementos visuais da simulação.

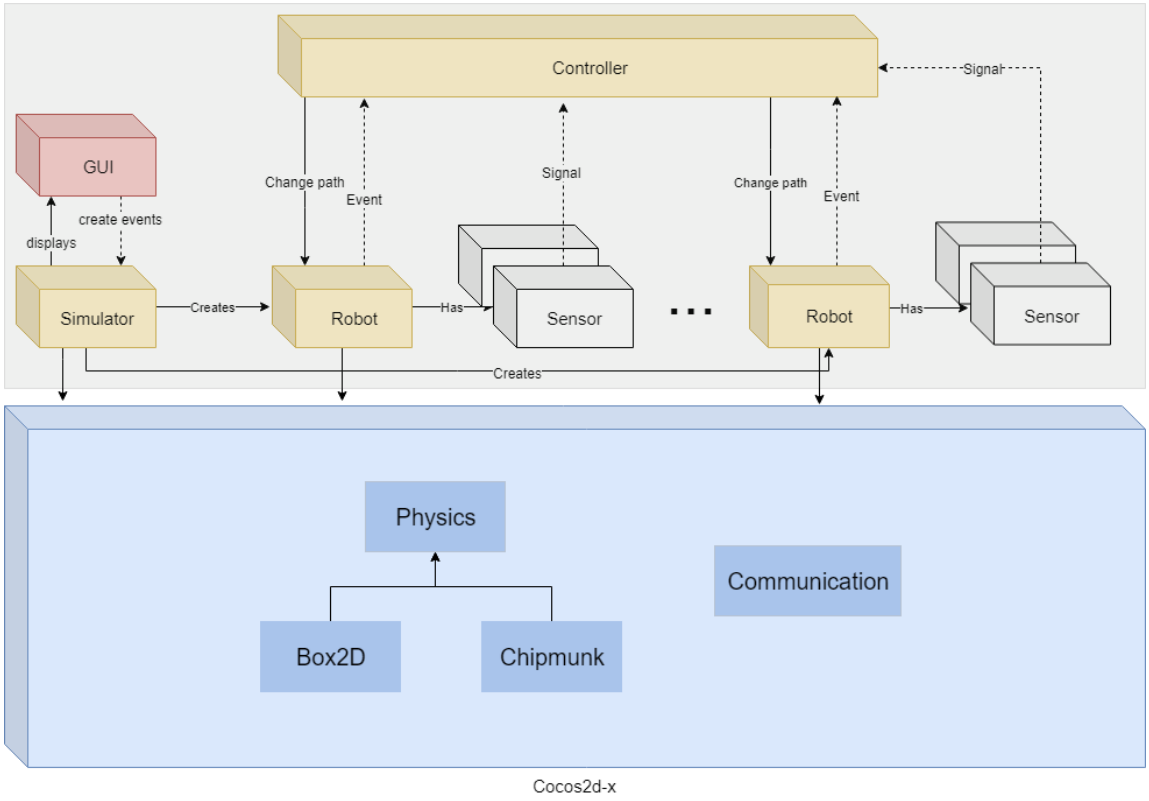

Figura 4 Arquitetura do simulador.

A GUI permite que o usuário defina e edite o mapa no qual os robôs irão navegar, a localização inicial dos robôs, os pacotes que os robôs devem buscar e o ponto de entrega no qual esses pacotes devem ser entregues. Além disso ela possui botões que permitem acelerar ou desacelerar a simulação, modificar o *Zoom,* entre outras funcionalidades.

Quando o usuário inicia a simulação, o *Simulator* cria os robôs, representado pelo módulo *Robot*, que de acordo com o algoritmo definido no módulo *Controller* e em conjunto com os módulos *Sensor,* irá definir os caminhos pelos quais os robôs seguirão até o cumprimento de suas tarefas.

A figura Figura 4 a arquitetura e interação dos módulos descrito, com a adição do *framework* como base para seu funcionamento. Dentro do módulo Cocos2d-x são destacados seus componentes mais relevantes para este projeto. O módulo de *Physics* pode ser representado tanto pelo motor Box2D como pelo Chipmunk como estabelecido no capítulo de tecnologias. E o módulo de comunicação que se torna essencial para a comunicação entre diferentes módulos.

Já os módulos acima do framework, destacados em cinza, foram implementados neste trabalho, entre eles temos o *Robot* e o *Simulador*, ambos herdam classes provenientes do framework, *sprite* e *scene* respectivamente. O *Robot* mantém uma comunicação com o *Controller* por meio de eventos, que irão definir suas próximas ações. Além disso os sensores também podem mandar sinais que possivelmente alteram o caminho que o robô tomara.

5.2 Módulo *Controller*

Este módulo é responsável pela definição do caminho que o robô tomará, usando o algoritmo A* ele define qual o melhor caminho para que o robô chegue aos seus

destinos em tempo mínimo, considerando obstáculos estáticos já definidos no mapa e os evitando.

O algoritmo de busca de caminho A* é comumente utilizado na busca de melhor caminho entre dois nodos de um grafo, nele é utilizada uma heurística para determinar o custo necessário para chegar ao ponto de destino. O funcionamento deste algoritmo se dá da seguinte forma: começando pelo nodo inicial, neste caso a posição atual do robô, é definido como nodo atual, adquire-se uma lista de todos os seus vizinhos e aplica uma função heurística. Esta função retorna um valor que determina a distância do vizinho para o nodo final, o vizinho que tiver a menor distância se torna o nodo atual, e o procedimento se repete até que o nodo atual seja o nodo final. Cada um dos nodos que se tornaram o nodo atual formam o melhor caminho entre o nodo inicial e o nodo final.

A sua função de definir o caminho de um robô é chamada em três ocasiões, no início da simulação, quando um sensor detecta uma colisão e quando uma tarefa é concluída. Considera-se que uma tarefa é concluída quando um robô entrega um objeto a um ponto de entrega definido pelo usuário.

5.3 Módulo *Robot*

O módulo *Robot* realiza os movimentos definidos pelo módulo *Controller* e assim que cada movimenta é finalizado ele modifica o seu estado e cria até dois eventos: um deles para dizer que o robô finalizou um movimento e outro para informar que tipo de

movimento o robô finalizou, sendo o primeiro para o controlador e o segundo para o simulador.

No primeiro caso, um evento é criado e recebido pelo módulo *Controller*, que a partir dele vai determinar uma nova tarefa que o robô deve realizar e o seu novo caminho a seguir.

No segundo caso, um evento, que dependendo do tipo de movimento que o robô finalizou, define sua localização, sendo as seguintes possibilidades: um objeto que deve coletar, um ponto de entrega ou um ponto final.

Se um robô chegou a um ponto final, isso significa que ele finalizou todas as suas tarefas e que agora ele deve aguardar que os outros robôs também finalizem as deles, isso é, caso haja mais de um robô na simulação.

Todo robô é subclasse de um *Sprite* e possui um componente *PhysicsBody* que representa o seu corpo físico. Este corpo possui diversas propriedades incluindo a sua aceleração. Além disso um robô também tem um conjunto de sensores que o ajudam a se deslocar pelo ambiente em que se encontra.

Por fim um robô possui um caminho, uma propriedade chamada *path*, que é definido pelo *Controller* e só é modificado quando o robô finaliza uma tarefa ou quando um sensor envia um sinal ao *Controller* dizendo que o caminho deve ser alterado para evitar uma colisão.

5.4 Módulo *Simulator*

O módulo *Simulator* define os eventos de resposta para o módulo GUI. Assim cada botão que é ativado possui uma resposta que será efetivada pelo Simulador. Quando o usuário inicia a simulação pressionando o botão *Start*, o *Simulator* cria todos os robôs especificados pelo usuário, especificando suas localizações e criando os seus *PhysicsBody*.

Controle temporal é definido aqui para garantir que as tarefas obedecem aos *deadlines* definidos pelo usuário, avisando ao usuário, por meio da GUI, caso um *deadline* seja perdido. Desse modo, quando a simulação é iniciada, este módulo possui um relógio que é ativado, sendo apenas finalizado quando todos os robôs realizaram todas as suas tarefas e desse modo, definindo o tempo total da simulação. Além disso, quando cada robô termina todas as suas tarefas e retorna para o ponto inicial, o simulador registra o momento em que o robô finalizou, desse modo marcando quanto tempo cada robô levou. Esse controle temporal se dá pelo uso da classe *Stopwatch*, implementada neste projeto para determina o tempo tomado pelos robôs e simulação.

Caso o usuário necessite, ao fim da simulação, todos esses tempos podem ser exportados para facilitar a análise do cenário simulado.

5.5 Cocos2d-x

Este módulo representa o framework utilizado neste projeto, ele provê as funcionalidades que definem a simulação física dos robôs, a comunicação entre os

módulos e a interface gráfica. Abaixo discutiremos os principais módulos que fazem parte do *framework* e quais são as suas funcionalidades neste projeto.

5.5.1 Módulo *Communication*

Este módulo permite as funcionalidades de intercomunicação entre os módulos do sistema por meio de *event listerner* e *event dispatchers*, uma forma de comunicação que se encaixa no padrão *publish-subscribe* comum em arquiteturas de software. Nele, um procedimento cria um evento, estabelece os dados relevantes a função que vai lidar com esse evento e através do mecanismo *EventDispatcher* que notifica todos os *listeners* que o evento ocorreu e chamar as funções estabelecidas para lidar com esse evento enviando como parâmetro os dados previamente estabelecidos pelo objeto que gerou o evento.

Um exemplo de seu uso no sistema é quando um robô chega ao seu destino, ele cria um evento "*robot_completed_task*" para todos que se inscreveram com essa *String,* que funciona como um identificador do evento. Neste caso o *Controller* é o objeto que aguarda este evento e assim que ele é notificado do seu acontecimento ele pode ou não definir uma nova tarefa ao robô.

Outro exemplo de seu uso é quando um robô chega a um ponto de interesse, como por exemplo a um objeto que deve transportar, um evento "*robot_at_package*" é criado e enviado para qualquer objeto que esteja o aguardando. O *Simulator* recebe este evento e sinaliza que o pacote foi coletado, fazendo com que a GUI o remove do *display*.

5.5.2 Módulo GUI

Este módulo é responsável pelos elementos visuais do simulador, incluindo a Grid na qual o robô, os objetivos e os obstáculos se encontram, além disso ele também inclui os menus e botões necessários para a criação de um ambiente de simulação, e por fim áreas que contém informações como o tempo e estado em que os robôs e a simulação se encontram. Todos os eventos criados aqui são respondidos pelo módulo *Simulator* que decide como cada elemento interativo se comporta.

Para a visualização de cada robô, eles foram definidos como subclasses da classe *Sprite* do *framework* e atribuídas texturas que os representam. Então essas *sprites* são adicionadas ao *Simulator*, subclasse de *Scene* também parte do *framework*. Com isso todos os robôs possuem uma representação visual que facilita o usuário entender seu comportamento.

Também fazem parte deste módulo as seguintes classes:

- *Infobar*, que mostra as informações relevantes quanto a simulação, como por exemplo o tempo decorrido, o estado da simulação, a posição de um robô, etc.
- *Toolbar,* que consiste em um menu que permite o usuário adicionar, modificar ou remover robôs, obstáculos e objetivos.

- *ActionBar,* que disponibiliza as funcionalidades como começar, pausar ou parar a simulação, modificar a velocidade de simulação e ampliar ou reduzir a tela.

5.5.3 Módulo *Physics*

Este módulo utiliza conceitos da parte física do framework para aplicar propriedade físicas aos robôs, como aceleração, elasticidade, fricção e colisão. Ele também permite a criação de um evento quando uma colisão é detectada, o que já determina um dos sensores deste projeto. Para a sua utilização é necessário criar um componente chamado *PhysicsBody* para cada robô da simulação, na sua criação, propriedades como fricção e elasticidade são definidas e então esses corpos físicos são atribuídos as *Sprites*, neste caso os robôs. É necessário lembrar, como visto anteriormente que uma *Sprite* é inicialmente apenas um elemento visual, mas quando um corpo físico é atribuído a ela, quando um *PhyscisBody* é adicionado como componente, elas se tornam suscetíveis a gravidade e a outros corpos físicos no mesmo espaço.

Também é possível a criação de robôs mais complexos, com múltiplas partes móveis serem simuladas pelo moto físico, para isso pode-se utilizar *Joints* que funcionam como ligamentos para se construir um robô feito por múltiplos corpos físicos. Na imagem abaixo é demonstrado um possível robô com múltiplos ligamentos criando um corpo similar à de uma cobra.

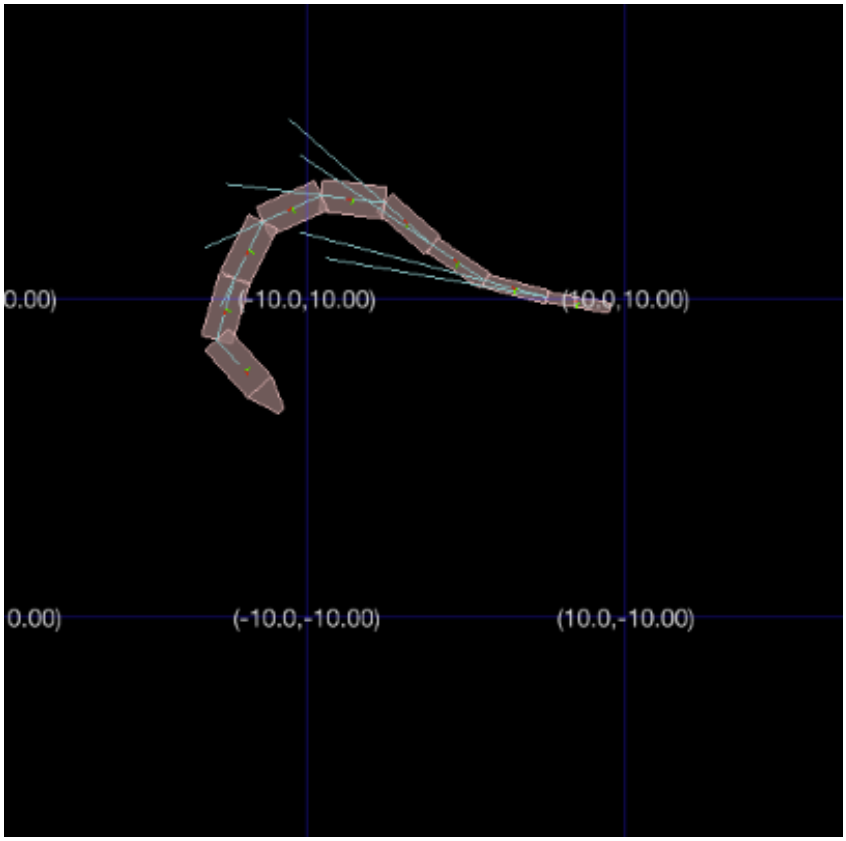

Figura 5 Exemplo de um corpo físico formado por múltiplos corpos interligados.

5.6 Módulo *Sensor*

Foram desenvolvidos três sensores neste projeto, um sensor de contato, um GPS e um sensor de proximidade, abaixo serão descritos seus funcionamentos:

5.6.1 Sensor de Contato

Este sensor detecta quando um robô entra em contato físico com outro robô ou objeto no ambiente, ou seja, quando há uma colisão. A implementação deste sensor foi

feita a partir do módulo *Physics*, atribuindo um corpo físico que consiste no formato e tamanho condizente ao robô e por fim adicionando um evento que será chamado quando um robô colide, este evento é então recebido pelo *Controller* que define como reagir, sendo uma das possibilidades a modificação do caminho do robô para evitar que ele fique preso com um outro robô.

5.6.2 GPS

O sensor GPS define a localização de um robô no mapa (*Grid*), no entanto, para imitar o funcionamento de um GPS real essa posição não deve ser muito precisa, isso ocorre pois um sistema de GPS não é atualizado a não ser que o objeto sendo observado não se mova uma determinada distancia, esta distância pode ser alterada modificando o tamanho das divisórias da *Grid*.

Para simular GPS os robôs são dispostos nesta *Grid* de valor modificável, e possuem uma posição nela que difere da posição no *display*, neste a caso a posição e uma *Sprite*, a medida que se movem essa posição é modificada, robôs que possuem GPS podem reagir a essa posição de acordo com o algoritmo definido no *Controller*.

5.6.3 Sensor de Proximidade

Este sensor envia um sinal a um robô quando um elemento está a uma certa distância *d* de outra entidade. Para a implementação deste sensor foi utilizado o que

comumente é chamado de um corpo fantasma para um robô, ele essencialmente funciona como um corpo físico, no entanto ele não impede o movimento do robô quando há uma colisão, ou seja, quando a distância do sensor detecta outro corpo. Ao invés disso ele apenas envia um sinal avisando o robô que há um outro robô ou objeto dentro de sua área, isso permite que o robô já se prepare e evite a colisão, reduzindo sua velocidade e/ou modificando o seu caminho.

A sinalização do sensor é dada como um *event*, no qual o *Controller* decide como lidar quando o sensor é ativado.

Capítulo 6

6. Simulações e Resultados Obtidos

Neste capítulo será apresentado um cenário, emulando um possível armazém de entregas de pedidos online, para exemplificar o uso do simulador desenvolvido nesta tese. Inicialmente será delineado o problema, explicando duas possíveis situações as quais desejamos testar. Em seguida será aplicado os cenários especificados no simulador, e com a obtenção dos dados que consistem do tempo de cada robô, o tempo total de cada simulação e o tempo necessário para cumprir os pedidos dos usuários, em ambos os cenários. Por fim, será feita uma comparação entre os cenários a fim de verificar qual deles é mais eficiente no uso de tempo para o cumprimento das tarefas atribuídas.

6.1 Cenário

Neste cenário, três usuários realizaram compras em um site online, e agora seus produtos (um pedido), atualmente localizados em corredores de um armazém devem ser transportados, por alguns robôs de carga, a um ponto de entrega. Neste trabalho, um ponto de entrega se refere à um espaço físico onde serão depositados estes pedidos os quais serão, posteriormente, carregados para um meio de transporte (caminhões, containers, aviões, drones e outros meios de transporte).

O transporte do pedido do ponto de entrega até a endereço do cliente não faz parte do escopo de simulação deste trabalho. Esta etapa poderá ser realizada por outros times de robôs ou por pessoas.

A tabela abaixo mostra os usuários e os produtos que compraram:

Usuários	Produtos comprados
Usuário 1	A, B, C
Usuário 2	A, X, Y
Usuário 3	A, Y, Z

Tabela 2 Tabela de produtos comprados

Os produtos estão distribuídos em um galpão, que possui prateleiras, formando corredores que limitam o número de robôs que podem passar por eles, a localização deles pode ser observada na Figura 6. Nela podemos ver a disposição dos produtos A, B, C, X, Y e Z nas prateleiras. A área *Start*, onde os robôs (R1, R2 e R3) iniciam na simulação para transportar os objetos comprados. A área *Delivery* onde os robôs podem descarregar os objetos para serem enviados aos compradores. Por fim temos a *Drop Area*, onde os robôs podem descarregar objetos para que outros robôs os levem até a Delivery.

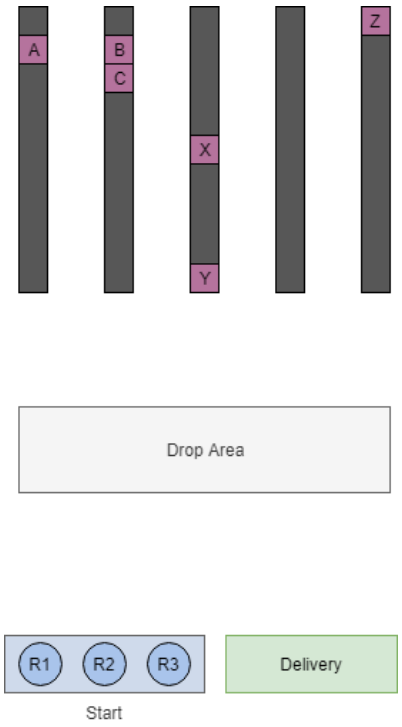

Figura 6 Cenário de simulação.

Neste cenário, três simulações serão realizadas de forma a determinar qual é o melhor plano de ação para que todos os produtos sejam depositados dentro da área Delivery sem a perda de deadline e com tempo mínimo. Cada robô poderá carregar apenas um objeto por vez. A seguir discutiremos as três abordagens que serão testadas e qual o comportamento de cada robô em cada uma delas.

Além disso todos os robôs possuem o mesmo tamanho e percorrem o mapa na mesma velocidade. Dentre os três sensores desenvolvidos neste trabalho, foi utilizado o sensor de proximidade de modo a evitar colisões antes que elas ocorram.

Foi estipulado um *deadline* de 23 segundos para este cenário, este valor foi estipulado através de três testes com o simulador, utilizando apenas um robô, adquirindo os tempos do pior caso possível (30.25 segundos), onde todos os pacotes estão no fim dos corredores, um caso médio, onde eles estão distribuídos em pontos médios das estantes (20.43) e por fim um melhor caso, onde eles estão na ponta mais próxima do robô (17.33).

6.1.1 Simulação 1

Na primeira simulação cada robô será designado a um usuário, portanto R1 buscará os produtos do usuário 1. R2 os produtos do usuário 2 e R3 os produtos do usuário 3. E então eles transportaram os produtos até a área de *Delivery*. A simulação é finalizada quando todos os produtos estejam localizados na área *Delivery*.

6.1.2 Simulação 2

Na segunda simulação, os robôs R1 e R2 serão responsáveis por transportar os produtos apenas até a *Drop Area,* sendo que R3 será permitido transportar apenas objetos da *Drop Area* até a área de *Delivery*. Assim como a simulação anterior, considera-se que a simulação seja finalizada quando todos os produtos estiverem na área *Delivery*.

6.1.3 Simulação 3

Na terceira simulação será permitido que qualquer robô transporte qualquer objeto. O comportamento dos robôs será a de buscar o objeto mais próximo de si, caso dois robôs estejam no mesmo distância de um objeto, a preferência é de R1, seguido por R2, e por fim R3. Cada robô deverá carregar os objetos até a área Delivery, a simulação termina quando todos os objetos forem entregues.

6.2 Resultados

A seguir serão demonstrados os dados temporais obtidos nas simulações executadas, juntamente com uma análise desses resultados. Na Figura 7 pode-se verificar o tempo total de cada simulação, com a simulação 1 tendo uma duração de 23,54 segundos, 22,24 segundos para a simulação 2 terminar e com o menor tempo a simulação 3 foi finalizada em 21,27 segundos. Com isso podemos concluir que o algoritmo de comportamento aplicado na simulação 3 é o mais eficiente quando se considera apenas o tempo total da simulação. O tempo médio das simulações foi calculado como 22.32 segundos.

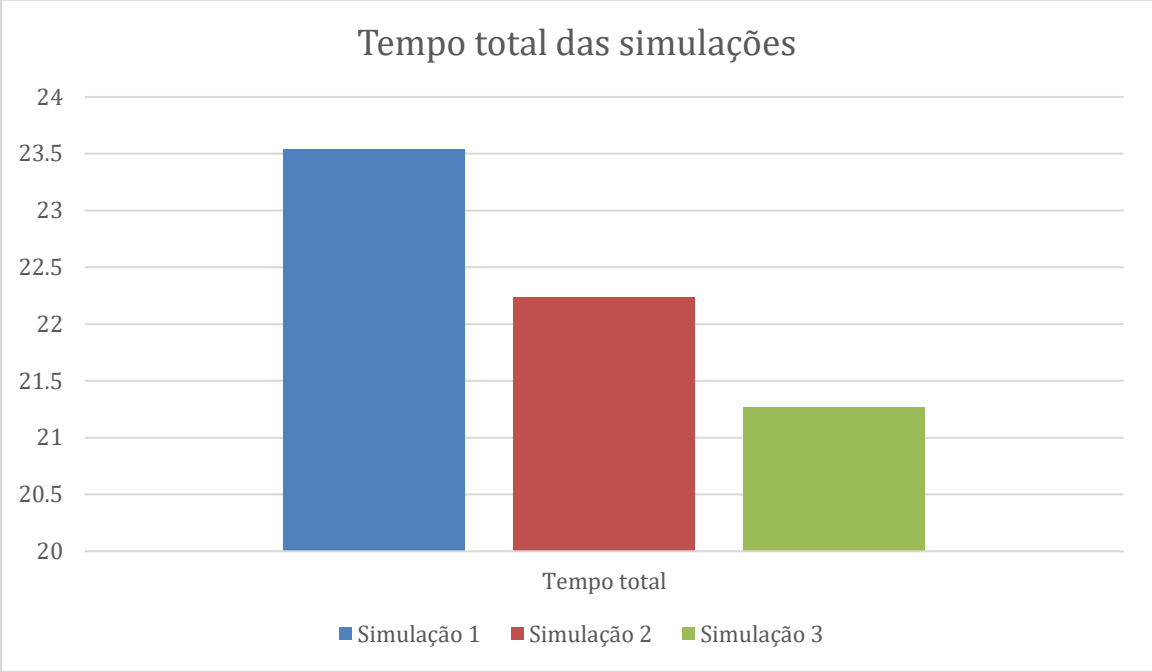

Figura 7 Tempo total das simulações

Além do tempo total da simulação, também foi coletado o tempo individual de cada robô, com estes dados podemos verificar se algum dos robôs perdeu um dos seus deadlines e como cada um afetou o tempo total da simulação.

Na Figura 8 é possível ver o gráfico que demonstra os tempos de cada robô em cada simulação. Para o robô 1, os tempos de execução foram de 23.54 segundos na primeira simulação, que foi além do deadline estipulado, 20.43 segundos na segunda simulação e por fim 20.14 segundos na terceira simulação. É notável que a sua performance na simulação 1 é determinante para o tempo mais elevado da mesma, isso se decorreu da seguinte forma devido aos desvios necessários para evitar colisões, assim como as disposições mais distantes dos produtos do usuário 1, esses fatores são

menos relevantes nas simulações 2 e 3 pois o robô não está necessariamente vinculado com os produtos do usuário 1, e desse modo o tempo gasto foi melhor distribuído entre os robôs.

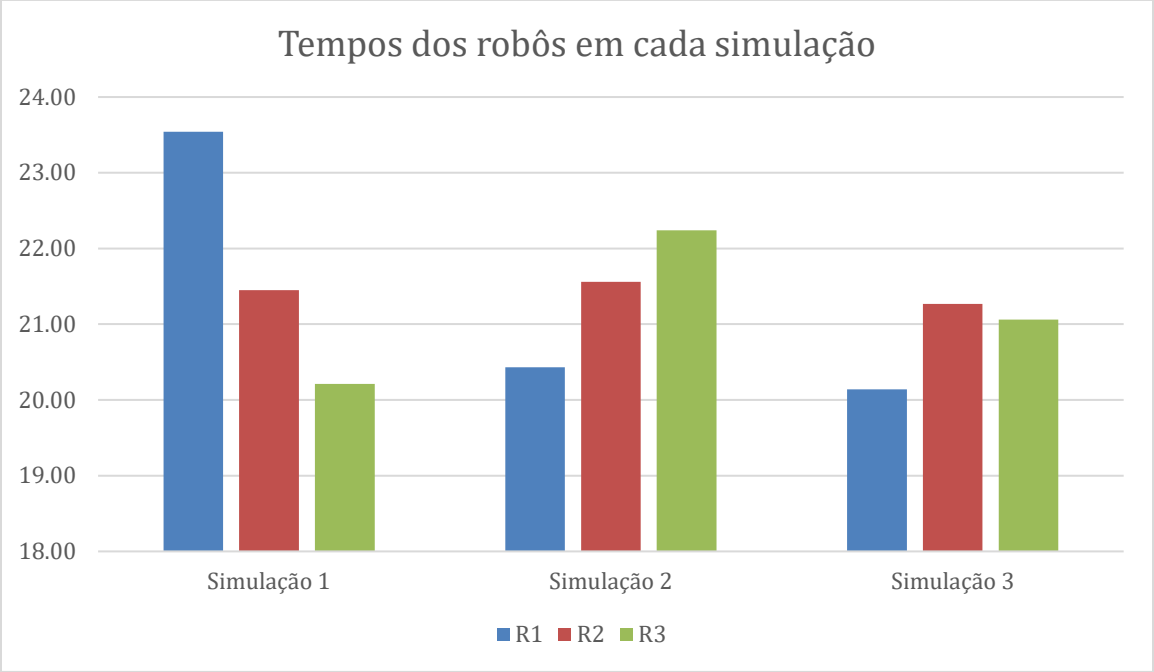

Figura 8 Tempos dos robôs em cada simulação

Para o robô 2 temos os seguintes valores: 21.45s, 21.56s e 21.27s. Na simulação 1 os produtos designados ao robô 2 são bem distribuídos em termos de sua disposição geográfica, portanto, seus valores se mantem nas simulações seguintes já que a sua tarefa não muda, diferente do robô 3.

O robô 3, assim como o robô 1 tem valores com uma maior discrepância. Na simulação 1 ele possui o menor valor dentre as simulações, 20.21 segundos, devido a proximidade do produto Y, juntamente com o fato do produto Z estar isolado e, portanto,

não havendo risco de colisão com outros robôs no corredor que leva até ele. Já na simulação 2 o seu valor é o mais alto pois ele deve aguardar os outros robôs entregarem os pacotes para a *Drop Area.* Por fim, na simulação 3, assim como os outros robôs, os valores acabaram sendo bem distribuídos e ele conclui suas tarefas em 21.06 segundos.

Na tabela a abaixo pode-se ver os valores individuais de cada robô para finalizar as suas respectivas tarefas, junto com os desvios padrão de cada simulação executada.

Duração	Simulação 1	Simulação 2	Simulação 3
Robô 1	23.54	20.43	20.14
Robô 2	21.45	21.56	21.27
Robô 3	20.21	22.24	21.06
Desvio Padrão	1.68298	0.91428	0.60103
Total	23.54	22.24	21.27
Média	21.73333	21.41	20.82333

Tabela 3 Tabela de dados obtidos das simulações

Com os resultados obtidos aqui, pode-se notar que os valores obtidos na simulação 1 são bastante discrepantes, possuindo um alto desvio padrão além do maior tempo de simulação total, portanto, pode-se considerar que o método de solução definido na simulação 1 não seja a mais eficiente para o problema apresentado.

No entanto, tanto a simulação 2 e a simulação 3 demonstraram tempos totais menores em comparação a simulação 1 e com uma menor divergência entre os valores individuais dos robôs. Dentre ambas as simulações, a simulação 3 foi a que obteve os melhores resultados, com o menor tempo total de execução e o menor desvio padrão

entre os tempos dos robôs e, portanto, pode-se determinar que foi a melhor solução para o cenário apresentado dentre as simulações executadas.

Estes resultados confirmam que o algoritmo utilizado na simulação 3 é o mais eficiente na realização da tarefa especificada neste cenário. No entanto algumas coisas devem ser consideradas, existe a possibilidade de um outro algoritmo de busca, que não seja o A* possa desempenhar melhor e modificar os resultados expostos aqui. Uma diferente disposição dos produtos também poderia alterar os resultados vistos nesta análise.

Capítulo 7

7. Conclusão

O propósito deste trabalho de conclusão de curso foi o desenvolvimento de um Simulador para múltiplos robôs de transporte utilizando o framework Cocos2d-x que foi desenvolvido com sucesso e demonstrado neste documento. Foram utilizadas funções do framework para estabelecer comunicação entre diferentes módulos a fim de manter um baixo acoplamento e alta coesão com o objetivo de facilitar a adição de novos módulos no sistema. Foi também adicionada a funcionalidade que permite que o usuário defina o comportamento do robô. Foram desenvolvidos múltiplos sensores que podem ser habilitados ou desabilitados a medida da necessidade do usuário. Por fim todo o código deste projeto está disponível de forma aberta na plataforma Github[1] caso seja necessária modificação ou adição de novas funcionalidades.

Como prova do funcionamento do simulador foi feita a simulação de um cenário de um armazém de produtos, testando três diferentes comportamentos e definido qual deles foi o mais eficiente em completar a tarefa definida.

[1] SimZ: https://github.com/Penumbroso/Multi-Robot-Simulator

7.1 Trabalhos futuros

Este projeto implementou um simulador para robôs terrestres de transporte. Ele pode ser ampliado e novas funções podem ser adicionadas a fim de aumentar a sua capacidade de uso para que os usuários não se sintam limitados. Dentre as possíveis adições ao projeto, as seguintes foram consideradas, mas não implementadas devido a limitação de tempo:

Simulação em um espaço 3D seria uma das possibilidades a serem implantadas em um trabalho futuro, isso pode alterar consideravelmente as capacidades de ambientes e robôs que podem ser simuladas, permitindo ao usuário adicionar modelos e ambientes 3D, outros tipos de robôs como aéreos também poderiam ser considerados.

Neste projeto alguns sensores foram desenvolvidos para facilitar a utilização do simulador e para testar seu funcionamento. Nesta versão do trabalho, o usuário terá que alterar o código fonte do simulador caso queira adicionar mais sensores. A definição de um módulo de sensores (ou um sistema de plug-ins), possibilitando ao usuário instanciar diferentes tipos de sensores irá facilitar o desenvolvimento de novos cenários.

Outra proposta seria um modulo energético que define a carga ou bateria de um robô, e desse modo, o usuário poderia medir não só o tempo que os robôs utilizam, mas também o consumo energético de um cada robô. Gerando a necessidade de os robôs retornarem a um ponto especifico no mapa para recarregar.

REFERÊNCIAS BIBLIOGRÁFICAS

JIA, M. J. M.; ZHOU, G. Z. G.; CHEN, Z. C. Z. An efficient strategy integrating grid and topological information for robot exploration. IEEE Conference on Robotics, Automation and Mechatronics, 2004**.**, v. 2, p. 1–3, 2004.

NILSON, P.; HAESAERT, S.; THAKKER, R.; Otsu, K.; VASILE, C.; AGHA-MOHAMMADI, A.; MURRAY, R. M.; AMES, A. D. Toward Specification-Guided Active Mars Exploration for Cooperative Robot Teams. Disponível em: <http://roboticsproceedings.org/rss14/p47.pdf>.

MARKS, M. ROBOTS IN SPACE: SHARING OUR WORLD WITH AUTONOMOUS DELIVERY VEHICLES. We Robot, 2019. Disponível em: <https://robots.law.miami.edu/2019/wp-content/uploads/2019/03/Mason-Marks-Robots-in-Space-WeRobot-2019-3-14.pdf>.

BREGA, R.; TOMATIS, N.; ARRAS, K. O. The Need for Autonomy and Real-Time in Mobile Robotics: A Case Study of XO/2 and Pygmalion. International Conference on Intelligent Robots and Systems. Disponível em: <http://www2.informatik.uni-freiburg.de/~arras/papers/bregaIROS00.pdf>.

YAN, Z.; FRABRESSE, L.; LAVAL, J.; BOURAQADI, N. Building a ROS-Based Testbed for Realistic Multi-Robot Simulation: Taking the Exploration as an Example. Setembro, 2017.

PINCIROLI, C.; TRIANNI, V.; O'GRADY, R.; PINI, G.; BRUTSCHY, A.; BRAMBLIA, M.; MATTHEWS, N.; FERRANTE, E.; DI CARO, G.; DUCATELLE, F.; STIRLING, T.; GUTIÉRREZ, A.; GAMBARDELLA, L. M.; DORIGO, M. ARGoS: A Modular, Multi-Engine Simulator for Heterogeneous Swarm Robotics. IEEE/RSJ International Conference on Intelligent Robots and Systems September 25-30, p. 5027-5034 ,2011. San Francisco, CA, USA.

MIURA, J.; UOZUMI, H.; SHIRAI, Y. Mobile robot motion planning considering the motion uncertainty of moving obstacles. IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics, v. 4, p. 692–697, 1999. ISSN 1062-922X.

TSUBOUCHI, T.; HIROSE, a.; ARIMOTO, S. A navigation scheme with learning for a mobile robot among multiple moving obstacles. Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '93), v. 3, n. C, p. 2234–2240, 1993.

GUERRERO, J.; OLIVER, G. Multi-robot coalition formation in real-time scenarios. Robotics and Autonomous Systems, Elsevier B.V., v. 60, n. 10, p. 1295–1307, 2012. ISSN 09218890. Disponível em: <http://dx.doi.org/10.1016/j.robot.2012.06.004>.

CARPIN, S.; PAVONE, M.; SADLER, B. M. Rapid Multirobot Deployment with Time Constraints. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS. Chicago, IL, USA: [s.n.], 2014. p. 1147–1154. ISBN 9781479969340.

ANEXO A – Código Fonte

A.1 HEADERS

Listagem A.1: Actionbar.h

```
#include "cocos2d.h"

USING_NS_CC;

class Actionbar : public cocos2d::Layer
{
public:
        virtual bool init();

        std::vector<MenuItemImage*> buttons;

        MenuItemImage* runItem;
        MenuItemImage* exportItem;
        MenuItemImage* resetItem;
        MenuItemImage* speedUpItem;
        MenuItemImage* slowDownItem;
        MenuItemImage* moveItem;
        MenuItemImage* zoomInItem;
        MenuItemImage* zoomOutItem;

        CREATE_FUNC(Actionbar);
private:
        void createButtons();
};
```

Listagem A.2: AppDelegate.h

```
#ifndef  _APP_DELEGATE_H_
#define  _APP_DELEGATE_H_

#include "cocos2d.h"

/**
@brief    The cocos2d Application.

Private inheritance here hides part of interface from Director.
*/
class  AppDelegate : private cocos2d::Application
{
public:
    AppDelegate();
```

```
    virtual ~AppDelegate();

    virtual void initGLContextAttrs();

    /**
    @brief    Implement Director and Scene init code here.
    @return true    Initialize success, app continue.
    @return false   Initialize failed, app terminate.
    */
    virtual bool applicationDidFinishLaunching();

    /**
    @brief  Called when the application moves to the background
    @param  the pointer of the application
    */
    virtual void applicationDidEnterBackground();

    /**
    @brief  Called when the application reenters the foreground
    @param  the pointer of the application
    */
    virtual void applicationWillEnterForeground();
};

#endif // _APP_DELEGATE_H_
```

Listagem A.3: AStar.hpp

```
#ifndef __ASTAR_HPP__
#define __ASTAR_HPP__

#include <vector>
#include <functional>
#include <set>
#include "cocos2d.h"

namespace AStar
{
        typedef cocos2d::Point Vec2i;

    using uint = unsigned int;
    using HeuristicFunction = std::function<uint(Vec2i, Vec2i)>;
    using CoordinateList = std::vector<Vec2i>;
```

```
struct Node
{
    uint G, H;
    Vec2i coordinates;
    Node *parent;

    Node(Vec2i coord_, Node *parent_ = nullptr);
    uint getScore();
};

using NodeSet = std::set<Node*>;

class Generator
{
    bool detectCollision(Vec2i coordinates_);
    Node* findNodeOnList(NodeSet& nodes_, Vec2i coordinates_);
    void releaseNodes(NodeSet& nodes_);

public:
    Generator();
    void setWorldSize(Vec2i worldSize_);
    void setDiagonalMovement(bool enable_);
    void setHeuristic(HeuristicFunction heuristic_);
    CoordinateList findPath(Vec2i source_, Vec2i target_);
    void addCollision(Vec2i coordinates_);
                void addCollisions(std::vector<Vec2i> coordinates_);
    void removeCollision(Vec2i coordinates_);
    void clearCollisions();

private:
    HeuristicFunction heuristic;
    CoordinateList direction, walls;
    Vec2i worldSize;
    uint directions;
};

class Heuristic
{
    static Vec2i getDelta(Vec2i source_, Vec2i target_);

public:
    static uint manhattan(Vec2i source_, Vec2i target_);
    static uint euclidean(Vec2i source_, Vec2i target_);
    static uint octagonal(Vec2i source_, Vec2i target_);
};
```

```
}

#endif // __ASTAR_HPP__
```

Listagem A.4: Grid.h

```
#ifndef __GRID_H__
#define __GRID_H__

#include "cocos2d.h"
#include "Square.h"
#include "Util.h"

USING_NS_CC;

class Grid : public cocos2d::Layer
{
public:
        virtual bool init();

        std::map<Point, Square*> squares;

        int square_size;

        float number_of_columns;
        float number_of_lines;

        Point getPositionOf(Point point);
        Point getGridPositionOf(Point screen_position);

        void setState(Square::State state, Point point);

        vector<Point> starts;
        vector<Point> delivery_points;
        vector<Point> packages;
        vector<Point> blockades;

        vector<Point> available_packages;
        vector<Point> static_collidables;

        void toggleDragAndDrop();

        CREATE_FUNC(Grid);

protected:
```

```
        void drawLines();
        void addSymbol(const std::string &filename, Point point);
        void removeSymbol(Point point);

private:
        Menu* menu;
        std::map<Point, Sprite*> symbols;

        void createSquares();
        bool onTouchBegan(cocos2d::Touch * touch, cocos2d::Event * event)
override;
        void onTouchMoved(cocos2d::Touch * touch, cocos2d::Event * event)
override;

        Point initial_touch_location;
};

#endif
```

Listagem A.5: Infobar.h

```
#include "cocos2d.h"

USING_NS_CC;

class Infobar : public cocos2d::Layer
{
public:
        virtual bool init();

        std::string* time;

        void updateSpeed(float speed);
        void updateZoom(float zoom);

        CREATE_FUNC(Infobar);

private:
        cocos2d::Label* speed_factor_label;
        cocos2d::Label* zoom_label;

        cocos2d::Label* time_label;

        void updateClock(float dt);
};
```

Listagem A.6: Robot.h

```
#ifndef __ROBOT_H__
#define __ROBOT_H__

#include "cocos2d.h"
#include "Sensor.h"
#include <vector>

USING_NS_CC;

using std::vector;

class Robot : public cocos2d::Sprite
{
public:

	virtual bool init();

	enum State
	{
		FULL,
		EMPTY
	};

	State state = EMPTY;

	vector<Sensor> sensors;

	vector<Point> grid_path;
	vector<Point> screen_path;
	vector<Point> complete_grid_path;

	Point grid_position;

	Point grid_destination;
	Point screen_destination;
	Point screen_package;
	Point grid_start;
	Point screen_start;
	Point screen_delivery_point;

	void move();
	void stop();
```

```
        bool isParked();
        bool isAtDeliverty();
        bool isFull();
        bool isAtPackage();
        bool isInThe(vector<Point> path);
        void addSensor(Sensor sensor);

        CREATE_FUNC(Robot);

private:
        Sequence* movement;

        void finishedMovement();
        void updateState();
        void updateGridPosition();
};

#endif
```

Listagem A.7: RobotController.h

```
#include "cocos2d.h"
#include "Robot.h"
#include "AStar.hpp"
#include "Grid.h"


class RobotController : public cocos2d::Node
{
public:
        virtual void onEnter();

        vector<Robot*> robots;
        Grid* grid;

        void definePathOf(Robot * robot);
        void preventCollisionOf(Robot* robot);
        void repath(Robot* r1, Robot* r2);
        Robot* getRobotAt(Point grid_position);

        void robotCompletedMovement(EventCustom* event);

        CREATE_FUNC(RobotController);

private:
```

```
        vector<Point> findShortestPath(Point origin, vector<Point> destinations);
        bool isCollisionImminent(Point next_position);
        AStar::Generator path_generator;
        vector<Point> convertGridPathToScreenPath(vector<Point> path);
        void setupGenerator();
};
```

Listagem A.8: Sensor.h

```
#ifndef __SENSOR_H__
#define __SENSOR_H__

#include "cocos2d.h"

USING_NS_CC;

class Sensor : public cocos2d::Node
{
public:

        virtual bool init();

        void signal();
        void setOwner(cocos2d::Node* owner);

        CREATE_FUNC(Sensor);

private:
        cocos2d::Node * owner;
};

#endif
```

Listagem A.9: Simulator.h

```
#ifndef __SIMULATOR_SCENE_H__
#define __SIMULATOR_SCENE_H__

#include "cocos2d.h"
#include "Grid.h"
#include "Robot.h"
#include "Toolbar.h"
```

```
#include <vector>
#include "RobotController.h"
#include "Infobar.h"
#include <fstream>
#include "Actionbar.h"
#include "Stopwatch.h"

using std::vector;

class Simulator : public cocos2d::Scene
{
public:
    virtual bool init();

    CREATE_FUNC(Simulator);

private:
        vector<Robot*> robots;
        Grid* grid;
        Toolbar* toolbar;
        Infobar* infobar;
        Actionbar* actionbar;
        Stopwatch* stopwatch;
        RobotController * robotController;

        vector<Point> packages_delivered;
        std::map<Robot*, String> robot_times;
        std::map<PhysicsBody*, Robot*> robots_bodies;

        float speed_multiplier;

        void setup();
        void start();
        void stop();
        void proceed();
        void reset();
        void createRobots();
        bool allPackagesWereDelivered();
        bool allRobotsAreParked();
        void setCallbacks();
        void createCustomEvents();

        // Callbacks functions
        void menuRunCallback(cocos2d::Ref * pSender);
        void menuResetCallback(cocos2d::Ref * pSender);
```

```
        void gridSquareCallback(Point coord);
        void menuExportCallback(cocos2d::Ref * pSender);
        void menuChangeSpeedCallback(float multiplier);
        void menuMoveGridCallback(cocos2d::Ref* pSender);
        void menuZoomCallback(float multiplier);
        bool onContactBegin(PhysicsContact& contact);

        // Event functions
        void robotIsAtDelivery(EventCustom* event);
        void robotIsAtPackage(EventCustom* event);
        void robotIsParked(EventCustom* event);
};

#endif // __SIMULATOR_SCENE_H__
```

Listagem A.10: Square.h

```
#include "cocos2d.h"

class Square : public cocos2d::MenuItemImage
{
public:
        void onEnter() override;

        enum State
        {
                EMPTY,
                PACKAGE,
                BEGIN,
                END,
                BLOCKADE
        };

        State state;

        cocos2d::Point grid_position;

        CREATE_FUNC(Square);

};
```

Listagem A.11: Stopwatch.h

```
#include "cocos2d.h"
```

```
#include <string>

class Stopwatch : public cocos2d::Node
{

public:
	virtual bool init();
	void start();
	void stop();

	int getMinutes();
	int getSeconds();
	int getMilliseconds();
	void reset();

	std::string text;
	std::string toString();
	void setSpeedMultiplier(float multiplier);

	CREATE_FUNC(Stopwatch);

private:
	int minutes;
	int seconds;
	int milliseconds;

	float speed_multiplier;


	void count(float dt);
};
```

Listagem A.12: Toolbar.h

```
#include "cocos2d.h"

USING_NS_CC;

class Toolbar : public cocos2d::Layer
{
	public:
		virtual bool init();

		enum Tool {
```

```
                PACKAGE,
                BEGIN,
                END,
                ERASE,
                BLOCKADE,
                CLOCK,
                PATH
        };

        Tool selected = PACKAGE;

        std::vector<MenuItemImage*> buttons;

        MenuItemImage* endItem;
        MenuItemImage* packageItem;
        MenuItemImage* beginItem;
        MenuItemImage* eraseItem;
        MenuItemImage* blockadeItem;
        MenuItemImage* pathItem;
        MenuItemImage* clockItem;

        void setTool(Tool tool, MenuItemImage* btn);

        CREATE_FUNC(Toolbar);
    private:
        DrawNode* selected_bg;

        void setCallbacks();
        void createButtons();
};
```

Listagem A.13: Util.h

```
#include <vector>
#include "cocos2d.h"

using std::vector;

class Util
{
public:
    Util();
    ~Util();
```

```
        template <typename T>
        static bool contains(vector<T>* vector, T element)
        {
                if (vector)
                {
                        auto it = std::find(vector->begin(), vector->end(), element);
                        if (it != vector->end()) return true;
                }
                return false;
        }

        template <typename T>
        static void addIfUnique(vector<T>* vector, T element)
        {
                if (vector)
                {
                        auto it = std::find(vector->begin(), vector->end(), element);
                        if (it == vector->end()) vector->push_back(element);
                }
        }

        template<typename T>
        static void removeIfContains(vector<T>* vector, T element)
        {
                if (vector)
                {
                        auto it = std::find(vector->begin(), vector->end(), element);
                        if (it != vector->end()) vector->erase(it);
                }
        }
};
```

A.2 IMPLEMENTAÇÕES

Listagem A.14: Actionbar.cpp

```
#include "Actionbar.h"

bool Actionbar::init()
{
        auto visibleSize = Director::getInstance()->getVisibleSize();

        this->createButtons();
```

```
	auto menu = Menu::create();
	menu->setPosition(Vec2::ZERO);

	buttons.push_back(runItem);
	buttons.push_back(resetItem);
	buttons.push_back(exportItem);
	buttons.push_back(speedUpItem);
	buttons.push_back(slowDownItem);
	buttons.push_back(moveItem);
	buttons.push_back(zoomInItem);
	buttons.push_back(zoomOutItem);

	int offset = 30;
	for (int i = 0; i < buttons.size(); i++)
	{
		buttons[i]->setPosition(Vec2(15, visibleSize.height - 15 * i * 2 - offset));
		menu->addChild(buttons[i]);
	}

	this->addChild(menu, 1);

	return true;
}

void Actionbar::createButtons()
{
	runItem = MenuItemImage::create(
		"RunBtn.png",
		"RunBtn_pressed.png");

	resetItem = MenuItemImage::create(
		"resetBtn.png",
		"resetBtn_pressed.png");

	exportItem = MenuItemImage::create(
		"Timer.png",
		"Timer_pressed.png");

	speedUpItem = MenuItemImage::create(
		"SpeedUpBtn.png",
		"SpeedUpBtn.png");

	slowDownItem = MenuItemImage::create(
		"SlowDownBtn.png",
```

```
                "SlowDownBtn.png");

        moveItem = MenuItemImage::create(
                "Timer.png",
                "Timer_pressed.png");

        zoomInItem = MenuItemImage::create(
                "Timer.png",
                "Timer_pressed.png");

        zoomOutItem = MenuItemImage::create(
                "Timer.png",
                "Timer_pressed.png");
}
```

Listagem A.15: AppDelegate.cpp

```
#include "AppDelegate.h"
#include "Simulator.h"

// #define USE_AUDIO_ENGINE 1
// #define USE_SIMPLE_AUDIO_ENGINE 1

#if USE_AUDIO_ENGINE && USE_SIMPLE_AUDIO_ENGINE
#error "Don't use AudioEngine and SimpleAudioEngine at the same time. Please just select one in your game!"
#endif

#if USE_AUDIO_ENGINE
#include "audio/include/AudioEngine.h"
using namespace cocos2d::experimental;
#elif USE_SIMPLE_AUDIO_ENGINE
#include "audio/include/SimpleAudioEngine.h"
using namespace CocosDenshion;
#endif

USING_NS_CC;

static cocos2d::Size designResolutionSize = cocos2d::Size(1200, 600);
static cocos2d::Size smallResolutionSize = cocos2d::Size(480, 320);
static cocos2d::Size mediumResolutionSize = cocos2d::Size(1024, 768);
static cocos2d::Size largeResolutionSize = cocos2d::Size(2048, 1536);

AppDelegate::AppDelegate()
{
```

```
}

AppDelegate::~AppDelegate()
{
#if USE_AUDIO_ENGINE
    AudioEngine::end();
#elif USE_SIMPLE_AUDIO_ENGINE
    SimpleAudioEngine::end();
#endif
}

// if you want a different context, modify the value of glContextAttrs
// it will affect all platforms
void AppDelegate::initGLContextAttrs()
{
    // set OpenGL context attributes:
red,green,blue,alpha,depth,stencil,multisamplesCount
    GLContextAttrs glContextAttrs = {8, 8, 8, 8, 24, 8, 0};

    GLView::setGLContextAttrs(glContextAttrs);
}

// if you want to use the package manager to install more packages,
// don't modify or remove this function
static int register_all_packages()
{
    return 0; //flag for packages manager
}

bool AppDelegate::applicationDidFinishLaunching() {
    // initialize director
    auto director = Director::getInstance();
    auto glview = director->getOpenGLView();
    if(!glview) {
#if (CC_TARGET_PLATFORM == CC_PLATFORM_WIN32) ||
(CC_TARGET_PLATFORM == CC_PLATFORM_MAC) || (CC_TARGET_PLATFORM
== CC_PLATFORM_LINUX)
        glview = GLViewImpl::createWithRect("AWS", cocos2d::Rect(0, 0,
designResolutionSize.width, designResolutionSize.height));
#else
        glview = GLViewImpl::create("AWS");
#endif
        director->setOpenGLView(glview);
    }
```

```
    // turn on display FPS
    director->setDisplayStats(false);

    // set FPS. the default value is 1.0/60 if you don't call this
    director->setAnimationInterval(1.0f / 65);

    // Set the design resolution
    glview->setDesignResolutionSize(designResolutionSize.width,
designResolutionSize.height, ResolutionPolicy::NO_BORDER);
    auto frameSize = glview->getFrameSize();
    // if the frame's height is larger than the height of medium size.
    if (frameSize.height > mediumResolutionSize.height)
    {
        director-
>setContentScaleFactor(MIN(largeResolutionSize.height/designResolutionSize.height,
largeResolutionSize.width/designResolutionSize.width));
    }
    // if the frame's height is larger than the height of small size.
    else if (frameSize.height > smallResolutionSize.height)
    {
        director-
>setContentScaleFactor(MIN(mediumResolutionSize.height/designResolutionSize.heig
ht, mediumResolutionSize.width/designResolutionSize.width));
    }
    // if the frame's height is smaller than the height of medium size.
    else
    {
        director-
>setContentScaleFactor(MIN(smallResolutionSize.height/designResolutionSize.height,
smallResolutionSize.width/designResolutionSize.width));
    }

    register_all_packages();

    // create a scene. it's an autorelease object
            auto scene = Simulator::create();

    // run
    director->runWithScene(scene);

    return true;
}

// This function will be called when the app is inactive. Note, when receiving a phone call
it is invoked.
```

```
void AppDelegate::applicationDidEnterBackground() {
    Director::getInstance()->stopAnimation();

#if USE_AUDIO_ENGINE
    AudioEngine::pauseAll();
#elif USE_SIMPLE_AUDIO_ENGINE
    SimpleAudioEngine::getInstance()->pauseBackgroundMusic();
    SimpleAudioEngine::getInstance()->pauseAllEffects();
#endif
}

// this function will be called when the app is active again
void AppDelegate::applicationWillEnterForeground() {
    Director::getInstance()->startAnimation();

#if USE_AUDIO_ENGINE
    AudioEngine::resumeAll();
#elif USE_SIMPLE_AUDIO_ENGINE
    SimpleAudioEngine::getInstance()->resumeBackgroundMusic();
    SimpleAudioEngine::getInstance()->resumeAllEffects();
#endif
}
```

Listagem A.16: AStar.cpp

```
#include "AStar.hpp"
#include <algorithm>

using namespace std::placeholders;

AStar::Node::Node(Vec2i coordinates_, Node *parent_)
{
    parent = parent_;
    coordinates = coordinates_;
    G = H = 0;
}

AStar::uint AStar::Node::getScore()
{
    return G + H;
}

AStar::Generator::Generator()
{
```

```
    setDiagonalMovement(false);
    setHeuristic(&Heuristic::manhattan);
    direction = {
        { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 },
        { -1, -1 }, { 1, 1 }, { -1, 1 }, { 1, -1 }
    };
}

void AStar::Generator::setWorldSize(Vec2i worldSize_)
{
    worldSize = worldSize_;
}

void AStar::Generator::setDiagonalMovement(bool enable_)
{
    directions = (enable_ ? 8 : 4);
}

void AStar::Generator::setHeuristic(HeuristicFunction heuristic_)
{
    heuristic = std::bind(heuristic_, _1, _2);
}

void AStar::Generator::addCollision(Vec2i coordinates_)
{
    walls.push_back(coordinates_);
}

void AStar::Generator::addCollisions(std::vector<Vec2i> coordinates_)
{
            for (auto coordinate_ : coordinates_)
                    this->addCollision(coordinate_);
}

void AStar::Generator::removeCollision(Vec2i coordinates_)
{
    auto it = std::find(walls.begin(), walls.end(), coordinates_);
    if (it != walls.end()) {
        walls.erase(it);
    }
}

void AStar::Generator::clearCollisions()
{
    walls.clear();
```

```
}

AStar::CoordinateList AStar::Generator::findPath(Vec2i source_, Vec2i target_)
{
    Node *current = nullptr;
    NodeSet openSet, closedSet;
    openSet.insert(new Node(source_));

    while (!openSet.empty()) {
        current = *openSet.begin();
        for (auto node : openSet) {
            if (node->getScore() <= current->getScore()) {
                current = node;
            }
        }

        if (current->coordinates == target_) {
            break;
        }

        closedSet.insert(current);
        openSet.erase(std::find(openSet.begin(), openSet.end(), current));

        for (uint i = 0; i < directions; ++i) {
            Vec2i newCoordinates(current->coordinates + direction[i]);
            if (detectCollision(newCoordinates) ||
                findNodeOnList(closedSet, newCoordinates)) {
                continue;
            }

            uint totalCost = current->G + ((i < 4) ? 10 : 14);

            Node *successor = findNodeOnList(openSet, newCoordinates);
            if (successor == nullptr) {
                successor = new Node(newCoordinates, current);
                successor->G = totalCost;
                successor->H = heuristic(successor->coordinates, target_);
                openSet.insert(successor);
            }
            else if (totalCost < successor->G) {
                successor->parent = current;
                successor->G = totalCost;
            }
        }
    }
```

```
    CoordinateList path;
    while (current != nullptr) {
        path.push_back(current->coordinates);
        current = current->parent;
    }

    releaseNodes(openSet);
    releaseNodes(closedSet);

    return path;
}

AStar::Node* AStar::Generator::findNodeOnList(NodeSet& nodes_, Vec2i coordinates_)
{
    for (auto node : nodes_) {
        if (node->coordinates == coordinates_) {
            return node;
        }
    }
    return nullptr;
}

void AStar::Generator::releaseNodes(NodeSet& nodes_)
{
    for (auto it = nodes_.begin(); it != nodes_.end();) {
        delete *it;
        it = nodes_.erase(it);
    }
}

bool AStar::Generator::detectCollision(Vec2i coordinates_)
{
    if (coordinates_.x < 0 || coordinates_.x >= worldSize.x ||
        coordinates_.y < 0 || coordinates_.y >= worldSize.y ||
        std::find(walls.begin(), walls.end(), coordinates_) != walls.end()) {
        return true;
    }
    return false;
}

AStar::Vec2i AStar::Heuristic::getDelta(Vec2i source_, Vec2i target_)
{
    return{ abs(source_.x - target_.x),  abs(source_.y - target_.y) };
}
```

```
AStar::uint AStar::Heuristic::manhattan(Vec2i source_, Vec2i target_)
{
   auto delta = std::move(getDelta(source_, target_));
   return static_cast<uint>(10 * (delta.x + delta.y));
}

AStar::uint AStar::Heuristic::euclidean(Vec2i source_, Vec2i target_)
{
   auto delta = std::move(getDelta(source_, target_));
   return static_cast<uint>(10 * sqrt(pow(delta.x, 2) + pow(delta.y, 2)));
}

AStar::uint AStar::Heuristic::octagonal(Vec2i source_, Vec2i target_)
{
   auto delta = std::move(getDelta(source_, target_));
   return 10 * (delta.x + delta.y) + (-6) * std::min(delta.x, delta.y);
}
```

Listagem A.17: Grid.cpp

```
#include "Grid.h"

USING_NS_CC;

bool Grid::init()
{
	if (!Layer::init())
		return false;

	square_size = 30;

	menu = Menu::create();
	menu->setPosition(Vec2::ZERO);
	addChild(menu);

	const auto visibleSize = Director::getInstance()->getVisibleSize();

	number_of_lines = visibleSize.height / square_size + 1;
	number_of_columns = visibleSize.width / square_size + 1;

	createSquares();
	drawLines();

	auto listener = EventListenerTouchOneByOne::create();
```

```
	listener->onTouchBegan = CC_CALLBACK_2(Grid::onTouchBegan, this);
	listener->onTouchMoved = CC_CALLBACK_2(Grid::onTouchMoved, this);
	_eventDispatcher->addEventListenerWithSceneGraphPriority(listener, this);
	return true;
}

Point Grid::getPositionOf(Point point)
{
	return squares.at(point)->getPosition();
}

Point Grid::getGridPositionOf(Point screen_position)
{
	for (const auto& p : squares)
	{
		auto square = p.second;
		if (square->getPosition() == screen_position)
			return p.first;
	}
	return Point(0,0);
}

void Grid::setState(Square::State state, Point point)
{
	auto square = squares.at(point);

	square->setColor(Color3B::GRAY);
	square->state = state;

	switch (state)
	{
	case Square::BEGIN:
		addSymbol("Plus.png", point);
		Util::addIfUnique<Point>(&starts, point);
		break;

	case Square::END:
		addSymbol("Minus.png", point);
		Util::addIfUnique<Point>(&delivery_points, point);
		break;

	case Square::PACKAGE:
		Util::addIfUnique<Point>(&available_packages, point);
		Util::addIfUnique<Point>(&packages, point);
		Util::addIfUnique<Point>(&static_collidables, point);
```

```
                break;

        case Square::BLOCKADE:
                Util::addIfUnique<Point>(&static_collidables, point);
                Util::addIfUnique<Point>(&blockades, point);
                break;

        case Square::EMPTY:
                removeSymbol(point);
                square->setColor(Color3B::WHITE);
                Util::removeIfContains(&starts, point);
                Util::removeIfContains(&delivery_points, point);
                Util::removeIfContains(&available_packages, point);
                Util::removeIfContains(&static_collidables, point);
                break;
        }

}

void Grid::toggleDragAndDrop()
{
        menu->setEnabled(!menu->isEnabled());
}

void Grid::drawLines()
{
        const auto visibleSize = Director::getInstance()->getVisibleSize();
        const Color4F lightGray = Color4F(0.95f, 0.95f, 0.95f, 1);

        for (int i = 0; i < number_of_columns; i++) {
                auto drawVerticalLine = DrawNode::create();
                drawVerticalLine->drawLine(Point(i * square_size, 0), Point(i *
square_size, visibleSize.height), lightGray);
                addChild(drawVerticalLine);
        }

        for (int i = 0; i < number_of_lines; i++) {
                auto drawHorizontalLine = DrawNode::create();
                drawHorizontalLine->drawLine(Point(0, i * square_size),
Point(visibleSize.width, i * square_size), lightGray);
                addChild(drawHorizontalLine);
        }
}

void Grid::addSymbol(const std::string &filename, Point point)
```

```
{
	Sprite* symbol = Sprite::create(filename);
	symbol->setContentSize(Size(square_size - 10, square_size - 10));
	symbol->setPosition(squares.at(point)->getPosition());
	symbols[point] = symbol;
	addChild(symbol);
}

void Grid::removeSymbol(Point point)
{
	if (symbols.count(point) > 0) {
		auto symbol = symbols.at(point);
		if(symbol)
			symbol->removeFromParentAndCleanup(true);

		symbols[point] = NULL;
	}
}

void Grid::createSquares()
{
	for (int j = 0; j < number_of_columns; j++) {
		for (int i = 0; i < number_of_lines; i++) {
			auto square = Square::create();
			square->setNormalImage(Sprite::create("square.png"));
			square->setContentSize(Size(square_size, square_size));

			squares[Point(j, i)] = square;
			square->grid_position = Point(j, i);

			square->setPosition(Vec2(j * square_size + square_size / 2, i
* square_size + square_size / 2));

			menu->addChild(square);
		}
	}
}

bool Grid::onTouchBegan(cocos2d::Touch *touch, cocos2d::Event *event)
{
	return true;
}


void Grid::onTouchMoved(cocos2d::Touch *touch, cocos2d::Event *event)
```

```
{
	setPosition(getPosition() + touch->getDelta());
}
```

Listagem A.18: Infobar.cpp

```
#include "Infobar.h"

bool Infobar::init()
{
	auto visibleSize = Director::getInstance()->getVisibleSize();

	auto top_bar = DrawNode::create();

	top_bar->drawSolidRect(
		Vec2(0, visibleSize.height),
		Vec2(visibleSize.width, visibleSize.height - 30),
		Color4F(0.3f, 0.3f, 0.3f, 1));

	this->addChild(top_bar);

	time_label = Label::createWithTTF("0:0:0", "fonts/arial.ttf", 20);
	time_label->setColor(Color3B::GRAY);
	time_label->setPosition(Vec2(visibleSize.width / 2, visibleSize.height - 15));
	this->addChild(time_label);

	speed_factor_label = Label::createWithTTF("Speed: x1", "fonts/arial.ttf", 20);
	speed_factor_label->setColor(Color3B::GRAY);
	speed_factor_label->setPosition(Vec2(visibleSize.width / 2 - 200,
visibleSize.height - 15));
	this->addChild(speed_factor_label);

	zoom_label = Label::createWithTTF("Zoom: x1", "fonts/arial.ttf", 20);
	zoom_label->setColor(Color3B::GRAY);
	zoom_label->setPosition(Vec2(visibleSize.width / 2 + 200, visibleSize.height -
15));
	this->addChild(zoom_label);

	this->schedule(CC_SCHEDULE_SELECTOR(Infobar::updateClock), 0.001f);

	return true;
}
```

```
void Infobar::updateSpeed(float speed)
{
	std::string speed_mult = std::to_string(speed);
	speed_factor_label->setString("Speed: x" + speed_mult.substr(0, 5));
}

void Infobar::updateZoom(float zoom)
{
	std::string speed_mult = std::to_string(zoom);
	speed_factor_label->setString("Zoom: x" + speed_mult.substr(0, 5));
}

void Infobar::updateClock(float dt)
{
	if (time)
	{
		time_label->setString(*time);
	}

}
```

Listagem A.19: Robot.cpp

```
#include "Robot.h"
#include "Util.h"

USING_NS_CC;

bool Robot::init()
{
	if (!Sprite::init())
		return false;

	return true;
}

void Robot::move()
{
	if (screen_path.empty()) return;
	Vector<FiniteTimeAction*> movements;

	auto origin = getPosition();
	reverse(screen_path.begin(), screen_path.end());
```

```
	auto callbackUpdateGridPosition = CallFunc::create(CC_CALLBACK_0(Robot::updateGridPosition, this));

	for (auto destination : screen_path)
	{
		const double distance = sqrt(pow((destination.x - origin.x), 2.0) + pow((destination.y - origin.y), 2.0));

		const float moveDuration = 0.01 * distance;

		auto moveToNextSquare = MoveTo::create(moveDuration, destination);
		movements.pushBack(moveToNextSquare);
		movements.pushBack(callbackUpdateGridPosition);

		origin = destination;
	}

	auto callbackFinishedMovement = CallFunc::create(CC_CALLBACK_0(Robot::finishedMovement, this));

	auto sequence_of_movements = Sequence::create(movements);
	movement = Sequence::create(sequence_of_movements, callbackFinishedMovement, nullptr);
	runAction(movement);
}

void Robot::stop()
{
	stopAction(movement);
	unscheduleUpdate();
}

void Robot::updateState()
{
	if (getPosition() == screen_package)
		state = FULL;

	if (getPosition() == screen_delivery_point)
		state = EMPTY;
}

bool Robot::isParked()
{
	return grid_position == grid_start;
```

```
}

bool Robot::isAtDeliverty()
{
	return getPosition() == screen_delivery_point;
}

bool Robot::isFull()
{
	return state == FULL;
}

bool Robot::isAtPackage()
{
	return getPosition() == screen_package;
}

bool Robot::isInThe(vector<Point> path)
{
	return Util::contains<Point>(&path, grid_position);
}

void Robot::addSensor(Sensor sensor)
{
	sensors.push_back(sensor);
	sensor.setOwner(this);
}

void Robot::updateGridPosition()
{
	grid_position = grid_path.back();
	grid_path.pop_back();
	complete_grid_path.push_back(grid_position);
}

void Robot::finishedMovement()
{
	if (isAtDeliverty())
	{
		EventCustom event("robot_at_delivery");
		event.setUserData(this);
		_eventDispatcher->dispatchEvent(&event);
	}
	else if (isAtPackage())
	{
```

```
                EventCustom event("robot_at_package");
                event.setUserData(this);
                _eventDispatcher->dispatchEvent(&event);
        }
        else
        {
                EventCustom event("robot_is_parked");
                event.setUserData(this);
                _eventDispatcher->dispatchEvent(&event);
        }

        grid_path.clear();
        screen_path.clear();
        updateState();

        if (!isParked())
        {
                EventCustom event("robot_completed_movement");
                event.setUserData(this);
                _eventDispatcher->dispatchEvent(&event);
        }

}
```

Listagem A.20: RobotController.cpp

```
#include "RobotController.h"


void RobotController::onEnter()
{
        Node::onEnter();

        setupGenerator();

        auto robotCompletedMovementListener =
EventListenerCustom::create("robot_completed_movement",
CC_CALLBACK_1(RobotController::robotCompletedMovement, this));
        _eventDispatcher-
>addEventListenerWithSceneGraphPriority(robotCompletedMovementListener, this);
}

void RobotController::definePathOf(Robot * robot)
{
        vector<Point> destinations;
```

```
	if (robot->state == Robot::EMPTY && !grid->available_packages.empty())
	{
		destinations = grid->available_packages;
		robot->grid_path = this->findShortestPath(robot->grid_position,
destinations);
		robot->screen_path = this->convertGridPathToScreenPath(robot-
>grid_path);

		robot->screen_package = robot->screen_path[0];

		Util::removeIfContains<Point>(&grid->available_packages, robot-
>grid_path[0]);
	}

	else if(robot->state == Robot::FULL)
	{
		destinations = grid->delivery_points;
		robot->grid_path = this->findShortestPath(robot->grid_position,
destinations);
		robot->screen_path = this->convertGridPathToScreenPath(robot-
>grid_path);

		robot->screen_delivery_point = robot->screen_path[0];
	}

	else if (robot->state == Robot::EMPTY && grid->available_packages.empty())
	{
		destinations = { robot->grid_start };
		robot->grid_path = this->findShortestPath(robot->grid_position,
destinations);
		robot->screen_path = this->convertGridPathToScreenPath(robot-
>grid_path);
	}

	robot->grid_destination = robot->grid_path[0];
	robot->screen_destination = robot->screen_path[0];
}

void RobotController::preventCollisionOf(Robot * robot)
{
	auto next_position = robot->grid_path.back();

	if (isCollisionImminent(next_position))
	{
```

```
                auto collision_robot = getRobotAt(next_position);
                auto path = collision_robot->grid_path;

                if (robot->isInThe(path) || collision_robot->grid_path.empty())
                {
                        grid->static_collidables.push_back(next_position);
                        robot->grid_path = findShortestPath(robot->grid_position, {
robot->grid_destination });
                        grid->static_collidables.pop_back();
                }
                else
                {
                        robot->grid_path.push_back(robot->grid_position);
                }
        }
}

void RobotController::repath(Robot * r1, Robot * r2)
{
        auto next_position = r1->grid_path.back();
        auto path = r2->grid_path;

        if (r1->isInThe(path) || r2->grid_path.empty())
        {
                for (auto p : r2->grid_path)
                {
                        grid->static_collidables.push_back(next_position);
                }

                r1->grid_path = findShortestPath(r1->grid_position, { r1-
>grid_destination });
                r1->screen_path = this->convertGridPathToScreenPath(r1-
>grid_path);

                for (auto p : r2->grid_path)
                {
                        grid->static_collidables.pop_back();
                }

        }
}

vector<Point> RobotController::findShortestPath(Point origin, vector<Point>
destinations) {
        vector<Point> shortest_path;
```

```
	int min_size = std::numeric_limits<int>::max();

	for (Point destination : destinations)
	{
		path_generator.clearCollisions();
		path_generator.addCollisions(grid->static_collidables);
		path_generator.removeCollision(destination);

		auto path = path_generator.findPath({ origin.x, origin.y }, {
destination.x, destination.y });
		if (path.size() < min_size) {
			shortest_path = path;
			min_size = path.size();
		}
	}

	return shortest_path;
}

bool RobotController::isCollisionImminent(Point next_position)
{
	if (this->getRobotAt(next_position))
		return true;
	else
		return false;
}

vector<Point> RobotController::convertGridPathToScreenPath(vector<Point> path)
{
	vector<Point> screen_path;
	for (auto point : path)
	{
		screen_path.push_back(grid->getPositionOf(point));
	}
	return screen_path;
}

void RobotController::setupGenerator()
{
	path_generator.setWorldSize({ grid->number_of_columns, grid-
>number_of_lines });
	path_generator.setHeuristic(AStar::Heuristic::manhattan);
	path_generator.setDiagonalMovement(true);
}
```

```
Robot * RobotController::getRobotAt(Point grid_position)
{
	for (auto robot : robots)
	{
		if (robot->grid_position == grid_position)
			return robot;
	}
	return nullptr;
}

void RobotController::robotCompletedMovement(EventCustom* event)
{
	Robot* robot = static_cast<Robot*>(event->getUserData());
	this->definePathOf(robot);
	robot->move();
}
```

Listagem A.21: Sensor.cpp

```
#include "Sensor.h"

bool Sensor::init()
{
	if (!Sensor::init())
		return false;

	return true;
}

void Sensor::signal()
{
	EventCustom event("change_path");
	event.setUserData(this);
	_eventDispatcher->dispatchEvent(&event);
}

void Sensor::setOwner(Node * own)
{
	this->owner = own;
}
```

Listagem A.22: Simulator.cpp

```
#include "Simulator.h"
```

```
#include "SimpleAudioEngine.h"
#include "AStar.hpp"
#include <algorithm>

USING_NS_CC;

bool Simulator::init()
{
	if (!Scene::initWithPhysics())
	{
		return false;
	}

	speed_multiplier = 1.0f;

	grid = Grid::create();
	stopwatch = Stopwatch::create();
	infobar = Infobar::create();
	toolbar = Toolbar::create();
	actionbar = Actionbar::create();
	robotController = RobotController::create();

	addChild(grid);
	addChild(stopwatch);
	addChild(infobar);
	addChild(toolbar);
	addChild(actionbar);
	addChild(robotController);

	setup();

	auto contactListener = EventListenerPhysicsContact::create();
	contactListener->onContactBegin =
CC_CALLBACK_1(Simulator::onContactBegin, this);
	_eventDispatcher->addEventListenerWithSceneGraphPriority(contactListener,
this);

	createCustomEvents();

   return true;
}

void Simulator::setCallbacks()
{
```

```
		actionbar->runItem->setCallback(CC_CALLBACK_1(Simulator::menuRunCallback, this));
		actionbar->exportItem->setCallback(CC_CALLBACK_1(Simulator::menuExportCallback, this));
		actionbar->resetItem->setCallback(CC_CALLBACK_1(Simulator::menuResetCallback, this));
		actionbar->speedUpItem->setCallback(CC_CALLBACK_0(Simulator::menuChangeSpeedCallback, this, 1/2.0));
		actionbar->slowDownItem->setCallback(CC_CALLBACK_0(Simulator::menuChangeSpeedCallback, this, 2.0));
		actionbar->moveItem->setCallback(CC_CALLBACK_1(Simulator::menuMoveGridCallback, this));
		actionbar->zoomInItem->setCallback(CC_CALLBACK_0(Simulator::menuZoomCallback, this, 1.1));
		actionbar->zoomOutItem->setCallback(CC_CALLBACK_0(Simulator::menuZoomCallback, this, 1/1.1));

		for (const auto &p : grid->squares)
		{
			auto square = p.second;
			square->setCallback(CC_CALLBACK_0(Simulator::gridSquareCallback, this, square->grid_position));
		}
}

void Simulator::createCustomEvents()
{
		auto robotAtDeliveryListener = EventListenerCustom::create("robot_at_delivery", CC_CALLBACK_1(Simulator::robotIsAtDelivery, this));
		_eventDispatcher->addEventListenerWithSceneGraphPriority(robotAtDeliveryListener, this);

		auto robotAtPackageListener = EventListenerCustom::create("robot_at_package", CC_CALLBACK_1(Simulator::robotIsAtPackage, this));
		_eventDispatcher->addEventListenerWithSceneGraphPriority(robotAtPackageListener, this);

		auto robotIsParkedListener = EventListenerCustom::create("robot_is_parked", CC_CALLBACK_1(Simulator::robotIsParked, this));
		_eventDispatcher->addEventListenerWithSceneGraphPriority(robotIsParkedListener, this);
}
```

```
void Simulator::setup()
{
	grid->setPosition(30, 0);
	infobar->time = &stopwatch->text;
	robotController->grid = grid;

	setCallbacks();
}

void Simulator::start()
{
	createRobots();

	for (auto robot : robots)
	{
		robotController->definePathOf(robot);
		robot->move();
	}

	robotController->robots = robots;
	stopwatch->setSpeedMultiplier(speed_multiplier);
	stopwatch->start();
}

void Simulator::stop()
{
	stopwatch->stop();
	for (auto robot : robots)
	{
		robot->pause();
	}
}

void Simulator::proceed()
{
	stopwatch->start();
}

void Simulator::createRobots() {
	if (!robots.empty()) return;

	for (Point start : grid->starts)
	{
		auto robot = Robot::create();
```

```
                robot->initWithFile("Robot.png");
                robot->setPosition(grid->getPositionOf(start));
                robot->setColor(Color3B(150, 150, 150));
                robot->setContentSize(Size(grid->square_size, grid->square_size));
                robot->grid_position = start;
                robot->grid_start = start;

                auto physicsBody = PhysicsBody::createBox(Size(60.0f, 60.0f),
PhysicsMaterial(0.1f, 1.0f, 0.0f));
                physicsBody->setGravityEnable(false);
                physicsBody->setDynamic(true);
                physicsBody->setContactTestBitmask(true);
                robot->addComponent(physicsBody);

                grid->addChild(robot);

                robots_bodies[physicsBody] = robot;
                robots.push_back(robot);
        }
}

bool Simulator::allPackagesWereDelivered()
{
        return grid->packages.size() == packages_delivered.size();
}

bool Simulator::allRobotsAreParked()
{
        for (auto robot : robots)
        {
                if (!robot->isParked())
                        return false;
        }
        return true;
}

void Simulator::reset()
{
        for (Point package : grid->packages)
                grid->setState(Square::PACKAGE, package);

        grid->available_packages = grid->packages;

        infobar->time = &stopwatch->text;
        for (auto robot : robots)
```

```
                robot->removeFromParentAndCleanup(true);

        robots.clear();
        packages_delivered.clear();
        stopwatch->reset();
}

void Simulator::menuRunCallback(cocos2d::Ref * pSender)
{
        start();
}

void Simulator::menuResetCallback(cocos2d::Ref * pSender)
{
        stop();
        reset();
}

void Simulator::gridSquareCallback(Point coord)
{
        auto robot = robotController->getRobotAt(coord);
        switch (toolbar->selected)
        {
        case Toolbar::PACKAGE:
                grid->setState(Square::PACKAGE, coord);
                break;

        case Toolbar::BEGIN:
                grid->setState(Square::BEGIN, coord);
                break;

        case Toolbar::END:
                grid->setState(Square::END, coord);
                break;

        case Toolbar::ERASE:
                grid->setState(Square::EMPTY, coord);
                break;

        case Toolbar::BLOCKADE:
                grid->setState(Square::BLOCKADE, coord);
                break;
        case Toolbar::PATH:
                if (robot)
                {
```

```
                        for (auto coord : robot->complete_grid_path)
                                grid->squares[coord]->setColor(Color3B::RED);
                }
                break;
        case Toolbar::CLOCK:
                if (robot)
                {
                        auto robot_time = robot_times[robot];
                        std::string time = robot_time.getCString();
                        infobar->time = &time;
                }
                break;
        }

}

void Simulator::menuExportCallback(cocos2d::Ref * pSender)
{
        std::ofstream out("times.txt");

        out << "Total time: ";
        out << stopwatch->toString();
        out << std::endl;

        int id = 0;
        for (auto robot : robots)
        {
                out << "id: ";
                out << id;
                out << " time: ";
                out << robot_times[robot].getCString();
                out << std::endl;
                id++;
        }

        out.close();
}

void Simulator::menuChangeSpeedCallback(float multiplier)
{
        stop();
        speed_multiplier *= multiplier;
        infobar->updateSpeed(1 / speed_multiplier);
        start();
}
```

```
void Simulator::menuMoveGridCallback(cocos2d::Ref * pSender)
{
	grid->toggleDragAndDrop();
}

void Simulator::menuZoomCallback(float multiplier)
{
	const float scale = grid->getScale();
	grid->setScale(scale * multiplier);
	infobar->updateZoom(scale * multiplier);

}

bool Simulator::onContactBegin(PhysicsContact & contact)
{
	auto bodyA = contact.getShapeA()->getBody();
	auto bodyB = contact.getShapeB()->getBody();

	auto r1 = robots_bodies[bodyA];
	auto r2 = robots_bodies[bodyB];

	r1->stop();

	robotController->repath(r1, r2);

	r1->move();

	return false;
}

void Simulator::robotIsAtDelivery(EventCustom* event)
{
	const Robot* robot = static_cast<Robot*>(event->getUserData());
	Point grid_package = grid->getGridPositionOf(robot->screen_package);
	Util::addIfUnique<Point>(&packages_delivered, grid_package);
}

void Simulator::robotIsAtPackage(EventCustom* event)
{
	const Robot* robot = static_cast<Robot*>(event->getUserData());
	Point grid_package = grid->getGridPositionOf(robot->screen_package);
	grid->setState(Square::EMPTY, grid_package);
}
```

```
void Simulator::robotIsParked(EventCustom* event)
{
	Robot* robot = static_cast<Robot*>(event->getUserData());
	robot_times[robot] = stopwatch->toString();

	if (allRobotsAreParked() && allPackagesWereDelivered())
		stop();
}
```

Listagem A.23: Square.cpp

```
#include "Square.h"

using namespace cocos2d;

void Square::onEnter()
{
	MenuItemImage::onEnter();
	state = EMPTY;
}
```

Listagem A.24: Stopwatch.cpp

```
#include "Stopwatch.h"

USING_NS_CC;

bool Stopwatch::init()
{
	Node::init();
	this->speed_multiplier = 1.0f;
	this->reset();

	return true;
}

void Stopwatch::start()
{
	this->schedule(CC_SCHEDULE_SELECTOR(Stopwatch::count), 0.001f *
speed_multiplier);
}
```

```
void Stopwatch::stop()
{
	this->unschedule(CC_SCHEDULE_SELECTOR(Stopwatch::count));
}

int Stopwatch::getMinutes()
{
	return minutes;
}

int Stopwatch::getSeconds()
{
	return seconds;
}

int Stopwatch::getMilliseconds()
{
	return milliseconds/10;
}

void Stopwatch::reset()
{
	milliseconds = 0;
	seconds = 0;
	minutes = 0;

	text = this->toString();
}

std::string Stopwatch::toString()
{
	std::string text = std::to_string(minutes) + ":" + std::to_string(seconds) + ":" +
std::to_string(milliseconds/10);
	return text;
}

void Stopwatch::setSpeedMultiplier(float factor)
{
	this->speed_multiplier = factor;
}

void Stopwatch::count(float dt)
{
	milliseconds++;
```

```
        if (milliseconds == 1000)
        {
                milliseconds = 0;
                seconds++;
        }

        if (seconds == 60)
        {
                seconds = 0;
                minutes++;
        }

        text = this->toString();
}
```

Listagem A.25: Toolbar.cpp

```
#include "Toolbar.h"

bool Toolbar::init()
{
        auto visibleSize = Director::getInstance()->getVisibleSize();

        auto left_bar = DrawNode::create();

        left_bar->drawSolidRect(
                Vec2(0, 0),
                Vec2(30, visibleSize.height),
                Color4F(0.3f, 0.3f, 0.3f, 1));

        this->addChild(left_bar);

        selected_bg = DrawNode::create();
        selected_bg->drawSolidRect(
                Vec2(0, 0),
                Vec2(30, 30),
                Color4F(0.20f, 0.20f, 0.20f, 1));
        selected_bg->setContentSize(Size(30, 30));
        selected_bg->setVisible(false);
        selected_bg->setAnchorPoint(Vec2(0.5, 0.5));
        this->addChild(selected_bg);

        createButtons();
```

```
	auto menu = Menu::create();
	menu->setPosition(Vec2::ZERO);

	buttons.push_back(packageItem);
	buttons.push_back(beginItem);
	buttons.push_back(endItem);
	buttons.push_back(eraseItem);
	buttons.push_back(blockadeItem);
	buttons.push_back(clockItem);
	buttons.push_back(pathItem);

	int offset = 300;
	for (int i = 0; i < buttons.size(); i++)
	{
		buttons[i]->setPosition(Vec2(15, visibleSize.height - 15 * i * 2 - offset));
		menu->addChild(buttons[i]);
	}

	this->addChild(menu, 1);

	this->setCallbacks();

	return true;
}

void Toolbar::setTool(Tool tool, MenuItemImage * btn)
{
	selected_bg->setVisible(true);
	this->selected = tool;
	selected_bg->setPosition(btn->getPosition());
}

void Toolbar::setCallbacks()
{
	this->packageItem->setCallback(CC_CALLBACK_0(Toolbar::setTool, this,
Toolbar::PACKAGE, packageItem));
	this->beginItem->setCallback(CC_CALLBACK_0(Toolbar::setTool, this,
Toolbar::BEGIN, beginItem));
	this->endItem->setCallback(CC_CALLBACK_0(Toolbar::setTool, this,
Toolbar::END, endItem));
	this->eraseItem->setCallback(CC_CALLBACK_0(Toolbar::setTool, this,
Toolbar::ERASE, eraseItem));
	this->blockadeItem->setCallback(CC_CALLBACK_0(Toolbar::setTool, this,
Toolbar::BLOCKADE, blockadeItem));
```

```
	this->clockItem->setCallback(CC_CALLBACK_0(Toolbar::setTool, this,
Toolbar::CLOCK, clockItem));
	this->pathItem->setCallback(CC_CALLBACK_0(Toolbar::setTool, this,
Toolbar::PATH, pathItem));
}

void Toolbar::createButtons()
{
	packageItem = MenuItemImage::create(
		"PackageBtn.png",
		"PackageBtn_pressed.png");

	beginItem = MenuItemImage::create(
		"PlusBtn.png",
		"PlusBtn_pressed.png");

	endItem = MenuItemImage::create(
		"MinusBtn.png",
		"MinusBtn_pressed.png");

	eraseItem = MenuItemImage::create(
		"EraseBtn.png",
		"EraseBtn_pressed.png");

	blockadeItem = MenuItemImage::create(
		"BlockadeBtn.png",
		"BlockadeBtn_pressed.png");

	pathItem = MenuItemImage::create(
		"BlockadeBtn.png",
		"BlockadeBtn_pressed.png");

	clockItem = MenuItemImage::create(
		"Timer.png",
		"Timer_pressed.png");
}
```

ANEXO B – Artigo

Simulador de Robôs Moveis Terrestres

Ricardo Ademar Bezerra de Almeida[1]

[1]Universidade Federal de Santa Catarina (UFSC)
Departamento de Informática e Estatística

Campus Universitário – Florianópolis, SC - Brasil

ricardo.almeida@grad.ufsc.br

Abstract. *Currently mobile robotics is an area of expertise very broad, with many different types of robots performing a variety of tasks to automate work that was generally done by people. However, to test so many different cases in which robots can find themselves can be very expensive, unless we utilize a simulator to do so. With the growth of robotic came the increase need for simulators to perform different tasks and adapt to a variety of needs, such as different robots, environments and ways of navigating. The objective of this thesis is to develop of a simulator for terrain robots on an already known environment mimicking a transport robot, allowing a swarm of robot to share the same environment without colliding between themselves while looking for optimal paths as a way to reduce the cost and time of the tasks being performed, respecting deadlines that were previously determined. On this project a simulator was developed using the framework Cocos2d-x and a test case was proposed to demonstrate his functionalities.*

Resumo. *Atualmente a robótica móvel é uma área bastante abrangente, com vários tipos de robôs realizando uma diversidade de tarefas, de forma a automatizar trabalhos que eram previamente realizadas por pessoas. No entanto, para testar diferentes casos e situações em que esses robôs devem atuar, pode ser muito caro e demorado utilizando os mesmos. Por isso a utilização de simuladores para robôs móveis é uma decisão comum para definir e solucionar problemas de modo mais barato e rápido. O objetivo deste projeto é desenvolver um simulador para robôs terrestres autônomos em um ambiente parcialmente conhecido, onde os elementos estáticos já estão mapeados mas os móveis, como pessoas e outros robôs não, onde múltiplos robôs vão compartilhar o mesmo ambiente com o objetivo de carregar e transportar itens evitando colisões e buscando o melhor caminho aos seus objetivos, obedecendo as deadlines estipuladas. Este simulador foi desenvolvido utilizando o framework Cocos2d-x e testado em um caso proposto demonstrando sua utilização.*

1. Introdução

Atualmente o uso de robôs móveis para realizar diversas tarefas tem crescido, uma das razões para esse aumento é a redução do custo de hardware incentivando o mercado a utilizar robôs como um método para uma melhor eficiência na realização de determinadas tarefas antes realizadas por pessoas [MARKS, 2019].

Robôs móveis podem ser utilizados em uma variedade de funções a fim de completar certos objetivos, como por exemplo na exploração de ambientes [JIA, 2004], até mesmo fora do planeta [NILSON, 2018], no deslocamento de cargas de um ponto a outro, como nos armazéns da Amazon ou na perseguição de alvos móveis [MARKS, 2019].

Uma grande parte dessas tarefas é limitada por restrições temporais. Por exemplo, um robô da Amazon precisa entregar pacotes em um ponto de entrega em um tempo limite [MARKS, 2019], ou então um robô em Marte deve explorar o máximo do ambiente em que se encontra para adquirir uma maior quantidade de informações antes da sua capacidade energética se esgotar [NILSON, 2018]. Como essas tarefas devem ser realizadas em um tempo limite, robôs móveis são considerados Sistemas de Tempo Real (RTS, Real-Time Systems) [BREGA, 2000] e a falta de eficiência nesses casos pode causar grande prejuízo financeiro aos indivíduos ou organizações responsáveis por esses robôs [MARKS, 2019].

No entanto o problema não termina por aí, pois nem sempre esses robôs agem individualmente, mas sim em conjunto com vários outros robôs, humanos ou qualquer outra entidade que possa afetar o movimento dos robôs. Com isso é necessário a utilização de sensores ou conhecimento prévio do ambiente, e por vezes ambos, para que o robô realize suas tarefas com sucesso e sem colisão, evitando causar danos aos mesmos ou em determinados casos à pessoas, animais, ao ambiente ou outros equipamentos que compartilham o local onde o robô se movimenta.

Este trabalho propõe o desenvolvimento de um simulador 2D de robôs de carga em um ambiente parcialmente conhecido e de criticalidade soft, altamente modular utilizando o framework Cocos2d-x, permitindo ao usuário uma grande capacidade de modificar componentes de cada robô, adicionar ou desabilitar módulos e modificar o comportamento de cada robô de modo a realizar seus objetivos sejam eles para pesquisa ou para o mercado.

2. Fundamentação teórica

Para a compreensão do trabalho é necessário discutir alguns conceitos básicos relacionados a Robótica Móvel, Sistemas de Tempo Real (STR) que serão apresentados brevemente nesta seção, discutindo os tipos de robô moveis, sensores e ambientes de navegação deles. Assim como as classificações de Sistemas de Tempo Real.

2.1 Tempo real

Sistemas computacionais onde há uma restrição temporal que deve ser respeitada são chamados de sistemas de tempo real. Nesses sistemas o tempo de resposta para um evento deve ser respeitado e, caso ele ultrapasse um limite pré-definido, mesmo que a tarefa tenha sido executada ela será caracterizada como uma falha [FARINES, 2000].

Um sistema de tempo real também pode ser classificado pelo impacto que pode ser causado pela perda de um deadline, definido o quão crítico é a necessidade de que esse sistema cumpra as suas tarefas em um tempo predefinido. As classificações de criticalidade são:

- *Hard*: são sistemas onde a perda de um deadline pode ser catastrófica.

- *Soft*: onde a perda de deadline não corresponde a uma falha completa, mas que mesmo assim, prevê consequências na performance do sistema.
- *Firm*: similar ao soft, a perda de deadline não corresponde a falha do sistema, mas diferente do soft, múltiplas perdas podem fazer com que o sistema falhe por completo.

2.2 Robótica Móvel

A robótica é uma área relativamente nova, ela se tornou inicialmente bastante popular graças a sua utilização na área de fabricação industrial com o uso de braços robóticos para realizar tarefas repetitivas rapidamente, como por exemplo a solda e pintura de peças para automóveis [SIEGWART; NOURBAKHSH, 2011]. Algumas características podem ser atribuídas a esses robôs dependendo o local onde navegam, sensores que utilizam são descrevidos a seguir.

2.2.1 Tipos de Robôs

Robôs móveis podem ser divididos entre dois tipos: os tele operados, robôs controlados por uma pessoa a todo momento durante seu funcionamento e os autônomos, robôs que se movem por conta própria seguindo um algoritmo interno que define seu comportamento e o uso de sensores que permitem que ele compreenda e responda adequadamente ao ambiente que se encontra.

Além disso robôs móveis podem se enquadrar em diferentes subcategorias, sendo elas os robôs terrestres, aquáticos e aéreos.

2.2.2 Sensores

Como foi estabelecido anteriormente, robôs autônomos necessitam de sensores para compreender o ambiente no qual eles se encontram, especialmente se esse ambiente se encontra em constante mudança, como um ambiente que contenha objetos com mobilidade.

2.2.3 Navegação

Além de distinguir os tipos de robôs e os sensores que eles utilizam que são características dos robôs em si, também é necessário diferenciar os diferentes tipos de ambientes nos quais eles se locomovem, os quais podem ser divididos em duas categorias, ambiente conhecidos, não-conhecidos e os parcialmente conhecidos.

3. Cocos2d-x

A principal ferramenta para o desenvolvimento deste simulador foi o framework Cocos2d-x. Em geral este framework é voltado para o desenvolvimento de jogos e apresenta, portanto, muitas características semelhantes àquelas necessárias para a criação de simuladores para robótica móvel como, por exemplo, ambientes físicos, interfaces gráficas e facilidade para interação do usuário com o sistema. Estes fatores foram decisivos para a escolha da sua utilização neste

projeto. Além disso, o fato dele ser um framework de código-aberto, permitindo a sua modificação interna caso necessário, o torna bastante interessante para a comunidade científica.

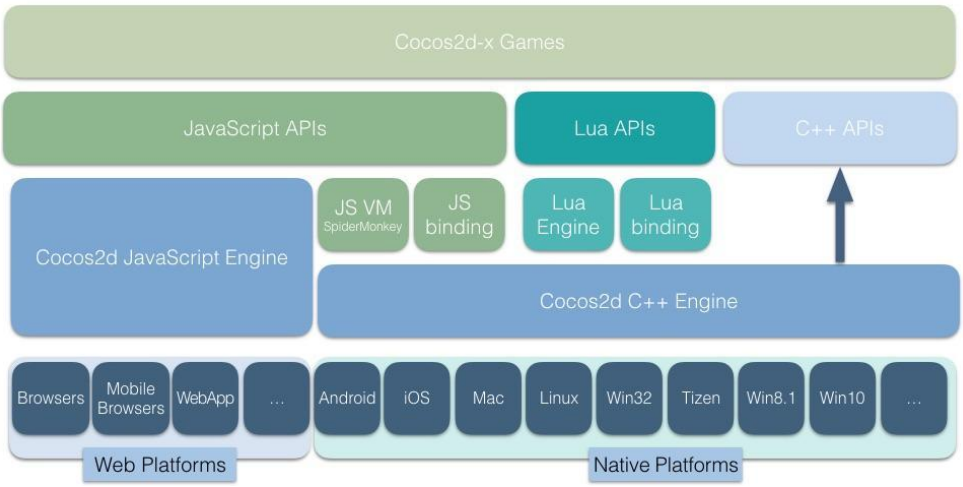

Figura 9 Arquitetura do *framework* Cocos2d-x

Na Figura 1 podemos ver a arquitetura geral do Cocos2d-X e seus diferentes módulos, Cocos2d-x *Games* representa o aplicativo desenvolvido utilizando uma de suas APIs, sendo o Cocos2d C++ *Engine* e o C++ APIs utilizados no desenvolvimento do simulador criado neste projeto. Por fim, na parte inferior da figura é possível distinguir as plataformas suportadas, que dependem da API sendo utilizada.

3.1 Principais componentes

Para compreender e utilizar o Cocos2D-x é necessário primeiramente o entendimento de alguns componentes básicos que são comumente utilizados para o desenvolvimento de jogos usando este *framework*, eles são brevemente descritos a seguir:

- *Node*: elemento básico com propriedades como posição, escala e ponto ancora.
- *Sprite*: elemento visual 2D de um aplicativo.
- *Action*: ação realizada com o intuito de modificar as propriedades de um nodo.
- *Layer*: contêiner de nodos.
- *Scheduler*: responsável por ativar *callbacks* em determinado tempo.
- *Scene*: elemento abstrato que define uma cena de um jogo que contem *layers*.
- *Director*: responsável pela mudança de cenas.

3.2 Motores físicos

Além da facilidade na criação da interface gráfica, Cocos2d-x já disponibiliza dois motores físicos para simular objetos em um espaço físico, o Chipmunk e Box2D que podem ser permutados sem necessidade de modificar a estrutura ou código do projeto.

Através desses motores físicos pode se atribuir um corpo físico para qualquer Sprite, lhes dando propriedades físicas que permite a colisão de corpos, aceleração e gravidade a essas Sprites, o que neste projeto se traduz em adicionar essas propriedades aos robôs móveis.

3.3 Troca de mensagens

O Cocos2d-x possui um módulo que permite uma comunicação por via de troca de mensagens, este que será utilizado neste projeto para a maior parte das comunicações entre os módulos do simulador.

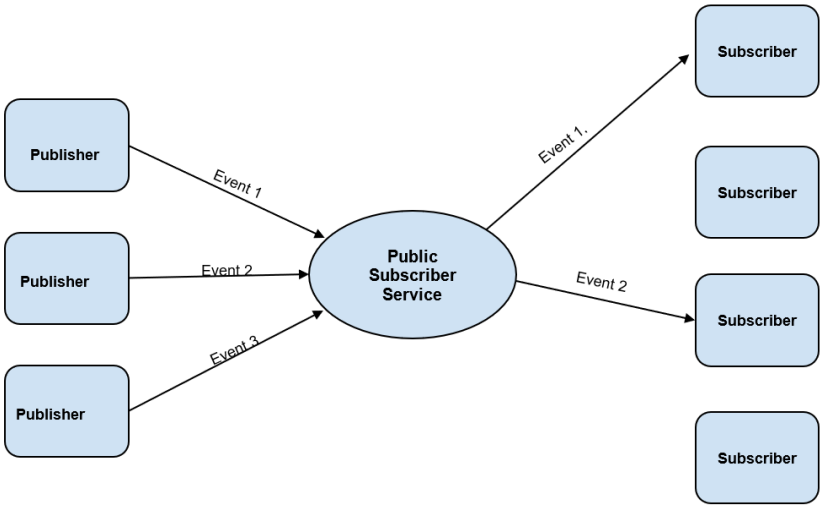

Figura 10 Exemplo de comunicação *publish-subscribe*

O seu funcionamento segue o padrão *publish-subscribe*, onde um processo cria um evento e os objetos inscritos aguardam este evento, quando ele ocorre, todos os objetos inscritos são avisados do ocorrido, como exemplificado na Figura 2.

4. Outros simuladores

Nesta seção serão comparados os simuladores acima mencionados, junto com o simulador desenvolvido neste projeto, na tabela abaixo contribui em destacar alguma das características dos simuladores discutidos. É possível notar que todos eles são de código aberto, sendo que o WeBots, que até 2018 tinha seu código fechado recentemente mudou sua política. Todos eles foram desenvolvidos em C++ assim como este projeto. O motor de física ODE foi o mais comum dentre os simuladores analisados.

Simulador	Código aberto	Plataformas	2D/3D	Motor de Física	Linguagens
WeBots	Sim	Linux, macOS, Windows	3D	ODE	C++
Stage	Sim	Linux, Solaris, macOS	2D	-	C++
Gazebo	Sim	Linux, macOS, Windows	3D	ODE, Bullet, Simbody, DART	C++
OpenRAVE	Sim	Linux, macOS, Windows	3D	ODE, Bullet	C++, Python
SimZ	Sim	Windows, iOS, macOS, Linux, Android	2D	Chipmunk, Box2D	C++

Figura 11 Tabela de simuladores pesquisados

Com exceção do Stage, a maior parte destes simuladores tende a uma alta precisão de simulação, com modelos 3D e poucos robôs em seus ambientes. Este foi um dos diferenciais que este TCC atacou, focando em uma alta quantidade de robôs em um mesmo ambiente e uma menor precisão de simulação, ou seja, neste trabalho não estamos preocupados com especificidades dos robôs e sim na definição das tarefas que eles devem cumprir e nos tempos que devem ser respeitados. No contexto de pesquisa deste trabalho, as especificidades dos robôs em relação ao hardware usado podem ser consideradas uma preocupação secundária já que eles serão usados em ambientes fechados, parcialmente conhecidos e toleram a perda de alguns deadlines. Estamos tratando de sistemas de tempo real não críticos e, portanto, o detalhamento do hardware dos robôs se mostra uma questão menos relevante do que a definição das tarefas e seus tempos de conclusão.

5. Solução proposta

O simulador proposto neste TCC é divido em 6 módulos, que se comunicam entre si através do sistema de mensagens disponibilizado pelo Cocos2d-x. Estes módulos são iniciados pelo módulo principal, chamado Simulator que constitui a cena principal do aplicativo.

O módulo Simulator interage com o GUI de forma a responder eventos iniciados pelo usuário, em contrapartida o Simulator responde a estes eventos, modificando os dados que definem os elementos visuais da simulação.

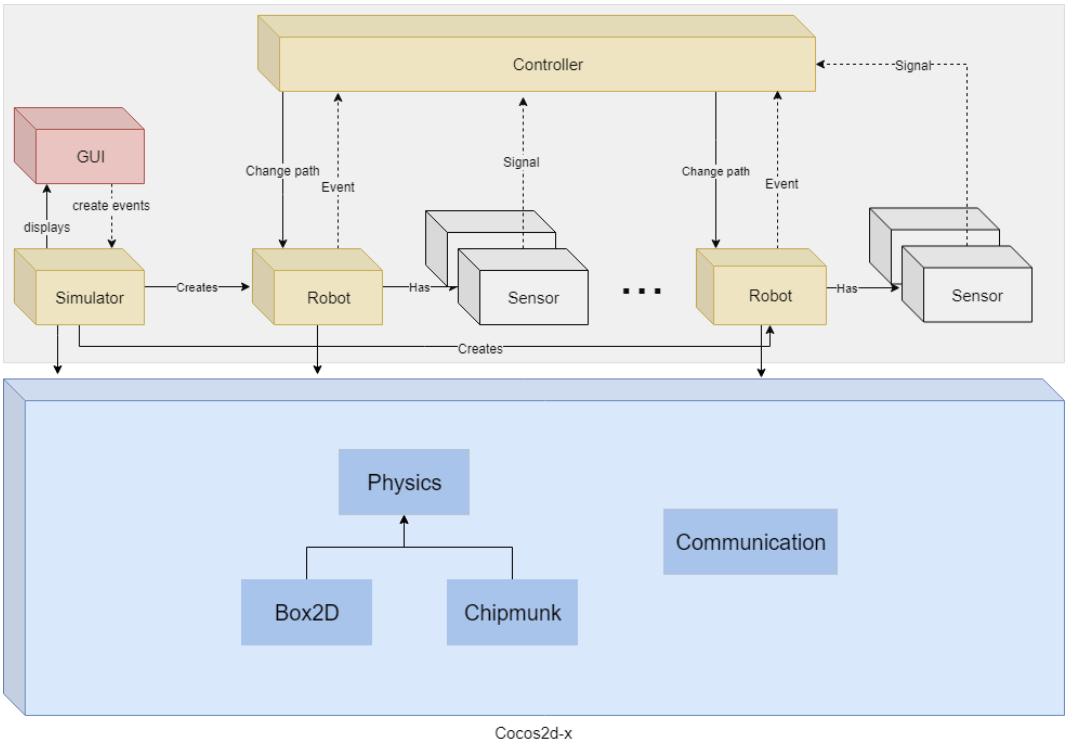

Figura 12 Arquitetura do simulador

A GUI permite que o usuário defina e edite o mapa no qual os robôs irão navegar, a localização inicial dos robôs, os pacotes que os robôs devem buscar e o ponto de entrega no qual esses pacotes devem ser entregues. Além disso ela possui botões que permitem acelerar ou desacelerar a simulação, modificar o Zoom, entre outras funcionalidades.

Quando o usuário inicia a simulação, o Simulator cria os robôs, representado pelo módulo Robot, que de acordo com o algoritmo definido no módulo Controller e em conjunto com os módulos Sensor, irá definir os caminhos pelos quais os robôs seguirão até o cumprimento de suas tarefas.

A figura Figura 4 a arquitetura e interação dos módulos descrito, com a adição do framework como base para seu funcionamento. Dentro do módulo Cocos2d-x são destacados seus componentes mais relevantes para este projeto. O módulo de Physics pode ser representado tanto pelo motor Box2D como pelo Chipmunk como estabelecido no capítulo de tecnologias. E o módulo de comunicação que se torna essencial para a comunicação entre diferentes módulos.

Já os módulos acima do framework, destacados em cinza, foram implementados neste trabalho, entre eles temos o Robot e o Simulador, ambos herdam classes provenientes do framework, sprite e scene respectivamente. O Robot mantém uma comunicação com o Controller por meio de eventos, que irão definir suas próximas ações. Além disso os sensores também podem mandar sinais que possivelmente alteram o caminho que o robô tomara. Foram desenvolvidos três sensores neste projeto, um sensor de contato, um GPS e um sensor de proximidade.

6. Simulações e resultados obtidos

Para testar o simulador foi criado um cenário onde três usuários realizaram compras em um site online, e agora seus produtos (um pedido), atualmente localizados em corredores de um armazém devem ser transportados, por alguns robôs de carga, a um ponto de entrega. Neste trabalho, um ponto de entrega se refere à um espaço físico onde serão depositados estes pedidos os quais serão, posteriormente, carregados para um meio de transporte (caminhões, containers, aviões, drones e outros meios de transporte).

Usuários	Produtos comprados
Usuário 1	A, B, C
Usuário 2	A, X, Y
Usuário 3	A, Y, Z

Figura 13 Tabela dos produtos comprados

Os produtos estão distribuídos em um galpão, que possui prateleiras, formando corredores que limitam o número de robôs que podem passar por eles, a localização deles pode ser observada na Figura 6. Nela podemos ver a disposição dos produtos A, B, C, X, Y e Z nas prateleiras. A área *Start*, onde os robôs (R1, R2 e R3) iniciam na simulação para transportar os objetos comprados. A área *Delivery* onde os robôs podem descarregar os objetos para serem enviados aos compradores. Por fim temos a *Drop Area*, onde os robôs podem descarregar objetos para que outros robôs os levem até a Delivery.

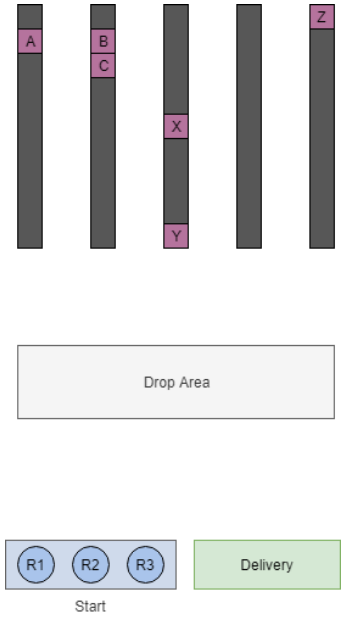

Figura 14 Cenário da simulação

Neste cenário, três simulações serão realizadas de forma a determinar qual é o melhor plano de ação para que todos os produtos sejam depositados dentro da área Delivery sem a perda de deadline e com tempo mínimo. Cada robô poderá carregar apenas um objeto por vez. A seguir discutiremos as três abordagens que serão testadas e qual o comportamento de cada robô em cada uma delas.

- Simulação 1: Na primeira simulação cada robô será designado a um usuário.
- Simulação 2: Na segunda simulação, os robôs R1 e R2 serão responsáveis por transportar os produtos apenas até a Drop Area, sendo que R3 será permitido transportar apenas objetos da Drop Area até a área de Delivery.
- Simulação 3: Na terceira simulação será permitido que qualquer robô transporte qualquer objeto. O comportamento dos robôs será a de buscar o objeto mais próximo de si.

6.1 Resultados

A seguir serão demonstrados os dados temporais obtidos nas simulações executadas, juntamente com uma análise desses resultados. Na Figura 7 pode-se verificar o tempo total de cada simulação, com a simulação 1 tendo uma duração de 23,54 segundos, 22,24 segundos para a simulação 2 terminar e com o menor tempo a simulação 3 foi finalizada em 21,27 segundos. Com isso podemos concluir que o algoritmo de comportamento aplicado na simulação 3 é o mais eficiente quando se considera apenas o tempo total da simulação

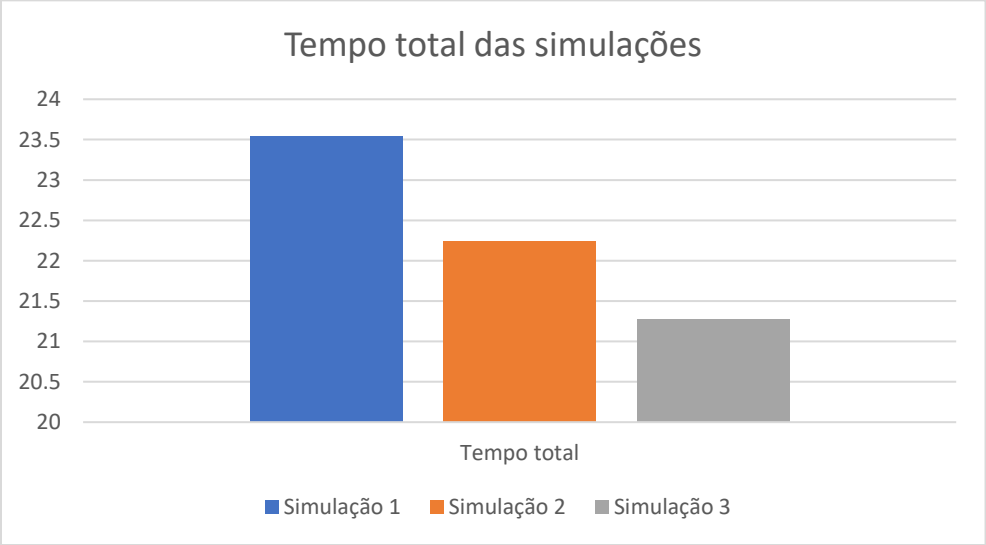

Figura 15 Gráfico dos tempos das simulações

Além do tempo total da simulação, também foi coletado o tempo individual de cada robô, com estes dados podemos verificar se algum dos robôs perdeu um dos seus deadlines e como cada um afetou o tempo total da simulação.

Na Figura 8 é possível ver o gráfico que demonstra os tempos de cada robô em cada simulação. Para o robô 1, os tempos de execução foram de 23.54 segundos na primeira simulação, que foi além do deadline estipulado, 20.43 segundos na segunda simulação e por fim 20.14 segundos na terceira simulação.

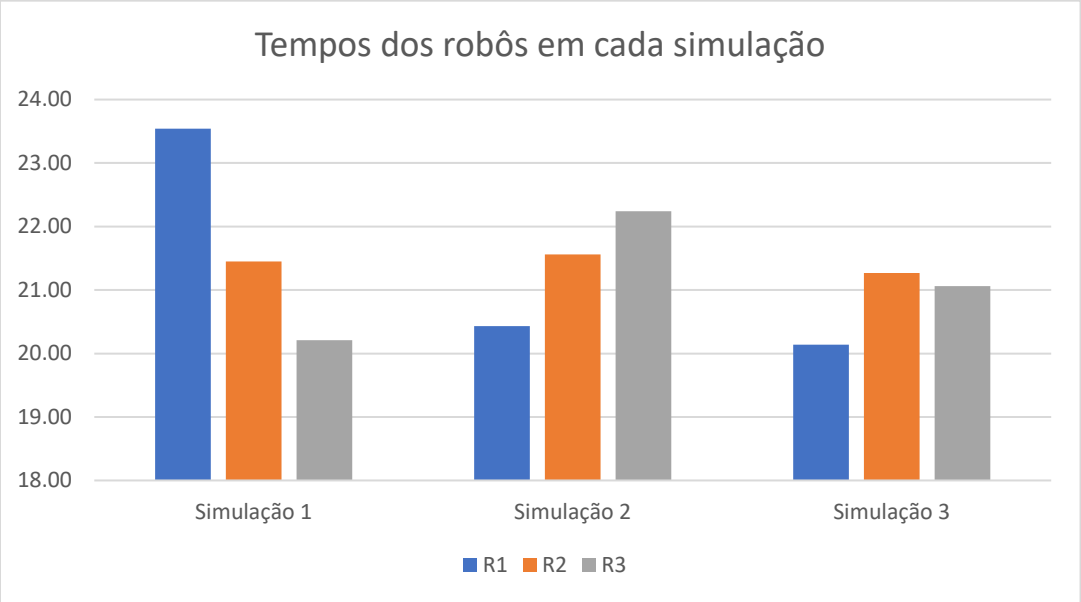

Figura 16 Gráfico dos tempos dos robôs

Na tabela a abaixo pode-se ver os valores individuais de cada robô para finalizar as suas respectivas tarefas, junto com os desvios padrão de cada simulação executada.

Duração	Simulação 1	Simulação 2	Simulação 3
Robô 1	23.54	20.43	20.14
Robô 2	21.45	21.56	21.27
Robô 3	20.21	22.24	21.06
Desvio Padrão	1.68298	0.91428	0.60103
Total	23.54	22.24	21.27
Média	21.73333	21.41	20.82333

Figura 17 Tabela de dados das simulações

Com os resultados obtidos aqui, pode-se notar que os valores obtidos na simulação 1 são bastante discrepantes, possuindo um alto desvio padrão além do maior tempo de simulação total,

portanto, pode-se considerar que o método de solução definido na simulação 1 não seja a mais eficiente para o problema apresentado.

No entanto, tanto a simulação 2 e a simulação 3 demonstraram tempos totais menores em comparação a simulação 1 e com uma menor divergência entre os valores individuais dos robôs. Dentre ambas as simulações, a simulação 3 foi a que obteve os melhores resultados, com o menor tempo total de execução e o menor desvio padrão entre os tempos dos robôs e, portanto, pode-se determinar que foi a melhor solução para o cenário apresentado dentre as simulações executadas.

7. Considerações finais

O propósito deste trabalho de conclusão de curso foi o desenvolvimento de um Simulador para múltiplos robôs de transporte utilizando o framework Cocos2d-x que foi desenvolvido com sucesso e demonstrado neste documento. Foram utilizadas funções do framework para estabelecer comunicação entre diferentes módulos a fim de manter um baixo acoplamento e alta coesão com o objetivo de facilitar a adição de novos módulos no sistema. Foi também adicionada a funcionalidade que permite que o usuário defina o comportamento do robô. Foram desenvolvidos múltiplos sensores que podem ser habilitados ou desabilitados a medida da necessidade do usuário. Por fim todo o código deste projeto está disponível de forma aberta na plataforma Github caso seja necessária modificação ou adição de novas funcionalidades.

Como prova do funcionamento do simulador foi feita a simulação de um cenário de um armazém de produtos, testando três diferentes comportamentos e definido qual deles foi o mais eficiente em completar a tarefa definida.

Referências

JIA, M. J. M.; ZHOU, G. Z. G.; CHEN, Z. C. Z. An efficient strategy integrating grid and topological information for robot exploration. IEEE Conference on Robotics, Automation and Mechatronics, 2004., v. 2, p. 1–3, 2004.

NILSON, P.; HAESAERT, S.; THAKKER, R.; Otsu, K.; VASILE, C.; AGHA-MOHAMMADI, A.; MURRAY, R. M.; AMES, A. D. Toward Specification-Guided Active Mars Exploration for Cooperative Robot Teams. Disponível em: <http://roboticsproceedings.org/rss14/p47.pdf>.

MARKS, M. ROBOTS IN SPACE: SHARING OUR WORLD WITH AUTONOMOUS DELIVERY VEHICLES. We Robot, 2019. Disponível em: <https://robots.law.miami.edu/2019/wp-content/uploads/2019/03/Mason-Marks-Robots-in-Space-WeRobot-2019-3-14.pdf>.

BREGA, R.; TOMATIS, N.; ARRAS, K. O. The Need for Autonomy and Real-Time in Mobile Robotics: A Case Study of XO/2 and Pygmalion. International Conference on Intelligent

Robots and Systems. Disponível em: <http://www2.informatik.uni-freiburg.de/~arras/papers/bregaIROS00.pdf>.

YAN, Z.; FRABRESSE, L.; LAVAL, J.; BOURAQADI, N. Building a ROS-Based Testbed for Realistic Multi-Robot Simulation: Taking the Exploration as an Example. Setembro, 2017.

PINCIROLI, C.; TRIANNI, V.; O'GRADY, R.; PINI, G.; BRUTSCHY, A.; BRAMBLIA, M.; MATTHEWS, N.; FERRANTE, E.; DI CARO, G.; DUCATELLE, F.; STIRLING, T.; GUTIÉRREZ, A.; GAMBARDELLA, L. M.; DORIGO, M. ARGoS: A Modular, Multi-Engine Simulator for Heterogeneous Swarm Robotics. IEEE/RSJ International Conference on Intelligent Robots and Systems September 25-30, p. 5027-5034

,2011. San Francisco, CA, USA.

MIURA, J.; UOZUMI, H.; SHIRAI, Y. Mobile robot motion planning considering the motion uncertainty of moving obstacles. IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics, v. 4, p. 692–697, 1999. ISSN 1062-922X.

TSUBOUCHI, T.; HIROSE, a.; ARIMOTO, S. A navigation scheme with learning for a mobile robot among multiple moving obstacles. Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '93), v. 3, n. C, p. 2234–2240, 1993.

GUERRERO, J.; OLIVER, G. Multi-robot coalition formation in real-time scenarios. Robotics and Autonomous Systems, Elsevier B.V., v. 60, n. 10, p. 1295–1307, 2012. ISSN 09218890. Disponível em: <http://dx.doi.org/10.1016/j.robot.2012.06.004>.

CARPIN, S.; PAVONE, M.; SADLER, B. M. Rapid Multirobot Deployment with Time Constraints. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS. Chicago, IL, USA: [s.n.], 2014. p. 1147–1154. ISBN 9781479969340.