Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Menor Ideia

Giovanni Meneguette Guidini

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.a Dr.a Genaína Rodrigues

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Menor Ideia

Giovanni Meneguette Guidini

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.a Dr.a Genaína Rodrigues (Orientadora)
CIC/UnB

Prof. Dr. Donald Knuth
Stanford University

Dr. Leslie Lamport
Microsoft Research

Dr.a Gabriel Rodrigues
Coordenador do Bacharelado em Ciência da Computação

Brasília, 22 de maio de 2021

Dedicatória

Na *dedicatória* o autor presta homenagem a alguma pessoa (ou grupo de pessoas) que têm significado especial na vida pessoal ou profissional. Por exemplo (e citando o poeta): *Eu dedico essa música a primeira garota que tá sentada ali na fila. Brigado!*

Agradecimentos

Nos *agradecimentos*, o autor se dirige a pessoas ou instituições que contribuíram para elaboração do trabalho apresentado. Por exemplo: *Agradeço aos gigantes cujos ombros me permitiram enxergar mais longe. E a Google e Wikipédia.*

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

O *resumo* é um texto inaugural para quem quer conhecer o trabalho, deve conter uma breve descrição de todo o trabalho (apenas um parágrafo). Portanto, só deve ser escrito após o texto estar pronto. Não é uma coletânea de frases recortadas do trabalho, mas uma apresentação concisa dos pontos relevantes, de modo que o leitor tenha uma ideia completa do que lhe espera. Uma sugestão é que seja composto por quatro pontos: 1) o que está sendo proposto, 2) qual o mérito da proposta, 3) como a proposta foi avaliada/validada, 4) quais as possibilidades para trabalhos futuros. É seguido de (geralmente) três palavras-chave que devem indicar claramente a que se refere o seu trabalho. Por exemplo: *Este trabalho apresenta informações úteis a produção de trabalhos científicos para descrever e exemplificar como utilizar a classe LaTeX do Departamento de Ciência da Computação da Universidade de Brasília para gerar documentos. A classe* `UnB-CIC` *define um padrão de formato para textos do CIC, facilitando a geração de textos e permitindo que os autores foquem apenas no conteúdo. O formato foi aprovado pelos professores do Departamento e utilizado para gerar este documento. Melhorias futuras incluem manutenção contínua da classe e aprimoramento do texto explicativo.*

Palavras-chave: LaTeX, metodologia científica, trabalho de conclusão de curso

Abstract

O *abstract* é o resumo feito na língua Inglesa. Embora o conteúdo apresentado deva ser o mesmo, este texto não deve ser a tradução literal de cada palavra ou frase do resumo, muito menos feito em um tradutor automático. É uma língua diferente e o texto deveria ser escrito de acordo com suas nuances (aproveite para ler `http://dx.doi.org/10.6061%2Fclinics%2F2014(03)01`). Por exemplo: *This work presents useful information on how to create a scientific text to describe and provide examples of how to use the Computer Science Department's LaTeX class. The `UnB-CIC` class defines a standard format for texts, simplifying the process of generating CIC documents and enabling authors to focus only on content. The standard was approved by the Department's professors and used to create this document. Future work includes continued support for the class and improvements on the explanatory text.*

Keywords: LaTeX, scientific method, thesis

Sumário

Lista de Tabelas

Capítulo 1

Referencial Teórico

O referencial teórico apresenta conceitos importantes para o projeto e trabalhos relacionados encontrados na literatura. A Seção 1.1 descreve um breve levantamento feito de alguns simuladores bem estabelecidos na literatura. Esses projetos foram usados de inspiração para a criação do HMR Sim. A Seção 1.2 apresenta a *design pattern* ECS (*Entity-Compoenent-System*), analisada por ser bastante utilizada na criação de jogos, pela complexidade dos jogos de videogame atuais e como estes são de certa forma simuladores. Esta arquitetura foi utilizada no projeto do HMR Sim pelas suas vantagens. Finalmente a Seção 1.3 comenta sobre duas técnicas de simulação encontradas nos simuladores que foram levantados e na literatura.

1.1 Simuladores na Literatura

O levantamento da literatura iniciou com um breve levantamento de simuladores já estabelecidos para sistemas robóticos. Foram selecionados os simuladores Gazebo [1], Simbad [2], CoppeliaSim [3], MORSE [4] e Dragonfly [5]. Uma descrição breve de como os robôs são definidos em cada simulador é incluída, junto com um link para a documentação oficial do projeto.

Cada um desses simuladores foi criado com propostas diferentes, desde simulação precisa das partes que compõem um robô e sua interação com o ambiente (Gazebo, CoppeliaSim, Morse), até simulações de mais alto nível focando principalmente no comportamento dos robôs (Simbad, Dragonfly). Simulações multi-robô são suportadas por simuladores atuais, mas geralmente em menor número - devido ao alto uso de recursos computacionais necessários para simular cada robô (i. e. experimentos feitos com Gazebo mostraram que o simulador tem dificuldades ao simular mais de 10 robôs [6]) - ou são muito específicos quando conseguem simular mais robôs (i.e. Dragonfly supostamente é capaz de simular até 400 entidades, mas só consegue simular drones [5]).

Gazebo. Robôs são definidos em modelos, que seguem uma estrutura de arquivos definida. O robô em si é descrito em arquivos .sdf. Cada modelo é definido através de uma série de <links> que definem as partes do modelo. Sensores ou outros componentes - outros modelos - podem ser ligados através de <joints>. Modelos podem ter plug-ins com funcionalidade extra. O projeto é open source e pode acessado no link `http://gazebosim.org`.

CoppeliaSim. Modelos são definidos como uma seleção de scene objects (e.g. joints, shapes, sensors, cameras, paths, etc). Existem muitas maneiras de controlar uma simulação, dentre elas destaca-se embedded scripts. Esse scripts podem ser definidos como parte de um scene object e executam alguma funcionalidade relacionada à este objeto. CoppeliaSim (antigamente V-Rep) é uma solução comercial da empresa Coppelia Robotics, disponível no link `https://www.coppeliarobotics.com/features`.

MORSE. Robôs são plataformas que definem o formato e certas propriedades, como área de colisão massa, etc. É nessas plataformas que sensores e atuadores são montados. Estes são fornecidos pelo simulador MORSE para serem adicionados à robôs. Apenas sensores e atuadores interagem com o mundo real com alguma funcionalidade. Os sensores e atuadores são fornecidos em diversos níveis de realismo, permitindo maior ou menor grau de abstração na simulação. MORSE é uma solução open source, baseado no software de modelagem Blender. Infelizmente o projeto se encontra abandonado desde 2020, mas ainda está disponível no link `https://morse-simulator.github.io`.

Simbad. Robôs são definidos em classes Java que estendem a classe `Agent`. Sensores são adicionados como atributos da classe. Status e movimentação do robô é alcançado através de APIs próprias. Robôs implementam as funções `initBehavior()` e `performBehavior()`, que definem o que acontece com o robô ao ser criado e o comportamento dele em cada loop de simulação. Um projeto open source disponível no link `http://simbad.sourceforge.net`.

Dragonfly. Drones possuem uma classe de controle - `DroneKeyboardController` ou `DroneAutomaticController`, respectivamente para ser controlado pelo usuário ou automaticamente. E classes que definem seus comportamentos, através de modelos. Além disso outras configurações, como nível de bateria, consumo por bloco, alvo e os wrappers (para fornecer comportamento adaptativo) podem ser alterados pela interface gráfica, para cada drone na cena. Esse simulador está limitado à simuações de drones. Também open source, disponível em `https://github.com/DragonflyDrone/Dragonfly`.

Cada simulador possui características arquiteturais e objetivos próprios que foram analisados e comparados, fornecendo um arcabouço de técnicas que podem ser utilizadas (ver Tabela 1.1). Pontos relevantes que foram investigados sobre os simuladores incluem:

Simulador	Nível de Abstração	Nº de robôs	Genérico	Arquitetura	Tipo de Simulação
Gazebo	Baixo	<20	SIM	Declarativa	Passos/DES
CoppeliaSim	Baixo	<20	SIM	Declarativa	Passos/DES
Simbad	Médio	<10	SIM	OOP	Passos
MORSE	Médio/Alto	20 - 100	SIM	OOP/Declarativa	Passos
Dragonfly	Alto	400	NÃO	MVC/AOP	Passos

Tabela 1.1: Comparação resumida das ferramentas com suporte de simulação multi-robôs.

- **Nível de Abstração.** Pode ser *baixo* indicando grande detalhamento dos componentes que compõe o robô e suas características físicas; *médio* indicando necessidade de detalhamento dos movimentos individuais dos componentes que formam um robô; *alto* indicando abstração dos componentes do robô.
- **Número de robôs** que o simulador é capaz de simular num tempo razoável
- Se o simulador é **genérico** ou não, ou seja, se existe restrição no tipo de robôs que o simulador é capaz de simular.
- **Arquitetura** utilizada na representação do robô (i.e. um robô é uma classe que deve ser implementada, ou um arquivo XML, etc)
- **Tipo de simulação** indica qual a técnica de simulação usada, em passos ou de eventos discretos (ver Seção 1.3)

1.2 Entity-Component-System

Entity-Component-System (ECS) é um padrão de desenho (*design pattern*) de software amplamente utilizada em jogos, tipicamente em sistemas interativos em tempo-real (e.g. jogos do tipo MMO, *Massive Multiplayer Online*) [7]. Nesse padrão, objetos da simulação são transformados em *entidades*. Cada entidade nada mais é que uma coleção de *componentes*. Um componente, por sua vez, armazena dados, mas tipicamente não implementa nenhuma lógica.

A lógica da simulação está nos *sistemas*, que modificam os dados de componentes de acordo com seu objetivo. Cada sistema age de maneira independente de outros sistemas sobre um conjunto de componentes que lhe interessa, ou seja, se uma entidade possui esse conjunto de componentes, então ela será afetada pelo sistema durante a simulação. O estado da simulação é o conjunto de estados de todos os componentes de todas as entidades

presentes na simualção. Ele é alterado apenas pelos sistemas, cada um alterando uma pequena parte desse estado global.

Essa organização permite grande modularização e separação de lógica entre as difrentes partes do sistema. Cada sistema (ou conjunto de sistemas) e seu conjunto de componentes associados pode ser adicionado ou remivdo do simulador conforme necessário. Por exemplo, um sistema comunicação entre diferentes robôs pode ser implementado como um componente que guarde uma fila de mensagens e pode ser adicionado à cada robô, associado à dois sistemas: um sistema que faça a entrega das mensagens de um robô para o outro, e outro sistema que processa as mensagens de cada robô. Note que se o processamento não for adequado à uma simulação, basta trocar aquele sistema por outro que seja adequado. Além disso, se alguma simulação não faz uso desse sistema de mensagens, basta removê-lo do simulador completamente, deixando a simulação mais leve.

Uma outra vantagem de utilizar o padrão ECS é a flexibilidade de adicionar ou remover capacidades das entidades durante a execução da simulação. Como cada entidade é simplesmente uma coleção de componentes, é possível associar certas capacidades dos robôs (e.g. sensores, atuadores) à presença ou ausência de certos componentes naquela entidade. Por exemplo, dada a existência de um componente `camera` e um sistema associado que simule a captura de imagens, qualquer entidade que possue esse componente vai possuir a capacidade de coletar imagens via componente camera. Além disso, al simular falhas catastróficas em componentes, basta remover o componente da entidade sendo analisada.

Apesar dessas vantagens, como apontado por Wiebush [7], o uso de ECS pode trazer complicações de compatibilidade entre sistemas desenvolvidos de maneira indepentente, como uso de componentes incompatíveis, e dificuldade em conhecer qual sistema é responsável por determinada funcionalidade e como utilizá-la. detalhes de como esses problemas foram sentidos durante o desenvolvimento do projeto e medidas tomadas para mitigá-los são discutidas no capítulo 2.

Foi utilizada a biblioteca `esper` para suporte do padrão ECS. Esper é uma biblioteca de ECS leve com foco em performance, escrita na linguagem Python por Benjamin Moran [8]. Ela cria uma classe `World` que mantém uma lista de entidades e de todos os componentes para cada entidade. Um componente pode ser qualquer estrutura em Python, no caso do projeto foram usadas classes (i.e. `class`). É possível ainda adicionar sistemas à classe `World`, que são implementados como funções, convencionalmente chamadas `process`. Alguns dos sistemas do projeto são adicionados ao `World`.

1.3 Técnicas de Simulação

Uma técnica de simulação bem estabelecida é a de tempo discreto com intervalo de incremento fixo [9]. Nesse modelo, o estado de um sistema no tempo t_{i+1} é uma função do estado do sistema no tempo t_i. Cada variável que compõe o estado do sistema é uma função de variáveis e estados até o momento anterior. O incremento de tempo da simulação entre t_i e t_{i+1} é sempre o mesmo, e pré-definido.

Se o tempo t_{calc} necessário para computar o estado t_{i+1} do sistema a partir do estado t_i é menor do que o tempo do incremento t_{incr}, então a simulação será computada mais rápido do que o tempo do relógio (e.g. o tempo real); da mesma forma, se $t_{calc} > t_{incr}$, então a simulação é computada mais devagar do que o tempo do relógio. Essas situações são conhecidas como simulação *offline* [9], porque não há sincronia entre o tempo da simulação e o tempo do relógio. Essa é uma situação aceitável para este projeto, onde o objetivo é obter a simulação desejada no menor tempo possível.

Essa técnica de simulação é indicada para simular sistemas que mudam constantemente, como por exemplo a temperatura de um ambiente ao longo do tempo, ou um sinal recebido por um sensor que trabalha a uma frequência conhecida. No entanto o "relógio"da simulação é sincronizado, e todas as funções do estado são processadas a cada incremento de tempo, o que pode levar a cálculos desnecessários. Por exemplo, em uma simulação que involva uma função que altera temperatura de uma sala a cada 200ms, e um sensor que registra a temperatura da mesma sala com leituras a cada 100ms, a função que altera temperatura deve ser executada em todos os incrementos de tempo, que devem ser no máximo 100ms para suportar a leitura do sensor. Nesse cenário, metade das chamadas à função de alterar temperatura não afeta o estado do sistema, mas ainda tem que ser processadas.

Outra técnica de simulação é por eventos discretos (DES, *Discrete Event Simulation*) [10]. Nesse modelo uma fila de eventos é processado um por vez, e cada estado s_{i+1} é o resultado de processar o evento no topo da fila sobre o estado s_i. Um evento e possui um tempo t e uma função f que altera o estado, e potencialmente cria outros eventos, que serão adicionados à fila. A fila de eventos é uma fila de prioridades ordenada pelo tempo t de cada evento, sendo que o tempo de cada novo evento gerado pode ser igual ou maior que o tempo do evento que o gerou (nunca menor, porque não se pode alterar o passado da simulação). O tempo da simulação corresponde ao tempo t do evento atual sendo processado, e como os eventos são ordenados pelo tempo, ele só será incrementado quando todos os eventos naquele tempo foram processados.

Diferentemente da simulação com intervalo de incremento fixo, onde as mesmas funções são executadas em intervalos conhecidos de tempo, na simulação do tipo DES funções diferentes alteram o estado da simulação, e o tempo da simulação no estado $s_i + 1$ não

depende apenas do estado s, mas também do evento sendo processado. Esse novo tempo pode não crescer de maneira uniforme ao longo da simulação. Essa técnica é adequada para simular sistemas que mudem de maneira infrequênte ao longo do tempo, por exemplo o inventário de um armazém [9], ou a operação de robôs de serviço dentro do armazém.

A simulação de incremento fixo de tempo pode ser implementada utilizando a técnica de eventos discretos, desde que os eventos sejam criados com tempos que possuam um intervalo constante. Uma outra característica interessante que pode ser alcançada com eventos discretos é separar a função em subsistemas que são executados de maneira independente e assíncrona. Retomando o exemplo da sala que muda de temperatura e possui o sensor, cada evento de leitura do sensor pode criar o próximo evento de leitura para o tempo $t + 100ms$; de forma similar cada evento de mudança de temperatura cria um novo evento de mudança para o tempo $t + 200ms$. Dessa forma, evita-se o problema de funções de alteração do estado da simulação tendo que ser executadas antes da hora.

O simulador utiliza a técnica de simulação de eventos discretos, através da biblioteca `simpy` [11]. Esse framework de simulação DES é baseado em processos e faz todo o gerenciamento dos eventos e sua execução. A simulação acontece dentro de um ambiente, onde diversos processos interagem entre si e com o ambiente através de eventos. QUalquer função geradora em Python pode ser um process no simpy.

Esse framework também tem suporte para recursos (*Resources*), que são compartilhados entre os processos. Recursos podem simular desde recursos a serem disputados (i.e. uma impressora, uma estação de carga) até recursos que são armazenados em contâiners (i.e. 10L de aguá de um reservatório com capacidade para 10000L). Recursos podem ainda ser preemptivos, ou filtrados de algum contâiner. Essa última capacidade foi bastante utilizada para a comunicação entre sistemas do simulador, como será discutido no Capítulo 2.

Capítulo 2

O simulador HMR Sim

Referências

[1] Koenig, Nathan e Andrew Howard: *Design and use paradigms for gazebo, an open-source multi-robot simulator*. Em *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, páginas 2149–2154. IEEE, 2004. 1

[2] Hugues, Louis e Nicolas Bredeche: *Simbad: an autonomous robot simulation package for education and research*. Em *International Conference on Simulation of Adaptive Behavior*, páginas 831–842. Springer, 2006. 1

[3] Rohmer, Eric, Surya PN Singh e Marc Freese: *V-rep: A versatile and scalable robot simulation framework*. Em *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, páginas 1321–1326. IEEE, 2013. 1

[4] Echeverria, Gilberto, Nicolas Lassabe, Arnaud Degroote e Séverin Lemaignan: *Modular open robots simulation engine: Morse*. Em *2011 IEEE International Conference on Robotics and Automation*, páginas 46–51. IEEE, 2011. 1

[5] Maia, Paulo Henrique, Lucas Vieira, Matheus Chagas, Yijun Yu, Andrea Zisman e Bashar Nuseibeh: *Dragonfly: a tool for simulating self-adaptive drone behaviours*. Em *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, páginas 107–113. IEEE, 2019. 1

[6] Noori, Farzan M, David Portugal, Rui P Rocha e Micael S Couceiro: *On 3d simulators for multi-robot systems in ros: Morse or gazebo?* Em *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, páginas 19–24. IEEE, 2017. 1

[7] Wiebusch, Dennis e Marc Erich Latoschik: *Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems*. Em *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, páginas 25–32. IEEE, 2015. 3, 4

[8] Moran, Benjamin: *Esper*. `https://github.com/benmoran56/esper/commit/850ee6365dcdc6c36c7b02053bef121a98042849`, maio 2020. 4

[9] Bélanger, Jean, P Venne e Jean Nicolas Paquin: *The what, where and why of real-time simulation*. Planet Rt, 1(1):25–29, 2010. 5, 6

[10] Matloff, Norm: *Introduction to discrete-event simulation and the simpy language*. Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August, 2(2009):1–33, 2008. 5

[11] Lünsdorf, Ontje e Stefan Scherfke: *Simpy*. `https://gitlab.com/team-simpy/simpy/-/commit/b29e72af9975aae6cc0f2ffeaff0c5c876c9e644`, abril 2020. 6