UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

ISABELLA NOVAES ALVES

O USO DE DIAGRAMAS PARA PROJETO DE JOGOS DE COMPUTADOR

PONTA GROSSA

2025

ISABELLA NOVAES ALVES

O USO DE DIAGRAMAS PARA PROJETO DE JOGOS DE COMPUTADOR

The use of diagrams in computer game design

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação da Universidade Tecnológica Federal do Paraná (UTFPR).
Orientador(a): Prof. Dr. André Koscianski.

PONTA GROSSA

2025

ISABELLA NOVAES ALVES

O USO DE DIAGRAMAS PARA PROJETO DE JOGOS DE COMPUTADOR

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 31/outubro/2025

André Koscianski
Doutor
Universidade Tecnológica Federal do Paraná – campus Ponta Grossa

---

Erikson Freitas de Morais
Doutor
Universidade Tecnológica Federal do Paraná – campus Ponta Grossa

---

Gleifer Vaz Alves
Doutor
Universidade Tecnológica Federal do Paraná – campus Ponta Grossa

PONTA GROSSA

2025

Dedico este trabalho à minha família, pelos momentos de ausência, distância e apoio, minha melhor amiga e meu namorado que me acompanharam nessa fase.

AGRADECIMENTOS

Tenho consciência que estes parágrafos não irão abranger a todas as pessoas que fizeram parte dessa importante fase de minha vida. Portanto, desde já peço desculpas àquelas que não estão presentes entre essas palavras, mas elas podem ter certeza que fazem parte do meu pensamento e de minha gratidão.

Agradeço a todos os meus familiares dos quais fiquei tão distante durante este tempo de graduação, que de forma mais específica minha mãe Monica Roberta Novaes Alves que sempre me incentivou mesmo nos piores momentos, meu pai Aparecido Roberto Alves que sempre foi um exemplo de dedicação para mim e aos meus irmãos Enzo e Roberto que foram parceiros de risos quando eu precisei. A minha tia Alessandra Melissa Novaes por todo incentivo e confiança que depositou em mim.

Agradeço em especial ao meu orientador Prof. Dr. André Koscianski , pela paciência, auxílio, direcionamento e toda sabedoria com que me guiou durante toda esta jornada, que não seria finalizada sem sua ajuda.

Não posso deixar de agradecer aqui a todos os meus amigos que me acompanharam, alguns desde o início da minha trajetória e outros que foram se juntando no decorrer desses anos, em específico minha melhor amiga Giulia Pedraja de Oliviera que foi uma peça fundamental na minha vida para eu estar onde estou nesse momento, obrigada por todo apoio mesmo com as dificuldades que apareceram.

Agradeço meu namorado Gustavo Andrade Andras que mesmo chegando mais para o final da minha caminhada, nunca deixou de acreditar em mim ou de me incentivar, sempre estando ao meu lado e me apoiando.

Enfim, a todos os que por algum motivo contribuíram para a realização desta pesquisa.

RESUMO

O desenvolvimento de jogos digitais é um processo complexo que integra design, programação e *storytelling*. Nesse contexto, diagramas são essenciais para documentar, discutir e validar comportamentos: permitem visualizar estados e decisões (FSM/BT) e modelar fluxos de recursos e progressão (Machinations), agilizando comunicação entre áreas, análise e balanceamento. A Engenharia Orientada a Modelos (Model-Driven Engineering - MDE) aproveita esses artefatos como núcleo do processo, apoiando automação e reuso. Este trabalho implementa um editor gráfico para a notação Machinations, com biblioteca de símbolos essenciais, conexões com linha contínua e pontilhada, subdiagramas via "*Select Mode*", rótulos por item e persistência em XML e JSON com preservação de propriedades visuais. A validação visual incluiu a reprodução de um diagrama de referência lado a lado com o original, demonstrando fidelidade de formas, estilos e semântica. A solução foi validada no Windows e Qt e está orientada à interoperabilidade, preparando integrações futuras, assim, a proposta busca facilitar a criação e manipulação de diagramas, contribuindo para a aplicação da MDE no design de jogos e possibilitando avanços na área.

Palavras-chave: engenharia orientada a modelos; jogos digitais; diagramas Machinations; desenvolvimento de software.

ABSTRACT

The development of digital games is a complex process that integrates design, programming, and storytelling. In this context, diagrams are essential to document, discuss, and validate behaviors: they enable the visualization of states and decisions (FSM/BT) and the modeling of resource flows and progression (Machinations), streamlining cross-team communication, analysis, and balancing. Model-Driven Engineering (MDE) leverages these artifacts as the core of the process, supporting automation and reuse. This work implements a graphical editor for the Machinations notation, featuring a library of essential symbols, connections with solid and dashed lines, sub-diagrams via “Select Mode,” per-item labels, and persistence in XML and JSON with preservation of visual properties. Visual validation included reproducing a reference diagram side by side with the original, demonstrating fidelity of shapes, styles, and semantics. The solution was validated on Windows with Qt and is oriented toward interoperability, paving the way for future integrations. Thus, the proposal aims to facilitate the creation and manipulation of diagrams, contributing to the application of MDE in game design and enabling advances in the field.

Keywords: Model-Driven Engineering; digital games; Machinations diagrams; software development.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

BT	*Behavior Tree* (Árvore de Comportamento)
CIM	*Computation Independent Model* (Modelo Independente de Computação)
DOM	*Document Object Model* (Modelo de Documento do Objeto)
EMF	*Eclipse Modeling Framework* (Framework de Modelagem do Eclipse)
FSM	*Finite State Machine* (Máquina de Estados Finita)
HFSM	*Hierarchical Finite State Machine* (Máquina de Estados Hierárquica)
I/O	*Input/Output* (Entrada/Saída)
JSON	*JavaScript Object Notation* (Notação de Objetos JavaScript)
MDA	*Model-Driven Architecture* (Arquitetura Orientada a Modelos)
MDE	*Model-Driven Engineering* (Engenharia Orientada a Modelos)
NPC	*Non-Player Character* (Personagem não jogável)
OMG	*Object Management Group* (Grupo de Gestão de Objetos)
PIM	*Platform Independent Model* (Modelo Independente de Plataforma)
PSM	*Platform Specific Model* (Modelo Específico de Plataforma)
Qt	Qt Framework
UML	*Unified Modeling Language* (Linguagem de Modelagem Unificada)
XSD	*XML Schema Definition* (Definição na linguagem XML Schema)
XSLT	*Extensible Stylesheet Language Transformations*
XML	*Extensible Markup Language* (Linguagem de Marcação Extensível)

SUMÁRIO

1 INTRODUÇÃO

Os jogos digitais têm se consolidado como uma das formas mais populares de entretenimento, abrangendo uma vasta gama de gêneros e plataformas, desde consoles e computadores até dispositivos móveis e realidade virtual. A indústria de jogos digitais não apenas gera bilhões de dólares em receita anualmente, mas também influencia a cultura popular, molda tendências tecnológicas e oferece novas formas de interação social (ALBAGHAJATI; AHMED, 2023). O desenvolvimento de jogos digitais é uma tarefa complexa que envolve diversas disciplinas, incluindo *design* gráfico, programação, música e *storytelling*. Este campo em constante evolução exige o uso de práticas, metodologias e ferramentas sofisticadas para criar experiências envolventes e interativas (DORMANS, 2012).

Dentro deste contexto, a *Model-Driven Engineering* (MDE), ou Engenharia Orientada a Modelos, oferece uma abordagem sistemática para o desenvolvimento de *software*, que se baseia na utilização de modelos para projetar e desenvolver sistemas de *software*, usados para representar diferentes aspectos do sistema em questão. Segundo Akiki (2018), a ideia é que os modelos capturem aspectos importantes do sistema, desde os requisitos até a implementação, e que esses modelos possam ser manipulados e transformados automaticamente para produzir código, documentação e outros artefatos do sistema.

Dessa forma, em jogos de computador, a aplicação da MDE pode proporcionar uma estrutura sólida para o *design* e desenvolvimento, permitindo aos desenvolvedores visualizarem e manipularem modelos de maneira mais eficiente. Esses modelos serão utilizados como abstrações de alto nível que representam implementações de código-fonte de vários conteúdos, como estágios do jogo, veículos ou entidades inimigas (por exemplo, chefes finais) (BLASCO *et al.*, 2021). Além disso, a MDE pode promover a reutilização de componentes, a automação de processos e a integração de sistemas, contribuindo assim para a eficácia e qualidade dos jogos (BUCCHIARONE; CICCHETTI; MARCONI, 2019).

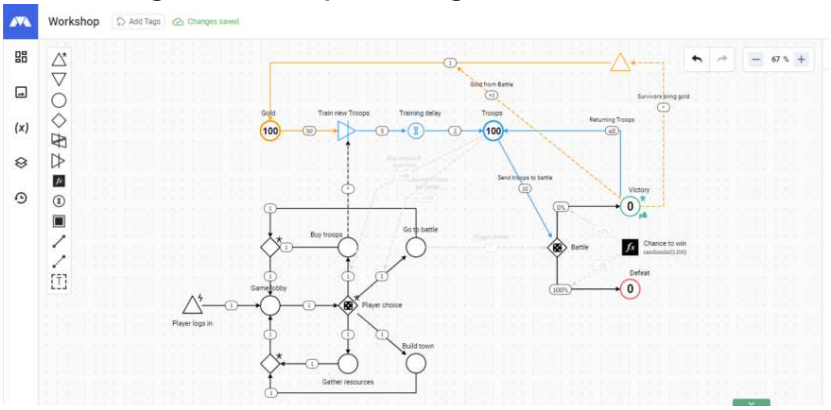
Há outras ferramentas que podem auxiliar nesse processo de projetar jogos digitais, como redes de Petri (FRITZ; KREBS; ZHANG, 2020), Árvores de Comportamento (ZHANG *et al.*, 2018) e Máquinas de Estado (IFTIKHAR *et al.*, 2015), pois elas podem descrever e controlar o comportamento de sistemas complexos, como os encontrados em jogos de computador.

Uma ferramenta específica para projetar e balancear sistemas de jogos é o diagrama chamado Machinations. Esta ferramenta foi desenvolvida por Joris Dormans, que oferece uma abordagem formal para essa perspectiva no design de jogos, sendo criada para auxiliar projetistas e jogos na concepção, documentação, simulação e teste das dinâmicas de economia. O Machinations funciona como um fluxograma em movimento ou mapa mental dos jogos, composto por diagramas que podem ser simulados em tempo real (ERNEST; DORMANS, 2012).

A ferramenta é uma linguagem visual e um ambiente de simulação voltados à modelagem de mecânicas e economias de jogos. Sua notação descreve fluxos de recursos, estruturas de progressão e ciclos de feedback de modo padronizado, por meio de símbolos e conexões. Esse formalismo permite documentar, discutir e validar comportamentos antes da implementação, favorecendo o balanceamento e a iteratividade (ERNEST; DORMANS, 2012). No contexto deste trabalho, o Machinations funciona como modelo central para edição e persistência (XML/JSON), alinhado à perspectiva de Engenharia Orientada a Modelos (MDE).

A Figura 1 apresenta um exemplo de diagrama Machinations que ilustra de forma visual como fluxos de recursos e condições de jogo podem ser representados para análise e balanceamento. Essa ilustração reforça a explicação dada no texto ao mostrar, na prática, como elementos de geração, consumo e redistribuição de recursos interagem dinamicamente, evidenciando a utilidade dessa notação para projetistas de jogos digitais.

Figura 1 - Exemplo de diagrama Machinations



Fonte: Machinations Documentation (2024)

Há um editor desses diagramas baseado em navegador, que embora permita uso gratuito, ainda é um *software* proprietário. Dessa forma, esse trabalho tem como proposta desenvolver um editor gráfico de código aberto para uma parte dos diagramas Machinations.

Além disso, para atender a interoperabilidade e manutenção do editor, adotou-se XML e JSON como formatos nativos de leitura/escrita. O XML favorece inspeção hierárquica, versionamento e validação, já o JSON privilegia a compacidade e integração com *pipelines web.*

O editor implementado neste trabalho foi construído sobre o Jade utilizando linguagem C++ e Qt6, que é uma base pré-existente de editor gráfico. Neste trabalho, foram desenvolvidas uma biblioteca de símbolos Machinations, conexões, rótulos por item, subdiagramas via "*Select Mode*", persistência nativa em XML e JSON, com leitura e gravação preservando equivalência visual e semântica. Não foi desenvolvido o Jade em si, como canvas, zoom/seleção básicos, infraestrutura de comandos. O presente trabalho estende essa base para a notação Machinations e implementa a camada de persistência XML/JSON.

1.1 Justificativa

Uma das principais vantagens do *Model-Driven Engineering* é o foco na modelagem em vez do desenvolvimento de código, ajudando a reduzir a complexidade do desenvolvimento de software ao permitir que os desenvolvedores trabalhem em um nível de abstração mais alto. Isso facilita a análise de propriedades da aplicação em estágios iniciais do desenvolvimento e permite uma menor preocupação com detalhes de implementação.

Outro ponto relevante é a questão da automação no processo de desenvolvimento. Mecanismos como transformações de modelos, trazem eficiência ao processo de desenvolvimento. A geração automática de código a partir dos modelos economiza tempo e reduz a possibilidade de erros. Um exemplo pode ser observado no trabalho de BLASCO *et al.* (2021), que desenvolveram o *Evolutionary Model Generation* (EMoGen), que apresenta uma abordagem para gerar modelos de *software* comparáveis em qualidade aos modelos criados por desenvolvedores humanos. Os resultados do estudo mostram que a abordagem alcança resultados comparáveis aos criados manualmente pelos desenvolvedores, mas em um tempo muito menor.

Um ponto interessante é observar que, embora a Engenharia de Software de Jogos seja uma área de pesquisa estabelecida, a aplicação de MDE aos jogos ainda é incomum. Isso sugere um espaço significativo para exploração e inovação nessa

interseção entre MDE e desenvolvimento de jogos. Por esse motivo, este trabalho poderá servir de base para futuras pesquisas que tratem desse assunto.

1.2 Objetivos

O objetivo deste trabalho é desenvolver um editor gráfico para um subconjunto definido da notação Machinations, com conexões do tipo *resource* e *state*, suporte a rótulos, subdiagramas e persistência em XML e JSON. Os objetivos específicos são:

- Especificar e implementar os seguintes símbolos Machinations: Source, Drain, Pool, Gate, Converter, Trader, Delay, Queue e EndCondition.
- Criar uma forma de gravar e ler esses diagramas em disco.
- Implementar o editor gráfico.
- Desenvolver mecanismos de subdiagramas reutilizáveis.
- Avaliar a eficácia da ferramenta desenvolvida por meio de testes.

1.3 Organização

Este trabalho está estruturado em cinco capítulos, divididos da seguinte forma: O capítulo 1 introduz os principais conceitos adotados para o desenvolvimento do trabalho, seguido da justificativa e dos objetivos. O capítulo 2 apresenta os principais métodos e ferramentas utilizados no desenvolvimento baseado em modelos, com foco na MDE e sua aplicação em jogos digitais. O capítulo 3 descreve o desenvolvimento do editor gráfico, abordando as técnicas para tradução entre formatos de representação e implementação do sistema. O capítulo 4 traz análises sobre o desempenho e aplicabilidade da ferramenta desenvolvida. Por fim, o capítulo 5 apresenta as conclusões e sugestões para trabalhos futuros

2 MÉTODOS E FERRAMENTAS

O desenvolvimento de jogos digitais pode ser aprimorado pelo uso de metodologias baseadas em modelos, que oferecem uma forma mais sistemática de projetar e implementar sistemas complexos.

Este trabalho tem foco principal na edição visual de diagramas Machinations e na persistência em XML/JSON, leitura/gravação preservando aparência e rótulos. As seções sobre Máquinas de Estado (FSM) e Árvores de Comportamento (BT) cumprem papel contextual, pois apresentam técnicas comuns de modelagem de comportamento em jogos, não implementadas neste trabalho, elas permanecem apenas como referências de contexto, para situar o leitor nas alternativas de modelagem em jogos. Já Machinations é a notação central adotada na ferramenta, e XML/JSON constituem a camada de interoperabilidade do editor. Assim, o capítulo reúne referências do domínio, FSM e BT, e fundamentação direta do que foi entregue, Machinations e persistência.

Esta seção aborda as principais abordagens e ferramentas utilizadas na modelagem de jogos, com ênfase na Engenharia Orientada a Modelos (MDE), diagramas utilizados em jogos e os formatos de intercâmbio de dados.

2.1 Model-Driven Engineering

A Engenharia Orientada a Modelos (MDE) é uma abordagem que enfatiza a criação, transformação e manipulação de modelos como principal forma de desenvolvimento de *software*. Ao contrário das abordagens tradicionais, que exigem uma programação manual extensiva, a MDE permite que os modelos sejam convertidos automaticamente em implementações concretas por meio de transformações sistemáticas (BRAMBILLA; CABOT; WIMMER, 2017).

Os modelos são considerados artefatos centrais no desenvolvimento de *software*, funcionando como abstrações que representam aspectos essenciais. Esses modelos são criados e refinados através de linguagens específicas de modelagem, que permitem uma representação mais intuitiva e de alto nível. A MDE possibilita um fluxo de desenvolvimento mais estruturado, em que transformações automáticas entre modelos podem gerar código-fonte, esquemas de dados e até mesmo documentação técnica (BRAMBILLA; CABOT; WIMMER, 2017).

Uma das vantagens da MDE é sua capacidade de padronização e reutilização de componentes, promovendo maior coerência entre diferentes etapas do desenvolvimento. Essa abordagem permite a redução de erros e inconsistências, já que os modelos passam por processos rigorosos de validação antes de serem convertidos em implementações concretas. Além disso, a MDE possibilita a adaptação dos modelos a diferentes plataformas, garantindo flexibilidade e eficiência no desenvolvimento de *software*, inclusive no contexto de jogos digitais.

Outro aspecto relevante da MDE é sua integração com a *Model-Driven Architecture* (MDA), uma iniciativa proposta pelo *Object Management Group* (OMG) que define padrões para modelagem e transformação de modelos. A MDA estabelece três níveis principais de abstração: *Computation Independent Model* (CIM), *Platform Independent Model* (PIM) e *Platform Specific Model* (PSM), permitindo uma transição gradual do conceito até a implementação final. Isso torna a MDE uma ferramenta poderosa para garantir a consistência do desenvolvimento e a escalabilidade dos projetos.

Além disso, ferramentas como *Eclipse Modeling Framework* (EMF) e *Papyrus* permitem a implementação prática da MDE, fornecendo ambientes robustos para a criação, edição e transformação de modelos. Essas ferramentas possibilitam a automação do desenvolvimento, reduzindo o esforço manual e permitindo que os desenvolvedores se concentrem em aspectos estratégicos do *design* e implementação dos jogos digitais.

2.2 Diagramas utilizados em jogos

Diagramas são instrumentos centrais para representar, discutir e validar o comportamento de sistemas em jogos digitais, pois permitem abstrair a complexidade do código e explicitar estados, regras de transição, fluxos de recursos e estratégias de decisão em notações visuais de fácil inspeção. No contexto de desenvolvimento, essas representações apoiam desde a IA de NPCs, como em Máquinas de Estado e Árvores de Comportamento, até mecânicas sistêmicas, como economias de jogo, progressão e balanceamento, e lógica de *gameplay*, em condições de vitória e derrota, *gating*, *triggers*.

Além de servirem como documentação viva para equipes multidisciplinares, os diagramas funcionam como artefatos intermediários entre *design* e

implementação, viabilizando testes conceituais, reuso de padrões e interoperabilidade com ferramentas de simulação e geração de código.

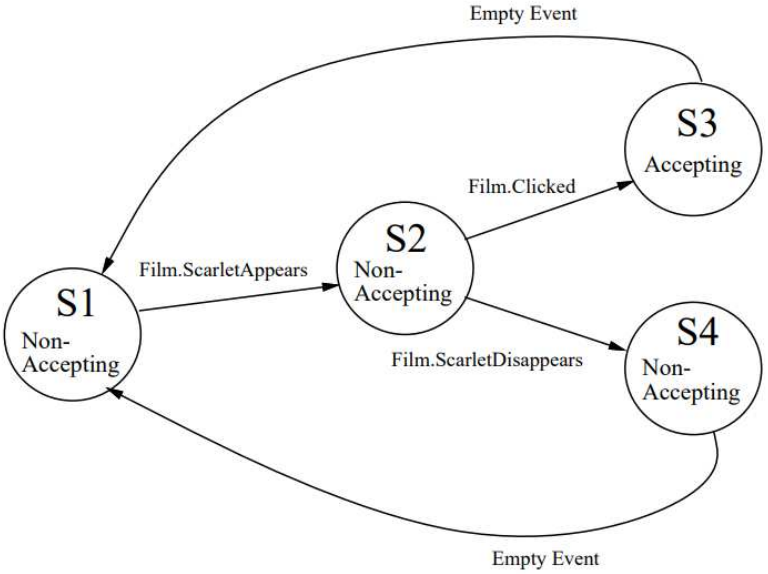
Nas subseções seguintes, destacam-se duas notações amplamente empregadas, Máquinas de Estado e Árvores de Comportamento, e posteriormente, será apresentada uma discussão específica sobre Machinations, voltada ao modelamento de economias e dinâmicas sistêmicas do jogo.

2.2.1 Máquinas de Estado

Em jogos de computador, máquinas de estado finitas (FSM) modelam o comportamento de NPCs por meio de um conjunto limitado de estados, como patrulhar, perseguir, atacar, e transições disparadas por eventos e condições, como distância ao alvo, linha de visão, saúde, o que favorece simplicidade, previsibilidade e depuração em tempo real. Em projetos maiores, é comum recorrer a FSMs hierárquicas (HFSM) ou à decomposição por subsistemas para mitigar a “explosão de estados”. Aplicações recentes ilustram essa prática, Adeniyi *et al.* implementaram um engine 2D com um subsistema de IA baseado em FSM para agentes orientados a estados, destacando a clareza de projeto e a facilidade de teste (ADENIYI *et al.*, 2024). Em outro estudo, Büsselberg e Reuss realizaram e avaliaram uma FSM como “jogador” em um jogo tático, na Unreal Engine, analisando requisitos funcionais e não funcionais e mostrando que a abordagem entrega decisões coerentes e adaptáveis, embora com limites de escalabilidade e coordenação (BÜSSELBERG; REUSS, 2024).

Para tornar mais concreto o conceito de máquina de estados, a Figura 2 exibe um diagrama típico que demonstra a transição entre diferentes comportamentos de um agente de jogo. Observa-se como cada estado é conectado por condições e eventos que determinam o fluxo de execução, aspecto fundamental para controlar a lógica de personagens e objetos interativos.

Figura 2 - Exemplo de Máquina de Estado



Fonte: BATES; BACON (1994)

Cada círculo é um estado identificado por S1, S2, S3, S4. O rótulo *Accepting/Non-Accepting* indica se o estado é final (aceitador) ou não final. As setas são transições e o texto sobre a seta é o evento que dispara a mudança de estado. Assim, S1 passa para S2 com o evento *Film.ScarletAppears*; S2 passa para S3 com *Film.Clicked* (o usuário clica); S2 passa para S4 com *Film.ScarletDisappears*; e há também transições rotuladas *Empty Event* (*epsilon*), que não exigem evento externo, como S1 para S3 (arco superior) e S4 para S1 (arco inferior).

Assim, a máquina descreve condições de passagem entre estados e quais trajetos são possíveis a partir dos eventos mostrados, destacando S3 como estado aceitador/final.

No contexto deste trabalho FSM serve como referencial conceitual para modelagem de comportamento, mas não foi implementado na ferramenta.

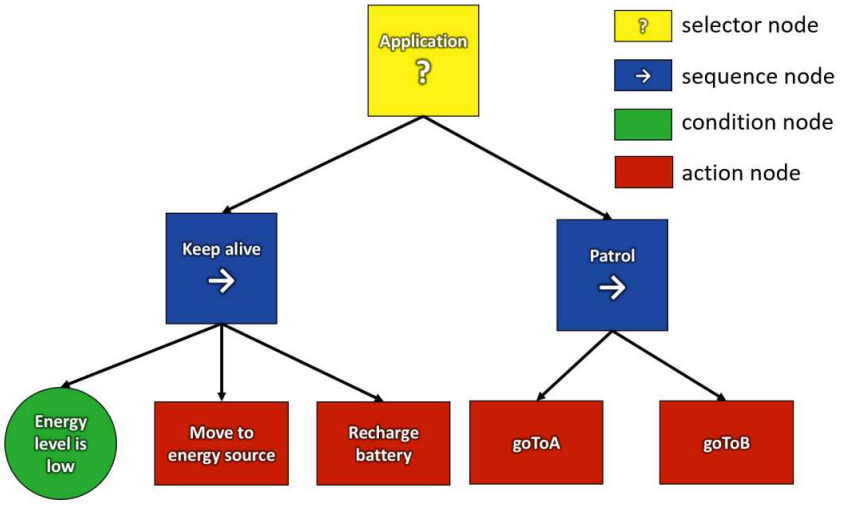
2.2.2 Árvores de Comportamento

Árvores de comportamento (*Behavior Trees*, BTs) estruturam ações e decisões em uma hierarquia de nós de controle, sequência e seletor, e decoradores, permitindo modularidade, reuso e composição de comportamentos complexos com melhor escalabilidade que FSMs planas. Em jogos, BTs são amplamente empregadas para IA de NPCs por combinarem metas reativas e planos deliberativos em um grafo executável e visualmente inspecionável. Na literatura, Sekhavat oferece uma síntese conceitual e semântica de BTs para jogos, incluindo variações e

extensões úteis à indústria, e revisa técnicas de construção e ajuste automático (SEKHAVAT, 2017). Complementarmente, Robertson e Watson demonstram a indução de BTs a partir de observações em jogos de estratégia em tempo real, extraindo *motifs*, isto é, padrões recorrentes de ações e decisões, de jogadores humanos e convertendo-os em estruturas BT executáveis, o que evidencia o potencial de aprendizado de comportamentos no domínio de jogos (ROBERTSON; WATSON, 2015).

A Figura 3 apresenta uma árvore de comportamento que exemplifica a hierarquia de decisões e a execução modular de ações em personagens controlados por IA. Essa representação ajuda a compreender como sequências de ações e seletores trabalham em conjunto para criar comportamentos complexos, complementando a discussão sobre a escalabilidade e a flexibilidade desse modelo em jogos.

Figura 3 - Exemplo de Árvore de Comportamento



Fonte: SIQUEIRA; PIERI (2015)

Em árvores de comportamento, nós internos são controladores e as folhas são ações/condições. Dessa forma, *sequence* (sequência) executa filhos da esquerda para a direita e só tem sucesso se todos tiverem sucesso, assim, se algum falhar, o nó falha. Já *selector* (seletor) chama cada filho em ordem e tem sucesso no primeiro filho que tiver sucesso, mas se todos falharem, o nó falha. O *condition* (condição) é o nó folha que verifica uma condição lógica (ex: “nível de energia baixo?”). E o *action* (ação) é o nó folha que executa um comportamento (ex: “Recarregar bateria”, “GoToA”).

A hierarquia (*root* → controladores → folhas) torna o fluxo decisório legível e modular, permitindo reuso de ramos e escala melhor que FSMs planas.

No contexto deste trabalho BT serve como referencial conceitual para decisões hierárquicas, mas também não foi implementado e aparece para situar o leitor no panorama de técnicas.

2.2.3 Machinations

A proposta é oferecer uma notação com sintaxe bem definida para descrever fluxos de recursos, estruturas de progressão e ciclos de *feedback* de forma clara e consistente, além de uma ferramenta online para desenhar diagramas e simulá-los em tempo real (DORMANS; ADAMS, 2012).

Na prática, o *framework* organiza o sistema do jogo em nós e conexões. Esses diagramas podem ser executados por passos de tempo, com coleta de dados e repetição de execuções para análise, o que favorece estudos estatísticos do comportamento do sistema. Em termos de expressividade, a literatura associada ao *framework* sustenta seu uso para simular sistemas complexos, com menções a análises de múltiplas execuções, como *multiple runs*, base para abordagens tipo Monte Carlo (DORMANS; ADAMS, 2012). As simulações podem operar em modos de tempo síncrono, assíncrono ou em turnos, e nós podem ser automáticos, interativos, passivos ou de ação inicial, o que permite representar desde sistemas em tempo real até lógicas baseadas em turno.

A ferramenta define um conjunto de símbolos com semântica clara. Os símbolos que compõem a notação Machinations, e que servem de base para a modelagem de economias e dinâmicas de jogo, podem ser observados na Figura 4. Nela, cada ícone é apresentado com sua função, permitindo ao leitor relacionar a descrição conceitual com uma visualização clara e organizada.

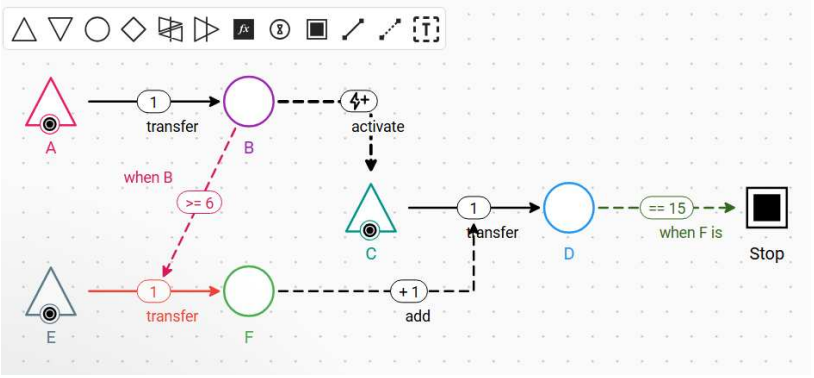
Figura 4 - Principais símbolos do Machinations

Fonte: Machinations Documentation (2025).

- *Source*: produz recursos e os injeta no sistema (por exemplo, geração de ouro, vida, munição).
- *Drain*: consome ou remove recursos do sistema (por exemplo, custo, desgaste, perda).
- *Pool*: armazena ou acumula recursos e é o “estado” observável do sistema (por exemplo, vida, ouro, munição).
- *Gate*: redistribui recursos conforme probabilidades (pesos, porcentagens) ou condições (intervalos, comparações).
- *Trader*: realiza troca entre recursos (trocas ponderadas), apropriado para mercados internos e câmbios.
- *Converter*: transforma entradas em saídas segundo uma razão definida, modelando crafting, trocas determinísticas ou processamento.
- *Delay*: introduz tempo de espera antes que recursos avancem no fluxo (modela *cooldowns*, tempo de fabricação).
- *Queue*: representa uma fila de processamento e atendimento, controlando a ordem e a capacidade (modela os gargalos).
- *End Condition*: define condições de término (vitória ou derrota) que param a simulação quando satisfeitas.

Além dos símbolos, há as conexões entre eles, podendo ser *Resource Connection*, representada por uma seta contínua, responsável por transportar recursos de um nó a outro, e *State Connection*, representado por uma seta pontilhada, responsável por ativação, inibição e modificação de elementos, como é mostrado na Figura 5, que exibe bem a diferença visual entre as conexões e os símbolos do diagrama no Machinations.

Figura 5 - Diferença visual entre conexões e símbolos



Fonte: Machinations Documentation (2025)

Assim, o Machinations oferece um vocabulário visual para expressar economias de jogo com clareza operacional, síntese e execução, e capacidade analítica, com execuções múltiplas, tornando-se um artefato eficaz de integração entre design e engenharia em jogos de computador (DORMANS; ADAMS, 2012).

2.3 Formatos de intercâmbio de dados

Formatos de intercâmbio de dados são fundamentais para persistir, compartilhar e integrar artefatos de desenvolvimento entre ferramentas, motores de jogo e serviços auxiliares, como telemetria, versionamento e análise. Em jogos de computador, dois formatos textuais e amplamente suportados destacam-se: XML e JSON. Ambos oferecem legibilidade humana, ampla disponibilidade de bibliotecas e compatibilidade com *pipelines* modernos, mas diferem em modelo de dados, mecanismos de validação e custo de processamento. Em linhas gerais, XML favorece documentos fortemente hierárquicos e validação formal via esquemas (XSD), além de dispor de um ecossistema maduro de transformação (XSLT), por outro lado, tende a ser mais verboso e exige atenção ao tipo de *parser* (DOM vs. *streaming*). Já o JSON privilegia compacidade e simplicidade (objetos/*arrays*), o que costuma se traduzir em arquivos menores e *parsing* eficiente, especialmente em integrações *web* e ferramentas com suporte nativo, em contrapartida, sua validação

costuma depender de convenções de projeto ou de JSON Schema, menos consolidado em alguns fluxos.

No contexto deste trabalho, a escolha por XML e JSON atende a dois objetivos, interoperabilidade com ferramentas externas, leitura e escrita em formatos amplamente reconhecidos, facilitando inspeção, versionamento e transformação, e a manutenção e evolução do editor, separando a política de persistência do restante da interface e permitindo estratégias distintas conforme o caso de uso. As subseções seguintes detalham vantagens, desvantagens e usos práticos de cada formato no desenvolvimento de jogos.

2.3.1 XML

No desenvolvimento de jogos, XML é frequentemente empregado para definir conteúdo e regras, configurar cenas e estruturar *logs* e telemetria graças à sua hierarquia clara, legibilidade e ao ecossistema maduro de validação por esquema (XSD), que ajuda a garantir consistência entre equipes e versões de *build*. Em contextos educacionais e *serious games*, *frameworks* baseados em XML permitem descrever componentes, atividades e avaliações de forma declarativa e extensível (CARUSO *et al.*, 2024). Já em jogos e simulações, *schemas* XML têm sido usados para padronizar arquivos de *log* e facilitar análise em larga escala (HAO *et al.*, 2016). Entre as vantagens, destacam-se a portabilidade, o farto suporte de ferramentas, em editores, validadores e transformações XSLT, e a validação formal via XSD. Como desvantagens, apontam-se a verbosidade, tamanho maior dos arquivos, o custo de *parsing* quando se usa DOM em arquivos extensos, normalmente mitigado com *parsers streaming*, e a adequação variável do modelo em árvore para certos dados altamente relacionais. (CARUSO *et al.*, 2024; HAO *et al.*, 2016).

2.3.2 JSON

Em *pipelines* modernos de jogos (clientes, *backends*, ferramentas), JSON é amplamente adotado para troca e persistência de dados por ser leve, legível e rápido de serializar e parsear, integrando-se bem a motores e *stacks* baseados em JavaScript, C# e serviços *web*. No domínio de jogos, JSON vem sendo usado não só para configuração de conteúdo e telemetria, mas também como formato de

representação de trajetórias de jogo para análise e comparação entre partidas (NGUYEN; LIÉBANA; LUCAS, 2025). Em avaliações comparativas, JSON tende a produzir arquivos menores e *parsing* mais eficiente que XML em diversos cenários de aplicações industriais, o que favorece iteração rápida e trânsito de dados (NURSEITOV *et al.*, 2009). Entre as vantagens, estão a compacidade, a simplicidade estrutural (objetos/*arrays*) e a onipresença de bibliotecas. Entre as desvantagens, destacam-se a ausência de um esquema padronizado de fato comparável ao XSD, embora exista JSON Schema, recursos limitados para *mixed content* e menor expressividade para documentos fortemente hierárquicos. (NGUYEN; LIÉBANA; LUCAS, 2025 ; NURSEITOV *et al.*, 2009).

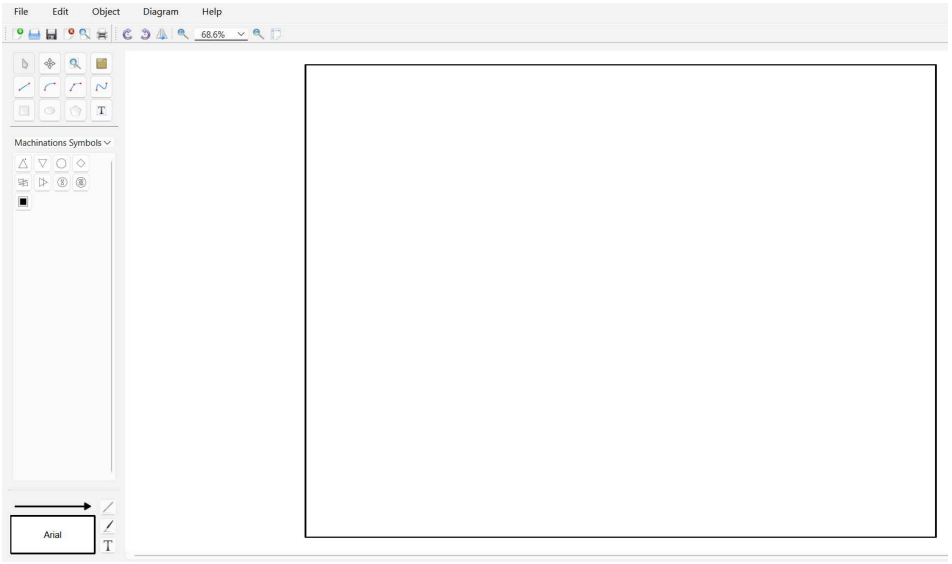
3 DESENVOLVIMENTO DO EDITOR

Este capítulo descreve as escolhas do projeto e as etapas de implementação do editor gráfico.

O editor deste trabalho foi construído sobre a ferramenta de código aberto Jade, utilizando a linguagem C++ e Qt6. O Jade não foi desenvolvido neste trabalho, ele apenas fornece a infraestrutura de um editor gráfico, como canvas com zoom/seleção, paleta, comandos de desfazer/refazer e suporte básico de persistência. Neste trabalho, o que foi feito foi estender e adaptar o Jade para a notação Machinations, implementando biblioteca de símbolos, conexões *resource/state* com setas e estilos, rótulos por item, subdiagramas pelo "*Select* Mode" e persistência nativa em XML/JSON preservando aparência dos diagramas.

A interface do editor desenvolvido neste trabalho, extensão do Jade, mostrada na Figura 6, ilustra a organização do ambiente de trabalho, incluindo a área de desenho, a paleta de símbolos e os controles de edição. Essa visão geral demonstra como o sistema foi pensado para oferecer usabilidade, fluidez na criação dos diagramas e facilidade ao usuário.

Figura 6 - Interface do editor gráfico desenvolvido



Fonte: Autoria própria (2025)

O editor inicialmente foi projetado para representar circuitos elétricos e lógicos, implementando componentes como op amps, fontes de tensão, transistores, e portas lógicas (AND, OR, etc.).

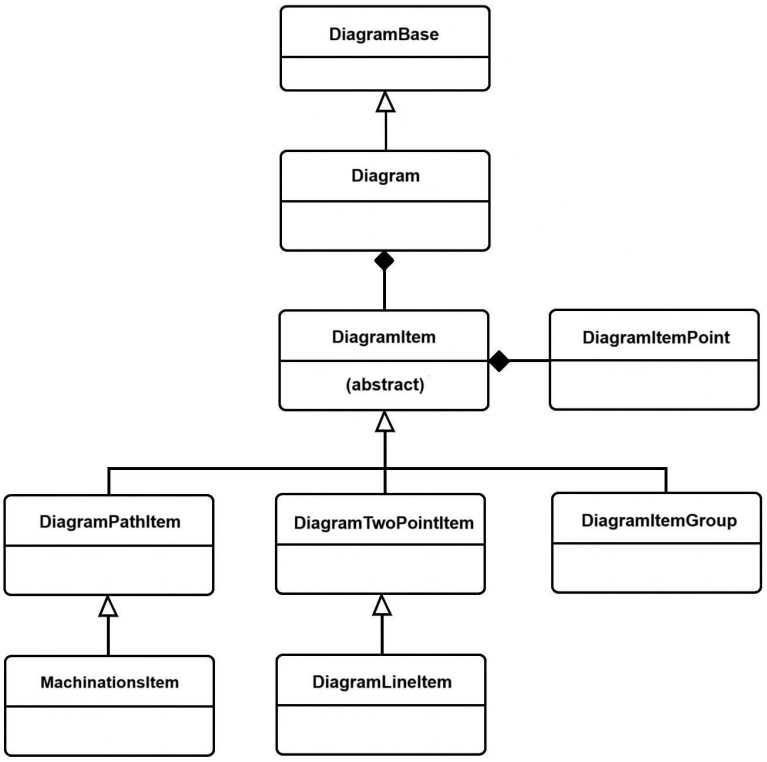
Internamente, o Jade organiza-se em módulos que separam interação (seleção, arrasto, edição), renderização (desenho vetorial via Qt), modelo de dados

(itens, propriedades e relações) e entrada e saída (leitura e gravação). Nessa base, este trabalho apenas estendeu funcionalidades já existentes.

A solução mantém a estrutura modular do Jade, em que a classe `Diagram` atua como interface principal e gerenciadora dos elementos visuais, enquanto `DiagramItem` permanece como classe base para símbolos, concentrando posição, propriedades, pontos de conexão e hierarquia. Para conexões, linhas e flechas, a especialização `DiagramLineItem` estende `DiagramTwoPointItem`. O agrupamento temporário e a manipulação conjunta de elementos são gerenciados por `DiagramItemGroup`, componente essencial para a definição de subdiagramas. Essa composição de classes foi a base para viabilizar as extensões propostas.

A arquitetura interna do editor é sintetizada na Figura 7, que apresenta um diagrama UML simplificado das classes e de suas relações. Essa representação contribui para a compreensão do desenho modular do *software*, destacando como a separação de responsabilidades favorece a manutenção, a extensibilidade e a integração de novas funcionalidades.

Figura 7 - Diagrama UML simplificado



Fonte: Autoria própria (2025)

A adaptação inicial do Jade exigiu a remoção dos símbolos elétricos originais e a criação de uma nova biblioteca gráfica alinhada ao léxico do

Machinations, além de ajustes na interface para carregamento e disposição dos novos elementos. Também foi incorporado o "*Select Mode*" como modo padrão de interação, com seleção retangular e manipulação de múltiplos itens, reforçando usabilidade e modularização dos diagramas digitais.

3.1 Tradução entre formatos de representação

Originalmente, o Jade gravava diagramas em seu formato nativo, `jdm`. No contexto deste trabalho, o editor foi estendido para suportar interoperabilidade em XML e JSON. Quando o usuário salva ou abre um arquivo, o editor verifica a extensão do arquivo e escolhe automaticamente o caminho certo. Essa lógica de despacho foi implementada em `Diagram::save` e `Diagram::load`: se o caminho termina com `.json`, invocam-se os métodos `saveJson` e `loadJson`, caso contrário, utiliza-se o fluxo XML, com os métodos `saveXml` e `loadXml`. Dessa forma, a escolha do formato, sendo XML ou JSON, não fica na interface, ela é feita na camada de modelo, o que reduz o acoplamento entre as duas.

No fluxo XML, o diagrama é serializado com `QXmlStreamWriter` em uma *ta*g raiz `<diagram>`, delegando a gravação de atributos e filhos às rotinas genéricas `writeXmlAttributes` e `writeXmlChildElements`. Na leitura, `readXmlAttributes` e `readXmlChildElement` reconstroem o grafo de itens, e em caso de arquivos inválidos, o editor emite mensagens de erro e aborta a carga, mantendo a pilha de desfazer consistente.

No JSON, utiliza-se funções auxiliares que convertem cores, pincéis e canetas do Qt para texto e vice-versa, sempre do mesmo jeito, como as funções `JD_colorToString` e `jdColorFromJson`, que ficam responsáveis pelas cores das linhas. Isso garante que, ao salvar e abrir, as propriedades visuais mantenham exatamente o mesmo significado do Qt, mesma cor, mesma espessura e mesmo estilo de linha. Dessa forma, o desenho continua com a mesma aparência após fechar e reabrir o arquivo, tanto em XML quanto em JSON.

Para copiar e colar dentro do editor, usa-se um XML simples no *clipboard*, um bloco `<items>` contendo os itens selecionados. Esse formato é leve e já reaproveita as rotinas de leitura e escrita existentes. Ao colar, os itens são reconstruídos e recebem um pequeno deslocamento (*offset*) para não ficarem sobrepostos. Se o documento usar outra escala ou unidade, o editor converte

automaticamente as medidas para a unidade do diagrama atual. Mantendo assim XML para o *clipboard* interno e XML e JSON para arquivos em disco, sem precisar duplicar lógica e deixando o código mais enxuto.

Em síntese, a camada de tradução entre representações foi desenhada com três objetivos, sendo eles a transparência de formato para o usuário final, estabilidade visual e semântica, e separação de *concerns* entre troca interna, *clipboard* XML, e persistência externa, XML e JSON.

3.2 Implementação

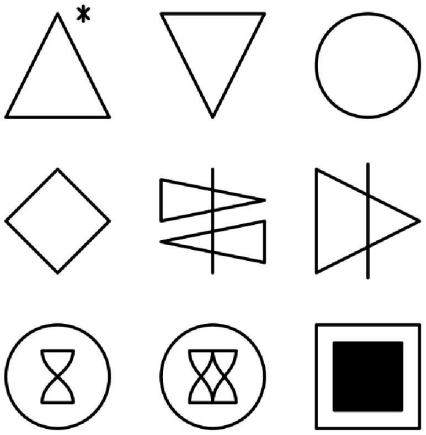
A implementação partiu do Jade (C++/Qt), base pré-existente deste projeto, não desenvolvida. Manteve-se as responsabilidades centrais do Jade, como canvas, seleção/zoom, comandos de desfazer/refazer, infraestrutura de itens, e o estendeu incrementalmente para atender à notação Machinations e aos novos requisitos de persistência (XML/JSON). O desenvolvimento foi iterativo, onde cada funcionalidade entrou como módulo coeso, com contratos simples entre camadas e uso consistente da pilha de undo/redo. Priorizou-se baixo acoplamento, via fábrica de itens e serialização desacoplada da UI, e alta coesão, onde cada item fica responsável pelo próprio desenho e estado, favorecendo manutenção e evolução. Nas subseções seguintes, detalha-se as adaptações para a biblioteca de símbolos, mecanismos de conexão, subdiagramas, mecânica de seleção e inserção de rótulos.

3.2.1 Adaptação do Jade e biblioteca de símbolos Machinations

A primeira etapa consistiu em substituir a biblioteca de componentes elétricos por uma biblioteca de símbolos compatível com o Machinations. Os componentes antigos foram desativados e, em seu lugar, a lista de itens passou a registrar símbolos equivalentes, a Figura 8 exibe os símbolos implementados, sendo eles respectivamente, *source*, *drain*, *pool*, *gate*, *trader*, *converter*, *delay*, *queue* e *end condition*, disponibilizados ao usuário na paleta do editor.

A Figura 7 reúne os símbolos do Machinations efetivamente implementados na ferramenta, evidenciando o cuidado em manter a fidelidade visual em relação à notação original. Essa apresentação permite verificar de forma direta o conjunto de elementos disponíveis para o usuário e reforça a correspondência entre a proposta conceitual e a solução desenvolvida.

Figura 8 - Símbolos implementados do Machinations



Fonte: Autoria própria (2025)

Cada símbolo foi redesenhado manualmente com rotinas gráficas do Qt, como `QPainterPath` com `moveTo`, `lineTo`, `addEllipse`, `cubicTo`, garantindo fidelidade visual à notação original. As funções estáticas de criação e o registro em fábrica mantém o código modular e facilitam a manutenção e extensão futura da biblioteca.

O conjunto acima foi selecionado para oferecer uma cobertura mínima útil de modelagem, com geração (*Source*), consumo (*Drain*), acúmulo (*Pool*), decisão (*Gate*), transformação (*Converter*), troca (*Trader*), atraso (*Delay*), ordenação (*Queue*) e término (*EndCondition*). Esses símbolos aparecem com alta frequência na literatura e em exemplos da notação e, ao mesmo tempo, foram viáveis de implementar com `QPainterPath` dentro do prazo e do escopo definidos.

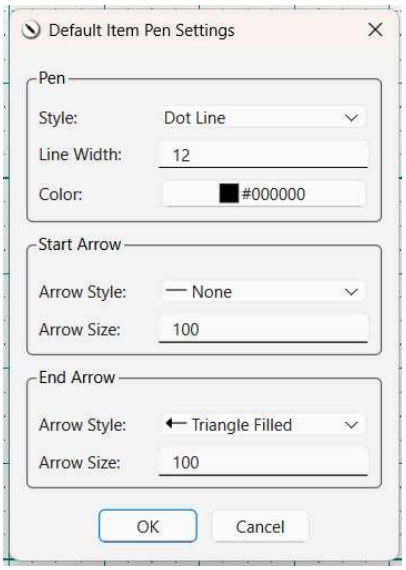
A notação Machinations completa possui outros símbolos e variantes, além de extensões visuais e semânticas. Porém eles não fazem parte deste escopo e estão planejados como trabalhos futuros, seguindo o mesmo padrão de desenho, registro na fábrica de itens e serialização XML/JSON já adotados.

3.2.2 Conexões e flechas

As conexões entre símbolos usam `DiagramLineItem`, que define *bounding box*, *shape* e renderização com caneta, além do desenho de setas em cada extremidade conforme a orientação da linha. Este item também serializa atributos de estilo, como `pen`, `startArrow`, `endArrow` para leitura e escrita, como é mostrado na Figura 9, que apresenta o conjunto de propriedades e a seleção de estilos no editor. Essa representação visual auxilia na compreensão das características

estruturais preservadas durante a serialização e na forma como o editor controla a aparência final das conexões, preservando configurações, como linha contínua ou tracejada e presença de flechas.

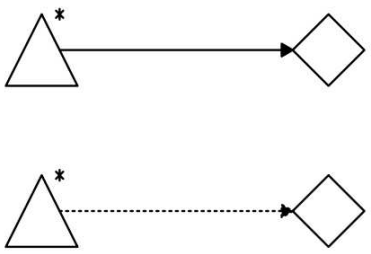
Figura 9 - Propriedades de estilos no editor



Fonte: Autoria própria (2025)

Para ilustrar o funcionamento das ligações entre os elementos do diagrama, a Figura 10 mostra exemplos de *Resource Connection* e *State Connection* dentro do editor. Essa diferenciação visual é essencial para compreender como fluxos de recursos e estados de ativação são representados, tornando mais clara a mecânica de interações descrita nesta etapa do trabalho.

Figura 10 - *Resource Connection* e *State Connection* no editor



Fonte: Autoria própria (2025)

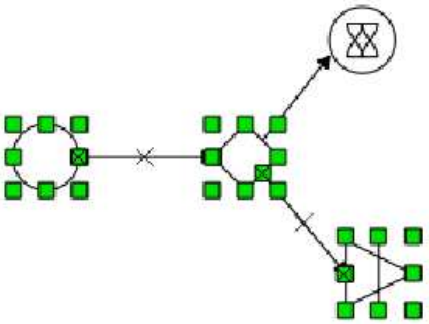
3.2.3 Subdiagramas reutilizáveis

Para suportar subdiagramas, o editor adota o "*Select Mode*" como modo padrão logo após a inicialização, permitindo seleção retangular e ações de manipulação em conjunto, como copiar, recortar, colar ou agrupar. A classe

`DiagramItemGroup` encapsula um conjunto de itens como unidade lógica, com métodos de escrita e leitura em XML que percorrem e persistem os itens internos, mantendo coesão entre o agrupador e seus elementos. Esse mecanismo viabiliza a modularização e a reutilização de estruturas recorrentes, economizando tempo na modelagem.

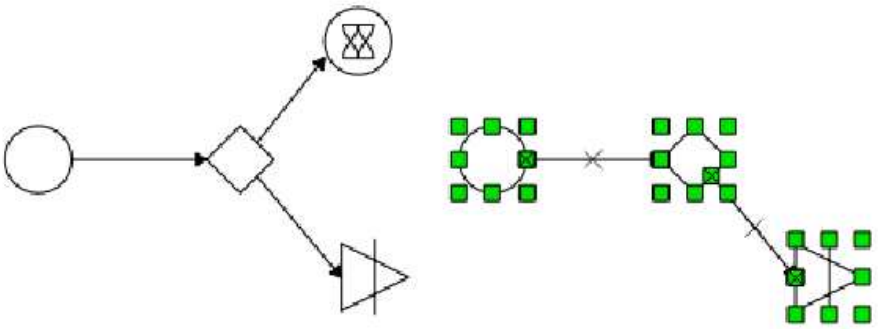
A Figura 11 apresenta um diagrama com alguns símbolos selecionados pelo "*Select Mode*" e a Figura 12 ilustra esses mesmos símbolos sendo reutilizados no diagrama, demonstrando a questão de subdiagramas reutilizáveis.

Figura 11 - Símbolos selecionados



Fonte: Autoria própria (2025)

Figura 12 - Símbolos reutilizados no diagrama



Fonte: Autoria própria (2025)

Operacionalmente, o agrupamento cria um `DiagramItemGroup` posicionado na âncora do primeiro item, transfere os itens para o grupo e atualiza a pilha de desfazer, e o desagrupamento faz a operação inversa, reinstalando os itens com as posições corretas.

3.2.4 Mecânica de seleção, transformação e desfazer/refazer

A classe `Diagram` gerencia seleção múltipla, rotação, espelhamento e reordenação dos itens, sempre registrando as operações na pilha, `QUndoStack`,

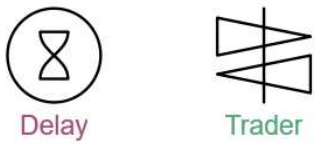
para desfazer e refazer. O modo padrão de operação é explicitamente rotulado como “*Select Mode*”, o que simplifica o fluxo cognitivo do usuário e a comunicação de estado na interface.

3.2.5 Texto e propriedades de itens

O editor possui a possibilidade de inserção de texto e o ajuste de propriedades por item, como estilo do tracejado, espessura de linha, coloração dos símbolos e flechas, integrando esses metadados ao mecanismo genérico de propriedades de `DiagramItem`, que inclui leitura e escrita e suporte a diálogos de edição. Essa abordagem aproveita a infraestrutura já existente para persistir atributos e para expor controles de edição em janelas dedicadas.

A Figura 13 destaca a possibilidade de inserir textos explicativos e personalizar propriedades gráficas, como cores, diretamente nos elementos do diagrama. Esse recurso amplia a expressividade do editor e favorece a criação de modelos mais ricos e informativos, reforçando a flexibilidade da ferramenta para diferentes contextos de projeto.

Figura 13 - Texto e propriedade de cor no editor



Fonte: Autoria própria (2025)

3.2.6 Relação com o Machinations original

O editor desenvolvido obteve similaridades mantidas em relação ao Machinations, sendo elas a semântica visual da notação, com símbolos e distinção entre *resource* e *state*, construção modular, com subestruturas reutilizáveis, os subdiagramas, anotações e propriedades visuais para inserir texto e documentar dentro dos diagramas.

Porém houve algumas diferenças deliberadas nesta etapa, como o escopo de símbolos, onde só foi implementado um subconjunto, não há simulação embutida, pois o foco está na edição e persistência, a execução de modelos fica para trabalhos futuros, o ambiente do editor é desktop, pelo Qt, não web, sem

colaboração em tempo real, além da persistência ampliada para XML e JSON nativos com preservação de propriedades como cores, espessuras, estilos e rótulos.

O editor desenvolvido permite a modelagem de sistemas com economias e fluxos de recursos, como jogos de estratégia, *tower defense*, *idle*/incremental, *tycoon*/gestão, *crafting* e sistemas de progressão, como custo ou ganho de XP, moedas, *cooldowns*, gargalos de fila. A ênfase atual é arquitetar o fluxo e documentar regras.

4 ANÁLISES

Esta seção apresenta uma análise da qualidade do editor desenvolvido sob quatro eixos, a fidelidade visual e semântica à notação Machinations, com comparação lado a lado com diagramas de referência, a usabilidade do fluxo de edição, analisando os efeitos de "*Select Mode*", undo/redo e copy/paste em sessões curtas de uso, a modularidade e reuso por subdiagramas, utilizando as funcionalidade de agrupar, duplicar e manter consistência estrutural, e a interoperabilidade por arquivos XML/JSON, observando a equivalência após salvar/abrir e entre formatos. Complementarmente, discutem-se portabilidade e implicações para trabalhos futuros.

4.1 Procedimento de avaliação

A avaliação combinou experimentação prática e observação sistemática. Primeiramente, foram construídos diagramas de referência da documentação Machinations para analisar a aderência visual e semântica, comparando a saída do editor com o diagrama original. Em seguida, executaram-se testes informais com sessões curtas de uso, para montar e modificar pequenos diagramas e relatar suas impressões. Tratou-se de uma avaliação formativa, qualitativa, com observação direta e discussões sobre tarefas básicas como inserir símbolos, conectar, rotular, agrupar/duplicar, registrando dificuldades, erros e comentários espontâneos sobre fluxo e clareza. Esses procedimentos sustentam a análise de fidelidade e de experiência de uso apresentada a seguir.

Por fim, realizaram-se exportações e importações entre XML e JSON para verificar consistência estrutural e estabilidade de propriedades, como cores, estilo das linhas, flechas e textos. O enfoque é predominantemente qualitativo, com verificações objetivas específicas, comparação visual e semântica e testes de equivalência de propriedades após salvar/abrir, suficientes para aferir qualidade nos eixos definidos.

4.2 Fidelidade visual da biblioteca Machinations

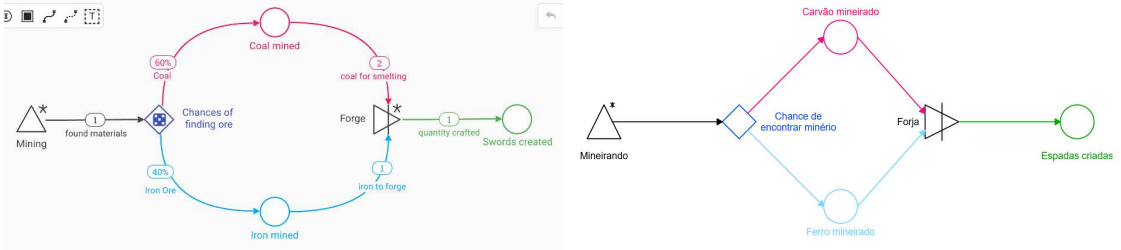
A comparação lado a lado entre os símbolos originais da documentação Machinations e suas versões no editor mostra correspondência sólida de formas, proporções e marcadores internos, como por exemplo, distinções claras entre

source, *drain*, *pool*, *gate* e operadores de transformação. As conexões preservam orientação, estilo de linha, podendo ela ser contínua ou tracejada, e setas de extremidade, mantendo legibilidade em diferentes níveis de zoom. Não foram observadas distorções que comprometem o reconhecimento da notação.

A opção por fidelidade visual e semântica à notação oficial é intencional e garante reconhecimento imediato pelos usuários, compatibilidade com a documentação e exemplos do Machinations e interoperabilidade na persistência (XML/JSON), sem ambiguidades. Não se pretendeu propor variações gráficas nesta fase.

A Figura 14 compara um diagrama criado na ferramenta original Machinations com o mesmo diagrama reproduzido no editor desenvolvido. A análise lado a lado evidencia a correspondência de formas, conexões e proporções, validando a proposta de oferecer uma alternativa aberta compatível com a notação oficial.

Figura 14 - Comparação entre o diagrama original da ferramenta Machinations (à esquerda) e o reproduzido no editor desenvolvido (à direita)



Fonte: Machinations Documentation (2025); Autoria própria (2025)

Sobre as diferenças visuais, esta comparação busca fidelidade de notação, como formas, rótulos, tipos de ligação e setas. As variações observadas como fontes, espaçamentos e posição final dos elementos, decorrem do ambiente Qt e de decisões de layout do editor, mas a leitura do modelo, fluxo de recursos e relações de estado, é preservada. Isso atende ao objetivo desta etapa, a compatibilidade de semântica e aparência suficientes para edição e persistência XML/JSON.

4.3 Usabilidade, seleção e fluxo de edição

O início no "*Select Mode*" reduz dificuldade, visto que o usuário identifica rapidamente como selecionar, mover e operar múltiplos itens. A seleção retangular e os atalhos para agrupar, duplicar e alinhar elementos aceleram tarefas recorrentes, além de ter a possibilidade de se utilizar atalhos do teclado. A pilha de desfazer e

refazer oferece segurança para exploração, permitindo correções rápidas sem perda de contexto, garantindo uma certa liberdade ao usuário de fazer alterações. O resultado é um ciclo de edição curto e intuitivo, com baixo custo de aprendizado para operações básicas e uma interface clara para uso.

4.4 Modularização por subdiagramas

O mecanismo de subdiagramas mostrou-se efetivo para o reuso de estruturas recorrentes no diagrama, permitindo uma certa economia de tempo ao usuário. Agrupar itens em uma unidade lógica, copiar e colar blocos e reposicioná-los sem perder propriedades nem relações internas reduz retrabalho e incentiva a composição de modelos maiores a partir de “peças” confiáveis. Do ponto de vista do processo, subdiagramas funcionam como artefatos de biblioteca interna do projeto.

4.5 Interoperabilidade e formatos de arquivo

A decisão de formato por extensão, JSON ou XML, simplifica a experiência do usuário e separa a política de persistência da interface. Em XML, a estrutura hierárquica do diagrama é direta de inspecionar e versionar, já em JSON, a representação é concisa e amplamente compatível com *pipelines* externos. Em ambos os casos, utilitários de conversão de cor, pincel e caneta preservam aparência entre as sessões. Os vários testes de leitura após a escrita confirmaram que ao abrir um arquivo recém salvo, reconstrói o diagrama com equivalência visual e semântica, sem perder qualquer propriedade dos símbolos presentes no diagrama.

Além do próprio editor, os arquivos XML/JSON foram deliberadamente modelados de forma neutra, com chaves estáveis para símbolos, posições e conexões. Isso permite o consumo por ferramentas de terceiros ou scripts como Python ou Node, em *pipelines* de transformação, incluindo geração de parâmetros de simulação ou geração de código a partir do diagrama.

4.6 Portabilidade e ambiente de execução

A implementação foi validada integralmente em ambiente Windows com o Qt. A base tecnológica é multiplataforma (Qt), porém ainda não foram executados ensaios formais em Linux e macOS para verificar desempenho gráfico, I/O e

renderização. Assim, trata-se de uma pendência de verificação (e não de uma restrição técnica), que deve ser endereçada em etapas seguintes para consolidar o uso em diferentes ambientes.

4.7 Implicações e trabalhos futuros

Os resultados sugerem que a combinação de fidelidade visual, fluxo de edição eficiente e subdiagramas reutilizáveis cria base favorável para práticas de Engenharia Dirigida por Modelos. Como próximos passos, recomenda-se: elaborar um protocolo de testes com métricas objetivas, como tempo de modelagem, número de ações, erros e reaproveitamento de subdiagramas, ampliar a interoperabilidade, avaliando o consumo de XML e JSON por ferramentas de terceiros e explorando transformações automáticas, como geração de parâmetros de simulação ou de código, e também conduzir testes multiplataforma em Linux e macOS para confirmar desempenho gráfico, I/O e renderização, mitigando riscos de portabilidade e consolidando o uso do editor em diferentes sistemas operacionais.

Além de ampliar a cobertura da biblioteca de símbolos Machinations, incorporando elementos ainda não contemplados como variações e operadores específicos, seguindo o mesmo padrão de desenho (`QPainterPath`), registro em fábrica e serialização XML/JSON já adotados.

5 CONCLUSÃO

Este trabalho apresentou o desenvolvimento de um editor gráfico orientado à notação Machinations, tomando como ponto de partida uma base existente, o Jade e o Qt, e articulando-o aos princípios da Engenharia Dirigida por Modelos (MDE). A proposta concentra-se em viabilizar a representação visual fiel de estruturas de economia e progressão pela notação Machinations e em estabelecer mecanismos de edição e persistência (XML/JSON) que favoreçam a documentação e comunicação entre áreas. A etapa de simulação iterativa dos modelos não fez parte do escopo desta entrega. Em síntese, o editor buscou facilitar a criação e manipulação de diagramas, mantendo aderência semântica aos símbolos da notação e consolidando recursos essenciais de edição e persistência para o uso cotidiano em projetos de jogos.

No plano técnico, os resultados demonstram que é possível substituir integralmente a antiga biblioteca de componentes do Jade por uma biblioteca visual Machinations com fidelidade de formas e marcadores, preservando a estrutura modular do código e ampliando o reuso. Os símbolos foram desenhados manualmente com rotinas vetoriais do Qt, mantendo consistência de estilo e comportamento esperado, e a organização em funções estáticas de criação contribuiu para coesão e extensibilidade futuras.

A avaliação prática indicou fidelidade visual e agilidade de edição. A comparação lado a lado entre diagramas de referência e suas versões produzidas no editor evidenciou correspondência sólida de proporções, setas, estilos de linha e legibilidade em diferentes níveis de zoom. O “*Select Mode*” como padrão reduziu a barreira inicial de uso, com seleção retangular, atalhos e pilha de desfazer e refazer tornaram o ciclo de edição curto e seguro, com baixo custo de aprendizado para operações frequentes. Tais aspectos convergem para um fluxo de edição condizente com a prática de prototipagem rápida em design de jogos.

No eixo da interoperabilidade, a estratégia de decisão por extensão simplificou a experiência do usuário e separou a política de persistência da interface, JSON para compacidade e ampla compatibilidade e XML para inspeção hierárquica e versionamento granular. Testes de leitura após escrita confirmaram equivalência visual e semântica ao reabrir arquivos, preservando propriedades como cores e estilo de linha, além de setas e rótulos. Esse cuidado com a persistência reforça a

estabilidade do ciclo de edição e sustenta a troca com ferramentas externas, elemento central quando se almeja um fluxo *data-driven*.

Quanto ao ambiente de execução, a validação ocorreu em Windows/Qt e como descrito nas seções de análise, permanecem pendentes ensaios formais em Linux e macOS para verificar desempenho gráfico, I/O e renderização, onde esses testes compõem os próximos passos do trabalho. Esta lacuna não impõe barreiras técnicas à multiplataforma, mas restringe generalizações e deve ser endereçada em etapas seguintes para aumentar a confiabilidade do uso em contextos heterogêneos de desenvolvimento.

Do ponto de vista metodológico, os resultados sugerem que a combinação de fidelidade visual, fluxo de edição eficiente e subdiagramas reutilizáveis cria uma base favorável a práticas de MDE aplicadas ao *design* de jogos. Como desdobramentos recomendam-se a elaboração de um protocolo de testes com métricas objetivas, como tempo de modelagem, número de ações, erros e reaproveitamento de subdiagramas; a ampliação da interoperabilidade, avaliando o consumo de XML e JSON por ferramentas de terceiros e explorando transformações automáticas, como parâmetros de simulação e geração de código.

Adicionalmente, abrem-se frentes de evolução com mecanismos de validação, por exemplo, prevenção de sobreposição indevida de símbolos, exportações direcionadas a motores de jogo e integrações com formalismos complementares para análises mais rigorosas. Tais iniciativas possuem trilhas técnicas claras e podem ser planejadas incrementalmente, aproximando ainda mais modelagem visual e implementação executável em jogos.

Além de incluir a ampliação da biblioteca de símbolos Machinations para cobrir elementos restantes da notação, como variações e operadores específico, aumentando assim o escopo do trabalho e também ampliando a aplicação do editor na hora de modelar diagramas mais complexos.

Em conclusão, o editor aqui desenvolvido alcança os objetivos propostos ao oferecer um ambiente de modelagem consistente com a notação Machinations, usável no dia a dia e interoperável com formatos amplamente adotados. Ao mesmo tempo em que aponta limitações e oportunidades de amadurecimento, o projeto entrega um artefato funcional que contribui para a prática de *design* orientado a modelos em jogos digitais, servindo como base concreta para simulação, automação

e reutilização em ciclos de desenvolvimento cada vez mais integrados entre *design* e engenharia.

REFERÊNCIAS

ADAMS, E; DORMANS, J. Machinations. *In*: **Game Mechanics:** Advanced Game Design. Berkeley: New Riders, 2012. p. 79-106.

ADENIYI, A; *et al.* Development of Two Dimension (2D) Game Engine with Finite State Machine (FSM) Based Artificial Intelligence (AI) Subsystem. **Procedia Computer Science,** v. 234, p. 2996-3006, jan. 2024.

AKIKI, P. CHAIN: Developing model-driven contextual help for adaptive user interfaces. **Journal of Systems and Software,** v. 135, p. 165-190, jan. 2018.

ALBAGHAJATI, A; AHMED, M. Video Game Automated Testing Approaches: An Assessment Framework. I**EEE Transactions on Games,** v. 15, n. 1, p. 81-94, mar. 2023.

BATES, J; BACON, J. **A Development Platform for Multimedia Applications in a Distributed, ATM Network Environment.** University of Cambridge, Cambridge, 1994

BLASCO, D; *et al*. An evolutionary approach for generating software models: The case of Kromaia in Game Software Engineering. **Journal of Systems and Software,** v. 171, p. 110804, jan. 2021.

BRAMBILLA, M; CABOT, J; WIMMER, M. **Model-Driven Software Engineering in Practice:** Second Edition. *[s.l.]:* Morgan & Claypool Publishers, 2017.

BUCCHIARONE, A; CICCHETTI, A; MARCONI, A. GDF: A Gamification Design Framework Powered by Model-Driven Engineering. *In:* INTERNATIONAL CONFERENCE MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS COMPANION (MODELS-C). 22., 2019, Munique. **Anais[…]** Munique: ACM/IEEE, 2019. p. 753-758.

BÜSSELBERG, L; REUSS, P. Realization and Evaluation of a Finite State Machine as a player for a Tactical Game. *In:* LERNEN, WISSEN, DATEN, ANALYSEN (LWDA). 2024., 2024, Würzburg. **Anais[...]** Würzburg: University of Hildesheim, 2024.

CARUSO, F; *et al.* SeriousGXcraft: an XML-based Framework for Developing Serious Games. *In:* INTERNATIONAL CONFERENCE ON ADVANCED VISUAL INTERFACES (AVI '24). 17., 2024, Nova York. **Anais[...]** Nova York: Association for Computing Machinery, 2024. p 1-3.

DORMANS, J. **Engineering emergence:** applied theory for game design. 2012. Tese (Doutorado em Humanidades), Universiteit van Amsterdam, Amsterdã, 2012.

FRITZ, R; KREBS, N; ZHANG, P. A Monte-Carlo Tree Search based Tracking Control Approach for Timed Petri Nets. **IFAC-PapersOnLine,** v. 53, p. 2095-2100, jan. 2020.

HAO, J; *et al.* Taming Log Files From Game/Simulation-Based Assessments: Data Models and Data Analysis Tools. **ETS Research Report Series,** v. 2016, n. 1, p.1-17, jun. 2016.

IFTIKHAR, S; *et al*. An Automated Model Based Testing Approach for Platform Games. *In:* INTERNATIONAL CONFERENCE ON MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS (MODELS). 18., 2015, Ottawa. **Anais[...]** Ottawa: ACM/IEEE, 2015. p. 426-435.

MACHINATIONS. Create dynamic economic models. **Machinations io.** 22 ago. 2025. Disponível em: https://machinations.io/. Acesso em 2 out. 2025.

MACHINATIONS. How to read a diagram. **Machinations Docs.** 3 dez. 2023. Disponível em: https://machinations.io/docs/how-to-read-a-machinations-diagram. Acesso em 2 out. 2025.

MACHINATIONS. Interface Basics. **Machinations Docs.** 11 ago. 2025. Disponível em: https://machinations.io/docs/interface-basics. Acesso em 2 out. 2025.

MACHINATIONS. State Connections. **Machinations Docs**. 21 abr. 2025. Disponível em:https://machinations.io/docs/state-connections. Acesso em 22 dez. 2025.

NGUYEN, D; LIÉBANA, D; LUCAS, S. JSON-Bag: A Generic Game Trajectory Representation. *In:* IEEE CONFERENCE ON GAMES (COG). 2025., 2025, Lisboa. **Anais[...]** Lisboa: Queen Mary University of London, 2025. p. 1-8.

NURSEITOV, N; *et al.* Comparison of JSON and XML Data Interchange Formats: A Case Study. *In:* INTERNATIONAL CONFERENCE ON COMPUTER APPLICATIONS IN INDUSTRY AND ENGINEERING (CCIIE). 22., 2009, São Francisco. **Anais[...]** São Francisco: ISCA, 2009. p 157-162.

ROBERTSON, G; WATSON, I. Building Behavior Trees from Observations in Real-Time Strategy Games. *In:* INTERNATIONAL SYMPOSIUM ON INNOVATIONS IN INTELLIGENT SYSTEMS AND APPLICATIONS (INISTA). 9., 2015, Madrid. **Anais[...]** Madrid: IEEE, 2015. p 1-7.

SEKHAVAT, Y; Behavior Trees for Computer Games. **International Journal on Artificial Intelligence Tools,** v. 26, n. 2, p. 1-27, jan. 2017.

SIQUEIRA, F; PIERI, E. A context-aware approach to the navigation of mobile robots. *In:* SIMPÓSIO BRASILEIRO DE AUTOMAÇÃO INTELIGENTE (SBAI). 12., 2015, Natal. **Anais[...]** Natal: UFSC, 2015. p. 666-671.

ZHANG, Q; *et al*. Behavior Modeling for Autonomous Agents Based on Modified Evolving Behavior Trees. *In:* DATA DRIVEN CONTROL AND LEARNING SYSTEMS CONFERENCE (DDCLS). 7., 2018, Enshi. **Anais[...]** Enshi: IEEE, 2018. p. 1140-1145.