INSTITUTO FEDERAL
Sudeste de Minas Gerais
Campus Muriaé

MINISTÉRIO DA EDUCAÇÃO

PÁTRIA AMADA BRASIL
GOVERNO FEDERAL

Sistemas Inteligentes

Apostila Didática

Publicado e distribuído por Paulo Vinícius Moreira Dutra

paulo.dutra@ifsudestemg.edu.br

Sumário

Parte Um

1. Introdução aos sistemas inteligentes em J

A Inteligência Artificial (IA) tem sido uma das áreas mais empolgantes e transformadoras da ciência da computação nas últimas décadas. Seu impacto se estende para além das fronteiras acadêmicas e industriais, encontrando um lugar de destaque em um dos campos mais cativantes e envolventes da cultura moderna: os jogos eletrônicos.

O cenário dos jogos eletrônicos tem sido moldado e enriquecido pela evolução constante das técnicas de IA. A demanda por experiências de jogo mais envolventes, desafiadoras e realistas impulsionou o desenvolvimento de NPCs e inimigos mais inteligentes, capazes de tomar decisões complexas e adaptar-se ao comportamento do jogador. Dentre as principais referências utilizadas para elaborar essa apostila é o livro "AI for Games" [4] sendo uma referência essencial para desenvolvedores, estudantes e entusiastas da área de jogos.

Neste capítulo introdutório, exploraremos os conceitos fundamentais dos sistemas inteligentes aplicados a jogos. Os sistemas inteligentes desempenham um papel crucial na evolução dos jogos, permitindo a criação de personagens virtuais inteligentes, jogabilidade dinâmica e desafios adaptativos.

Começaremos definindo o que são sistemas inteligentes e como eles se relacionam com os jogos. Os sistemas inteligentes são programas de computador que possuem a capacidade de perceber, raciocinar, tomar decisões e aprender com o ambiente. Eles combinam técnicas de inteligência artificial, aprendizado de máquina e outras áreas relacionadas para criar agentes virtuais capazes de comportamento inteligente.

Abordaremos as aplicações dos sistemas inteligentes em jogos, destacando como eles melhoram a experiência do jogador. Esses sistemas podem ser usados para criar personagens não jogáveis (NPCs) com comportamento realista, que interagem com o jogador e respondem às suas ações de maneira inteligente. Além disso, os sistemas inteligentes permitem a geração automática de conteúdo, o ajuste dinâmico de dificuldade e a adaptação às preferências individuais dos jogadores.

Discutiremos os benefícios dos sistemas inteligentes em jogos. Eles oferecem desafios mais interessantes e dinâmicos, tornando os jogos mais envolventes e imersivos. Os jogadores podem experimentar interações mais realistas e significativas com os personagens controlados pela inteligência artificial. Além disso, os sistemas inteligentes podem melhorar a jogabilidade, fornecendo suporte e orientação aos jogadores, especialmente em jogos educacionais.

Faremos também uma breve visão geral das principais técnicas e áreas relacionadas aos sistemas inteligentes em jogos, como inteligência artificial, aprendizado de máquina, algoritmos genéticos e simulação de comportamento. Essas áreas fornecem as bases teóricas e práticas para a criação e desenvolvimento de sistemas inteligentes em jogos.

Finalmente, destacaremos a importância contínua dos sistemas inteligentes em jogos e as tendências futuras nessa

área. À medida que a tecnologia avança, novas técnicas e abordagens surgem, proporcionando oportunidades emocionantes para aprimorar ainda mais a experiência de jogo.

Ao concluir este capítulo introdutório, os leitores terão uma compreensão sólida dos sistemas inteligentes em jogos, sua relevância e impacto na indústria de jogos. Eles estarão preparados para explorar os tópicos subsequentes que abordarão as técnicas específicas utilizadas na implementação de sistemas inteligentes em jogos.

1.1 I.A Acadêmica x Game I.A

As diferenças entre a Inteligência Artificial (IA) acadêmica e a IA aplicada a jogos (Game AI) podem ser significativas, pois cada uma dessas áreas tem objetivos, desafios e abordagens distintas. Abaixo estão algumas das principais diferenças entre elas:

Objetivos

- IA Acadêmica: A IA acadêmica geralmente tem o objetivo de desenvolver algoritmos e sistemas inteligentes para resolver problemas gerais de inteligência, sem se restringir a um domínio específico. Os pesquisadores acadêmicos estão mais preocupados em entender os princípios teóricos e as técnicas fundamentais da IA, visando a aplicação em diversas áreas, como medicina, finanças, automação industrial, entre outras.
- Game AI: A Game AI é focada em criar sistemas de IA específicos para aprimorar a experiência do jogador em jogos eletrônicos. O objetivo principal é desenvolver personagens não jogáveis (NPCs) com comportamentos realistas e desafiadores, melhorar a jogabilidade e oferecer uma experiência envolvente e imersiva para o jogador.

Domínio de Aplicação

- IA Acadêmica: A IA acadêmica é amplamente aplicada em diferentes setores e pode lidar com uma ampla variedade de problemas, como processamento de linguagem natural, reconhecimento de padrões, planejamento, robótica, aprendizado de máquina, entre outros.
- Game AI: A Game AI é especificamente aplicada aos jogos eletrônicos, onde é utilizada para controlar NPCs, criar inimigos e aliados virtuais, definir estratégias e táticas dos personagens controlados pelo computador e simular comportamentos complexos para melhorar a experiência de jogo.

Complexidade e Tempo de Resposta

- IA Acadêmica: Em muitos casos acadêmicos, a complexidade dos algoritmos e modelos não é um problema tão crítico, e o tempo de resposta pode ser mais flexível. As soluções podem ser computacionalmente intensivas, pois a ênfase está na eficácia do algoritmo e não necessariamente na velocidade de resposta em tempo real.
- Game AI: Nos jogos eletrônicos, a IA deve ser capaz de tomar decisões rapidamente e em tempo real, pois a interação com o jogador precisa ser fluída e responsiva. Isso muitas vezes requer a busca de soluções que sejam eficientes em termos computacionais, equilibrando a qualidade do comportamento com o desempenho em tempo real.

Complexidade do Ambiente

- IA Acadêmica: A IA acadêmica lida com uma ampla gama de ambientes e cenários, alguns dos quais podem ser bastante complexos, mas raramente são tão ricos em interação e detalhes quanto os mundos virtuais de jogos eletrônicos.
- Game AI: Os ambientes dos jogos eletrônicos são muitas vezes ricos em interatividade, com muitos elementos em movimento e em constante mudança. Isso torna o desenvolvimento de Game AI desafiador, pois os NPCs precisam ser capazes de lidar com essas interações complexas de forma inteligente e convincente.

Em resumo, enquanto a IA acadêmica se concentra na pesquisa teórica e na aplicação geral da inteligência artificial em vários domínios, a Game AI tem um foco específico em melhorar a experiência do jogador em jogos eletrônicos, exigindo soluções adaptadas às particularidades dos ambientes virtuais e das interações em tempo real. Ambas as áreas são complementares e se beneficiam mutuamente à medida que a tecnologia avança e novas descobertas são aplicadas em diversos contextos, inclusive nos jogos eletrônicos.

1.2 Definição de sistemas inteligentes

Os sistemas inteligentes são programas de computador ou sistemas que possuem a capacidade de imitar ou simular características associadas à inteligência humana. Esses sistemas utilizam algoritmos, técnicas de processamento de dados e modelagem matemática para tomar decisões, resolver problemas, aprender a partir de dados e interagir com o ambiente de forma autônoma ou semi-autônoma.

Esses sistemas são projetados para executar tarefas complexas que normalmente requerem inteligência humana, como reconhecimento de padrões, tomada de decisões, aprendizado, processamento de linguagem natural, planejamento e raciocínio lógico. Eles são baseados em princípios e técnicas de áreas como inteligência artificial, aprendizado de máquina, processamento de dados, lógica computacional e algoritmos.

Os sistemas inteligentes podem ser encontrados em diversos campos e aplicações, incluindo jogos, automação industrial, medicina, robótica, sistemas de recomendação, processamento de linguagem natural, reconhecimento de voz, visão computacional e muito mais. Eles têm o objetivo de auxiliar e melhorar a tomada de decisões, a eficiência e a capacidade de adaptação em diferentes domínios.

Esses sistemas podem ser desenvolvidos com base em diferentes abordagens e técnicas, como redes neurais artificiais, algoritmos genéticos, lógica fuzzy, processamento de linguagem natural, entre outros. Eles podem ser implementados em diferentes níveis de complexidade, desde sistemas simples com regras pré-definidas até sistemas sofisticados que aprendem e se adaptam continuamente ao ambiente.

A evolução dos sistemas inteligentes tem sido impulsionada pelo aumento da capacidade computacional, pela disponibilidade de grandes volumes de dados e pelo desenvolvimento de algoritmos mais avançados. Esses avanços têm permitido a criação de sistemas que podem lidar com problemas complexos de forma eficiente e fornecer soluções rápidas e precisas.

Em resumo, sistemas inteligentes são sistemas computacionais que imitam ou simulam a inteligência humana para realizar tarefas complexas. Eles são projetados para tomar decisões, aprender com dados e interagir com o ambiente de maneira autônoma ou semi-autônoma, proporcionando benefícios significativos em uma ampla variedade de aplicações e setores.

1.3 Aplicações de sistemas inteligentes em jogos

Os sistemas inteligentes têm sido amplamente utilizados na indústria de jogos para melhorar a experiência do jogador, criar personagens virtuais realistas e oferecer desafios adaptativos. Essas aplicações desempenham um papel crucial na indústria de jogos, oferecendo uma série de benefícios e melhorias significativas na experiência do jogador. Em resumo, os sistemas inteligentes têm um impacto significativo nos jogos, proporcionando uma experiência mais imersiva, desafiadora e personalizada para os jogadores. Eles contribuem para a evolução contínua da indústria de jogos, impulsionando a inovação e permitindo a criação de jogos mais envolventes e cativantes.

Aqui estão algumas das principais aplicações dos sistemas inteligentes em jogos.

- **Comportamento de personagens não jogáveis (NPCs):** Os sistemas inteligentes são utilizados para simular o comportamento de NPCs em jogos. Esses personagens podem ser programados para agir de forma autônoma, interagir com o jogador e tomar decisões com base em informações do ambiente. Os NPCs podem exibir comportamentos realistas, como reações emocionais, tomadas de decisão contextualizadas e adaptação a diferentes situações. Por exemplo, em um jogo de RPG, os NPCs podem simular emoções, reagir de forma adequada a diferentes situações e fornecer missões ou informações relevantes. Um exemplo é a série de jogos "The Elder Scrolls", em que os NPCs possuem rotinas diárias, personalidades distintas e interagem entre si. Um exemplo notável é o jogo "The Sims", no qual os NPCs têm personalidades únicas e realizam uma variedade de ações com base em suas necessidades e desejos.
- **Diálogos e interações naturais:** Sistemas inteligentes são empregados para permitir diálogos e interações naturais entre o jogador e os personagens virtuais. Isso pode envolver processamento de linguagem natural, reconhecimento de voz e geração de respostas contextuais. O jogo "The Witcher 3: Wild Hunt"é um exemplo em que os personagens têm diálogos complexos e interações com o jogador, adaptando-se às escolhas e ações do protagonista.
- **Oponentes virtuais(Inimigos):** Os sistemas inteligentes são aplicados para criar oponentes virtuais desafiadores em jogos. Esses oponentes podem usar estratégias avançadas, aprender com as ações do jogador e se adaptar ao seu estilo de jogo. Isso proporciona um desafio contínuo e estimulante, aumentando a imersão e a diversão do jogador. Por exemplo, em jogos de xadrez, sistemas inteligentes podem simular jogadores experientes ou até mesmo campeões mundiais, oferecendo partidas competitivas e estimulantes. Um segundo exemplo é o jogo "Alien: Isolation", onde o inimigo principal, o alienígena, possui um sistema de IA que aprende com os padrões de busca do jogador, tornando-se mais difícil de evitar conforme o jogo avança.
- **Geração procedural de conteúdo:** Os sistemas inteligentes podem ser empregados na geração automática de conteúdo em jogos, como níveis, missões, ambientes e itens. Algoritmos de geração procedural permitem criar conteúdo dinamicamente, proporcionando uma experiência única a cada partida. Isso aumenta a longevidade e a variedade do jogo, tornando-o mais interessante para os jogadores. Por exemplo, em um jogo de aventura, os sistemas inteligentes podem gerar labirintos ou ambientes complexos que se adaptam ao estilo de jogo do jogador, garantindo uma experiência personalizada. O jogo "No Man's Sky"utiliza um sistema inteligente para gerar um universo virtual com bilhões de planetas únicos, cada um com sua própria flora, fauna e características geográficas.
- **Ajuste dinâmico de dificuldade:** Os sistemas inteligentes podem monitorar o desempenho e o comportamento do jogador em tempo real e ajustar a dificuldade do jogo de acordo. Isso garante que o jogo seja desafiador o suficiente para manter o interesse do jogador, evitando que seja muito fácil ou muito difícil. O ajuste dinâmico de dificuldade melhora a experiência do jogador, proporcionando um equilíbrio adequado de desafio e diversão. Por exemplo, em um jogo de corrida, o sistema pode ajustar a velocidade e a agressividade dos oponentes com base no desempenho do jogador. O jogo "Left 4 Dead"adapta a quantidade e o comportamento dos inimigos de acordo com a habilidade e a performance dos jogadores, proporcionando uma experiência desafiadora e equilibrada.
- **Personalização da experiência do jogador:** Os sistemas inteligentes podem analisar o comportamento, preferências e histórico do jogador para personalizar a experiência de jogo. Com base nessas informações, o jogo pode adaptar os desafios, fornecer sugestões ou recomendações, criar personagens e histórias perso-

nalizadas, e oferecer recompensas e conquistas relevantes para o jogador. Isso cria uma experiência mais envolvente e cativante para cada jogador individualmente.

- **Tomada de decisões em tempo real e IA estratégica:** Os sistemas inteligentes são utilizados para a tomada de decisões em tempo real durante o jogo. Eles podem analisar informações do ambiente do jogo, como posição do jogador, objetivos, eventos e condições, e tomar decisões rápidas e inteligentes. Esses sistemas podem tomar decisões complexas, avaliar riscos e recompensas, e desenvolver estratégias para alcançar objetivos específicos. Eles podem analisar o ambiente do jogo, prever movimentos do adversário e tomar decisões com base em informações limitadas, proporcionando uma experiência desafiadora e realista. Por exemplo, em jogos de estratégia em tempo real, os sistemas inteligentes podem gerenciar recursos, coordenar unidades e desenvolver estratégias complexas para vencer o jogo. O jogo "StarCraft II"os oponentes controlados pelo computador usam táticas avançadas, coordenação de unidades e tomada de decisões em tempo real.
- **Assistência ao jogador:** Os sistemas inteligentes podem fornecer assistência e orientação aos jogadores durante o jogo. Isso pode incluir sugestões, dicas, tutoriais interativos e feedback personalizado. Essa assistência ajuda os jogadores iniciantes a se familiarizarem com o jogo e permite que os jogadores mais experientes explorem melhor as mecânicas e estratégias do jogo. O jogo "The Legend of Zelda: Breath of the Wild", possui um assistente virtual que fornecem dicas, orientações e ajudam o jogador durante a jornada.
- **Reconhecimento de voz e gestos:** Os sistemas inteligentes são utilizados para reconhecimento de voz e gestos, permitindo aos jogadores interagirem com o jogo de forma natural e imersiva. Os jogadores podem dar comandos de voz aos personagens ou usar gestos para controlar ações e movimentos dentro do jogo. Exemplos de jogos permitem aos jogadores interagir com os personagens ou controlar o jogo através de comandos de voz, são "Mass Effect"e "Hey You, Pikachu!".
- **Sistemas de recomendação e personalização:** Os sistemas inteligentes são empregados para criar sistemas de recomendação e personalização, sugerindo conteúdos, itens ou desafios com base nas preferências individuais do jogador. Isso melhora a experiência do jogador, fornecendo recomendações relevantes e adaptadas aos seus interesses e estilo de jogo. Por exemplo, um sistema inteligente pode recomendar novos jogos com base nos jogos anteriores que o jogador apreciou. Exemplos incluem o sistema de recomendação da plataforma Steam, que sugere jogos com base no histórico de jogos do jogador, e o algoritmo de recomendação da Netflix, que sugere filmes e séries com base nas preferências do usuário.
- **Simulação de comportamento de multidões:** Os sistemas inteligentes podem simular o comportamento realista de multidões em jogos, como torcidas, tráfego urbano ou exércitos em batalha, criando uma sensação de realismo e imersão. O jogo "Assassin's Creed Unity"utiliza sistemas inteligentes para simular o movimento e comportamento realista de multidões nas ruas de Paris durante a Revolução Francesa.

Essas são apenas algumas das aplicações dos sistemas inteligentes em jogos. À medida que a tecnologia avança, novas aplicações estão sendo exploradas, como sistemas de diálogo e interação natural, detecção de emoções do jogador, reconhecimento de gestos e expressões faciais, entre outras. Os sistemas inteligentes continuam a desempenhar um papel crucial na evolução dos jogos, proporcionando experiências mais imersivas, desafiadoras e personalizadas para os jogadores.

1.4 Visão geral das principais técnicas utilizadas

Os sistemas inteligentes em jogos envolvem uma série de técnicas e áreas relacionadas que visam criar personagens virtuais e comportamentos inteligentes para melhorar a experiência do jogador. Aqui está uma breve visão geral das principais técnicas e áreas envolvidas:

- **Inteligência Artificial (IA):** A inteligência artificial é o campo da ciência da computação que se concentra na criação de sistemas que podem realizar tarefas que exigem inteligência humana. Em jogos, a IA é usada para simular comportamentos inteligentes em personagens não jogáveis (NPCs) e adversários controlados pelo

computador. Isso envolve o desenvolvimento de algoritmos e técnicas para tomada de decisões, planejamento, aprendizado e adaptação.

- **Aprendizado de Máquina (Machine Learning):** O aprendizado de máquina é uma subárea da inteligência artificial que se concentra no desenvolvimento de algoritmos e modelos que permitem que um sistema aprenda a partir de dados e experiências passadas. Em jogos, o aprendizado de máquina pode ser usado para treinar NPCs a partir de dados de jogadores reais, permitindo que eles se adaptem e melhorem ao longo do tempo.
- **Algoritmos Genéticos:** Os algoritmos genéticos são técnicas inspiradas na evolução biológica que envolvem a geração, seleção e reprodução de soluções para resolver problemas complexos. Em jogos, os algoritmos genéticos podem ser usados para otimizar comportamentos de NPCs ou ajustar parâmetros do jogo para melhorar o equilíbrio e a jogabilidade.
- **Simulação de Comportamento:** A simulação de comportamento envolve a criação de modelos que simulam o comportamento humano ou animal em determinadas situações. Em jogos, a simulação de comportamento é usada para criar NPCs com comportamentos realistas e dinâmicos, permitindo interações mais naturais com o ambiente e com os jogadores.
- **Processamento de Linguagem Natural (NLP):** O processamento de linguagem natural é uma área da inteligência artificial que se concentra no processamento e compreensão da linguagem humana. Em jogos, o NLP pode ser usado para permitir interações por meio de comandos de voz ou para fornecer diálogos mais naturais e realistas entre os personagens do jogo.
- **Redes Neurais Artificiais:** As redes neurais artificiais são modelos computacionais inspirados no funcionamento do cérebro humano. Elas são usadas em jogos para reconhecimento de padrões, tomada de decisões e aprendizado. As redes neurais podem ser treinadas para realizar várias tarefas, como reconhecimento facial, movimentação de personagens e detecção de comportamentos anormais.
- **Lógica Fuzzy:** A lógica fuzzy é uma técnica que permite lidar com incerteza e imprecisão nas informações. Nos jogos, a lógica fuzzy pode ser utilizada para tomar decisões com base em condições vagas ou ambíguas, permitindo que NPCs ajam de forma mais natural e adaptativa.
- **Sistemas Multiagentes:** Os sistemas multiagentes envolvem a criação de entidades autônomas que podem interagir e cooperar entre si. Em jogos, os sistemas multiagentes são usados para criar interações complexas entre NPCs, permitindo comportamentos cooperativos ou competitivos.

Essas são apenas algumas das principais técnicas e áreas relacionadas aos sistemas inteligentes em jogos. A combinação dessas abordagens permite criar experiências de jogo mais imersivas, desafiadoras e interativas, proporcionando aos jogadores uma sensação de autenticidade e engajamento.

1.5 A importância dos sistemas inteligentes

Os sistemas inteligentes desempenham um papel fundamental e contínuo na indústria de jogos, proporcionando uma série de benefícios e contribuições significativas. Sua importância é evidente em várias áreas e tendências futuras.

Um dos aspectos mais impactantes é a melhoria da experiência do jogador. Os sistemas inteligentes permitem a criação de personagens não jogáveis (NPCs) com comportamentos realistas, estratégias adaptativas e tomada de decisões inteligentes. Isso resulta em interações mais imersivas, diálogos naturais e desafios adequados ao nível de habilidade de cada jogador, aprimorando assim a experiência global.

A adaptação e personalização são cada vez mais valorizadas pelos jogadores, e os sistemas inteligentes desempenham um papel central nesse aspecto. Eles são capazes de ajustar dinamicamente o jogo às preferências individuais, adaptando a dificuldade, o ritmo, os desafios e os conteúdos com base no perfil e nas habilidades de cada jogador. Isso cria uma experiência única e envolvente, tornando os jogos mais cativantes.

A geração procedural de conteúdo é uma tendência emergente que se beneficia dos sistemas inteligentes. Através de algoritmos, é possível gerar automaticamente elementos de jogos, como mapas, níveis, missões e itens, aumentando a variedade e o valor de replay dos jogos, proporcionando experiências mais dinâmicas e surpreendentes a cada partida.

Outra tendência promissora é o uso de sistemas inteligentes para fornecer assistência aos jogadores durante o jogo. Esses sistemas podem analisar o desempenho dos jogadores e oferecer sugestões estratégicas, dicas úteis e assistência personalizada, tornando os jogos mais acessíveis e inclusivos para jogadores de diferentes habilidades e níveis de experiência.

A simulação de comportamento avançada é uma área-chave dos sistemas inteligentes em jogos. Com o avanço da tecnologia, espera-se que os NPCs apresentem comportamentos mais complexos e realistas, incluindo interações sociais mais sofisticadas, emoções simuladas e personalidades distintas. Isso contribui para uma experiência de jogo mais imersiva e envolvente.

A integração de tecnologias emergentes, como realidade virtual (VR), realidade aumentada (AR), reconhecimento de voz e sensores de movimento, também impulsiona os sistemas inteligentes em jogos. Essas tecnologias proporcionam experiências mais imersivas, interativas e intuitivas, elevando a qualidade dos jogos e ampliando as possibilidades de interação.

O aprendizado de máquina e as redes neurais são elementos-chave dos sistemas inteligentes em jogos. Essas técnicas permitem que os sistemas aprendam e melhorem com base em dados e experiências passadas, tornando os NPCs mais inteligentes, adaptativos e capazes de aprender com as interações dos jogadores em tempo real.

Por fim, os sistemas inteligentes têm o potencial de revolucionar a narrativa nos jogos, permitindo a criação de narrativas dinâmicas e ramificadas que se adaptam às escolhas e ações dos jogadores. Isso proporciona experiências mais imersivas, envolventes e personalizadas, estimulando o engajamento e a participação ativa dos jogadores na história do jogo.

Em resumo, os sistemas inteligentes desempenham um papel fundamental na indústria de jogos, melhorando a experiência do jogador, possibilitando a adaptação e personalização, impulsionando a geração procedural de conteúdo, fornecendo assistência aos jogadores, simulação de comportamento avançada, integração de tecnologias emergentes, aprendizado de máquina e redes neurais, e revolucionando a narrativa dos jogos. O futuro dessa área promete ainda mais avanços e inovações emocionantes.

1.6 Utilização de Pseudocódigos para Ensinar Algoritmos de IA

Nesta apostila, utilizamos pseudocódigos como uma ferramenta eficaz para explicar os algoritmos de Inteligência Artificial (IA) aplicados a jogos. Um pseudocódigo é uma representação de alto nível de um algoritmo, que se concentra na lógica e no fluxo de controle, sem se preocupar com a sintaxe específica de uma linguagem de programação. Essa abordagem torna mais fácil compreender e aprender os princípios fundamentais da IA.

Por que usamos Pseudocódigos

- Clareza Conceitual: Pseudocódigos fornecem uma maneira clara e concisa de descrever algoritmos e lógica, permitindo que os alunos se concentrem nos conceitos subjacentes sem distrações de sintaxe de programação.
- Transferência de Conhecimento: Pseudocódigos são uma forma universal de representar algoritmos, o que facilita a transferência de conhecimento para várias linguagens de programação.

- Flexibilidade: Os pseudocódigos permitem que os alunos se concentrem na lógica e na estrutura do algoritmo, permitindo que adaptem esses conceitos a diferentes linguagens e contextos.

Estrutura dos Pseudocódigos

- Iniciamos definindo variáveis, constantes e estruturas de dados necessárias para o algoritmo.
- Descrevemos o fluxo de controle do algoritmo, incluindo decisões condicionais (if-else), loops, funções e chamadas de funções.
- Utilizamos pseudocódigos para representar operações matemáticas, cálculos de distância, movimento e outras operações típicas em jogos.

Exemplos de Pseudocódigos

Nossos exemplos incluem pseudocódigos que descrevem o comportamento de agentes em jogos. Esses pseudocódigos incluem a lógica de tomada de decisão, cálculos de distância, movimento em direção a um alvo e outras operações relevantes para jogos.

Ao estudar esses pseudocódigos, os alunos podem compreender como os algoritmos de IA são aplicados na prática e ganhar uma base sólida para implementar esses conceitos em suas próprias criações de jogos.

Os pseudocódigos apresentados neste material foram formulados de forma a serem facilmente compreensíveis e seguem uma sintaxe que é semelhante à linguagem de programação Python. Isso foi feito de propósito, pois a sintaxe Python é conhecida por sua clareza e legibilidade, tornando-a uma excelente escolha para aprender e ensinar programação e conceitos de Inteligência Artificial (IA).

A seguir, exemplos de pseudocódigos utilizados no livro

```python
# Exemplo de loop for para contar até 10
for i in range(10):
    print("Contagem: " + i)

# Exemplo de loop for com vetores
numeros = []
for i in range(10):
    numeros[i] = i

for elemento in numeros:
    print("Número: " + elemento)
```

Figura 1.1: Exemplo laço de repetição

```python
# Declaração de uma função para calcular a soma de dois números
def somar(a, b):
    resultado = a + b
    return resultado
```

Figura 1.2: Exemplo declaração de função

```
# Definição da classe Agent
class Agent:
    # Atributos da classe
    targetX   = 0
    targetY   = 0
    threshold = 10
    speed     = 8

    # Construtor da classe
    def __init__(self):
        self.targetX = 10
        self.targetY = 10

    # Método para iniciar o agente
    def start(self):
        #......

    # Método para atualizar o agente
    def update(self):
        #......
```

Figura 1.3: Exemplo Classes

2. Fundamentos de Inteligência Artificial (IA)

Os fundamentos de Inteligência Artificial (IA) em jogos são essenciais para a criação de comportamentos inteligentes e desafiadores nos ambientes virtuais. Eles fornecem a base teórica e prática para aplicar a IA no desenvolvimento de agentes autônomos capazes de interagir com o ambiente do jogo. Vários desses fundamentos desempenham um papel crucial nesse processo.

Um desses fundamentos é a concepção de agentes inteligentes, que são entidades autônomas equipadas com habilidades de percepção, raciocínio e tomada de decisão. Esses agentes são controlados pela IA para simular comportamentos humanos realistas, como planejamento estratégico, adaptação a situações em constante mudança e tomada de decisões baseadas em metas.

Outro aspecto importante é o uso de árvores de decisão, que são estruturas que representam sequências de escolhas e suas possíveis consequências. No contexto dos jogos, as árvores de decisão são empregadas para modelar o processo de tomada de decisão dos agentes, onde cada nó representa uma escolha e as ramificações representam diferentes opções disponíveis.

Os algoritmos de busca também desempenham um papel significativo. Eles são utilizados para encontrar soluções ou tomar decisões em ambientes complexos. Em jogos, esses algoritmos ajudam os agentes a determinar a melhor ação a ser tomada em uma determinada situação, como a busca em profundidade, a busca em largura e o algoritmo A*.

A lógica proposicional é outra ferramenta fundamental, permitindo a representação e o raciocínio sobre proposições e suas relações. Nos jogos, a lógica proposicional é usada para modelar as condições e regras do jogo, permitindo que os agentes avaliem o estado atual do jogo e tomem decisões com base nessas avaliações.

As redes neurais artificiais também têm um papel importante na IA em jogos. Inspiradas pelo cérebro humano, essas redes são capazes de reconhecer padrões complexos, aprender com dados e tomar decisões com base nas informações sensoriais. Os agentes de jogos podem ser treinados utilizando algoritmos de aprendizado de máquina, permitindo que eles se tornem mais inteligentes e eficientes.

O aprendizado por reforço é uma abordagem valiosa, onde os agentes aprendem a partir de interações com o ambiente, recebendo recompensas ou penalidades com base em suas ações. Nos jogos, essa técnica permite que os agentes melhorem seu desempenho ao longo do tempo, aprendendo quais ações levam a resultados positivos e evitando ações que resultem em punições.

Por fim, o planejamento desempenha um papel fundamental no desenvolvimento de estratégias de jogo. Os agentes utilizam técnicas de planejamento para gerar sequências de ações que levam à conquista de objetivos específicos, como completar missões ou superar desafios.

Esses são apenas alguns dos principais fundamentos da IA em jogos. Ao combinar e aplicar esses conceitos, é possível criar experiências de jogo mais imersivas e envolventes, proporcionando desafios inteligentes e personalizados aos jogadores.

2.1 Agentes inteligentes em Jogos

Os jogos eletrônicos têm sido um campo fértil para a aplicação de técnicas de Inteligência Artificial (IA). Os Agentes Inteligentes são elementos cruciais para criar experiências de jogo desafiadoras e envolventes, pois proporcionam aos jogadores interações mais realistas e imprevisíveis com o mundo virtual. Neste capítulo, exploraremos o papel dos Agentes Inteligentes em jogos, as abordagens utilizadas para sua criação e as vantagens que oferecem para a indústria de jogos.

2.1.1 O que são Agentes Inteligentes?

Agentes Inteligentes são entidades autônomas que têm a capacidade de perceber seu ambiente, tomar decisões com base em suas percepções e agir de acordo com suas decisões. Em jogos, esses agentes podem ser NPCs (Personagens Não Jogáveis) controlados pelo computador, que interagem com os jogadores e o ambiente do jogo. Os Agentes Inteligentes em jogos possuem várias características que os tornam distintos:

- **Percepção:** Os agentes inteligentes em jogos utilizam sensores virtuais para perceber o ambiente ao seu redor. Esses sensores podem incluir visão, audição, sensores de proximidade, entre outros, dependendo das necessidades do jogo. A percepção permite que os agentes reconheçam elementos do ambiente, como obstáculos, inimigos, itens e aliados, possibilitando que tomem decisões informadas com base nas informações obtidas.
- **Raciocínio:** O raciocínio é uma habilidade essencial para os agentes inteligentes em jogos. Com base nas informações obtidas por meio da percepção, os agentes processam esses dados e avaliam o estado atual do jogo. O raciocínio permite que os agentes analisem possíveis ações e escolham a mais adequada para alcançar seus objetivos ou responder às ações dos jogadores.
- **Tomada de Decisão:** A tomada de decisão é a capacidade dos agentes de escolher a melhor ação a ser executada com base em seu raciocínio e objetivos. Os agentes inteligentes utilizam algoritmos e técnicas específicas para determinar a ação mais adequada em cada situação, considerando fatores como recompensas, riscos, objetivos e estratégias.
- **Adaptabilidade:** Uma característica importante dos agentes inteligentes em jogos é a sua capacidade de se adaptar às mudanças no ambiente e no comportamento dos jogadores. Isso é possível por meio de técnicas de aprendizado de máquina, como o aprendizado por reforço, que permitem que os agentes aprendam com suas experiências e ajustem seu comportamento ao longo do tempo.
- **Comportamento Emergente:** Agentes inteligentes também podem exibir comportamento emergente, ou seja, comportamentos complexos e coordenados que surgem da interação entre vários agentes no ambiente do jogo. Isso pode levar a interações sociais, cooperação e estratégias surpreendentes e imprevisíveis, tornando a experiência de jogo mais dinâmica e interessante.
- **Personalização:** Agentes inteligentes podem ser projetados para se adaptar às preferências e habilidades dos jogadores. Isso permite uma experiência de jogo mais personalizada, onde o agente ajusta sua dificuldade ou estratégia para corresponder ao nível de habilidade do jogador e oferecer uma experiência desafiadora e agradável.
- **Variedade e Replay Value:** A presença de agentes inteligentes em jogos garante uma variedade de comportamentos e interações possíveis, o que aumenta a variedade e o valor de replay do jogo. Cada partida pode ser única, dependendo das escolhas e ações dos jogadores e dos agentes no jogo.

Além disso, os agentes utilizam-se de sensores e atuadores (Figura 2.1) que são componentes fundamentais em jogos, pois permitem que eles interajam com o ambiente virtual e respondam às mudanças e estímulos do jogo. Vamos descrever de forma detalhada sobre esses componentes:

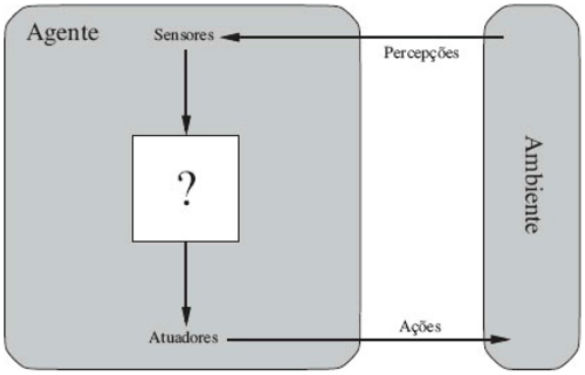

Figura 2.1: Agente interagindo com o ambiente [5]

Sensores

Visão: Os sensores de visão permitem que os agentes percebam o mundo virtual por meio de imagens. Essas imagens podem ser representadas como matrizes de pixels, e os agentes podem usar técnicas de processamento de imagens para identificar objetos, obstáculos, aliados e inimigos no ambiente.

Audição: Os sensores de audição permitem que os agentes percebam sons e efeitos sonoros no jogo. Isso é especialmente útil para detectar a aproximação de outros agentes ou inimigos, bem como para reagir a eventos sonoros relevantes no ambiente.

Proximidade: Os sensores de proximidade permitem que os agentes detectem a presença e a distância de outros objetos e entidades no ambiente. Eles podem ser utilizados para evitar colisões com obstáculos, aproximar-se ou afastar-se de outros agentes, e para reagir a eventos próximos.

Toque ou Colisão: Alguns jogos podem implementar sensores de toque ou colisão que permitem que os agentes percebam quando colidem com objetos ou outras entidades no ambiente. Isso pode ser usado para evitar obstáculos ou para acionar respostas específicas em situações de colisão.

Atuadores

Movimento: Os atuadores de movimento permitem que os agentes se desloquem pelo ambiente virtual. Eles podem controlar a velocidade, direção e aceleração dos agentes, permitindo que eles naveguem pelo terreno do jogo de forma realista e interajam com outros objetos e personagens.

Ações: Os atuadores de ações permitem que os agentes realizem ações específicas no jogo. Isso pode incluir a execução de ataques, saltos, interações com objetos ou outros personagens, e qualquer outra ação específica do jogo.

Comunicação: Alguns jogos podem incluir atuadores de comunicação que permitem que os agentes se comuniquem com outros personagens ou com o jogador. Isso pode envolver diálogos, respostas verbais ou gestos para expressar emoções ou intenções.

Tomada de Decisão: Os atuadores de tomada de decisão permitem que os agentes escolham ações com base em

suas percepções e raciocínio. Isso envolve selecionar a melhor ação ou estratégia para alcançar seus objetivos ou responder a situações específicas do jogo.

2.1.2 Tipos de agentes

A integração da Inteligência Artificial (IA) em jogos revolucionou a forma como os personagens e elementos interagem dentro dos ambientes virtuais. Uma parte crucial dessa revolução são os agentes de IA, que são responsáveis por conferir comportamentos autônomos aos personagens virtuais. Existem quatro tipos principais de agentes que desempenham papéis distintos na tomada de decisões e na criação de experiências envolventes em jogos.

Neste contexto, exploraremos os quatro tipos de agentes: Agentes Reativos Simples, Agentes Reativos Baseados em Modelos, Agentes Baseados em Objetivos e Agentes Baseados na Utilidade. Cada um desses tipos traz consigo uma abordagem única para a tomada de decisões, permitindo que os personagens virtuais reajam ao ambiente e aos desafios do jogo de maneira mais complexa e realista.

Nesta seção, veremos uma descrição detalhada de cada tipo de agente, entender como eles operam, suas aplicações práticas e exemplos que ilustram sua funcionalidade. Ao compreender esses tipos de agentes, poderemos apreciar melhor a riqueza e a diversidade de comportamentos que podem ser criados nos jogos por meio da IA, proporcionando experiências de jogo cada vez mais cativantes e imersivas.

Cada tipo de agente tem suas vantagens e limitações, e a escolha do tipo de agente depende do contexto do jogo e das decisões que os desenvolvedores desejam que os agentes tomem. Em jogos mais complexos, frequentemente é necessário usar uma combinação desses tipos de agentes para criar comportamentos mais realistas e desafiadores.

Agentes Reativos Simples

Os agentes reativos simples são os tipos mais básicos de agentes de IA em jogos. Eles tomam decisões com base apenas nas condições atuais do ambiente, sem considerar o histórico ou prever o futuro. Esses agentes reagem a estímulos do ambiente de maneira direta, seguindo regras pré-programadas. São ideais para jogos que envolvem ações simples e não demandam uma tomada de decisão muito complexa.

Exemplo: Um jogo de corrida onde os carros inimigos mudam de faixa para evitar colisões com outros carros ou obstáculos. Eles podem simplesmente trocar de faixa se houver um obstáculo à frente, sem levar em consideração estratégias de longo prazo.

Algoritmo 1: Agente Reactivo Simples - Esquivando Obstáculos

```
1  Inicializar posição do agente pos_agente;
2  Inicializar posição do obstáculo pos_obstaculo;
3  while Jogo estiver em execução do
4  |  Calcular a distância dist entre pos_agente e pos_obstaculo;
5  |  if dist é menor que um valor limite then
6  |  |  Mover o agente para a direita;
7  |  end if
8  |  else
9  |  |  Mover o agente para a frente;
10 |  end if
11 |  Atualizar pos_agente;
12 |  Atualizar pos_obstaculo;
13 end while
```

Agentes Reativos Baseados em Modelos

Estes agentes também reagem ao ambiente atual, mas eles usam um modelo interno do mundo para tomar decisões. Esse modelo é uma representação simplificada das relações entre as diferentes partes do ambiente. Embora ainda sejam reativos, eles podem ter uma visão mais ampla das consequências de suas ações.

Exemplo: Em um jogo de xadrez, um agente reativo baseado em modelo pode avaliar as posições das peças no tabuleiro e, em seguida, escolher um movimento que minimize o risco de perder uma peça valiosa no próximo turno.

Algoritmo 2: Agente Baseado em Modelo - Simulação de Tráfego

```
Inicializar posição do agente pos_agente;
Inicializar modelo do tráfego modelo_trafego;
while Jogo está em execução do
    acao = Consultar modelo do tráfego para pos_agente;
    if acao é "frear" then
        Frear o agente;
    end if
    else if acao é "acelerar" then
        Acelerar o agente;
    end if
    else
        Manter a velocidade do agente;
    end if
    Atualizar pos_agente;
    Atualizar modelo_trafego;
end while
```

Algoritmo 3: Agente Baseado em Modelo - Tomada de Decisões no Xadrez

```
Inicializar estado atual do tabuleiro estado_tabuleiro;
while Jogo não acabou do
    Calcular avaliação heurística avaliacao do estado_tabuleiro;
    if avaliacao indica uma situação vantajosa then
        Escolher o movimento que maximiza a vantagem;
    end if
    else
        Escolher um movimento aleatório;
    end if
    Realizar o movimento escolhido no estado_tabuleiro;
end while
```

Agentes Baseados em Objetivos

Os agentes baseados em objetivos definem metas específicas que desejam alcançar e tomam decisões para atingir esses objetivos. Eles avaliam o ambiente e escolhem ações que os aproximem dos resultados desejados. Esses agentes podem mudar de objetivos ao longo do tempo, adaptando-se a situações em evolução.

Exemplo: Em um jogo de estratégia, um agente baseado em objetivos pode ter o objetivo de coletar recursos para construir um exército. Ele tomará decisões para coletar recursos, construir estruturas e treinar unidades militares, tudo com o objetivo final de vencer a batalha.

Algoritmo 4: Agente Baseado em Objetivos - Coleta de Recursos em Jogo de Estratégia

```
1  Inicializar posição do agente pos_agente;
2  Definir objetivo objetivo como "coletar recursos";
3  while Jogo está em execução do
4      if objetivo é "coletar recursos"e há recursos próximos then
5          Mover o agente para o recurso mais próximo;
6          Coletar recurso;
7          Atualizar quantidade de recursos coletados;
8          if Quantidade de recursos coletados atingiu um limite then
9              Definir objetivo como "construir estrutura";
10         end if
11     end if
12     else if objetivo é "construir estrutura"e há local adequado then
13         Mover o agente para o local adequado;
14         Construir estrutura;
15         Atualizar estado do jogo e quantidade de recursos;
16         Definir objetivo como "coletar recursos";
17     end if
18     else
19         Explorar o mapa ou esperar;
20     end if
21 end while
```

Agentes Baseados na Utilidade

Esses agentes avaliam as ações possíveis com base na utilidade ou valor associado a cada ação. A utilidade é uma medida subjetiva de quão desejável é um determinado resultado. Os agentes calculam a utilidade de várias ações possíveis e escolhem a ação que maximize sua utilidade esperada. Isso permite que os agentes tomem decisões mais complexas, considerando trade-offs entre diferentes objetivos.

Exemplo: Em um RPG, um personagem pode decidir qual arma usar com base não apenas no dano que causa, mas também em fatores como a distância do inimigo, os recursos disponíveis e a saúde do personagem. O agente calcula a utilidade de cada ação possível, considerando diversos fatores, para escolher a ação que maximiza suas chances de sucesso.

Algoritmo 5: Agente Baseado na Utilidade - Escolha da Melhor Arma em Jogo de RPG

```
Inicializar valores de dano dano, saúde do personagem saude, distância do inimigo distancia;
Inicializar arma escolhida como nenhuma;
Inicializar valor máximo de utilidade como −∞;
foreach arma no inventário do
    Calcular utilidade da arma utilidade = f(dano, saude, distancia);
    if utilidade > valor máximo de utilidade then
        Atualizar arma escolhida como a arma atual;
        Atualizar valor máximo de utilidade para utilidade;
    else
        Continuar para a próxima arma;
    end if
end foreach
if arma escolhida é nenhuma then
    Esperar ou escolher ação padrão;
else
    Usar a arma escolhida;
end if
```

2.2 Mundo do aspirador de pó

O problema do mundo do aspirador de pó é um problema clássico da inteligência artificial que envolve um aspirador de pó que deve limpar uma sala dividida em quadrantes, coletando a sujeira em cada quadrante [5]. O objetivo é fazer com que o aspirador limpe toda a sala de forma eficiente.

O ambiente é geralmente representado como uma matriz bidimensional e o aspirador de pó está localizado em uma posição específica da matriz. O aspirador pode se mover para cima, para baixo, para a esquerda ou para a direita e limpar o quadrante onde está posicionado.

Vamos detalhar mais sobre as posições, ações e percepções do mundo do aspirador de pó:

1. Posições:
   O mundo é representado como uma matriz bidimensional, onde cada célula pode estar em um dos seguintes estados: 0 Representa uma célula limpa; -1 Representa uma célula suja; 1 Representa a posição atual do aspirador de pó.
2. Ações:
   O aspirador de pó pode realizar as seguintes ações: Mover-se para cima, baixo, esquerda, direita e Limpar a célula em que está posicionado.

A seguir, podemos visualizar o pseudocódigo 10 que modela um agente randômico que faz escolhas de ação aleatórias entre mover-se e limpar. O agente se move aleatoriamente pelo ambiente, limpando as células sujas quando realiza a ação de limpar. Esse processo continua até que todas as células sujas sejam limpas, momento em que o algoritmo termina. Isso reflete a abordagem básica de um agente aleatório no problema do aspirador de pó, onde o agente não segue uma estratégia lógica, mas apenas toma decisões aleatórias.

Algoritmo 6: Agente Randômico

```
Data: Ambiente ambiente[linhas][colunas], células sujas no ambiente
Data: Posição do agente (linha, coluna)
while Houver células sujas no ambiente do
    (linha, coluna) ← posição do agente;
    acao ← escolha aleatória entre mover para cima, baixo, esquerda ou direita;
    if acao é mover then
        Atualize a posição do agente com base na acao;
    end if
    else if acao é limpar then
        Limpe a célula em (linha, coluna);
    end if
end while
```

De forma detalhada o pseudocódigo 10 funciona da seguinte forma:

1. **Configurações Iniciais:**
   O algoritmo começa com uma seção de configurações iniciais onde são definidos os dados iniciais do agente e do ambiente. Isso inclui a posição inicial do agente, a matriz do ambiente que contém informações sobre as células limpas e sujas, e a informação sobre as células sujas no ambiente.
2. **Laço Principal:**
   O algoritmo entra em um laço principal que continua até que não haja mais células sujas no ambiente.
3. **Obtenção da Posição Atual:**
   A posição atual do agente (linha e coluna) é obtida para que possamos usar essas informações para realizar ações.

4. **Escolha Aleatória de Ação:**
   O agente escolhe aleatoriamente entre duas ações possíveis: mover-se ou limpar. A ação de mover-se pode ser em uma das quatro direções: para cima, para baixo, para a esquerda ou para a direita.
5. **Atualização da Posição (se for um movimento):**
   Se a ação escolhida for "mover", a posição do agente é atualizada de acordo com a direção escolhida. Isso simula o movimento do agente dentro do ambiente.
6. **Limpeza da Célula (se for uma ação de limpeza):**
   Se a ação escolhida for "limpar", o agente realiza uma ação de limpeza na célula onde está posicionado, marcando essa célula como limpa.
7. **Fim do Laço:**
   O laço continua até que não haja mais células sujas no ambiente. Isso significa que o agente continuará a se mover aleatoriamente e limpar as células sujas até que todas estejam limpas.

2.3 Jogos soma zero

Jogos de soma zero são um conceito importante na teoria dos jogos, um ramo da matemática e da economia que estuda estratégias de tomada de decisão em situações de conflito e competição. Um jogo de soma zero é um tipo específico de jogo em que o ganho de um jogador é exatamente igual à perda do outro jogador, ou seja, a soma dos ganhos e perdas é sempre zero. Os jogos de soma zero são frequentemente usados como um modelo básico para analisar estratégias em jogos competitivos. A teoria dos jogos também se expande para jogos não de soma zero, onde as recompensas não somam zero, ou para cenários em que mais de dois jogadores estão envolvidos.

Aqui estão os principais aspectos dos jogos de soma zero:

1. **Definição Básica:**
   Um jogo de soma zero é definido pela característica de que os ganhos e perdas dos jogadores envolvidos somam sempre zero. Isso significa que tudo o que um jogador ganha é exatamente o que o outro jogador perde.
2. **Matriz de Pagamentos:**
   Os jogos de soma zero são frequentemente representados por uma matriz de pagamentos, também conhecida como matriz de utilidades. Nessa matriz, as entradas indicam as recompensas que um jogador obtém em relação ao outro, dependendo das combinações de ações escolhidas por ambos.
3. **Vitória e Derrota:**
   Nos jogos de soma zero, uma vitória para um jogador é sempre uma derrota para o outro jogador, e vice-versa. Isso cria uma relação competitiva direta entre os jogadores.
4. **Estratégias Misturadas**:
   As estratégias misturadas, onde os jogadores escolhem ações com probabilidades específicas, também podem ser aplicadas aos jogos de soma zero. Isso introduz uma dimensão de aleatoriedade e torna a tomada de decisão mais complexa.
5. **Exemplos de Jogos de Soma Zero:**
   - Xadrez: Em xadrez, um jogador ganha quando coloca o oponente em xeque-mate, e o outro jogador perde na mesma medida. As regras estritas e as recompensas opostas exemplificam um jogo de soma zero.
   - Damas: Semelhante ao xadrez, as damas também são um jogo de soma zero. Cada peça conquistada por um jogador é uma peça perdida para o adversário.
   - Poker: Jogos de poker, como Texas Hold'em, possuem apostas e potenciais ganhos que são redistribuídos entre os jogadores. Um jogador ganha o valor que o outro jogador perde, somando zero no total.
6. **Utilização na Teoria dos Jogos:**

Os jogos de soma zero são frequentemente usados como um ponto de partida para analisar estratégias mais complexas em jogos. Eles permitem entender a lógica básica de tomada de decisões competitivas e formar a base para estratégias mais elaboradas.

Em resumo, os jogos de soma zero são uma representação simples e essencial de competição e conflito em situações estratégicas. Eles fornecem insights valiosos sobre como os jogadores tomam decisões para maximizar seus próprios interesses enquanto tentam prejudicar os interesses do adversário.

2.3.1 Minimax

O algoritmo Minimax é um dos pilares fundamentais da inteligência artificial, especialmente na área de jogos de estratégia, como xadrez, damas e Go. Ele é usado para tomar decisões em cenários competitivos onde dois jogadores alternam suas ações, tentando maximizar suas vantagens e minimizar as vantagens do oponente. Minimax é empregado em jogos competitivos chamados, **jogos soma zero**.

O objetivo principal do algoritmo Minimax é encontrar a melhor jogada para um jogador em um jogo de soma zero, onde o ganho de um jogador é a perda do outro. Ou seja, o total de pontos ganhos e perdidos por ambos os jogadores somam zero. O Minimax assume que o oponente sempre fará a melhor jogada possível para si mesmo.

De forma geral, o algoritmo funciona da seguinte forma:

1. **Árvore de Jogo:**
   O algoritmo Minimax opera em uma árvore de jogo, onde cada nó representa uma posição no jogo e as arestas conectam as posições possíveis após cada movimento.
2. **Maximizador e Minimizador:**
   No contexto de um jogo de dois jogadores, como xadrez, há um jogador "Maximizador"e um jogador "Minimizador". O Maximizador procura maximizar seu próprio ganho, enquanto o Minimizador tenta minimizar o ganho do Maximizador.
3. **Recursão:**
   O algoritmo Minimax é recursivo e começa a partir da posição atual do jogo. Ele gera todas as posições possíveis após uma jogada do Maximizador e, em seguida, todas as posições possíveis após uma jogada do Minimizador.
4. **Avaliação do Tabuleiro:**
   Quando a profundidade máxima da árvore é atingida ou o jogo acaba (alcançando um estado final, como vitória ou empate), é feita uma avaliação do tabuleiro para determinar o valor da posição para o Maximizador.
5. **Atribuição de Valores:**
   Cada nó terminal (estado final) é atribuído a um valor, geralmente 1 para vitória do Maximizador, -1 para vitória do Minimizador e 0 para empate. Nos nós intermediários, os valores são propagados para cima (maximizados ou minimizados) dependendo se é um nó de Maximizador ou Minimizador.
6. **Escolha da Melhor Jogada:**
   Depois que a recursão é realizada em toda a árvore, o Maximizador escolhe o ramo que leva ao maior valor, enquanto o Minimizador escolhe o ramo que leva ao menor valor. Essa escolha é feita considerando as ações possíveis e as avaliações atribuídas a elas.

O algoritmo Minimax é eficaz para jogos de soma zero com um espaço de busca razoável, mas tem algumas limitações em termos de eficiência computacional. Isso levou ao desenvolvimento de variações e otimizações, como o algoritmo Alfa-Beta Pruning, que reduz o número de ramos a serem explorados, tornando o algoritmo mais rápido sem alterar as decisões tomadas.

Algoritmo 7: Algoritmo Minimax

```
1  Minimax(estado, jogador)
2  if estado é um estado terminal then
3  |  return valor do estado;
4  end if
5  if jogador é o Maximizador then
6  |  return max_{a em ações} Minimax(AplicarAcao(estado, a), ProximoJogador(jogador));
7  end if
8  else
9  |  return min_{a em ações} Minimax(AplicarAcao(estado, a), ProximoJogador(jogador));
10 end if
```

2.4 Movimentação de Agentes em Ambientes

No mundo dos jogos e simulações interativas, a capacidade de criar agentes virtuais que se movem e interagem realisticamente com o ambiente é essencial para proporcionar uma experiência envolvente e imersiva. A movimentação de agentes refere-se ao conjunto de técnicas e estratégias usadas para controlar o deslocamento de personagens, NPCs (personagens não jogáveis) e outros elementos autônomos em um ambiente virtual. Essa área desafia os desenvolvedores a criar movimentos naturais e realistas, ao mesmo tempo que fornece controle e previsibilidade.

Imagine um jogo em que um jogador controla um explorador em uma floresta densa, ou um ambiente de simulação onde veículos autônomos navegam por uma cidade virtual. A movimentação desses agentes define como eles percorrem o espaço, evitam obstáculos, tomam decisões e interagem com outros elementos do ambiente.

Ao longo desta seção, exploraremos algumas técnicas e conceitos relacionados à movimentação de agentes em ambientes virtuais. Desde algoritmos básicos até abordagens avançadas, você aprenderá a criar movimentações suaves, trajetórias realistas e comportamentos complexos para os agentes do seu projeto.

Discutiremos tópicos como

- **Algoritmos de Navegação Simples:** Exploraremos algoritmos básicos, como o movimento em direção a um ponto alvo e a movimentação seguindo caminhos pré-definidos. Essas técnicas são a base para compreender os princípios da movimentação de agentes.
- **Desvio de Obstáculos:** A movimentação em ambientes complexos exige que os agentes evitem obstáculos. Abordaremos estratégias como "steering behaviors"(comportamentos de direção), potenciais de campo e grades de navegação para criar movimentos fluidos e sem colisões.
- **Comportamentos Emergentes:** Veremos como combinar comportamentos simples para criar movimentos complexos e naturalistas. Técnicas como "flocking"(aglomeração), "wandering"(vagar) e "seek and flee"(procurar e fugir) permitem que agentes se comportem de maneira realista em grupo.
- **Navegação em Terrenos:** Em ambientes com terrenos acidentados, é importante que os agentes possam navegar suavemente. Vamos explorar algoritmos de navegação em terrenos, que levam em consideração elevações e inclinações.
- **Planejamento de Trajetória:** Para movimentos complexos, como pilotagem de veículos ou caminhos detalhados, discutiremos técnicas de planejamento de trajetória, que permitem aos agentes escolherem rotas eficientes e seguras.
- **Inteligência Artificial e Tomada de Decisão:** Além de simples movimentos, abordaremos como os agentes podem tomar decisões inteligentes, reagir a estímulos do ambiente e interagir de forma dinâmica com o jogador ou outros agentes.

2.4.1 Movimento em Direção a um Ponto de Destino

Os fundamentos da movimentação muitas vezes começam com algoritmos de navegação simples. Um exemplo é o movimento em direção a um ponto de destino. Imagine um personagem em um jogo que precisa se mover até um ponto específico no mapa. Para fazer isso, o algoritmo calcula a direção do ponto atual até o ponto de destino e move gradualmente o personagem nessa direção. Essa abordagem é direta, mas pode não levar em consideração obstáculos.

Vamos explorar o conceito de movimento, gerando destinos aleatórios para explorar um ambiente sem obstáculos. Imagine um personagem autônomo em um jogo explorando um mundo 2D, escolhendo aleatoriamente onde se mover (Figura 2.2).

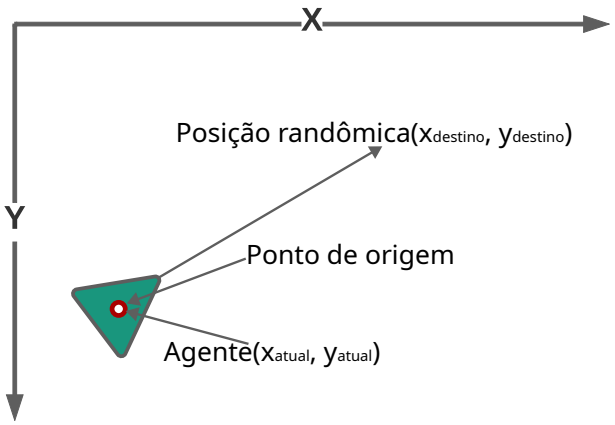

Figura 2.2: Movimento randômico

- **Passo 1: Inicialização do Ambiente:**
  Definimos as posições iniciais do personagem e de seu destino inicial. Suponha que o personagem comece nas coordenadas (0, 0) e seu destino seja inicializado aleatoriamente dentro de um determinado intervalo em uma cena cuja dimensão é $1280 \times 720px$, então, $X = range(0, 1280)$ e $Y = range(0, 720)$.
- **Passo 2: Movimentação do Agente em Direção ao Destino**
  A movimentação do agente autônomo é executada enquanto o ambiente estiver em execução. O algoritmo é executado enquanto enquanto a distância entre a posição atual do agente e a posição de destino for maior que uma tolerância ε). O valor de tolerância garante que o agente se mova até estar próximo o suficiente do destino. Importante, em algumas *engines* de desenvolvimento de jogos devemos considerar o ponto de origem do agente em relação a dimensão do *sprite*.
  Por exemplo, considere um *sprite* cuja dimensão seja $32 \times 32px$, então, o intervalo do ponto de origem nas coordenadas é $x = [0, 32]$ e $y = [0, 32]$. A Figura 2.3 demonstra o ponto de origem do *sprite* que foi definido no centro da imagem. Neste exemplo, o valor de tolerância pode ser igual $\varepsilon = 18$), considerando $2px$ a frente em relação a largura e altura, independente da direção que o agente se mova. Desta forma, iremos considerar que o agente chegou o mais próximo possível do destino. Caso o ponto de origem seja $x = 0, y = 0$, então, o valor de tolerância será $\varepsilon = 34$). Neste exemplo, estamos considerando que não estamos utilizando a equação do espaço euclidiano[1] para o calculo, e sim, comparando diretamente as posições X e Y do agente no ambiente.

[1] https://pt.wikipedia.org/wiki/Espaço_euclidiano

A tolerância ideal é o valor limite no qual consideramos que o agente chegou ao destino. Isso evita que o agente fique oscilando infinitamente ao redor do destino devido a imprecisões. O valor da tolerância ideal depende do sistema e da escala utilizado na *engine*, mas um valor típico pode ser em torno de 0.1 unidades caso utilize a equação do espaço euclidiano para realizar o calculo de distância entre dois pontos.

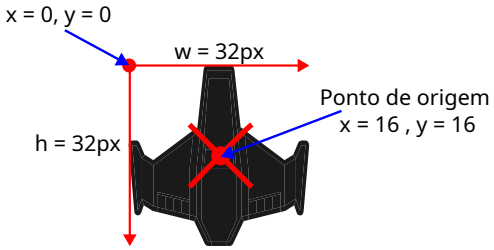

Figura 2.3: Sprite $32 \times 32px$ - Ponto de Origem

- **Passo 3: Verificação de Chegada ao Destino**
  Uma vez que a distância entre a posição atual do agente e a posição de destino é menor ou igual à tolerância ε, isso indica que o agente chegou ao destino. Neste ponto, o agente para de se mover.
- **Passo 4: Geração de Nova Posição Alvo**
  Após chegar ao destino, uma nova posição alvo é gerada. Isso inicia o próximo ciclo de movimentação em direção a uma nova posição.
- **Passo 5: Movimentação continua**
  O loop continua, gerando novas posições de destino e repetindo o processo de movimentação sempre que o agente chega a um destino.

A seguir o pseudocódigo 2.4 demonstra a movimentação de um agente gerando posições alvo aleatórias. Importante citar que, para o calculo da distância, estamos utilizando a equação euclidiana. Você pode ver um exemplo completo utilizando *engine* GameMaker Studio na seção **??**.

Neste exemplo, o pseudocódigo (Figura 2.4) descreve um algoritmo que move um agente em direção a um ponto de destino. Vamos explicar o pseudocódigo:

1. **Atributos Iniciais:**
   - **targetX:** A coordenada X do alvo para o qual o agente se moverá.
   - **targetY:** A coordenada Y do alvo para o qual o agente se moverá.
   - **threshold:** A distância mínima que o agente deve estar do alvo antes de escolher um novo alvo aleatório.
   - **speed:** A velocidade com que o agente se move em direção ao alvo.
2. **Método start():** Este método é chamado quando o jogo é iniciado. Ele gera uma posição alvo inicial definindo **targetX** e **targetY** como coordenadas aleatórias.
3. **Método update():** Este método é chamado enquanto o jogo está em execução, geralmente a cada quadro. Ele calcula a distância entre a posição atual do agente e o alvo atual usando uma função hipotética point_**distance(x, y, targetX, targetY)** e armazena o resultado em **_dist**.
   - Se a distância **_dist** for maior ou igual ao limite **threshold**, isso significa que o agente não está próximo o suficiente do alvo.
   - Calcula o ângulo **_direction** em direção ao alvo usando uma função hipotética **point_direction(x, y, targetX, targetY)**.

```
class RandomAgent:

    x = 0 #Posição na coordena X do agente
    y = 0 #Posição na coordena Y do agente
    targetX   = 0  #Coordenada X do alvo
    targetY   = 0  #Coordenada Y do alvo
    threshold = 10 #Limiar, distância mínima antes de trocar de posição
    speed     = 8  #velocidade de movimento do agente

    #Chamado ao iniciar o jogo
    def start(self):
        #Gera uma posição alvo inicial
        self.targetX, self.targetY = self.random_target()

    #Chamada enquanto o jogo estiver em execução
    def update(self):
        #Calcular a distância entre o Agente e o alvo
        _dist = self.point_distance(self.x, self.y, self.targetX, self.targetY)
        #se chegou perto do jogador muda de estado
        if _dist >= self.threshold:
            #Calcula o angulo de direção do agente
            _direction = self.point_direction(self.x, self.y, self.targetX, self.targetY)
            #Move o agente até o ponto de destino e atualiza sua posição atual
            self.x, self.y = self.move_towards_point(self.targetX, self.targetY,
            _direction, self.speed)
        else:
            self.targetX, self.targetY = self.random_target()
```

Figura 2.4: Movimentação Contínua do Agente

- Move o agente em direção ao ponto de destino **(targetX, targetY)** usando uma função hipotética **move_towards_point(targetX, targetY, _direction, speed)**. Isso faz com que o agente se mova em direção ao alvo com a velocidade especificada **speed**.
- Se o agente estiver próximo o suficiente do alvo (a distância for menor que **threshold**), ele escolherá um novo alvo aleatório, o que é feito de forma hipotética chamando a função random_target(). Isso faz com que o agente continue a se mover aleatoriamente pelo cenário.

2.4.2 Movimentação de Agentes Utilizando Caminhos Pré-Definidos em Ambientes sem Obstáculos

Em muitas situações, a movimentação suave e previsível de agentes é essencial, mesmo quando não há obstáculos para contornar. A movimentação utilizando caminhos pré-definidos é uma técnica poderosa para criar trajetórias controladas e eficientes em ambientes sem obstáculos. Essa abordagem é amplamente empregada em jogos, animações e simulações onde o movimento precisa ser precisa e repetível.

- **Criando Trajetórias Precisas**
  Ao utilizar caminhos pré-definidos, os desenvolvedores podem projetar trajetórias precisas que garantem que os agentes sigam um caminho específico. Isso é particularmente útil em situações em que o movimento deve ser sincronizado com eventos específicos ou onde é necessário evitar desvios indesejados.
- **Animação e Cinematografia Virtual**

A movimentação com caminhos pré-definidos é frequentemente usada em animações e cinematografia virtual para criar movimentos de câmera suaves e controlados. Isso permite que os criadores definam exatamente como a câmera se moverá ao redor de um ambiente ou objeto, garantindo que os movimentos de câmera sejam consistentes e fluidos.

- **Jogos Plataforma e Sidescrollers**
  Em jogos de plataforma e sidescrollers, a movimentação com caminhos pré-definidos é uma técnica valiosa para criar sequências de ação e exploração. Por exemplo, um personagem pode ser movido automaticamente ao longo de uma sequência de plataformas em alta velocidade, permitindo que o jogador se concentre na interação e nos desafios do ambiente.
- **Eficiência em Simulações e Treinamento**
  Em simulações e treinamento, a movimentação previsível e controlada é fundamental para criar cenários repetíveis e mensuráveis. Por exemplo, em simulações de treinamento de voo, as aeronaves podem seguir trajetórias pré-definidas para garantir que os pilotos em treinamento experimentem situações específicas de voo.
- **Exemplo Prático: Animação de Personagens em um Game Sidescroller**
  Imagine um jogo sidescroller onde o jogador controla um personagem enquanto ele corre automaticamente ao longo de uma trilha. A movimentação com caminhos pré-definidos permitiria que o personagem seguisse uma trajetória precisa, enquanto o jogador foca na interação e nas habilidades do personagem.

Em resumo, a movimentação com caminhos pré-definidos é uma ferramenta versátil para criar movimentos precisos e controlados em ambientes sem obstáculos. Essa técnica oferece consistência, previsibilidade e eficiência em uma variedade de contextos, desde animações cinematográficas até jogos e simulações. Ao incorporar movimentação pré-definida, os desenvolvedores podem guiar a experiência do usuário e criar movimentos convincentes que se encaixam perfeitamente nas necessidades do projeto.

2.5 Maquinas de estado finito

Uma Máquina de Estado Finito (MEF, do inglês, *Finite State Machines (FSM)*), também conhecida como Autômato Finito, é um modelo matemático que descreve o comportamento de um sistema que pode estar em diferentes estados e que muda de estado em resposta a eventos externos ou ações internas. É amplamente utilizado para modelar comportamentos sequenciais em sistemas, como jogos, sistemas de controle, linguagens de programação e muitas outras aplicações.

Uma FSM consiste em um conjunto finito de estados, uma lista de eventos ou condições que podem causar a transição entre esses estados e uma série de ações associadas a essas transições. Cada estado representa uma situação ou condição específica em que o sistema pode estar, e as transições determinam como o sistema responde a determinados eventos, movendo-se de um estado para outro.

Aqui está uma descrição detalhada dos principais componentes de uma Máquina de Estado Finito:

- **Estados (States):** São as situações ou condições em que o sistema pode estar. Cada estado representa um comportamento ou configuração específica do sistema. Por exemplo, em um jogo, os estados poderiam ser "Menu Principal", "Jogando", "Pausado"e "Game Over".
- **Eventos (Events):** São as entradas ou gatilhos que podem causar uma mudança de estado. Um evento pode ser uma ação do usuário, uma condição específica no ambiente ou qualquer outro estímulo que possa afetar o sistema. Exemplos de eventos em um jogo seriam "Clique do mouse", "Tecla pressionada"ou "Tempo esgotado".
- **Transições (Transitions):** São as regras que determinam como o sistema muda de um estado para outro em resposta a um evento. Cada transição é geralmente definida por um estado de origem, um evento e um estado

de destino. Por exemplo, quando o jogador pressiona a tecla "Espaço"no estado "Jogando", o sistema pode fazer a transição para o estado "Pausado".

- **Ações (Actions):** São as atividades ou operações executadas quando uma transição ocorre. Essas ações podem incluir alterações no estado do sistema, ativação de eventos adicionais, atualizações visuais, reprodução de sons, entre outras coisas. Em um jogo, uma ação poderia ser aumentar a pontuação do jogador quando ele coleta um item.

2.5.1 Exemplo: Máquina de Estado Finito em um Semáforo

Vamos usar o exemplo de um semáforo para ilustrar uma Máquina de Estado Finito simples:

- Estados: "Vermelho", "Amarelo"e "Verde".
- Eventos: "Tempo Esgotado"(quando o tempo de um estado acaba).
- Transições:
  -
  - No estado "Vermelho", a transição ocorre quando "Tempo Esgotado"para o estado "Verde".
  - No estado "Verde", a transição ocorre quando "Tempo Esgotado"para o estado "Amarelo".
  - No estado "Amarelo", a transição ocorre quando "Tempo Esgotado"para o estado "Vermelho".
- Ações:
  - Quando a transição ocorre do estado "Vermelho"para o estado "Verde", a ação pode ser acender a luz verde.
  - E assim por diante para as outras transições.

Essa Máquina de Estado Finito descreve o ciclo de funcionamento de um semáforo, onde as mudanças de estado são acionadas pelo tempo esgotado em cada estado.

As Máquinas de Estado Finito são uma ferramenta poderosa para modelar comportamentos sequenciais e lógicos de sistemas, permitindo uma representação clara e estruturada do funcionamento do sistema em diferentes situações. Elas são amplamente utilizadas no desenvolvimento de jogos, automação industrial, sistemas embarcados e em muitas outras áreas.

2.6 IA e a FSM

Dentro do campo da Inteligência Artificial (IA), a FSM é um conceito fundamental, embora não seja tão amplamente utilizado quanto em outras áreas, como a engenharia de software tradicional. A FSM na IA é usada para modelar o comportamento de agentes autônomos ou personagens virtuais em ambientes virtuais ou jogos (Figura 2.9).

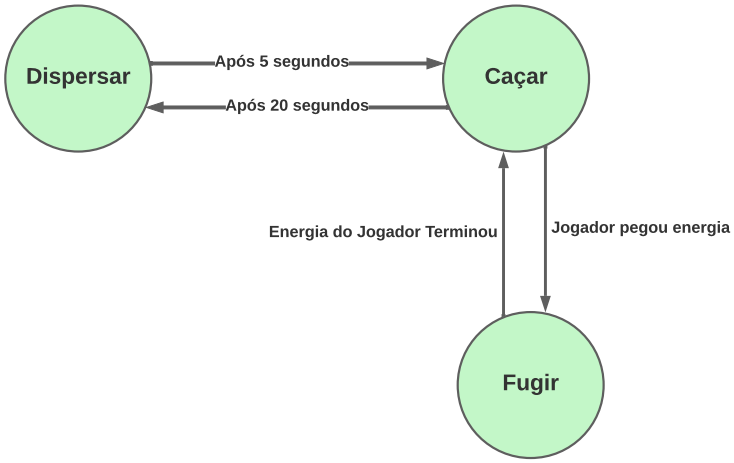

Figura 2.5: Máquina de estados finitos.

Embora as Máquinas de Estado Finito possam ser consideradas simples em comparação com abordagens mais complexas de IA, como redes neurais profundas, elas ainda têm seu lugar e aplicabilidade em algumas situações:

- Comportamento Previsível: Em jogos ou simulações, os personagens precisam muitas vezes seguir padrões comportamentais previsíveis. A FSM é adequada para modelar esses comportamentos, onde as transições de estado são baseadas em eventos específicos que ocorrem no ambiente ou nas interações do jogador.
- Personagens de Fundo: Personagens secundários ou elementos de fundo em um jogo muitas vezes não requerem inteligência complexa. Uma MEF pode ser usada para criar comportamentos simples, como patrulhamento ou movimento aleatório.
- Sistemas de Controle: MEFs podem ser usadas para criar sistemas de controle de personagens em jogos ou até mesmo para controlar elementos de IA em sistemas industriais, onde os comportamentos podem ser predefinidos e repetitivos.
- Comportamentos Específicos: Quando você precisa que um personagem execute comportamentos específicos em momentos específicos, uma MEF pode ser uma escolha adequada. Por exemplo, um personagem que executa uma animação de "dança"em resposta a um gatilho específico.
- Prototipagem Rápida: MEFs são relativamente fáceis de implementar e entender. Em situações onde você precisa de um comportamento de personagem rapidamente para fins de prototipagem, uma MEF pode ser uma escolha eficaz.

É importante notar que as Máquinas de Estado Finito têm limitações em comparação com abordagens mais avançadas de IA, como aprendizado de máquina ou redes neurais. Elas são mais adequadas para situações onde os comportamentos são relativamente simples e previsíveis. Para simulações mais complexas e comportamentos mais adaptativos, outras abordagens de IA, como algoritmos genéticos ou aprendizado por reforço, podem ser mais apropriadas.

FSMs são mais utilizados em padrões chamados, *Go-To Game AI Patterns*, para desenvolver jogos que são relativamente simples de implementar, visualizar, e entender. Usando um simples *if/else*, nos podemos facilmente implementar uma FSM ([1]).

Em resumo, as Máquinas de Estado Finito podem ser uma ferramenta útil no campo da IA, especialmente quando se trata de criar comportamentos específicos e previsíveis para personagens virtuais em jogos e simulações.

2.6.1 Exemplo básico: FSM do tanque de guerra

Baseado no livro [1], vamos criar uma FSM para simular o comportamento básico de um tanque de guerra. Antes de prosseguir, é muito importante ao modelar um FSM, é importante definir cuidadosamente as transições, as regras e os estados para garantir que o comportamento do seu sistema seja correto e eficiente. Aqui estão algumas dicas para criar esses elementos de forma eficaz:

1. **Definindo Estados:**
   Identifique os estados que representam diferentes comportamentos do seu sistema. Cada estado deve ser uma situação distinta em que o sistema pode estar. Mantenha os estados específicos e coesos. Evite estados muito genéricos que podem abranger muitos comportamentos diferentes. Use nomes claros e descritivos para os estados para facilitar a compreensão do seu FSM.
2. **Criando Transições:**
   Para cada estado, defina as condições que acionarão a transição para outro estado. As condições são geralmente baseadas em eventos, variáveis de estado ou outras informações relevantes. Certifique-se de que as condições sejam claras e bem definidas. Isso evitará comportamentos inesperados no sistema. Evite condições conflitantes. Se uma condição acionar mais de uma transição, isso pode levar a comportamentos indefinidos ou imprevisíveis.
3. **Estabelecendo Regras:**
   Determine como as transições são priorizadas. Se várias condições forem atendidas ao mesmo tempo, qual transição deve ocorrer primeiro? Isso é especialmente importante quando as condições podem se sobrepor. Defina como o sistema reage quando nenhuma das condições é atendida. Isso pode incluir manter o estado atual, retornar a um estado inicial ou executar uma ação padrão.
4. **Tratando de Estados Especiais:**
   Considere estados de espera ou inativos. Nem todos os estados precisam ter uma transição ativa o tempo todo. Alguns estados podem representar momentos em que o sistema está inativo ou esperando um evento específico. Crie um estado inicial. O estado inicial é o estado em que o sistema começa. Ele deve ser o ponto de partida lógico para o comportamento.
5. **Mantendo a Coerência:**
   Evite estados órfãos ou transições não utilizadas. Cada estado deve ser acessado de alguma forma, e cada transição deve ser ativada por alguma condição realista. Mantenha o FSM o mais simples possível. Muitos estados e transições podem tornar o sistema difícil de entender e manter.
6. **Testando e Depurando:**
   Teste todas as transições em diferentes cenários para garantir que elas ocorram como esperado. Use logs ou saídas de depuração para rastrear o fluxo de estados e transições enquanto o sistema está em execução.

O pseudocódigo (Figura 2.6) é uma implementação simplificada de uma Máquina de Estado Finito (FSM) para controlar o comportamento de um agente de tanque de guerra em um jogo. Vamos explicar o código passo a passo:

1. **Definindo os Estados (enum STATE):** Aqui, estamos definindo três estados possíveis para o agente do tanque de guerra: PATROL (patrulha), ATTACK (ataque) e CHASING (perseguição). Esses estados são identificados por números inteiros (1, 2 e 3) para facilitar a manipulação.
2. **Classe WarTankAgent:** Esta é uma classe que representa o agente do tanque de guerra.
3. **Estado Atual (current_state):** A variável **current_state** é usada para rastrear o estado atual do agente. Inicialmente, o agente começa no estado de **PATROL** (patrulha).
4. **Método update():** Este é um método que é chamado regularmente pela engine para atualizar o comportamento do agente.
5. **Cálculo da Distância (_dist):** Calcula a distância entre a posição do agente (x, y) e a posição do jogador $(player.x, player.y)$. Isso é usado para determinar a proximidade do jogador ao agente. É importante notar que a função **point_distance** mencionada no pseudocódigo é uma função hipotética que calcula a distância entre dois pontos (no caso, a posição do agente e a posição do jogador). Em muitos motores de

```
#Definindo o nome para cada estado
class STATE(Enum):
    PATROL  = 1
    ATTACK  = 2
    CHASING = 3

class WarTankAgent:

    x = 0 #Posição na coordena X do agente
    y = 0 #Posição na coordena Y do agente

    #Chamado ao iniciar o jogo
    def start(self):
        #Definindo o estado inicial do agente
        self.current_state = STATE.PATROL

    #Chamada enquanto o jogo estiver em execução
    def update(self):
        #Calcular a distância entre o Agente e o Jogador
        _dist = self.point_distance(self.x, self.y, self.player.x, self.player.y)
        #Se chegou perto do jogador muda de estado
        if _dist <= 5:
            current_state = STATE.ATTACK
        elif _dist <= 10:
            current_state = STATE.CHASING
        else:
            current_state = STATE.PATROL

        #Executando as ações de acordo com cada estado
        if current_state == STATE.PATROL:
            self.ia_patrol()
        elif current_state == STATE.CHASING:
            self.ia_chasing(self.player)
        elif current_state == STATE.ATTACK:
            self.ia_fire(self.player)
```

Figura 2.6: FSM do Tanque de Guerra

jogos, essa função ou algo semelhante é fornecido como parte das funcionalidades básicas. Portanto, os desenvolvedores não precisam criar essa função do zero.

6. **Transições de Estado:** Com base na distância calculada, o agente decide se deve mudar de estado: Se a distância for menor ou igual a 5 unidades, o agente muda para o estado de **ATTACK** (ataque). Se a distância for maior que 5, mas menor ou igual a 10 unidades, o agente muda para o estado de **CHASING** (perseguição). Caso contrário, o agente volta para o estado de **PATROL** (patrulha).
7. **Switch-Case (switch(estado)):** Aqui, você está usando uma estrutura de controle **switch-case** para lidar com os diferentes estados do agente com base na variável **current_state**. Dependendo do estado atual, o agente executará diferentes funções de comportamento.
    - No estado de **PATROL**, a função **ia_patrol()** é chamada para controlar o comportamento de patrulha do agente.
    - No estado de **CHASING**, a função **ia_chasing(player)** é chamada para controlar o comportamento de

perseguição do agente em relação ao jogador.
- No estado de **ATTACK**, a função **ia_fire(player)** é chamada para controlar o comportamento de ataque do agente em relação ao jogador.

Este código representa uma estrutura básica de uma Máquina de Estado Finito (FSM) em que o agente pode mudar de estado com base em condições específicas e realizar diferentes ações com base em seu estado atual. É uma maneira eficaz de modelar o comportamento de personagens em jogos, permitindo reações dinâmicas às mudanças de situação. Note que as funções **ia_patrol()**, **ia_chasing()**, e **ia_fire()** devem ser definidas em outro lugar na classe **WarTankAgent** para descrever o comportamento real do agente em cada estado.

2.6.2 Go-To AI Patterns

O padrão "Go-To AI Patterns"(também conhecido como "Go-To Game AI Patterns") se refere a um conjunto de abordagens e técnicas padrão usadas no desenvolvimento de comportamentos de inteligência artificial (IA) para personagens em jogos. Esses padrões são como soluções pré-definidas para problemas comuns encontrados ao criar a lógica de comportamento para NPCs (personagens não jogáveis) e outros elementos do jogo.

Esses padrões são geralmente reconhecidos como uma série de "receitas"ou "fórmulas"que os desenvolvedores podem usar para criar comportamentos específicos de IA, economizando tempo e esforço no processo de desenvolvimento. Os padrões "Go-To AI"são frequentemente usados para adicionar complexidade e realismo aos personagens, mesmo quando o jogo não envolve algoritmos de IA avançados.

Alguns exemplos de padrões "Go-To AI Patterns"incluem:

- Patrulhamento (Patrolling): O NPC se move entre um conjunto de pontos predefinidos no mapa, imitando um comportamento de patrulha. Isso é comumente usado para guardas de segurança ou inimigos em jogos de ação e aventura.
- Perseguindo (Chasing): O NPC persegue um alvo, como o jogador, quando está dentro de um determinado raio. Isso é frequentemente usado em jogos de sobrevivência ou furtividade, onde os inimigos tentam atacar o jogador.
- Evitando Obstáculos (Obstacle Avoidance): O NPC evita obstáculos em seu caminho, ajustando sua rota para evitar colisões. Isso é importante para criar movimentos realistas e evitar que os personagens fiquem presos em objetos do ambiente.
- Procurando (Wandering): O NPC se move aleatoriamente pelo ambiente, simulando um comportamento errático. Isso pode adicionar realismo a animais, multidões ou qualquer personagem que não tenha uma rota específica.
- Ataques Aleatórios (Random Attacks): O NPC realiza ataques em momentos aleatórios, em vez de sempre seguir um padrão previsível. Isso pode tornar os combates mais desafiadores e imprevisíveis.
- Seguindo Caminhos (Path Following): O NPC segue um caminho pré-definido ou traçado pelo designer. Isso é frequentemente usado em corridas ou missões onde o jogador deve seguir um caminho específico.
- Decisões Baseadas em Probabilidade (Probabilistic Decision Making): O NPC toma decisões com base em probabilidades, tornando seu comportamento menos previsível e mais realista.
- Comunicação entre NPCs (NPC Communication): NPCs podem trocar informações entre si, permitindo que eles coordenem comportamentos ou reajam a eventos em conjunto.

Esses são apenas alguns exemplos dos muitos padrões "Go-To AI Patterns"disponíveis. Eles são especialmente úteis em jogos em que a IA precisa parecer inteligente, mas não necessariamente precisa empregar técnicas avançadas de aprendizado de máquina ou tomada de decisão complexa. O uso desses padrões pode ajudar os desenvolvedores a criar comportamentos convincentes para os personagens do jogo sem ter que reinventar a roda em termos de lógica de IA.

2.6.3 Exercícios práticos

Exercício 2.1 Jogo de Ação Básica

Desenvolva um jogo de ação simples em que o jogador controla um personagem que enfrenta inimigos. Use uma FSM para controlar o comportamento dos inimigos. Estados podem incluir "Perseguindo Jogador"e "Atacando Jogador". ■

Exercício 2.2 Mini Jogo de Memória

Crie um mini jogo de memória em que o jogador precisa encontrar pares de cartas correspondentes. Use uma FSM para controlar o comportamento das cartas. Estados podem incluir "Virada para Cima", "Virada para Baixo"e "Correspondência Encontrada". ■

Exercício 2.3 Jogo de Esquiva de Obstáculos

Desenvolva um jogo em que o jogador controla um personagem que precisa esquivar-se de obstáculos que se movem em sua direção. Implemente o comportamento dos obstáculos usando uma FSM. Estados podem incluir "Movendo-se na Direção do Jogador"e "Retornando à Posição Inicial". ■

Exercício 2.4 Jogo de Labirinto Simples

Crie um jogo de labirinto em que o jogador precisa guiar um personagem até a saída. Use uma FSM para controlar o comportamento do personagem. Estados podem incluir "Explorando", "Movendo para a Saída"e "Vitória". ■

Exercício 2.5 Jogo de Atirador de Bolhas

Desenvolva um jogo em que o jogador controla um canhão que atira bolhas para estourar outras bolhas. Use uma FSM para controlar o comportamento das bolhas. Estados podem incluir "Movendo-se na Direção do Jogador", "Explodindo"e "Desaparecendo". ■

Exercício 2.6 Jogo de Pular Plataformas

Crie um jogo de plataforma em que o jogador controla um personagem que pula entre plataformas. Implemente o comportamento das plataformas usando uma FSM. Estados podem incluir "Subindo", "Descendo"e "Esperando". ■

Exercício 2.7 Jogo de Quebra-Cabeças Deslizante

Desenvolva um jogo de quebra-cabeças em que o jogador move peças para reorganizar uma imagem. Use uma FSM para controlar o comportamento das peças. Estados podem incluir "Mover para a Esquerda", "Mover para a Direita", "Mover para Cima"e "Mover para Baixo". ■

Exercício 2.8 Jogo de Velocidade e Reflexos

Crie um jogo de reflexos em que o jogador precisa tocar na tela o mais rápido possível quando uma luz acender. Use uma FSM para controlar o comportamento da luz. Estados podem incluir "Apagada"e "Acesa". ■

Exercício 2.9 Jogo de Plataforma com Portais

Desenvolva um jogo de plataforma em que o jogador pode usar portais para se mover entre locais distantes. Use uma FSM para controlar o comportamento dos portais. Estados podem incluir "Pronto para Teletransportar"e "Teletransportando". ■

Exercício 2.10 Comportamento do Inimigo

Crie um pequeno jogo de plataforma em que um jogador controla um personagem principal e precisa evitar inimigos que se movem. Desenvolva o comportamento dos inimigos usando uma FSM. Eles podem ter estados como "Patrulhando", "Perseguindo"e "Atacando"quando o jogador está perto. ■

Exercício 2.11 Jogo de Fuga

Desenvolva um jogo de fuga (escape room) em que o jogador precisa resolver quebra-cabeças para escapar de uma sala. Crie comportamentos para elementos interativos usando uma FSM. Por exemplo, uma porta pode ter estados como "Fechada", "Abrindo"e "Aberta". ■

Exercício 2.12 Sistema de Combate em RPG

Crie um sistema de combate básico para um RPG em que o jogador e inimigos possam atacar uns aos outros. Use uma FSM para gerenciar o comportamento de combate. Os estados podem incluir "Ataque do Jogador", "Ataque do Inimigo"e "Esperando Turno". ■

Exercício 2.13 IA de Multijogador em Labirinto

Desenvolva um jogo multijogador em que os jogadores precisam navegar por um labirinto e coletar tesouros. Implemente uma IA para os monstros usando FSM. Eles podem ter estados como "Patrulhando", "Perseguindo Jogador"e "Evitando Obstáculos". ■

Exercício 2.14 Simulação de Tráfego em Cidade

Crie uma simulação de tráfego veicular em uma cidade virtual. Implemente os comportamentos dos veículos usando FSM. Eles podem ter estados como "Acelerando", "Freando", "Virando à Esquerda"e "Virando à Direita". ■

Exercício 2.15 Jogo de Estratégia em Tempo Real (RTS)

Desenvolva um RTS em que os jogadores controlam unidades para combater inimigos e conquistar territórios. Implemente o comportamento das unidades usando FSM. Cada unidade pode ter estados como "Patrulhando",

"Atacando", "Defendendo"e "Retornando à Base". ■

Exercício 2.16 Jogo de Sobrevivência em Zumbi

Crie um jogo de sobrevivência em que o jogador precisa se defender de zumbis. Desenvolva o comportamento dos zumbis usando uma FSM. Eles podem ter estados como "Vagueando", "Detectando Jogador"e "Atacando Jogador". ■

Exercício 2.17 Exercício 8: NPC com Rotinas Diárias

Desenvolva um mundo aberto em que NPCs tenham rotinas diárias. Cada NPC pode ter uma FSM que define seus estados de "Trabalho", "Lazer"e "Descanso"em diferentes momentos do dia. ■

Exercício 2.18 Jogo de Quebra-Cabeças com Portais

Crie um jogo de quebra-cabeças baseado em portais, em que o jogador pode criar portais em superfícies para se mover pelo ambiente. Use FSM para controlar o comportamento das portas, como "Criar Portal", "Entrar no Portal"e "Sair do Portal". ■

Exercício 2.19 Simulação de Economia em Jogo de Estratégia

Desenvolva um jogo de estratégia em que os jogadores precisam gerenciar recursos e construir edifícios. Use FSM para controlar o comportamento dos trabalhadores, como "Coletar Recursos", "Construir Edifícios"e "Descansar". ■

Exercício 2.20 Controle de Robô de Linha de Montagem

Simule um robô que executa diferentes tarefas em uma linha de montagem. Use uma FSM para controlar as diferentes etapas do processo, como "Pegando Peça", "Montando", "Verificando Qualidade"etc. ■

Exercício 2.21 Jogo de Aventura Textual

Desenvolva um jogo de aventura textual em que o jogador toma decisões para avançar na história. Use uma FSM para controlar os diferentes estados do jogo, como "Explorando", "Encontrando um Desafio", "Vitória"e "Derrota". ■

2.7 Árvores de decisão

Em um mundo cada vez mais dominado pela tecnologia e pela interatividade, os jogos eletrônicos estão se tornando mais do que apenas uma forma de entretenimento. Eles evoluíram para plataformas onde a complexidade e a inteligência são cruciais para a experiência do jogador. Nesse contexto, as árvores de decisão se destacam como uma ferramenta fundamental na criação de jogos que oferecem desafios e narrativas dinâmicas.

As árvores de decisão são estruturas de dados amplamente utilizadas na inteligência artificial (IA) para modelar e automatizar processos de tomada de decisão. Em jogos, elas desempenham um papel crucial na criação de

comportamentos autônomos para personagens não jogáveis (NPCs), inimigos, aliados e até mesmo para o próprio jogador. Essas árvores permitem que os desenvolvedores de jogos criem sistemas de IA que se adaptam ao contexto do jogo e às ações dos jogadores, tornando a experiência mais desafiadora e imersiva.

Neste contexto, as árvores de decisão se assemelham a "mapas"de escolhas, onde cada nó representa uma decisão possível e cada ramo representa o resultado dessa decisão. Os nós podem conter condições, ações e referências para outros nós, permitindo que a IA avalie o estado atual do jogo e escolha a melhor ação com base em critérios predefinidos. Isso pode envolver desde decisões simples, como escolher a melhor rota para seguir, até decisões complexas, como a estratégia de um inimigo em um jogo de estratégia em tempo real.

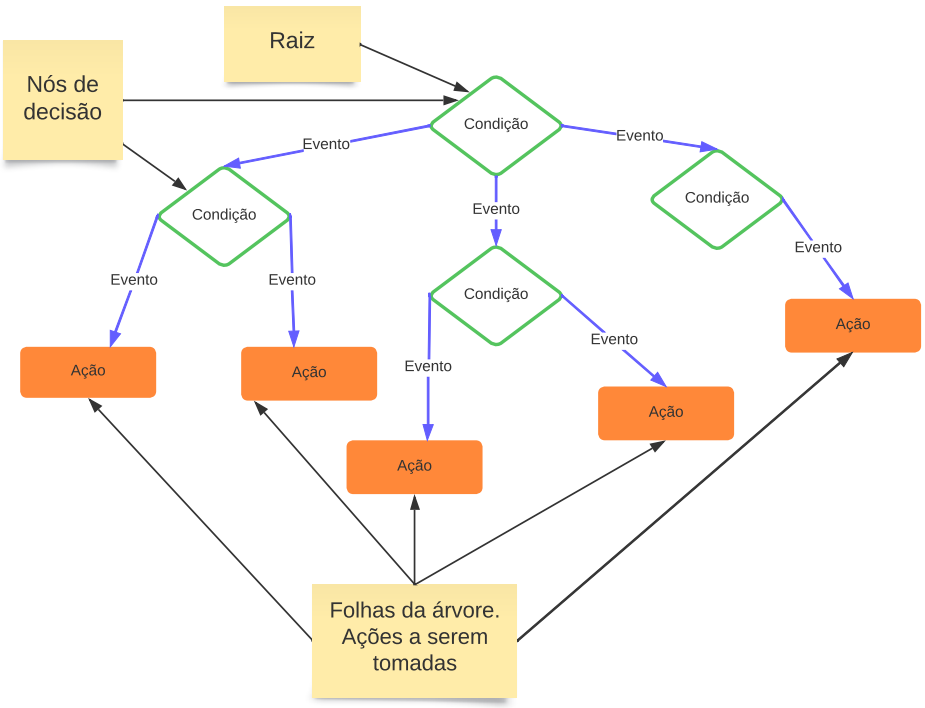

Figura 2.7: Exemplo de uma estrutura de uma árvore de decisão

As árvores de decisão proporcionam uma maneira estruturada de implementar a lógica do jogo e a IA, tornando o desenvolvimento mais organizado e flexível. Além disso, elas possibilitam que os jogos ofereçam experiências mais dinâmicas, onde as escolhas dos jogadores têm um impacto real na narrativa, nos desafios e na jogabilidade.

As árvores de decisão são compostas de galhos e folhas. Cada galho é responsável pela avaliação e a folha é composta pela ação. Através de uma sequencia de condições, nossa árvore sempre resultará em uma ação final que o agente executará.

Para criar uma árvore decisão, devemos implementar uma avaliação para nossa árvore, que avalia a partir da raiz até resultar na ação. Uma vez a ação executada, a árvore de decisão reavaliará a árvore a partir do ramo raiz para determinar a próxima ação a ser executada:

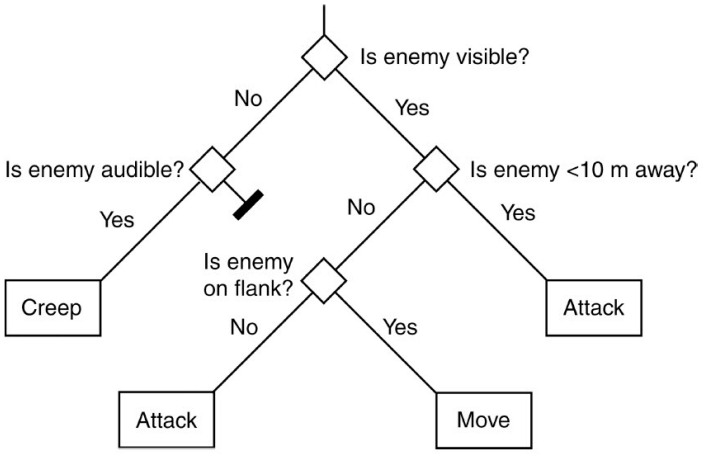

Figura 2.8: Árvore de decisão [4]

Os principais elementos de uma árvore de decisão em jogos incluem:

- **Nós da Árvore:**
  - Nó Raiz: O nó inicial da árvore, de onde todas as decisões começam.
  - Nós de Decisão: Nós que representam pontos de escolha, onde o jogo toma uma decisão com base em certas condições.
  - Nós de Ação: Nós que representam ações ou comportamentos que o NPC ou personagem deve executar.
- **Condições e Critérios de Decisão:** As condições que são verificadas em nós de decisão para determinar qual ramo da árvore seguir. Isso pode incluir condições como "o jogador está perto", "a saúde do NPC está baixa"etc.
- **Ações e Comportamentos:** As ações ou comportamentos específicos que um NPC ou personagem realiza em nós de ação. Isso pode incluir movimento, ataque, interação com objetos, diálogo, etc.
- **Ramos da Árvore:** Cada nó de decisão pode ter ramos que representam diferentes escolhas ou resultados possíveis. Os ramos são conectados aos nós de ação ou a outros nós de decisão.
- **Resultados e Consequências:** Cada ramo leva a resultados ou consequências diferentes. Por exemplo, se um NPC decide atacar o jogador, o resultado pode ser uma batalha. Se o NPC decide dialogar, o resultado pode ser uma conversa amigável.
- **Prioridades e Ponderações:** Às vezes, é importante atribuir prioridades ou ponderações aos ramos da árvore. Isso permite que certas escolhas sejam mais prováveis do que outras com base em condições ou metas específicas.
- **Lógica de Execução:** A lógica que determina como a árvore de decisão é percorrida. Isso pode ser uma lógica sequencial simples, onde a árvore é percorrida da raiz até uma folha, ou uma lógica mais complexa que leva em consideração prioridades e probabilidades.
- **Feedback e Aprendizado:** Em alguns sistemas de árvores de decisão em jogos mais avançados, é possível incluir feedback e aprendizado. Isso permite que o sistema se adapte com o tempo com base nas escolhas e resultados passados, melhorando a experiência do jogo.
- **Integração com o Motor do Jogo:** A árvore de decisão deve ser integrada com o motor do jogo para que as decisões tomadas na árvore afetem o comportamento do jogo em tempo real.
- **Depuração e Visualização:** Ferramentas de depuração e visualização são úteis para os desenvolvedores acompanharem o fluxo da árvore de decisão e entenderem como o comportamento está sendo determinado.
- **Manutenção e Escalabilidade:** As árvores de decisão devem ser projetadas de forma a serem facilmente mantidas e escaláveis à medida que mais comportamentos e decisões são adicionados ao jogo.

Esses são os principais elementos de uma árvore de decisão em jogos. A maneira como eles são implementados e organizados pode variar dependendo do jogo e das necessidades específicas do design do jogo.

2.7.1 Construindo uma árvore de decisão

Neste exemplo, usaremos um cenário hipotético em que um jogador toma decisões com base em certas condições.

Passo 1: Defina o Cenário do Jogo

Primeiro, defina o cenário do jogo e o problema que a árvore de decisão resolverá. Vamos criar um cenário onde um jogador deve decidir se deve ou não atravessar uma floresta à noite. As condições são as seguintes:

- Lanterna: O jogador tem uma lanterna que pode usar para iluminar o caminho.
- Medo do Escuro: O jogador tem medo do escuro.
- Tempo: Está escuro (noite) ou claro (dia).

Passo 2: Identifique as Decisões e Resultados Possíveis

Agora, identifique as decisões que o jogador pode tomar e os resultados possíveis para cada decisão. Neste cenário, o jogador pode tomar duas decisões:

- Decisão 1: Usar a lanterna.
- Decisão 2: Não usar a lanterna.

Os resultados possíveis para cada decisão dependem das condições. Por exemplo:

- Se o jogador usar a lanterna durante a noite, ele conseguirá atravessar a floresta com segurança.
- Se o jogador não usar a lanterna durante a noite, ele ficará com medo e não conseguirá atravessar a floresta.

Passo 3: Crie a Árvore de Decisão

Agora, crie a árvore de decisão com base nas decisões e resultados identificados:

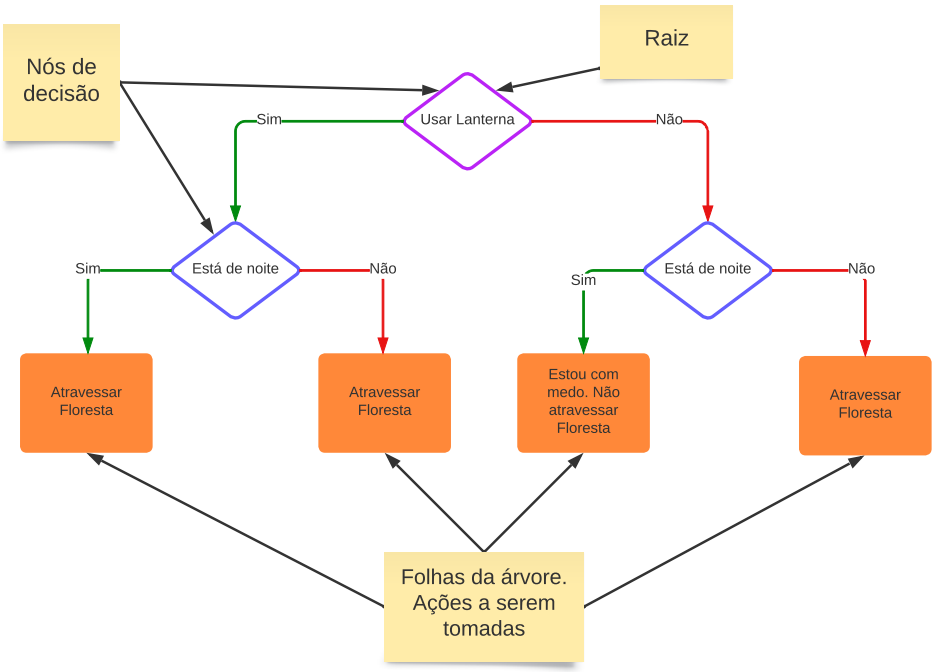

Figura 2.9: Árvore de decisão - Floresta

Passo 4: Codifique a Árvore de Decisão

Como a nossa árvore de decisão possui simples decisões, poderíamos implementar com uma simples sequência de *if-else*. Entretanto, não estaríamos de fato implementando uma árvore de decisão.

```
def atravessar_floresta(lanterna, tempo):
    if lanterna:
        # Se o jogador tem lanterna, ele atravessa com segurança, independentemente do tempo.
        return "Atravessa a floresta com segurança"
    else:
        if tempo == "Noite":
            # Se o jogador tem medo do escuro e é noite, ele não consegue atravessar.
            return "Fica com medo e não consegue atravessar"
        else:
            # Em todas as outras situações, ele atravessa com segurança.
            return "Atravessa a floresta com segurança"

#Exemplo de uso da função
resultado = atravessar_floresta(lanterna=True, tempo="Noite")
print(resultado)
```

Figura 2.10: Exemplo de uma sequência de decisões.

O exemplo apresentado na Figura 2.10 é um exemplo simples de como você pode criar e codificar uma árvore de decisão em um jogo. À medida que os cenários se tornam mais complexos, as árvores de decisão podem crescer

em tamanho e complexidade, mas o conceito básico permanece o mesmo: o jogador toma decisões com base em condições e eventos do jogo. Desta forma precisamos criar uma estrutura de árvore mais eficiente.

```
# Classe que representa o nó decisão
class NoDecisao:
    def __init__(self, pergunta, sim=None, nao=None, contexto_jogo = {}):
        self.pergunta = pergunta
        self.sim = sim
        self.nao = nao
        self.contexto_jogo = contexto_jogo

    def avaliar(self, decisao):
        return self.contexto_jogo["tempo"] == "dia" or decisao == "sim"

    def tomar_decisao(self):
        print(self.pergunta)
        decisao = input("Digite 'sim' ou 'nao': ").lower()
        if self.testar(decisao):
            self.sim.tomar_decisao()
        elif decisao == "nao":
            self.nao.tomar_decisao()
        else:
            print("Resposta inválida. Por favor, digite 'sim' ou 'nao'.")
            self.tomar_decisao()

# Classe que representa o nó de ação
class NoAcao:
    def __init__(self, acao):
        self.acao = acao

    def tomar_decisao(self):
        print(self.acao)

# Situação hipotética do contexto do jogo
contexto_jogo = {
    "tempo" : "noite"
}

# Criando os nós da árvore de decisão
no_acao_atravessar = NoAcao("Atravessar Floresta com segurança!")
no_acao_fora = NoAcao("Fica com medo e não consegue atravessar!")
no_raiz = NoDecisao("Você chegou na floresta. Usar lanterna?",
                    sim=no_acao_atravessar,
                    nao=no_acao_fora,
                    contexto_jogo=contexto_jogo)
no_raiz.tomar_decisao()
```

Figura 2.11: Exemplo de uma sequência de decisões.

Bibliografia

Artigos

Livros

[1] R. Barrera. *Unity 2017 Game AI Programming - Third Edition: Leverage the power of Artificial Intelligence to program smart entities for your games, 3rd Edition*. Packt Publishing, 2018. ISBN: 9781788393294. URL: https://books.google.com.br/books?id=qtRJDwAAQBAJ (ver páginas 34, 35).

[2] Mat. Buckland. *Programming Game AI by Example*. G - Reference,Information and Interdisciplinary Subjects Series. Wordware Pub., 2005. ISBN: 9781556220784.

[3] M. DaGraca. *Practical Game AI Programming*. Packt Publishing, 2017. ISBN: 9781787129467.

[4] I. Millington. *AI for Games, Third Edition*. 3ª edição. Taylor & Francis Group, 2020. ISBN: 9780367670566. URL: https://books.google.com.br/books?id=TWqxzQEACAAJ (ver páginas 7, 42).

[5] P. Norvig e S. Russell. *Inteligência artificial: Tradução da 3a Edição*. 3ª edição. Elsevier Brasil, 2014. ISBN: 9788535251418. URL: https://books.google.com.br/books?id=BsNeAwAAQBAJ (ver páginas 19, 25).

Manuais

Sites