


Tópicos Especiais em Engenharia de Software (Jogos II)

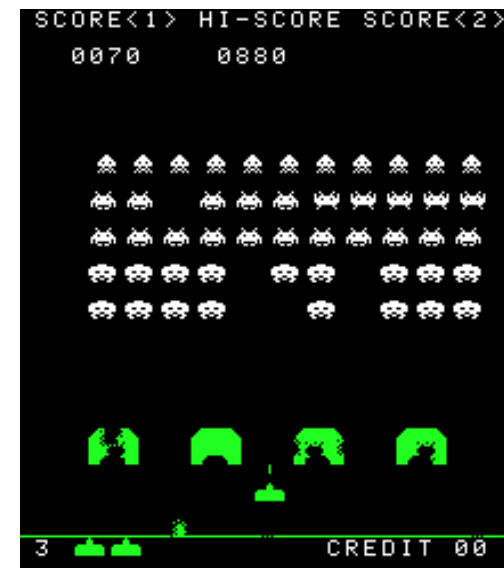
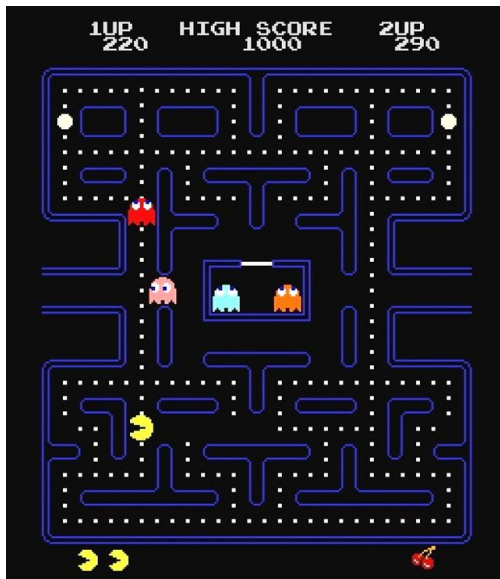
Aula 09 – Inteligência Artificial

Edirlei Soares de Lima
<edirlei@iprj.uerj.br>



Inteligência Artificial em Jogos

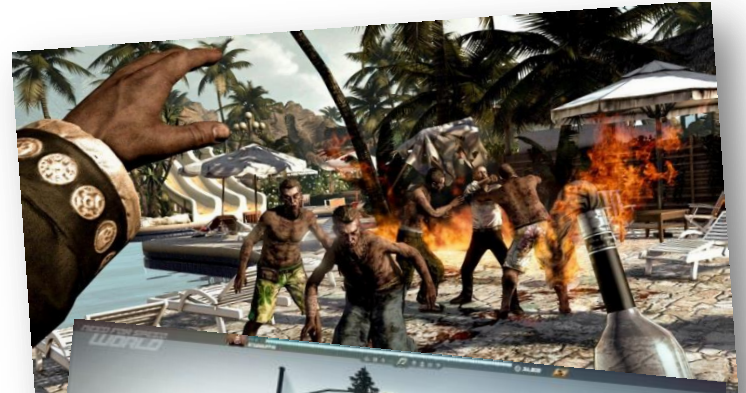
- Surgiu com a criação dos primeiros jogos (Pac-Man, Space Invaders...).
- **No início:**
 - Regras simples, sequencias pré-definidas de ações, tomada de decisão aleatória.



Inteligência Artificial em Jogos

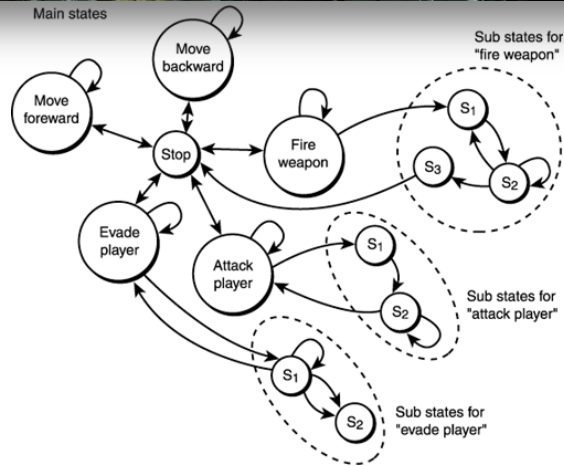
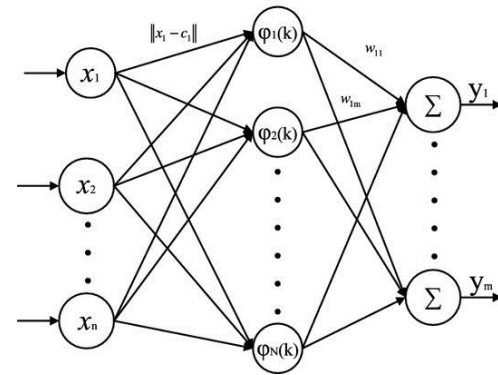
- **Atualmente:**

- Melhoras em **gráficos e som** são pouco notadas.
- Ambiente visual já está **suficientemente complexo**.
- Foco agora está no **gameplay**, na **jogabilidade** e na **inteligência artificial**.
- Personagens devem ser tão bons quanto **oponentes humanos**.



Inteligência Artificial em Jogos

- **Industria vs Academic/Research**



Ilusão de Inteligência

- Não se espera criar unidades inteligentes, mas sim criar uma “**ilusão de inteligência**”.
- Em outras palavras, espera-se criar comportamentos que imitem comportamentos humanos.
- Roubar ou não roubar?
- Percepção semelhante a dos humanos?



Princípios de Design

- NPCs devem gerar uma **experiência divertida para o jogador** e não para o programador.
- No meio acadêmico são criados programas para superar o usuário (derrotar o jogador).
- Meta da inteligência artificial para jogos **não é vencer o jogador**. O objetivo é dar ao jogador desafios e diversão!
- Todo jogador deve ser capaz de superar os desafios do jogo.



Princípios de Design

- Humanos não gostam de jogar se estão perdendo.
- O jogo deve ser agradável para todos os níveis de habilidade.
- Deve-se **evitar excessos** nos graus de dificuldade (muito fácil ou muito difícil).
- O ideal é **ajustar dinamicamente** a dificuldade dos desafios dependendo do jogador.



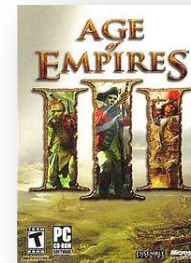
Princípios de Design

- Deve-se usar **métricas para medir o desempenho do jogador** para um ajuste dinâmico de dificuldade.
 - Tempo em cada nível, número de vidas perdidas, grau de dano...
- Deve-se evitar que o jogador **descubra métrica** e tente engana-la.
- O jogador quer derrotar tudo e todos na sua primeira tentativa dando o melhor de si.



Princípios de Design

- Todos os NPCs trapaceiam, mas o **jogador não pode perceber**.
- Não existe tecnologia para NPCs serem justos. Os NPCs devem ser simples (mais baratos e realistas).
- Jogador deve entender a o que os NPCs estão fazendo. **O importante é parecer inteligente**.
- NPC só ganha vida quando o Jogador o entende.



Técnicas Mais Usadas

- **Técnicas mais comuns:**
 - Waypoints e Pathfinding (Busca de Caminho com A*);
 - Máquinas de Estados Finitos (FSM - Finite-State Machine);
 - Aprendizado de Máquina Simplificado;
 - Sistemas de Gatilhos (Trigger Systems);
 - Previsão de Trajetória (jogos de esporte);

Locomoção em Ambientes Virtuais

- **Locomover-se no espaço do jogo** é uma ação fundamental dos NPCs em qualquer gênero de jogo.
- A busca de caminhos deve ser implementada de maneira muito eficiente, pois **será executada muitas vezes** por vários personagens durante o jogo

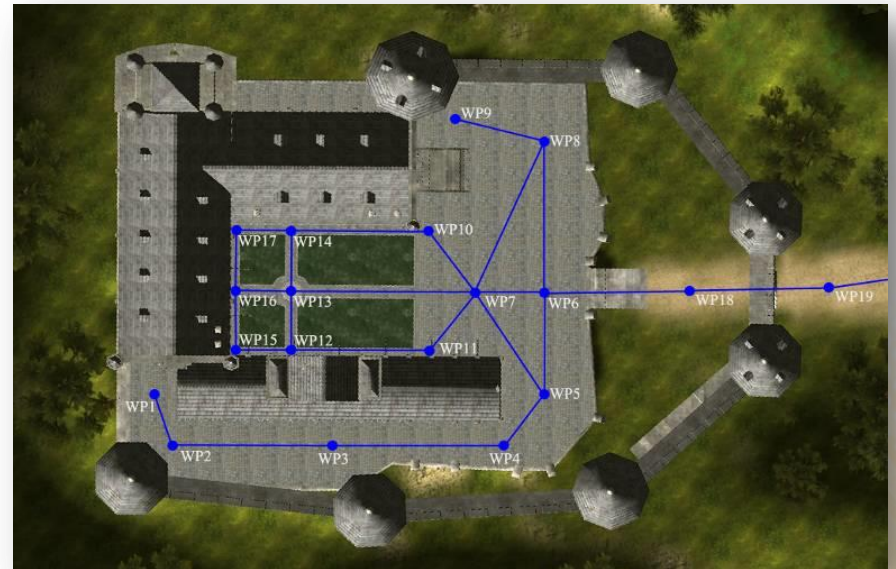


Exemplo

	1	2	3	4	5
1	😊		■		X
2			■		
3		■	■		
4					

Waypoints

- **Waypoints** são uma representação aproximada do terreno (amostragem).
- Fornecem **representações mais econômicas** do que as malhas poligonais dos cenários.
- **Estrutura de grafos.**

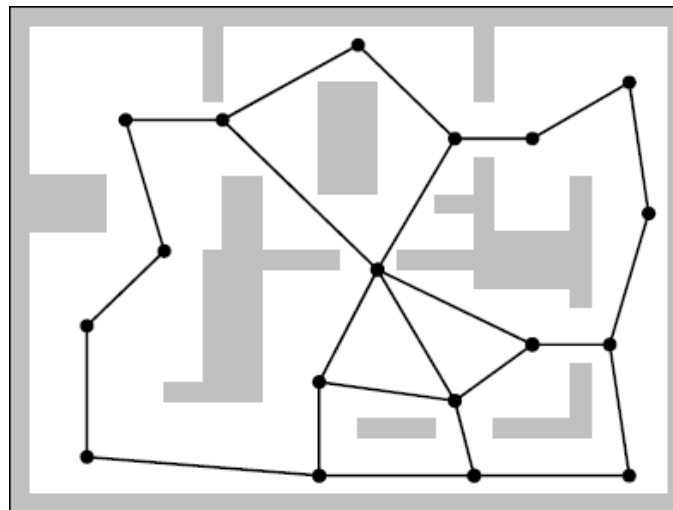


Construção de Grafos de Navegação

- Um **grafo de navegação** é uma estrutura de grafo formada por um conjunto de waypoints.
- Existem várias formas de representar a geometria do ambiente do jogo, da mesma forma existem várias estratégias para **converter a informação espacial em uma estrutura de grafo**:
 - Pontos de Visibilidade
 - Tiles (Grid)
 - Geometria Expandida
 - NavMesh

Pontos de Visibilidade

- A criação de um grafo de navegação baseado em **Pontos de Visibilidade** consiste na adição de nós em pontos importantes do ambiente.
- Os pontos do grafo devem ter pelo menos uma **linha reta de visão** para algum outro ponto.

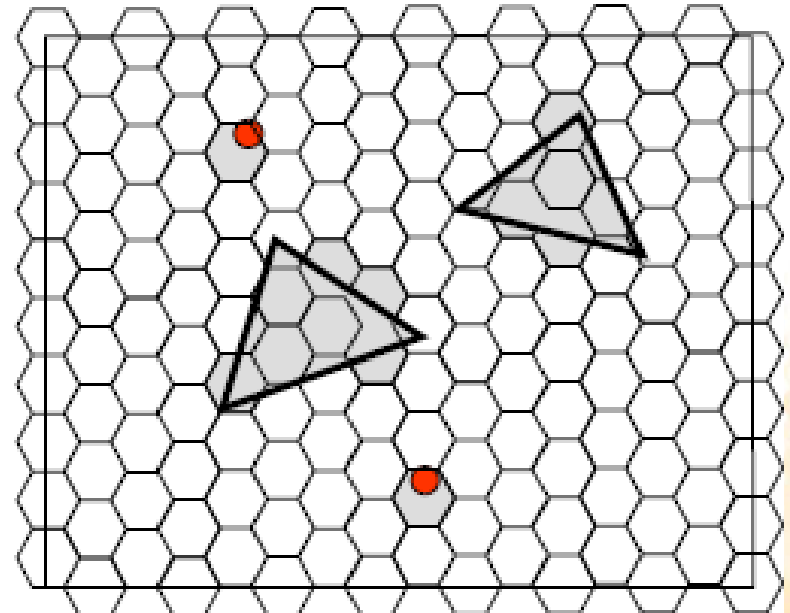
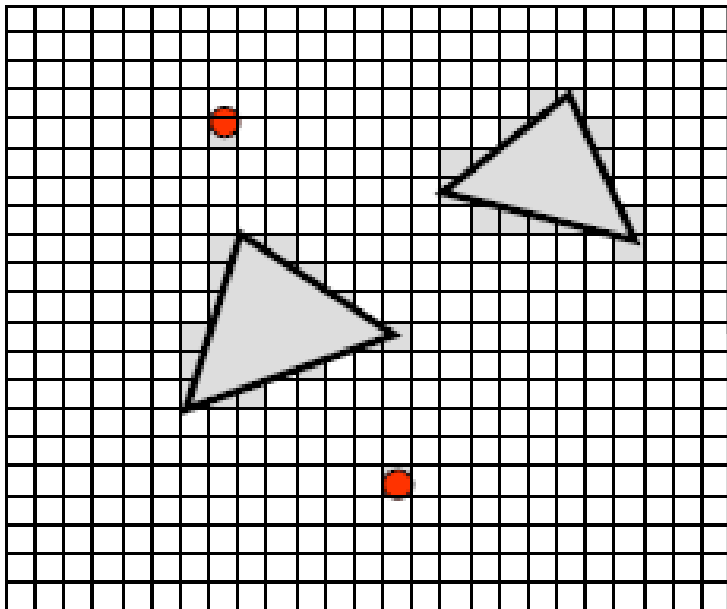


Pontos de Visibilidade

- Geralmente o processo de adição dos pontos de visibilidade é feito manualmente pelo **game designer**.
- Se o jogo **restringe o movimento** dos agentes somente **sobre as arestas do grafo**, como ocorre no Pac-man, esta solução é a escolha perfeita.
- Entretanto, se o grafo é projetado para um jogo onde os agentes têm **maior liberdade** de movimentos é necessário realizar mais algumas tarefas.

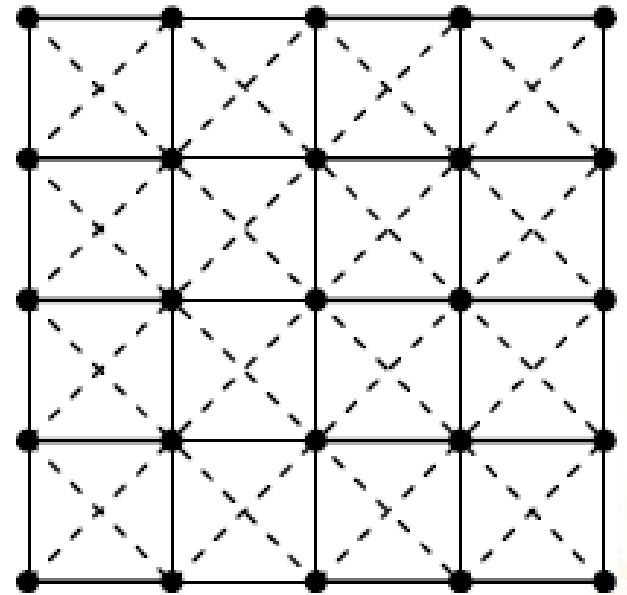
Tiles (Grid)

- Este tipo de abordagem é muito comum em jogos de estratégia em tempo real (RTS).
- Geralmente são grafos grandes e complexos, organizados por meio de quadrados ou hexágonos.
- Cada nó do grafo representa o centro de cada célula e as arestas representam a vizinhança de cada célula.



Tiles (Grid)

- O grande problema desta abordagem é que o número de vértices e arestas pode se tornar rapidamente muito elevados.
- Para um mapa com 100 x 100 células, tem-se 10.000 nós e 78.000 arestas.



Pathfinding

- Uma vez definido o grafo de navegação, é possível utilizar técnicas de busca para encontrar caminhos entre dois pontos.
- O algoritmo de busca mais utilizado em jogos é o **A***.
 - Normalmente é possível calcular boas funções heurísticas.
- O caminho gerado pelo A^* é composto por uma **lista de vértices** por onde o agente deve passar para chegar ao destino.

Exemplo

	1	2	3	4	5
1	😊		■		X
2			■		
3		■	■		
4					

Busca A*

- **Estratégia:**
 - Combina o custo do caminho $g(n)$ com o valor da heurística $h(n)$
 - $g(n)$ = custo do caminho do nó inicial até o nó n
 - $h(n)$ = valor da heurística do nó n até um nó objetivo (distancia em linha reta no caso de distancias espaciais)
 - $f(n) = g(n) + h(n)$
- **É a técnica de busca mais utilizada.**

Exemplo - A*

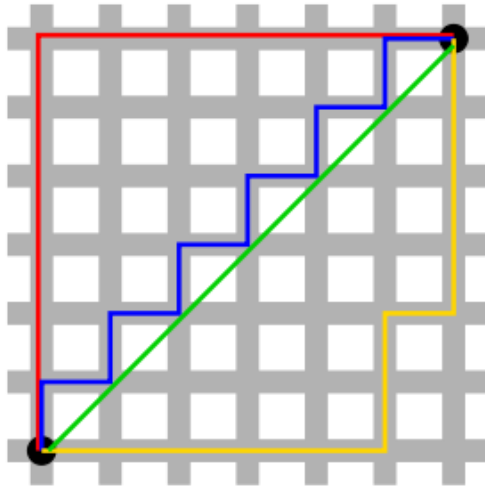
	1	2	3	4	5
1	😊		█		X
2			█		
3		█	█		
4					

Exemplo - A*

- **Heurística do A*:** $f(n) = g(n) + h(n)$
 - $g(n)$ = custo do caminho
 - $h(n)$ = função heurística
- Qual seria a função heurística $h(n)$ mais adequada para este problema?
 - A distancia em linha reta é uma opção.

Exemplo - A*

- Como calcular a heurística $h(n)$?
 - Distancia de Manhattan



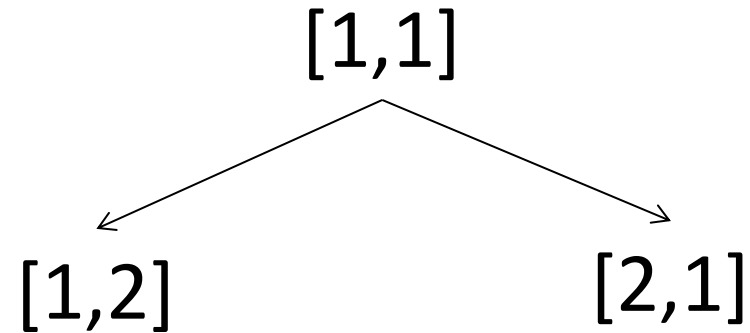
$$d_1(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|_1 = \sum_{i=1}^n |p_i - q_i|,$$

Exemplo - A*

- O próximo passo é gerar a árvore de busca e expandir os nós que tiverem o menor valor resultante da função heurística $f(n)$.

$$- f(n) = g(n) + h(n)$$

Exemplo - A*



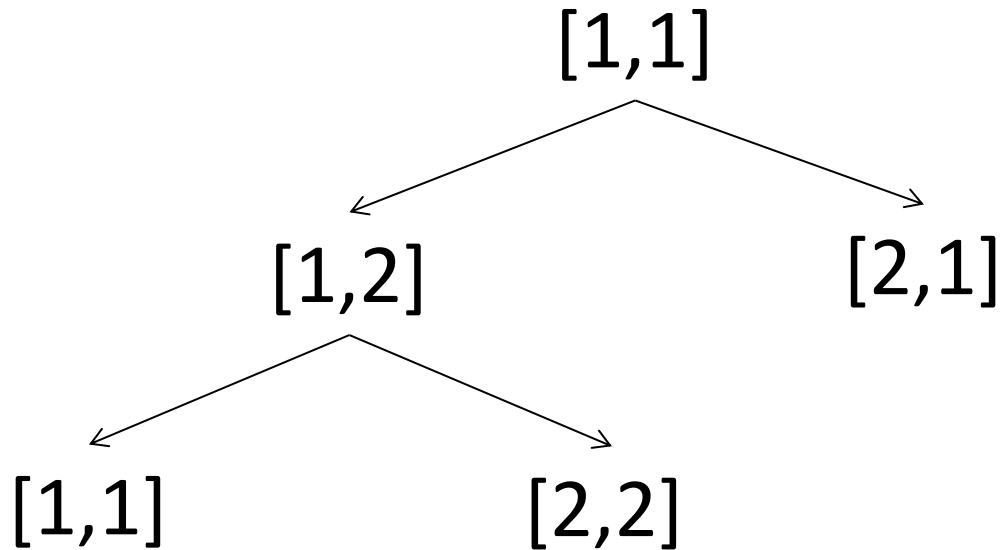
$$[1,2] = f(n) = ?? + ??$$

$$[2,1] = f(n) = ?? + ??$$

Exemplo - A*

	1	2	3	4	5
1	😊		█		X
2			█		
3		█	█		
4					

Exemplo - A*



$$[1,1] = f(n) = ?? + ??$$

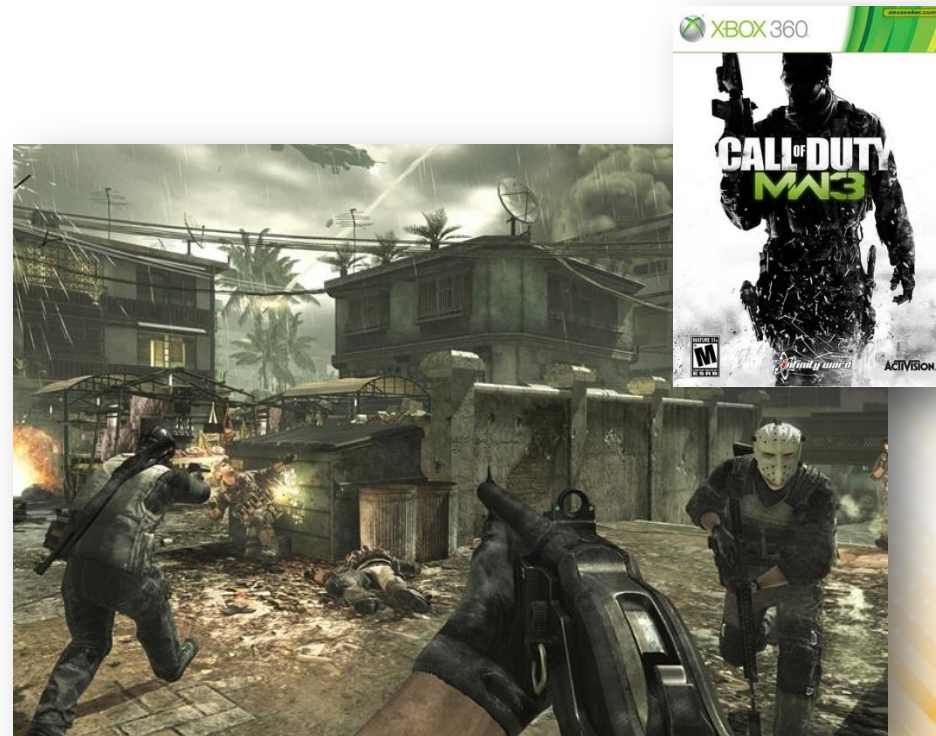
$$[2,2] = f(n) = ?? + ??$$

Exemplo - A*

	1	2	3	4	5
1	😊		█		X
2			█		
3		█	█		
4					

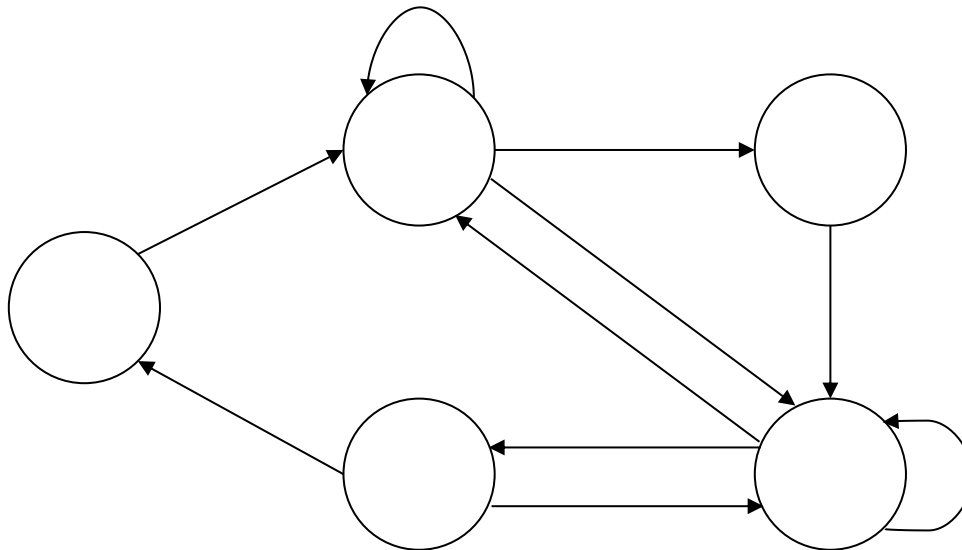
Comportamentos Autônomos

- **Máquinas de Estados Finitos** (Finite State Machines - FSM) são provavelmente o padrão de software mais utilizado em jogos para selecionar o comportamento de agentes reativos.



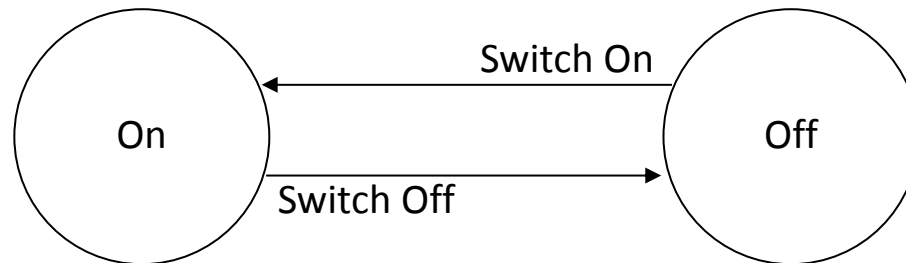
Máquina de Estados

- Uma **máquina de estados** é um modelo matemático usado para representar programas.
 - Conjunto de estados.
 - Regras de transição entre estados.
 - Estado atual.



Máquina de Estados

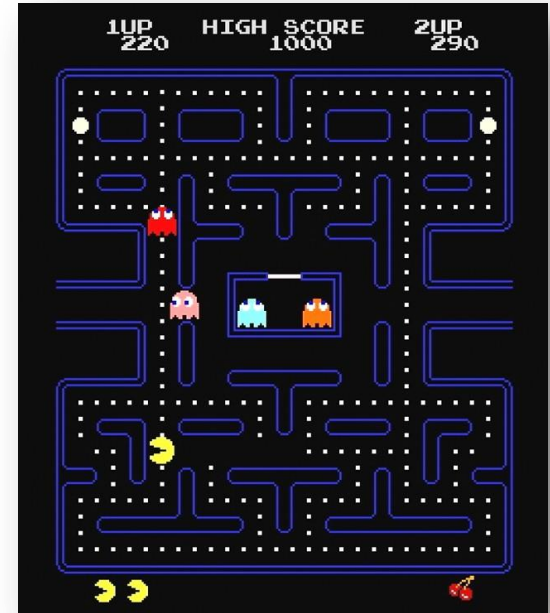
- Um exemplo bem simples de uma FSM é um interruptor de luz.



- Em um jogo normalmente uma FSM não é tão simples assim, visto que geralmente os agentes podem ter um **conjunto muito maior de estados**.

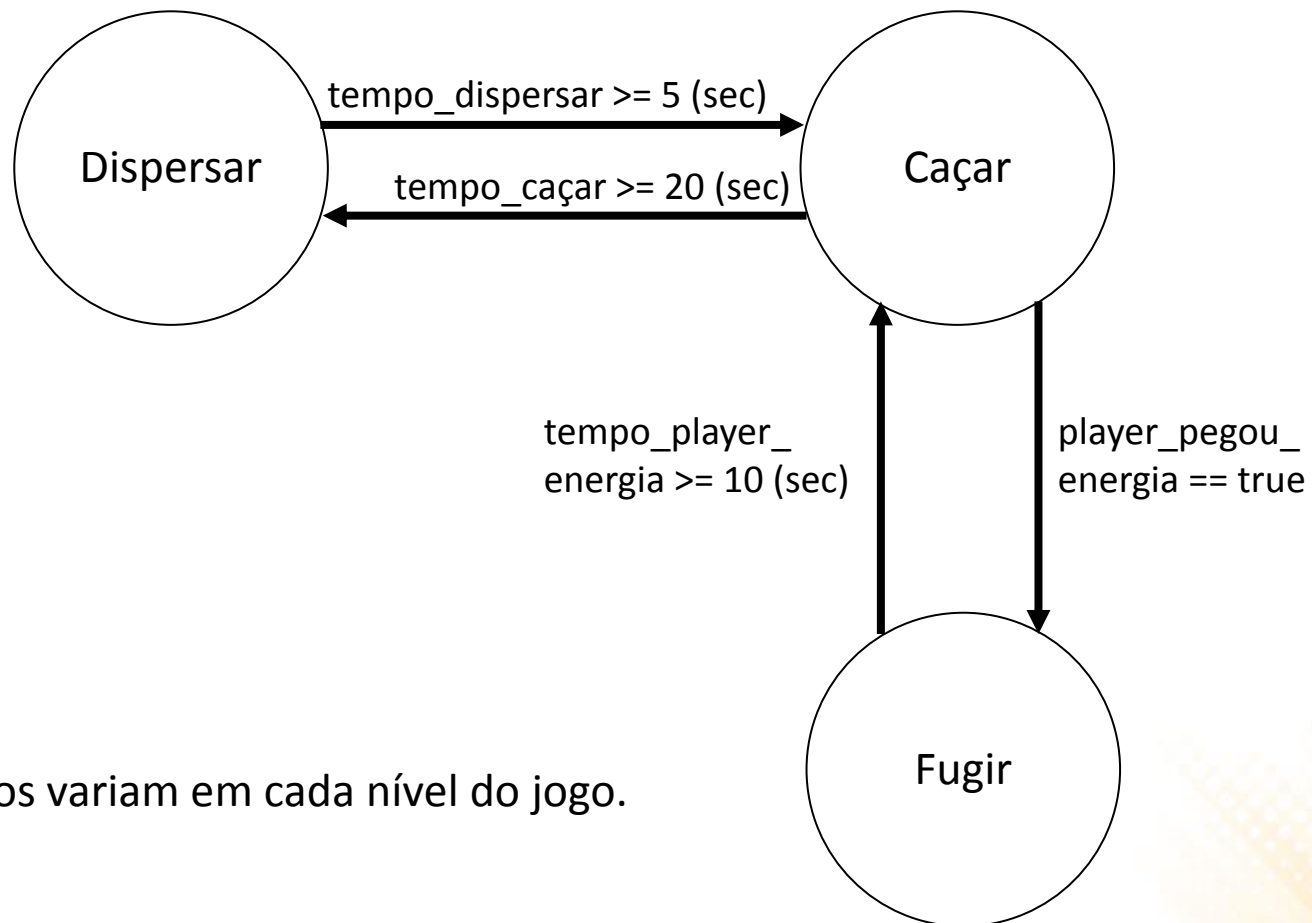
Exemplo – Pac-Man

- Os fantasmas Inky, Pinky, Blinky e Clyde do jogo **Pac-man** são implementados via FSM.
- Os fantasmas tem **3 comportamentos**:
 - Caçar (Chase)
 - Fugir (Evade)
 - Dispersar (Scatter)
- A **transição** de estados ocorre sempre que o jogador conseguir alguma pílula de energia.
- A implementação da **ação caçar** de cada fantasma é diferente.



Exemplo – Pac-Man

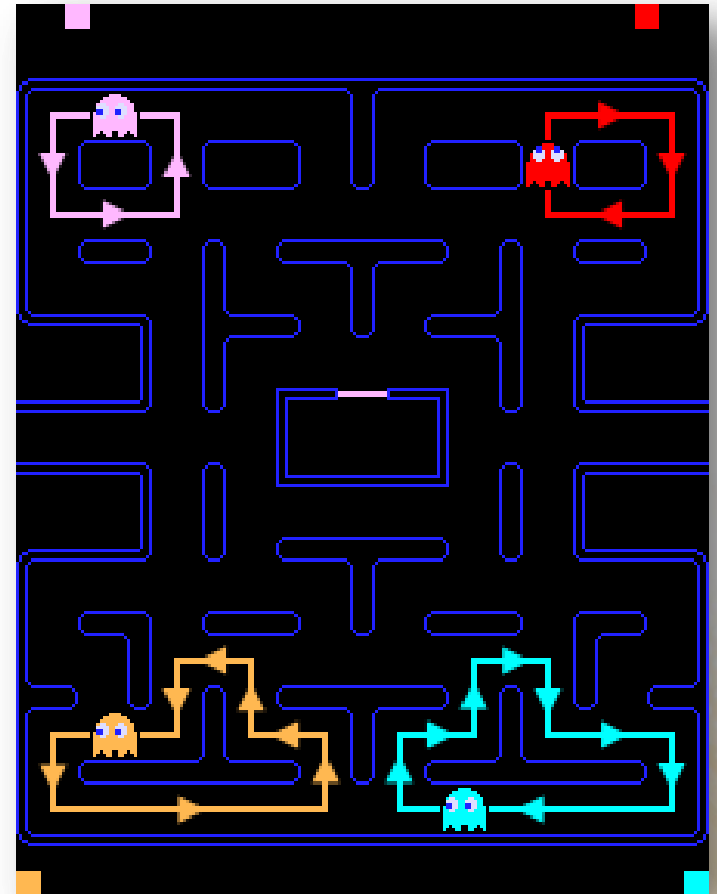
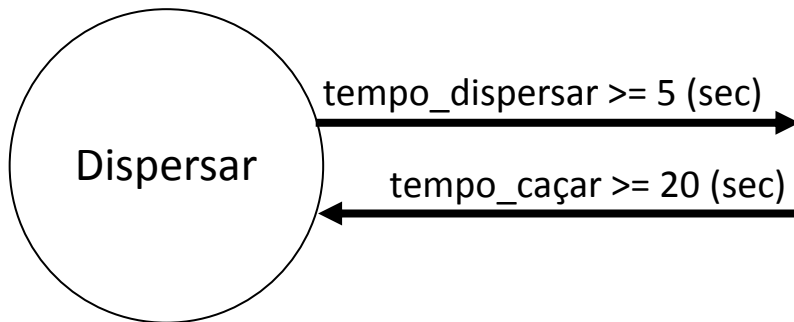
- Máquina de Estados:



- Os tempos variam em cada nível do jogo.

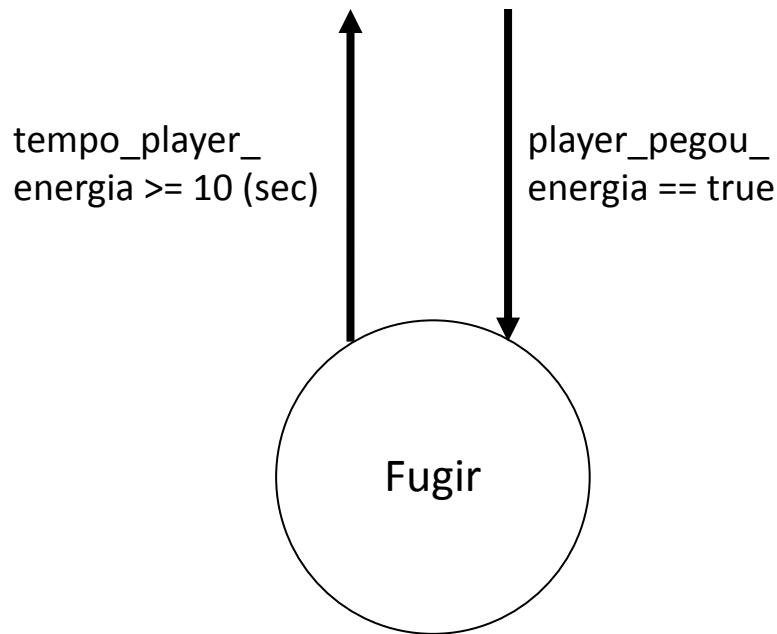
Exemplo – Pac-Man

- **Comportamento de Dispersar:**
 - Mover em direção aos cantos e ficar andando em círculos.



Exemplo – Pac-Man

- **Comportamento de Fugir:**
 - Movimentar-se mais lentamente com movimentos aleatórios.



Exemplo – Pac-Man

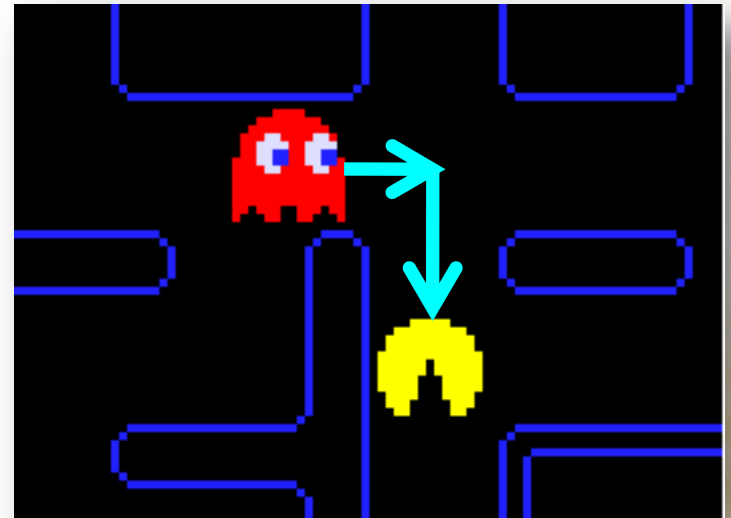
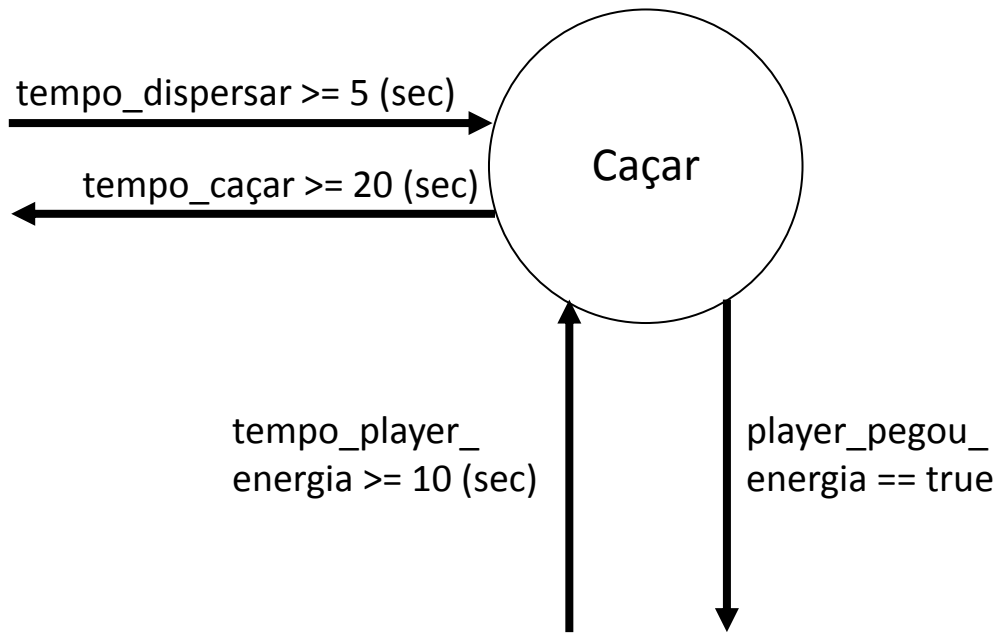
- Comportamento de Caçar:



- SHADOW

"BLINKY"

- Movimenta-se mirando na posição do Pac-Man.



Exemplo – Pac-Man

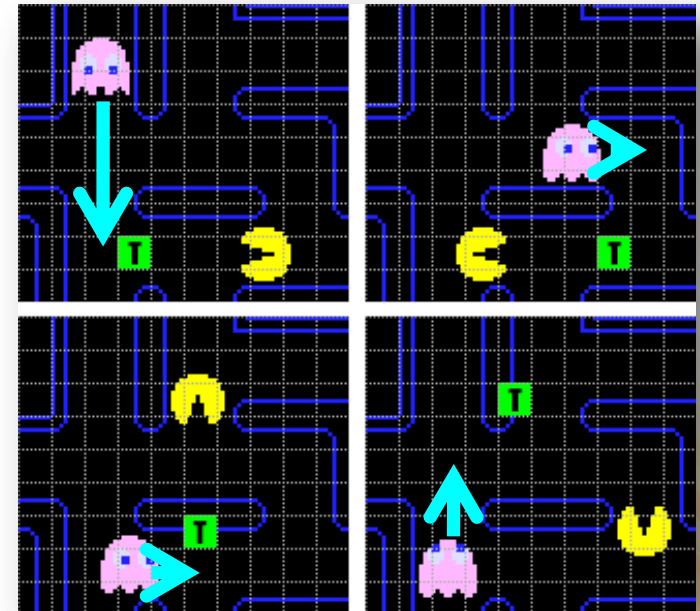
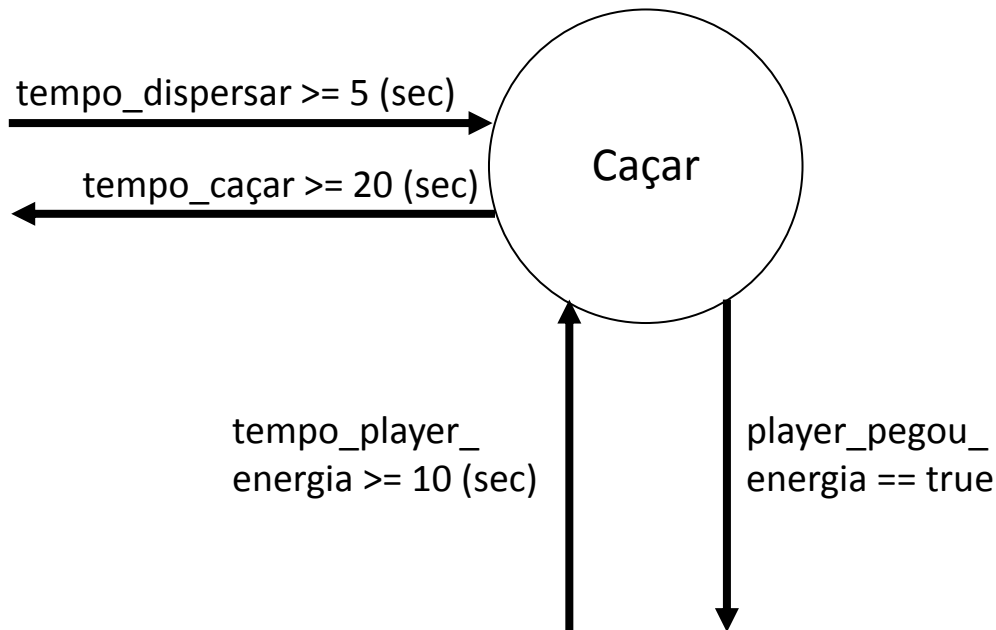
- Comportamento de Caçar:



-SPEEDY

"PINKY"

- Movimenta-se mirando na posição 4 tiles a frente do Pac-Man.



Exemplo – Pac-Man

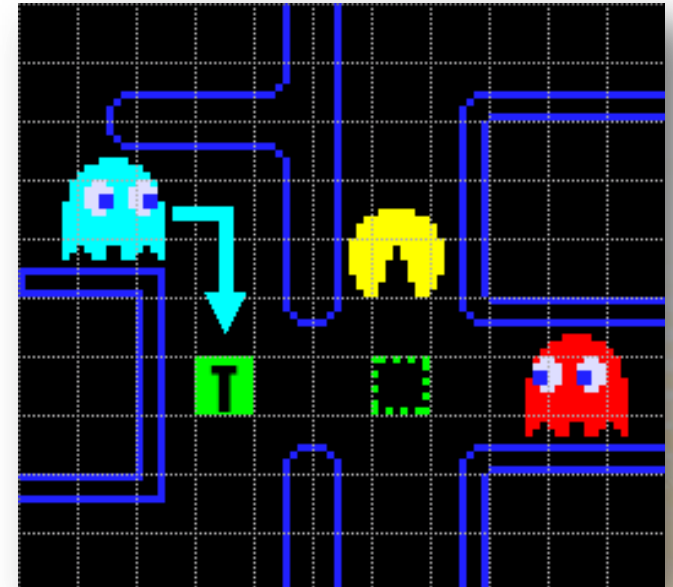
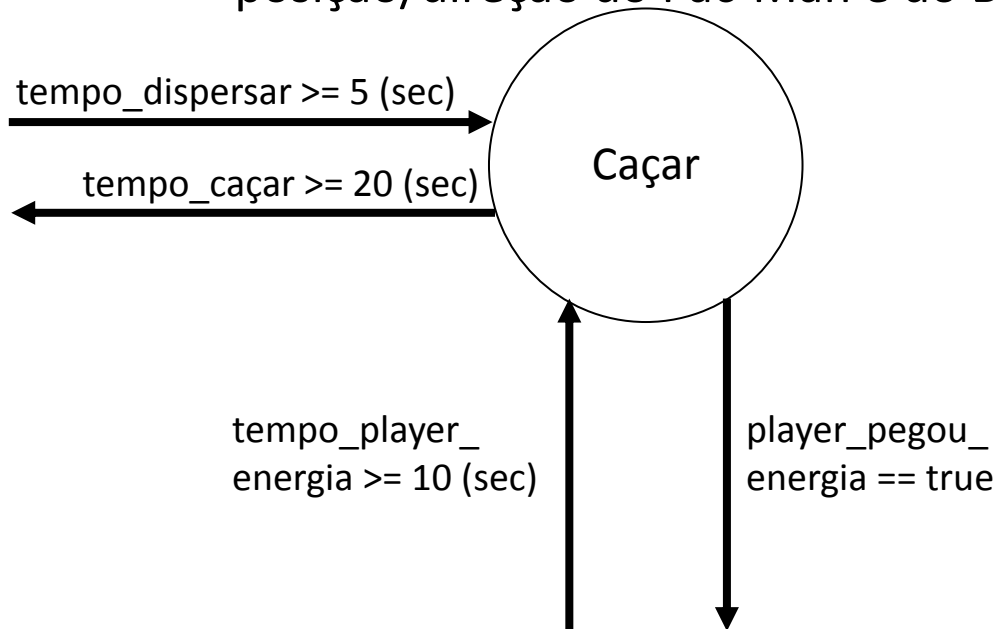
- Comportamento de Caçar:



- BASHFUL

” INKY ”

- Movimenta-se mirando em uma posição que combina a posição/direção do Pac-Man e do Blinky.



Exemplo – Pac-Man

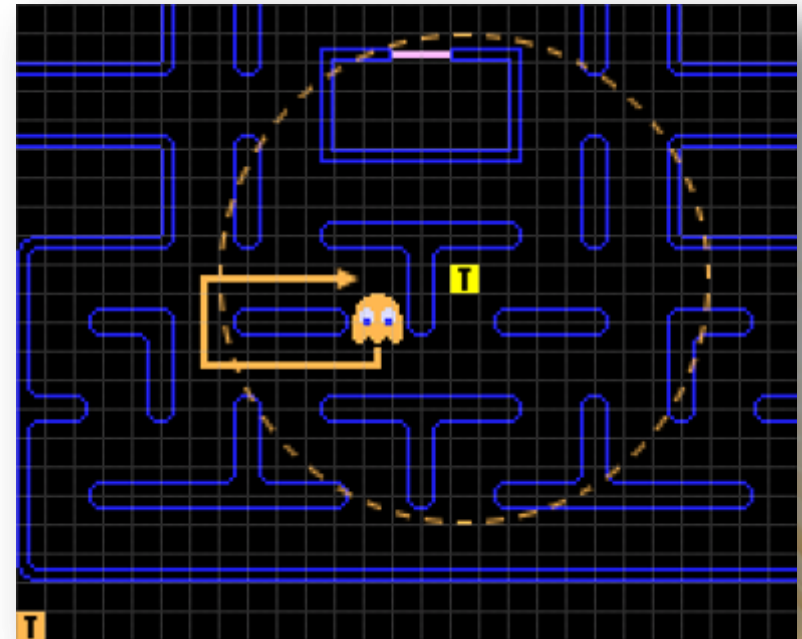
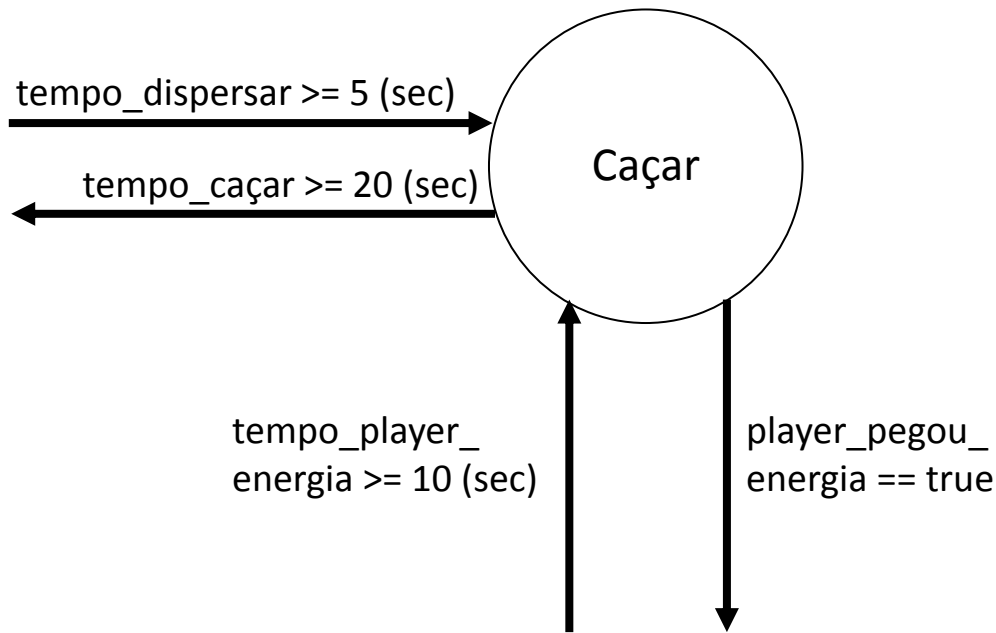
- Comportamento de Caçar:



- POKEY

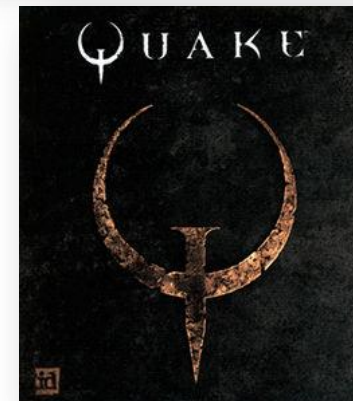
"CLYDE"

- Quando está longe do Pac-Man, movimenta-se em direção ao Pac-Man. Quando está perto, movimenta-se em direção ao canto da tela.

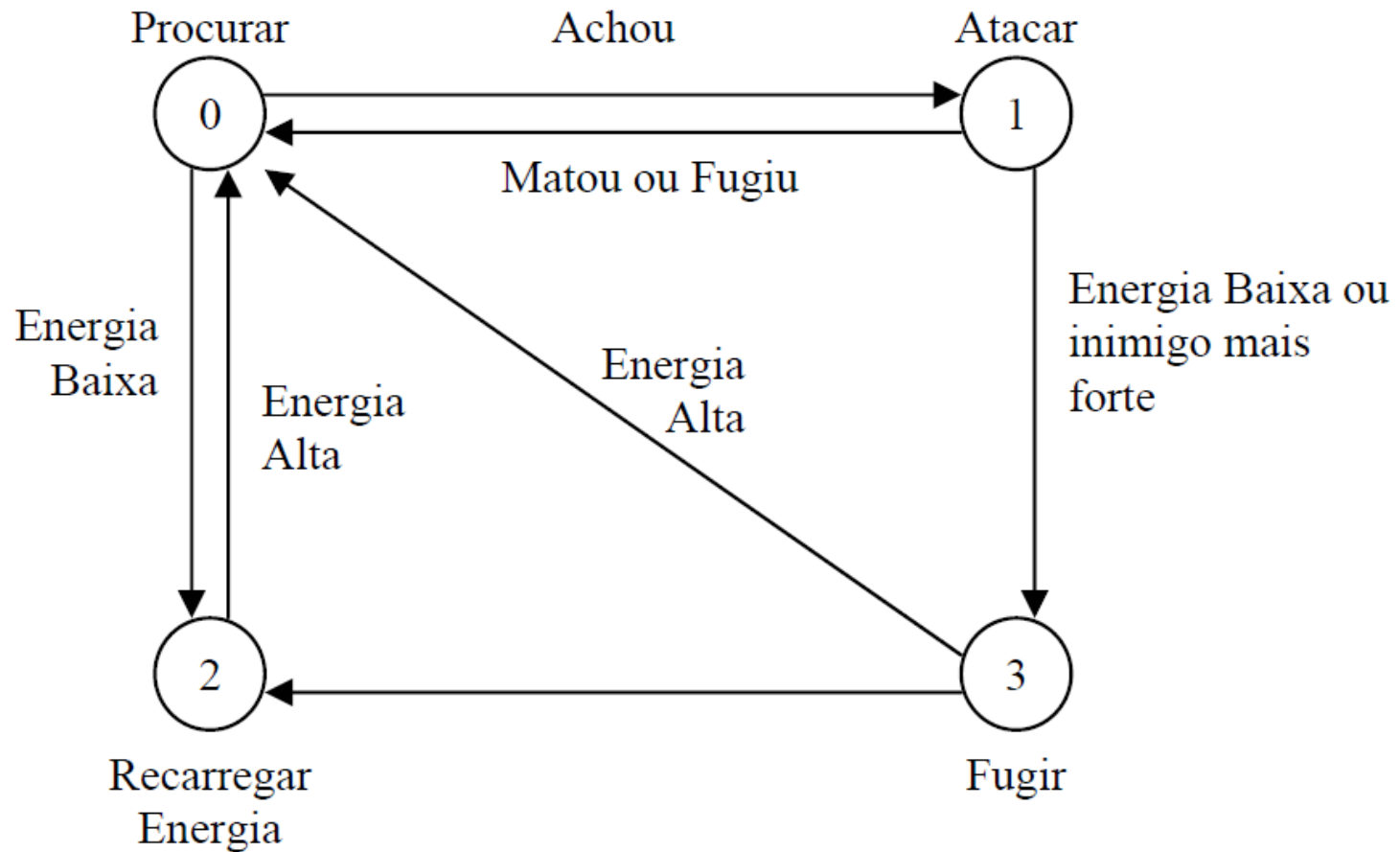


Exemplo – Quake

- Os NPCs do jogo **Quake** também são implementados via FSM.
- **Estados/Comportamentos:**
 - Procurar Armadura (FindArmor)
 - Procurar Kit Medico (FindHelth)
 - Correr (RunAway)
 - Atacar (Attack)
 - Perseguir (Chase)
 - ...
- Até mesmo as armas são implementadas como uma mini FSM.
 - Mover (Move)
 - Tocar Objeto (TouchObject)
 - Morrer/Explodir (Die)



Máquina de Estados

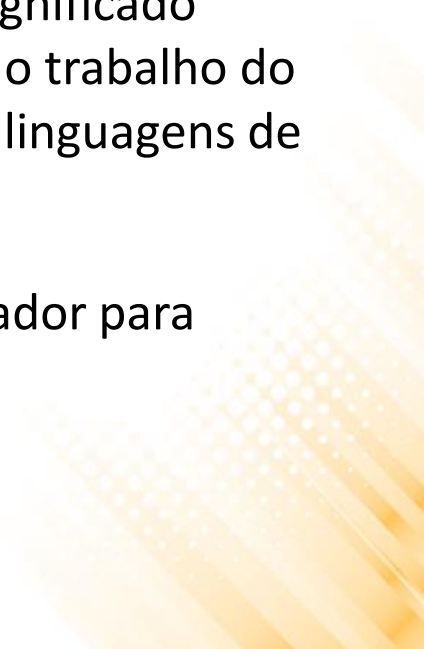


Implementação


```
void run(int *state){
    switch(*state){
        case 0: //procurar inimigo
            procurar();
            if(encontrou_inimigo)
                *state = 1;
            break;
        case 1: //atacar inimigo
            atacar();
            if (morto){
                morrer();
                *state = -1;
            }
            if (matou){
                *state = 0;
            }
            if(energia < 50 || inimigo_forte)
                *state = 3;
            break;
```

```
        case 2: //recarregar energia
            recarregar();
            if(energia > 90)
                *state = 0;
            break;
        case 3: //fugir
            fugir();
            if(!encontrou_inimigo){
                if(energia < 50)
                    *state = 2;
                else
                    *state = 0;
            }
            break;
    }
}
```

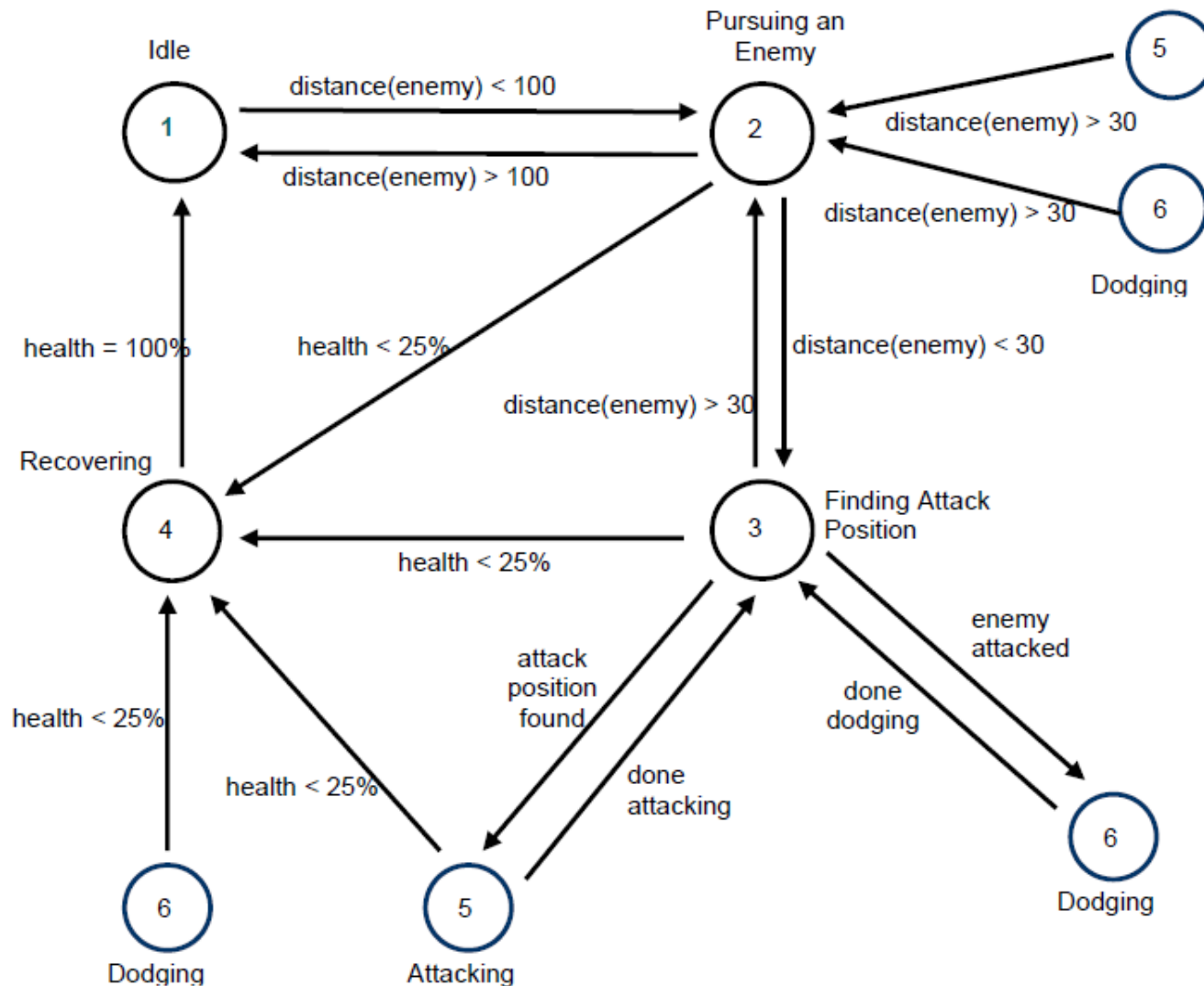
Vantagens

- **Elas são rápidas e simples de implementar** – existem várias formas de implementar e todas são muito simples.
 - **Gastam pouco processamento.**
 - **São fáceis de depurar** – quando o numero de estados é pequeno.
 - **São intuitivas** – qualquer pessoa consegue entender o seu significado apenas olhando para a sua representação visual. Isso facilita o trabalho do **game designer**, que muitas vezes não tem conhecimento de linguagens de programação.
 - **São flexíveis** – podem ser facilmente ajustada pelo programador para prover comportamentos requeridos pelo game designer.
- 

Problemas

- À medida que a complexidade do comportamento dos agentes aumenta, as FSMs tendem a **crescer de forma descontrolada**.
 - As FSMs se tornam terrivelmente complexas quanto levam em consideração **ações muito básicas** necessárias a um agente.
 - A **representação visual** torna-se intratável.
 - Comportamentos complexos **são necessários em jogos modernos**.
- 

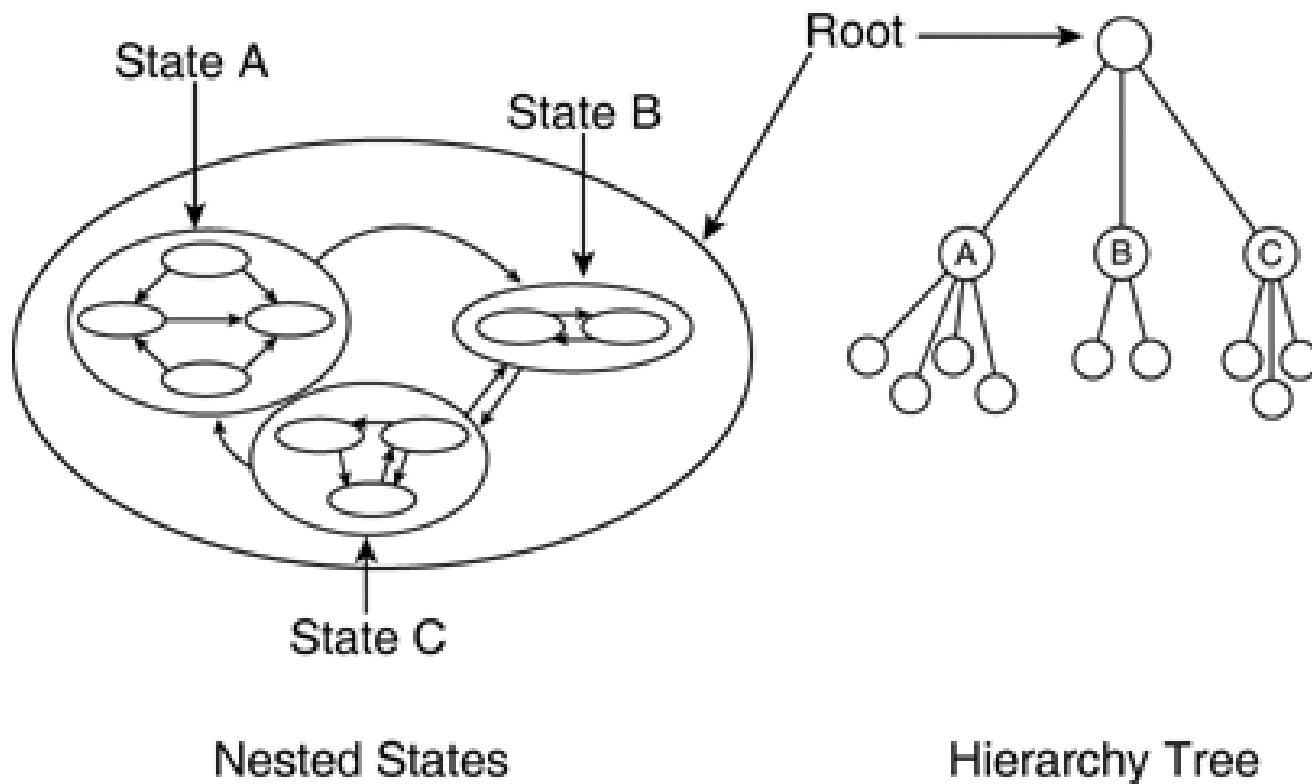
FSM um Pouco Mais Complexa...



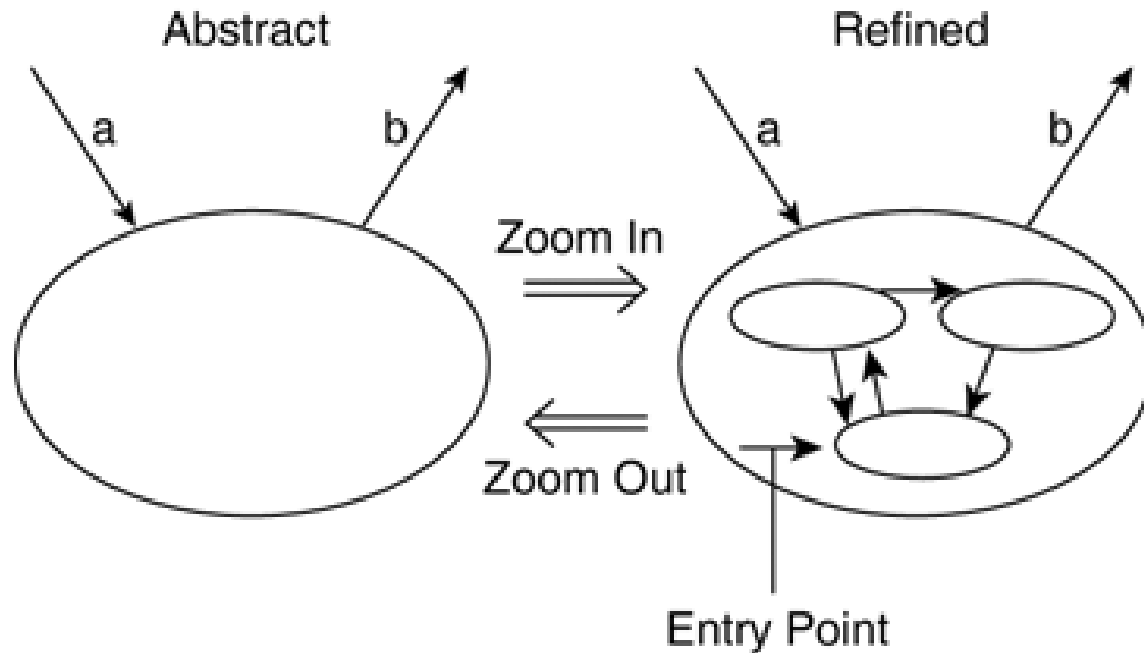
Máquina de Estados Finita Hierárquica

- É possível organizar uma FSM usando **máquinas de estados finitas hierárquicas (HFSM)**.
- Níveis mais altos lidam com ações mais genéricas, enquanto níveis mais baixos lidam com ações mais específicas.
- **Cada estado pode ser uma nova FSM.**
- Infelizmente, a hierarquia não reduz o número de estados. Ela pode somente reduzir significativamente o número de transições e tornar a FSM **mais intuitiva e simples de compreender**.

Máquina de Estados Finita Hierárquica



Máquina de Estados Finita Hierárquica



Leitura Complementar

- Millington, I.; Funge, J.: **Artificial Intelligence for Games**, 2nd Ed., Morgan Kaufmann, 2009.

