

DCC192

2025/1



Desenvolvimento de Jogos Digitais

A4: Game Update and Objects

Prof. Lucas N. Ferreira

Plano de aula

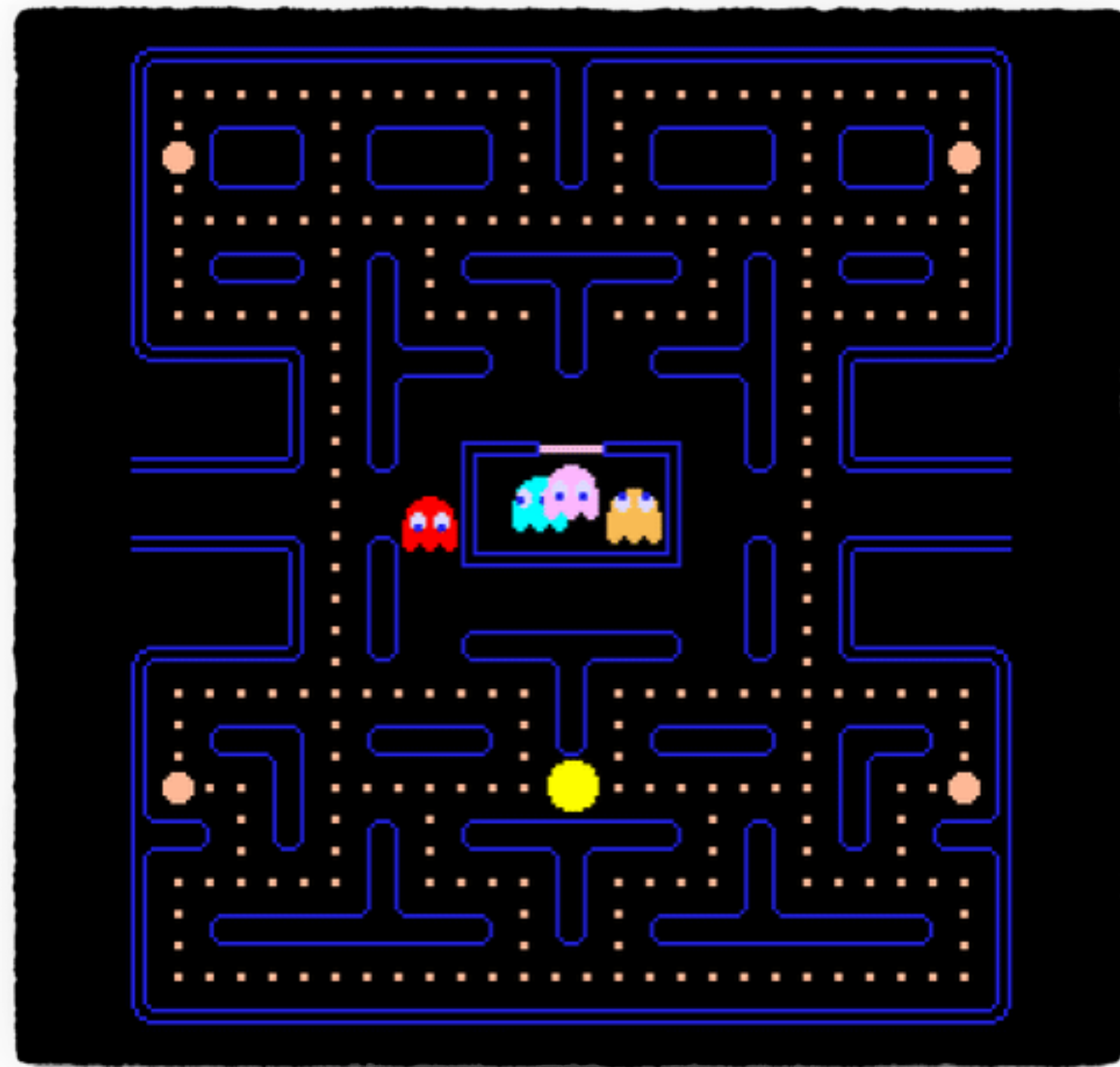


- ▶ Game Update
- ▶ Game Objects: Dinâmicos , Estáticos e Gatilhos
- ▶ Modelagem de Objetos
 - ▶ Modelo de hierarquia de classes
 - ▶ Modelo de componentes
 - ▶ Modelo híbrido

Game Loop



Na última aula vimos que um jogo é um loop que repete três funções: **Input**, **Update** e **Draw**



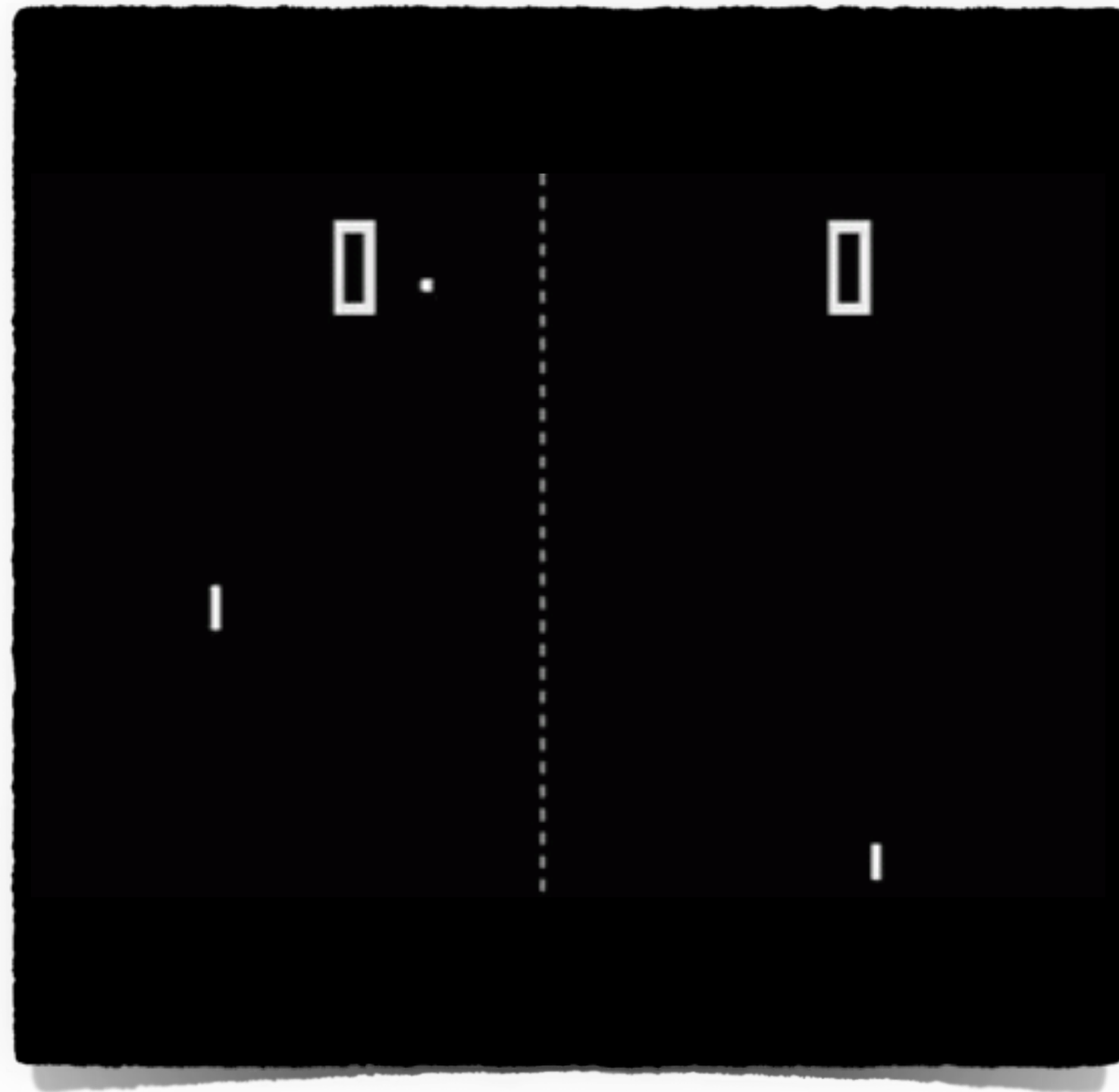
1. **while** Game is running
2. ProcessInput ()
3. UpdateGame ()
4. GenerateOutput ()

A etapa de **Update** implementa toda a “lógica” do jogo (mecânicas, regras, objetivos, ...)

Game Update



Em jogos simples, como o Pong, é razoável implementar o **Update** em uma única função:



```
while !goal:
    // Atualizar posição das raquetes
    update player1.position based on input1
    update player2.position based on input2

    // Atualizar posição da bola
    update ball.velocity based on collision
    ball.position += ball.velocity

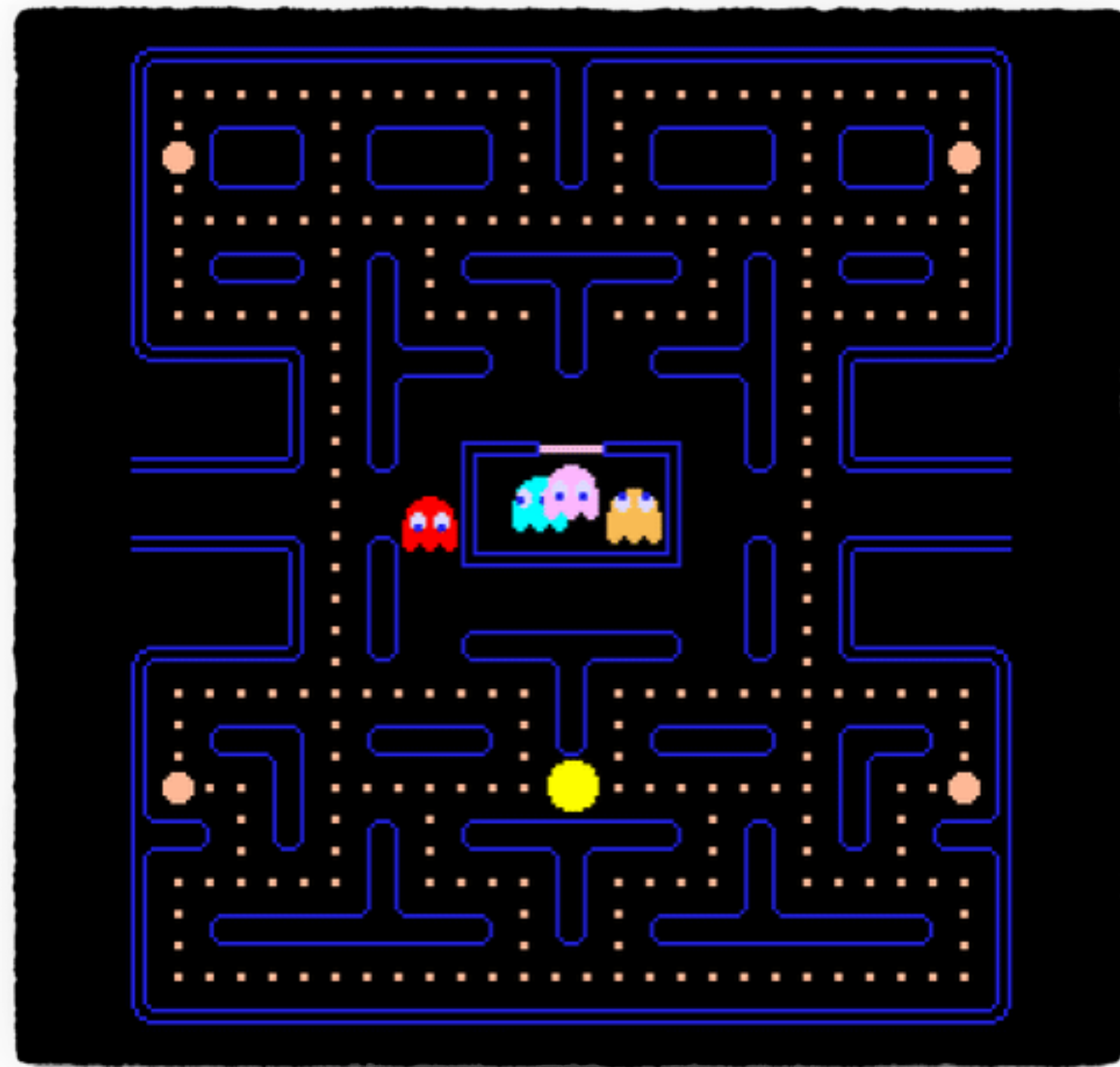
    // Verificar se houve gol
    if ball.position > LEFT_BOUND or
       ball.position < RIGHT_BOUND
        goal = true

    // Atualizar placar
    ...
```

Game Update



Para jogos um pouco mais complexos, como o PacMan, isso já se torna problemático:



```
while player.lives > 0
    // Atualizar posição
    update player.position based on input

    // Atualizar posição
    foreach Ghost g in world
        if player collides with g
            kill either player or g
        else
            update AI for g

    // Comer as pastilhas/power ups
    ...
```

- ▶ Verificação/Resolução de colisão não é trivial
- ▶ Cada fantasma tem sua IA, também não trivial

Game Update



Na maioria dos jogos e engines modernas, o jogo é um objeto que contém uma lista de *Game Objects*, onde cada objeto implementa seu próprio **Update**

```
class Game {
public:
    World():numObjects(0) {}
    void Update(float deltaTime);
    void AddActor(GameObject* obj);
    void RemoveActor(GameObject* obj);

private:
    GameObject* gameObjs[MAX_OBJS];
    int numObjects;
};
```

```
void Game::Update(float deltaTime) {
    while (true) {
        // Handle user input...

        // Update each entity.
        for (int i = 0; i < numObjects; i++) {
            gameObjs[i]->Update();
        }

        // Rendering...
    }
}
```


Game Update



Adicionar/Remove game objects durante o update de um objeto é um problema:

- ▶ **Adicionar** um elemento ao final da lista faz com que ele seja atualizado antes mesmo de ser visualizado
- ▶ **Remove** um elemento da lista que está antes de *i* pula o update de um objecto

```
void Game::Update(float deltaTime) {  
    while (true) {  
        // Handle user input...  
  
        // Update each entity.  
        for (int i = 0; i < numObjects; i++) {  
            gameObjs[i]->Update();  
        }  
  
        // Physics and rendering...  
    }  
}
```

Game Update



Adicionar/Remover game objects durante o update de um objeto é um problema:

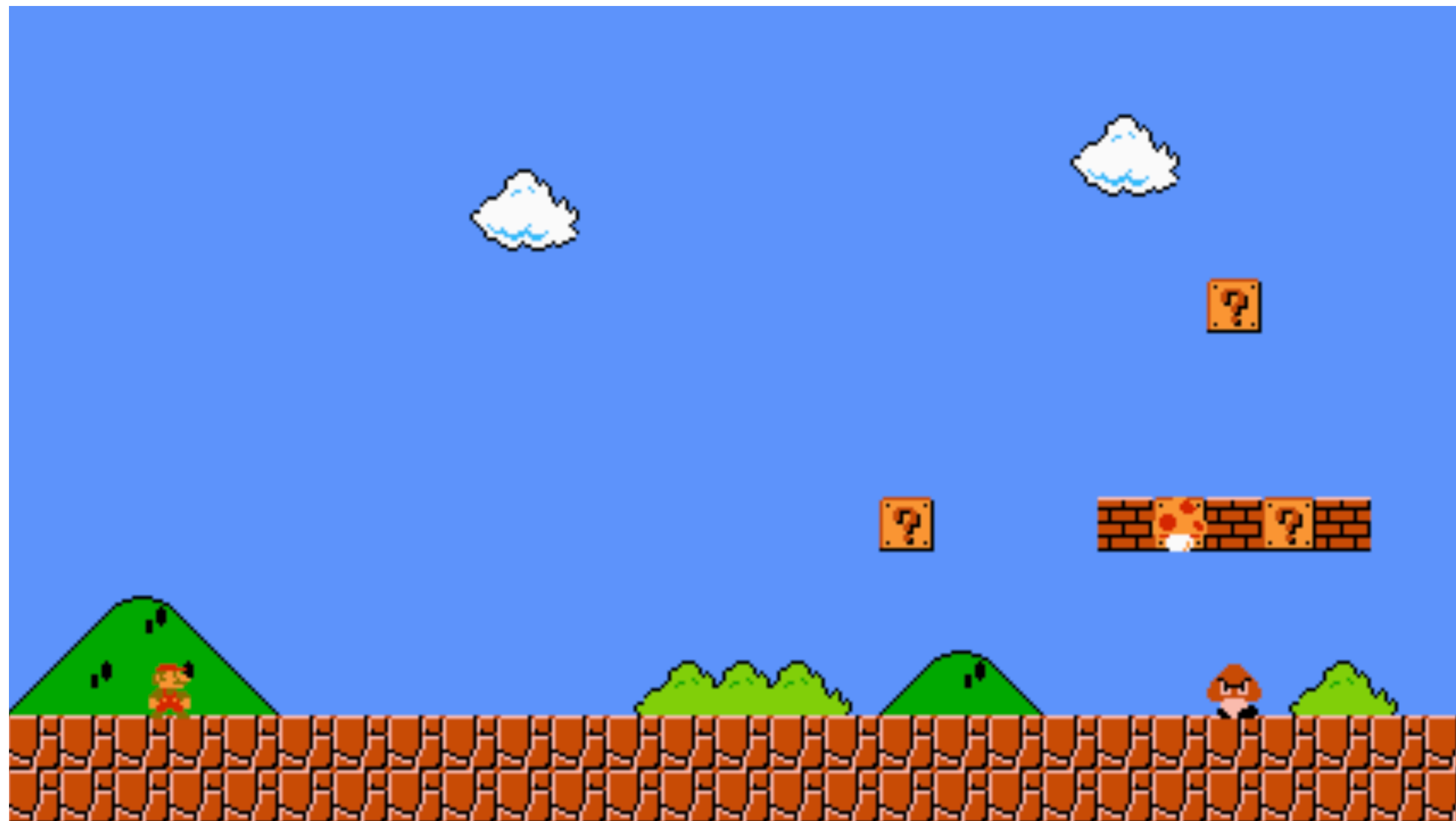
- ▶ **Adicionar** um elemento ao final da lista faz com que ele seja atualizado antes mesmo de ser visualizado
- ▶ **Remover** um elemento da lista que está antes de *i* pula o update de um objecto
- ▶ **Solução:** criar listas auxiliares para armazenar objetos add/del durante os updates dos objetos e processá-las no final do update do jogo

```
class Game {  
public:  
    World():numObjects(0) {}  
    void Update(float deltaTime);  
    void AddActor(GameObject* obj);  
    void RemoveActor(GameObject* obj);  
  
private:  
    GameObject* pendingObjects[MAX_OBJS];  
    GameObject* deadObjects[MAX_OBJS];  
    GameObject* gameObjs[MAX_OBJS];  
    int numObjects;  
};
```


Game Objects



Até agora temos como adicionar objetos no jogo, mas o que é um game object em si, quais propriedades e funções eles devem ter?

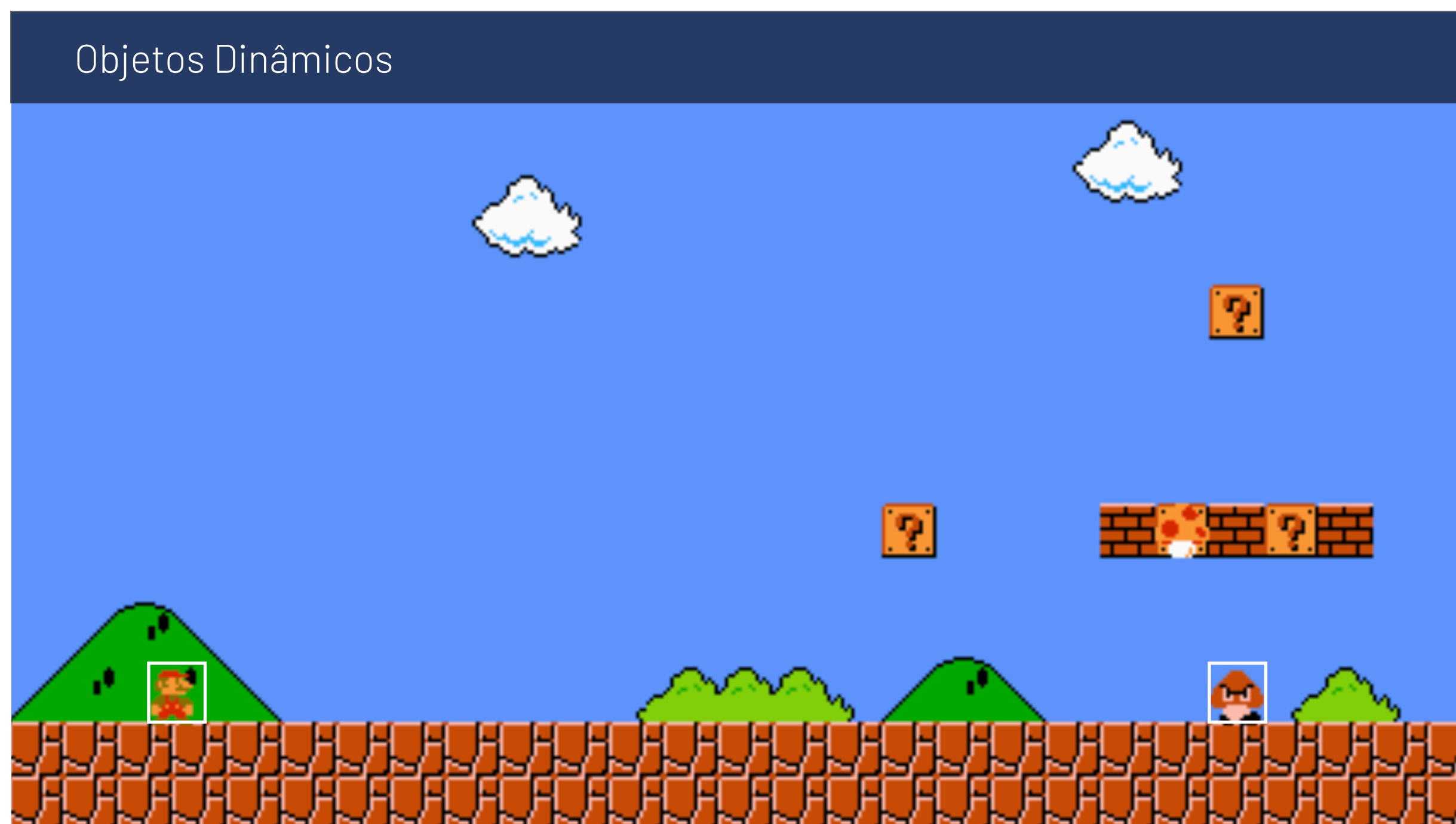


```
class GameObject {  
public:  
    GameObject();  
    ????  
  
private:  
    ????  
};
```

Game Objects Dinâmicos



Objetos dinâmicos possuem gráficos e se movem no mundo, portando devem possuir atributos físicos (Ex., Rigidbody e BoxCollider) e métodos como Update, Draw, Move, etc..



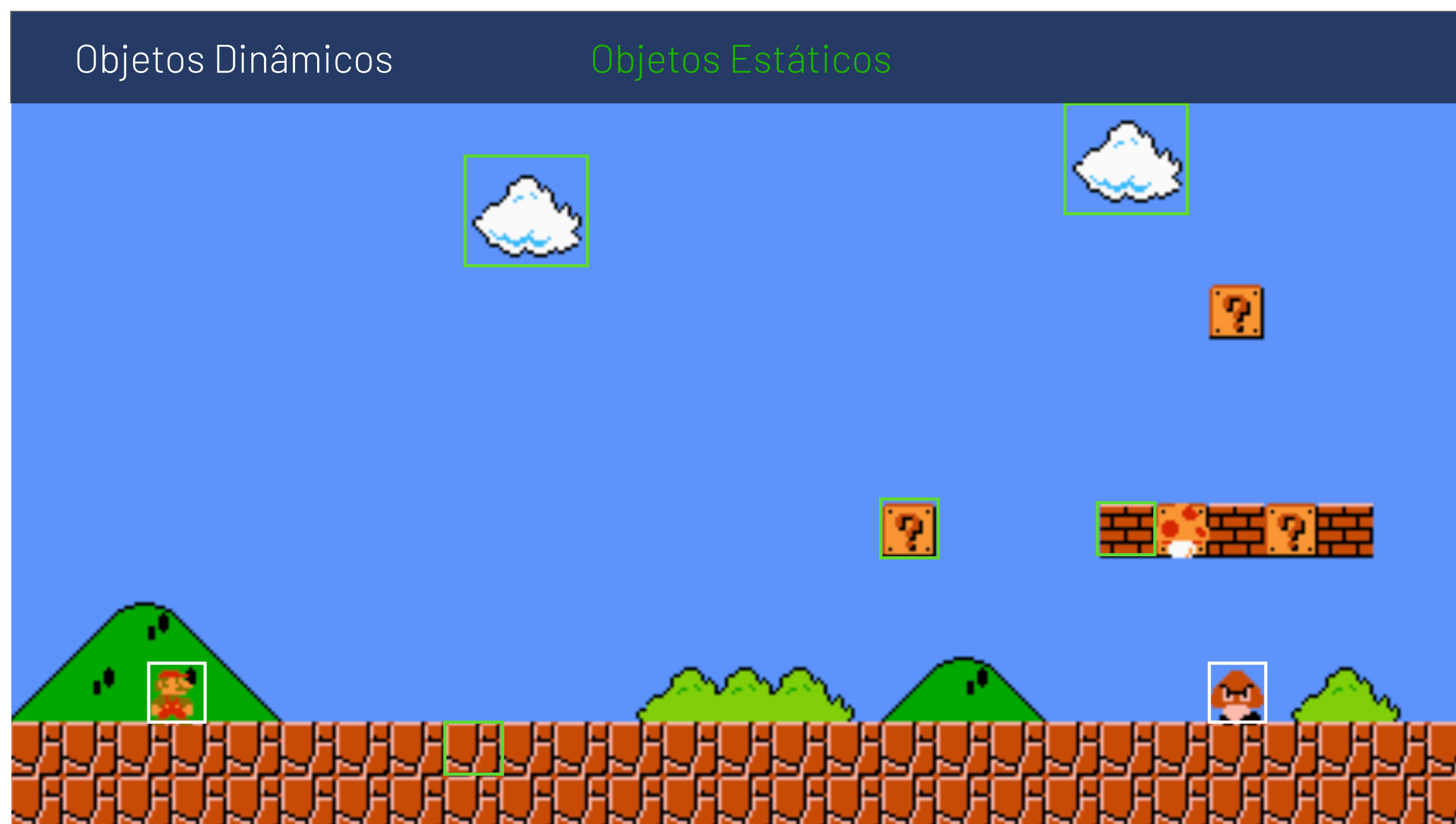
```
class GameObject {  
public:  
    GameObject();  
    void Update();  
    void Draw();  
    void Move();  
    void Jump()?  
private:  
    Rigidbody* body;  
    BoxCollider* collider;  
    Animator* animator;  
};
```

E o método Jump? Devemos incluí-lo já que o Goomba não pula?

Game Objects Estáticos



Objetos estáticos também possuem gráficos, mas não se movem, apesar de que alguns possuem colisores (Ex., chão, plataformas) e outros não (Ex. nuvem).

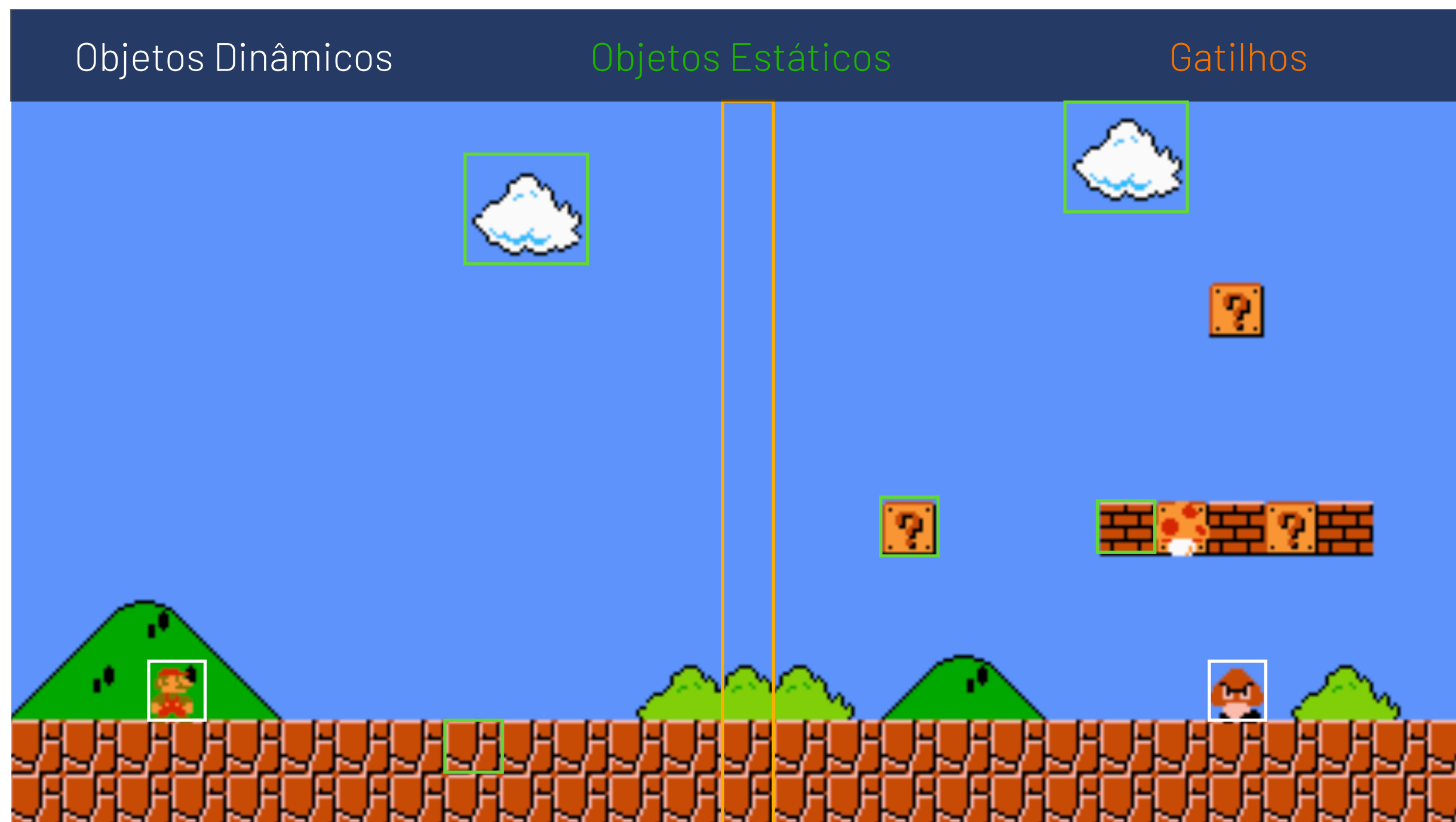


```
class GameObject {  
public:  
    GameObject();  
    void Update();  
    void Draw();  
    void Move()???  
    void Jump()???  
private:  
    Rigidbody* body; ???  
    BoxCollider* collider; ???  
    Animator* animator; ???  
};
```

Game Objects Gatilhos (Triggers)



Objetos gatilhos utilizam colisão para chamar um determinado evento no jogo (Ex., spawn do Goomba), mas não possuem gráficos



```
class GameObject {  
public:  
    GameObject();  
    void Update(); ???  
    void Draw(); ???  
    void Move(); ???  
    void Jump(); ???  
private:  
    Rigidbody* body; ???  
    BoxCollider* collider; ???  
    Animator* animator; ???  
};
```

Problema: os objetos de jogo geralmente são muito diferentes entre si!

Modelagem de Objetos



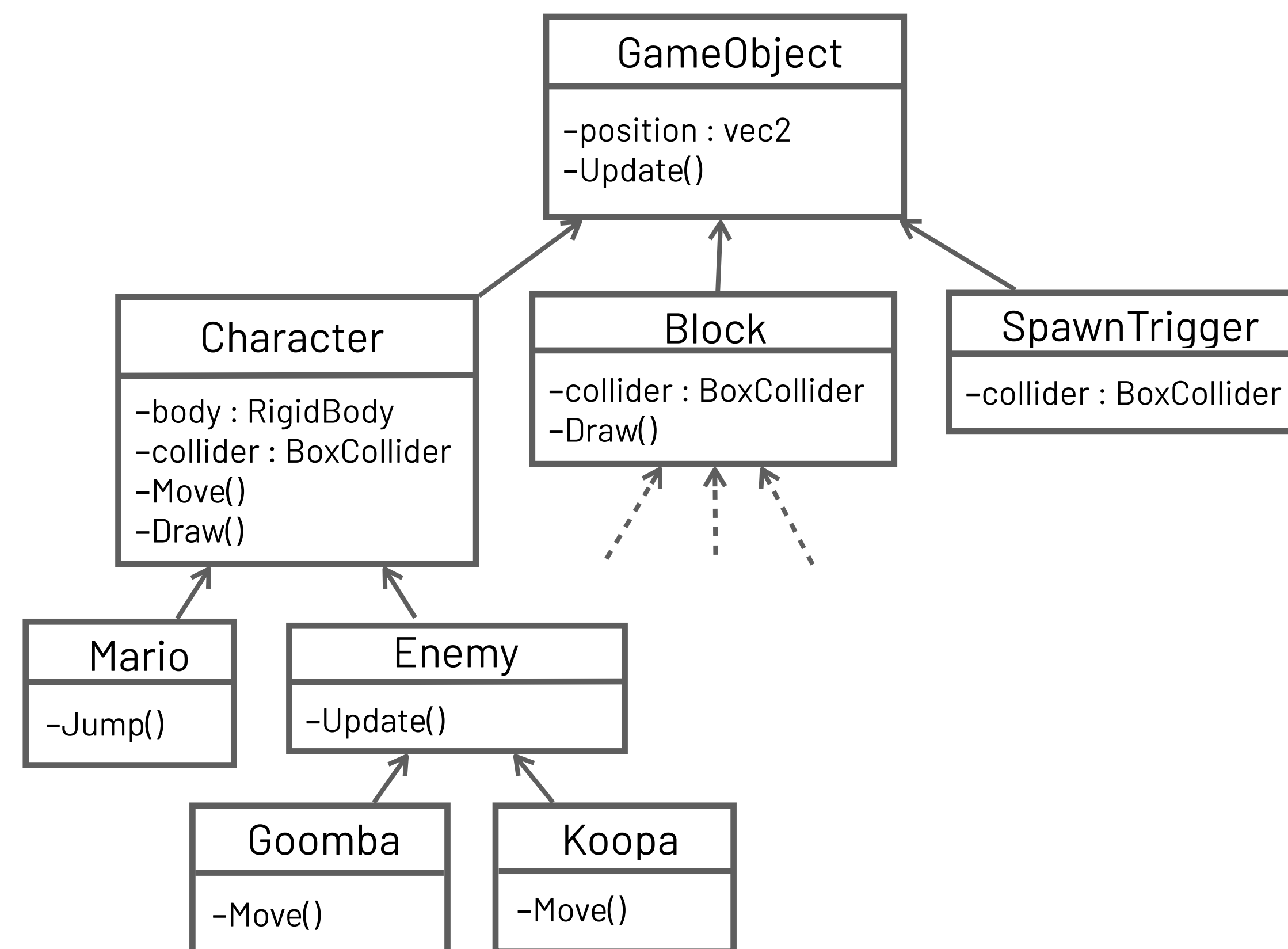
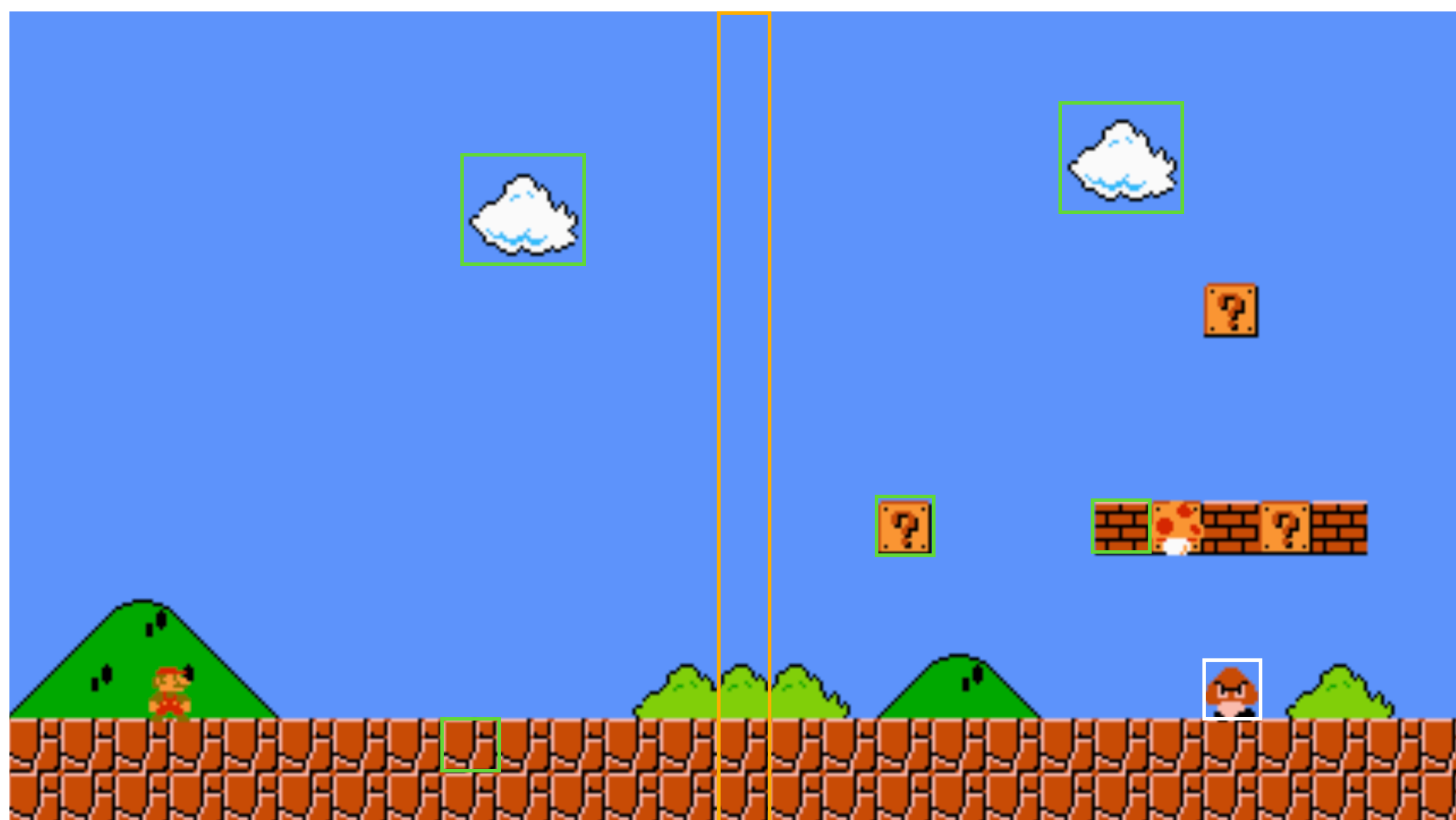
Existem três técnicas principais para representar game objects:

- ▶ Modelo de hierarquia de classes
- ▶ Modelo de componentes
- ▶ Modelo híbrido

Modelo de Hierarquia de Classes



No **modelo de hierarquia de classes**, o comportamento dos objetos do jogo é definido e compartilhado utilizando uma hierarquia de classes, com a raiz em um classe base (GameObject)



Problema do Modelo de Hierarquia de Classes



Em geral, objetos em jogos possuem muitas características independentes, o que gera uma explosão combinatória de classes com uma hierarquia profunda

► Hierarquia de classes profundas

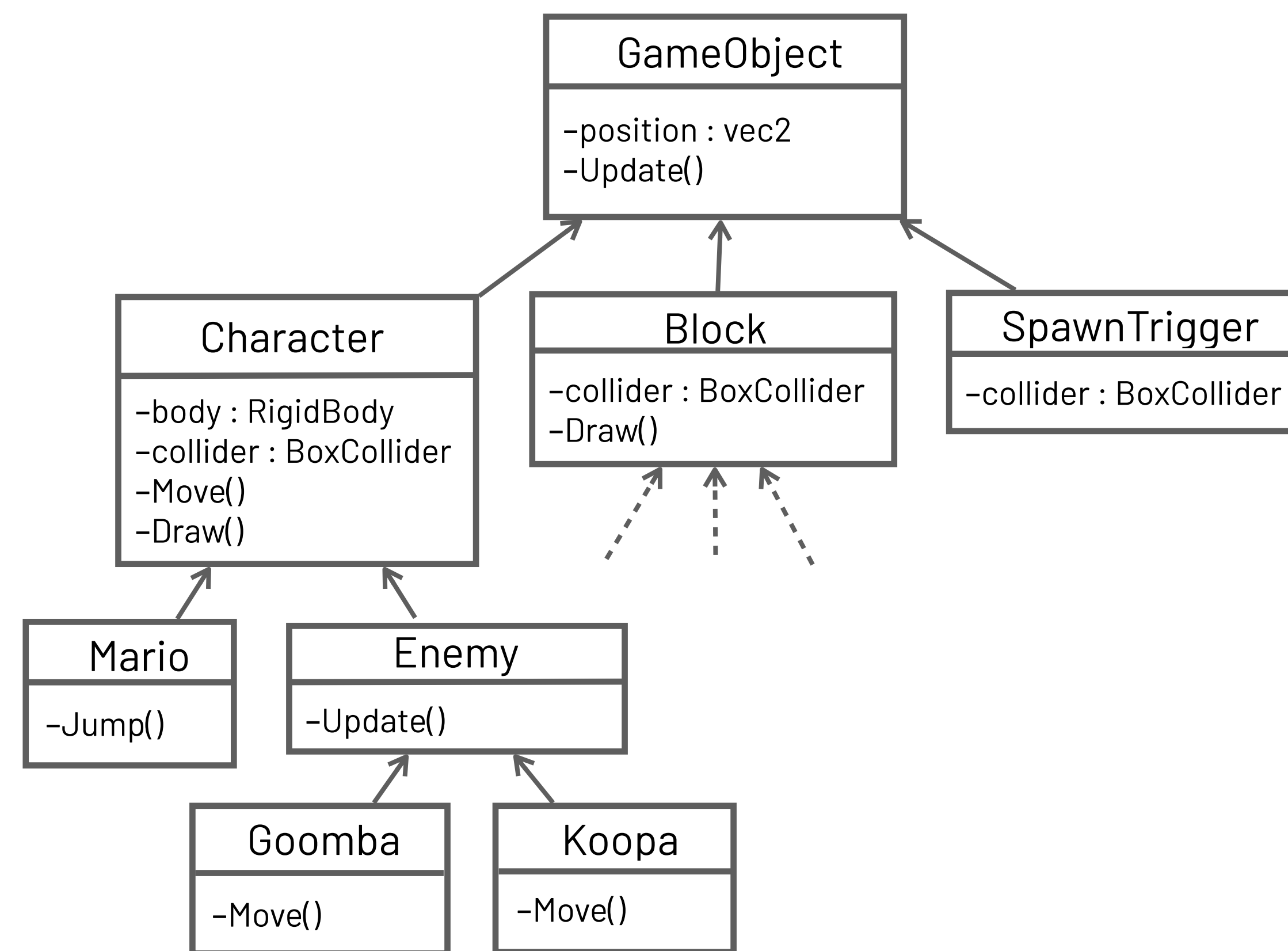
Alterações em classes base podem quebrar inesperadamente as classes derivadas.

► Reutilização limitada

A herança só permite reutilizar código de superclasses, não de classes "irmãs".

► Dificuldade em componentes dinâmicos

Objetos precisam mudar comportamentos durante a execução, o que é complicado com herança pura.

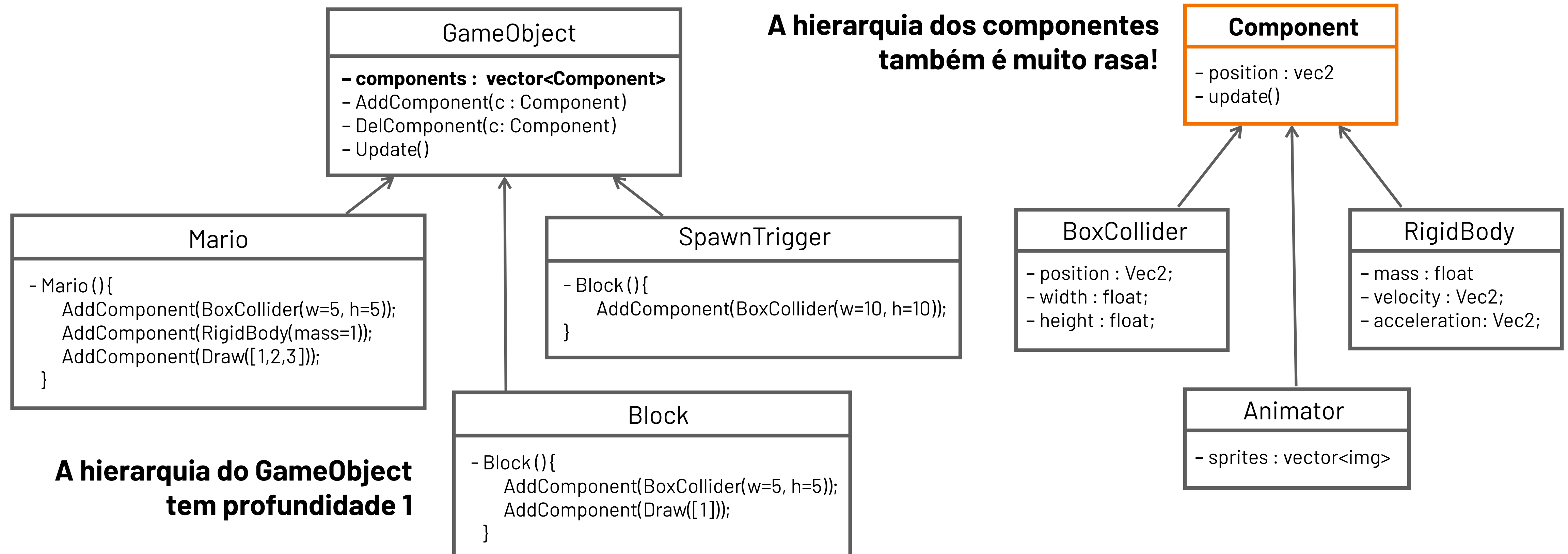


Modelo de Componentes



No **modelo de componentes**, cada objeto do jogo tem uma lista de componentes que, quando combinados, definem a sua funcionalidade (Ex., Unity).

A hierarquia dos componentes também é muito rasa!



Problemas do Modelo de Componentes



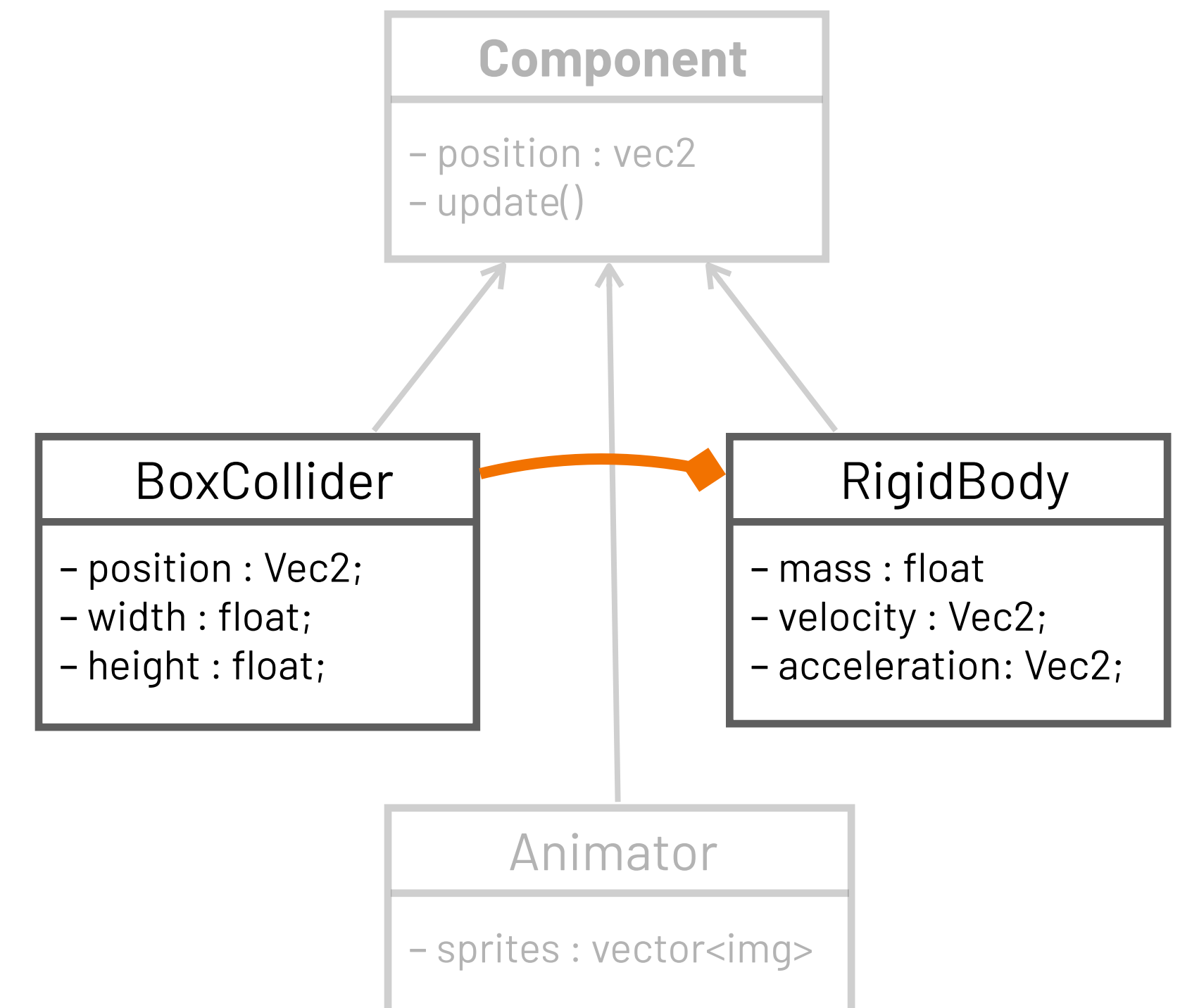
Em geral, diferentes componentes precisam comunicar entre si, fazendo com que haja várias buscas por componentes, prejudicando performance

```
BoxCollider::CheckHorizontalCollision() {}
BoxCollider::CheckVerticalCollision() {}

BoxCollider::Update(float deltaTime) {
    // Get object velocity from Rigidbody
    Rigidbody *body = owner->GetComponent<Rigidbody>();

    if(Math::Abs(body->GetVelocity().x) > 0)
        CheckHorizontalCollision();
}

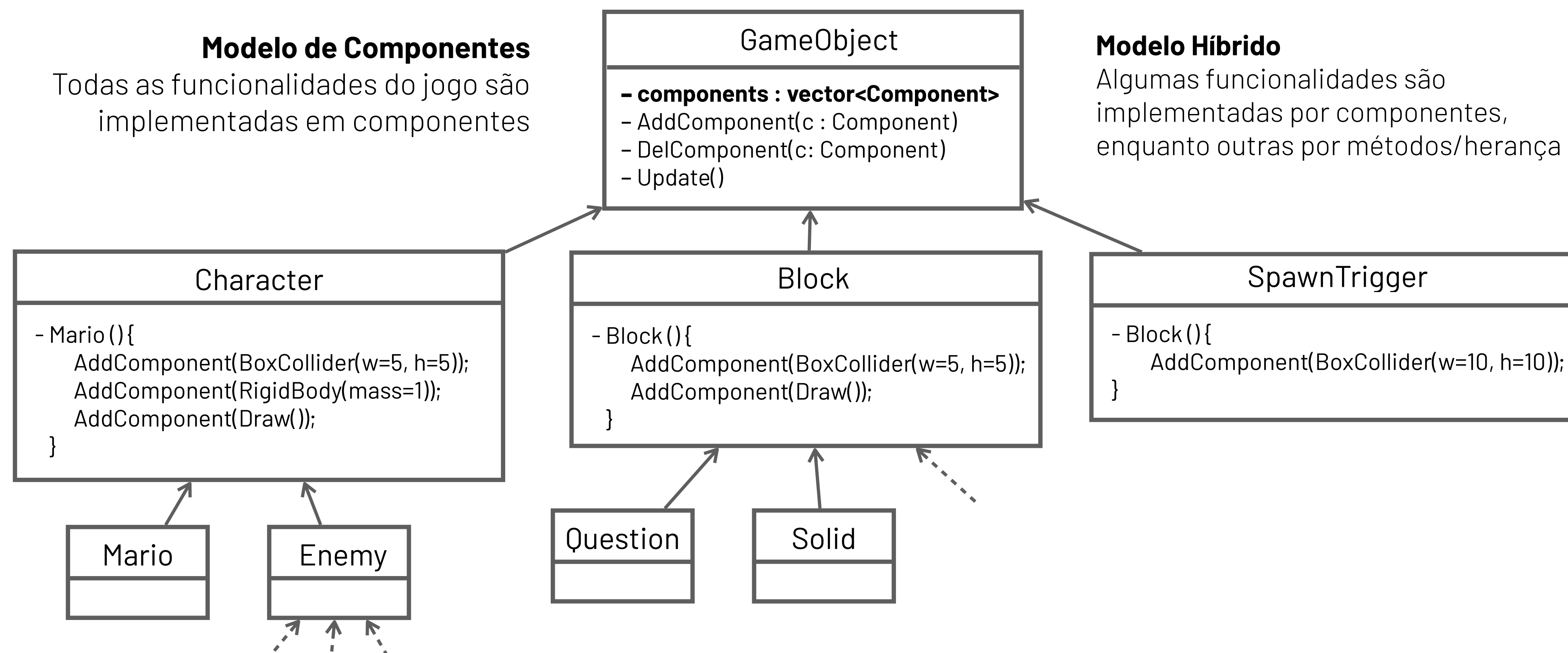
if(Math::Abs(body->GetVelocity().y) > 0)
    CheckVerticalCollision();
}
```



Modelo de Objetos Híbrido



No **modelo híbrido**, combinados uma hierarquia de classes com componentes, ou seja, algumas propriedades/funções são passadas por herança enquanto outras por componentes (Ex. Unreal)



Próxima aula



A5: Lab 1: Pong

- ▶ Introdução ao TP1: Pong
- ▶ Vetores