DCC192

2025/1

UFMG

Desenvolvimento de Jogos Digitais

A8: Detecção de Colisão II

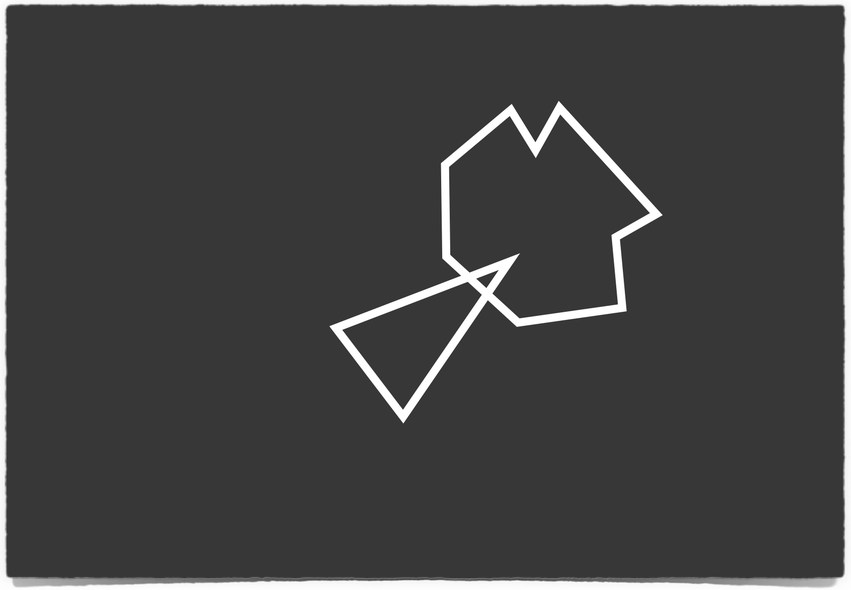
Prof. Lucas N. Ferreira

Plano de aula

- Collisão entre Múltiplos Corpos Rígidos
- Otimizações de Detecção de Colisão
  - Particionamento de Espaço
    - Hashing Espacial
    - Quadtree/Octree
  - Culling de Região Ativa
  - Broad Phase and Narrow Phase

Detecção de Colisão entre Dois Corpos

Na última aula vimos algoritmos para representar e detectar collisão entre dois corpos rígidos:



- Circunferência
- AABBs e OBBs
- Cápsulas
- Polígonos Convexos

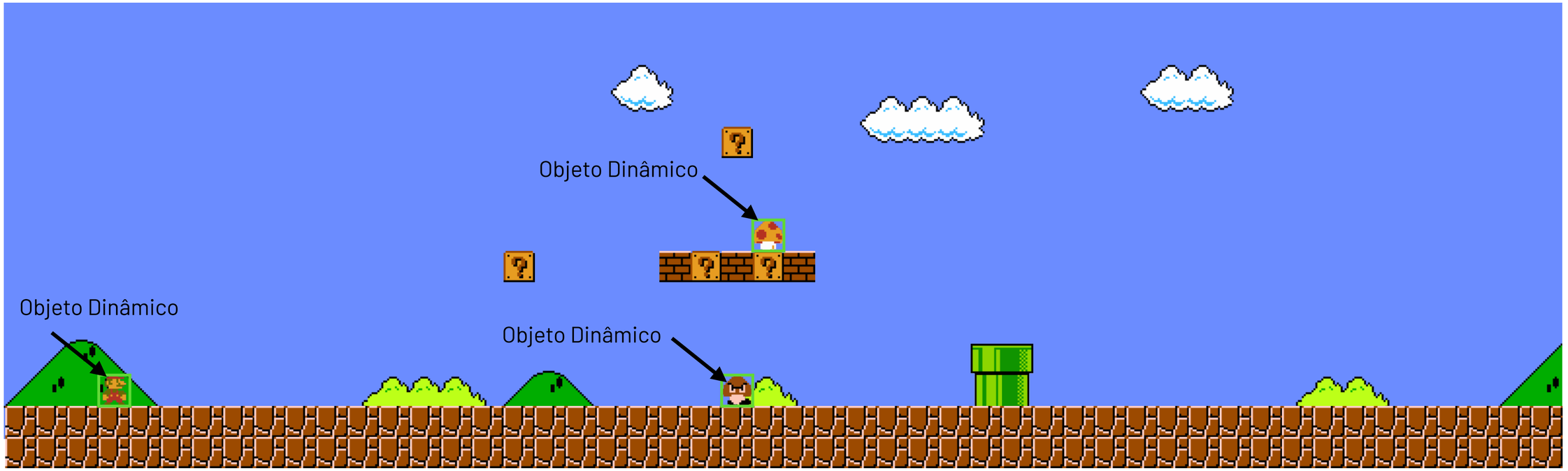
- Outras geometrias importantes incluem pontos, raios, planos, triângulos, ...

Detecção de Colisão entre Multiplos Corpos Rígidos

Na maioria dos jogos, precisamos verificar colisão entre vários objetos rígidos, alguns dinâmicos (que se movem) e outros estáticos (que não), cada um com sua geometria de colisão:

Detecção de Colisão entre Multiplos Corpos Rígidos

Na maioria dos jogos, precisamos verificar colisão entre vários objetos rígidos, alguns dinâmicos (que se movem) e outros estáticos (que não), cada um com sua geometria de colisão:



Detecção de Colisão entre Multiplos Corpos Rígidos

Na maioria dos jogos, precisamos verificar colisão entre vários objetos rígidos, alguns dinâmicos (que se movem) e outros estáticos (que não), cada um com sua geometria de colisão:

Detecção de Colisão entre Múltiplos Corpos Rígidos

A solução mais simples consiste em comparar todos os objetos dinâmicos (que se movem) com todos os objetos, tanto dinâmicos (ex. goomba) quanto estáticos (ex. blocos)


```
for(auto obj1: mDynamicObject) {
   for(auto obj2 : mObjects) {
      if (obj1 != obj2) {
         bool collided = Game::DetectCollision(obj1, obj2);
         if (collided) {
            Game::ResolveCollision(obj1, obj2);
         }
      }
   }
}
```

Otimização de Detecção de Colisão

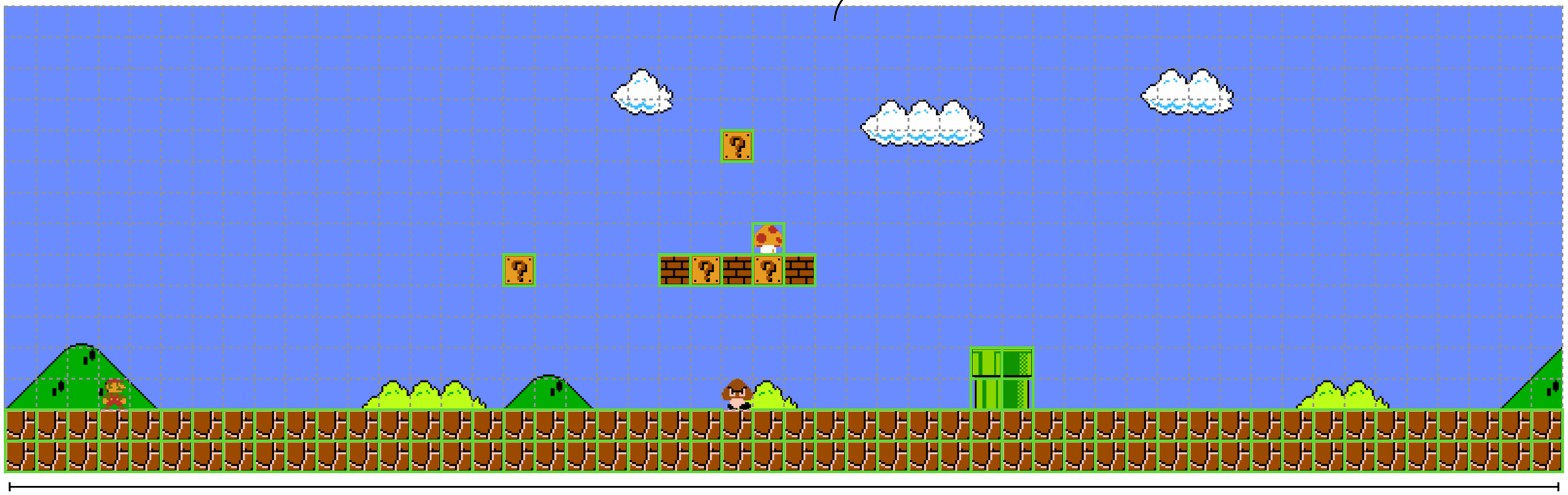
Essa solução não é muito eficiente $O(N^2)$ e pode ser um problema quando temos muitos objetos no jogo. Existem algumas formas mais comuns de otimizar essa solução:

- Particionamento de Espaço
  - Hashing Espacial
  - Quadtree/Octree
- Culling de Região Ativa
- Broad Phase and Narrow Phase

```
for(auto obj1: mDynamicObject) {
   for(auto obj2 : mObjects) {
      if (obj1 != obj2) {
         bool collided = Game::DetectCollision(obj1, obj2);
         if (collided) {
            Game::ResolveCollision(obj1, obj2);
         }
      }
   }
}
```

Particionamento de Espaço: Hashing Espacial

Hashing Espacial é uma solução bastante comum que **divide o espaço** de colisão em uma **grade** de células de tamanho fixo e usa uma função hash para mapear a posição do mundo em uma posição do grid.



Posição no mapa $p_{map} = (x, y)$

Posição na grade $p_{grid} = (\lfloor \frac{x}{c} \rfloor, \lfloor \frac{y}{c} \rfloor)$

Particionamento de Espaço: Hashing Espacial

Hashing Espacial é uma solução bastante comum que **divide o espaço** de colisão em uma **grade** de células de tamanho fixo e usa uma função hash para mapear a posição do mundo em uma posição do grid.

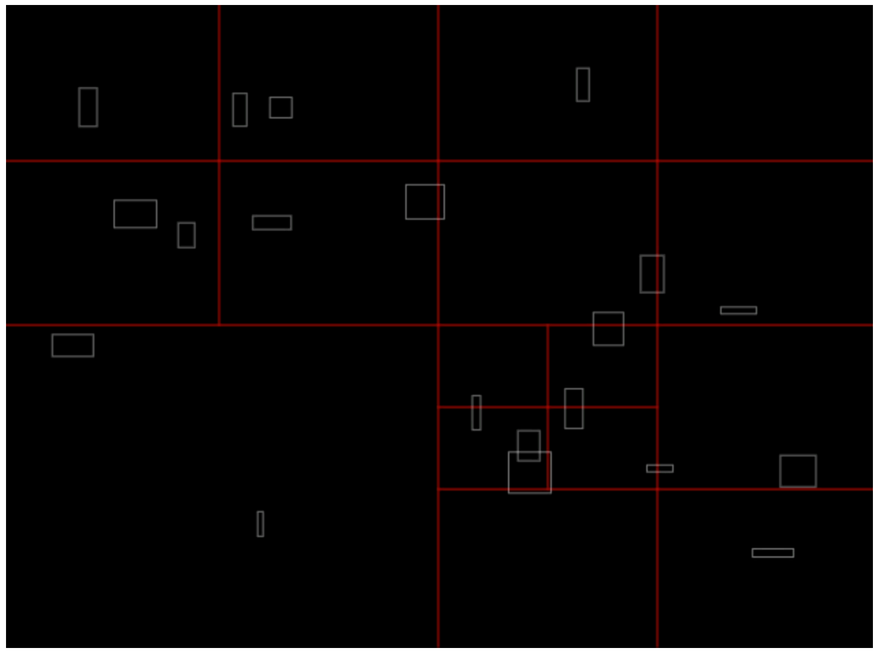
```
RigidBody::Update(float dt) {
   mVelocity += mAccelaration * dt;

   Vector2 gridPos = mPosition + mVelocity;
   gridPos.x = (int)(gridPos.x/CELL_SIZE);
   gridPos.y = (int)(gridPos.x/CELL_SIZE);
   if (CollisionGrid[gridPos.x][gridPos.y] == 1) {
      // Colisão – verificar em quais eixos e zerá-los
   }
   else {
      mPosition += position * dt;
   }

   mAcceleration.Set(0f, 0f);
}
```

Posição no mapa $p_{map} = (x, y)$

Posição na grade $p_{grid} = (\lfloor \frac{x}{c} \rfloor, \lfloor \frac{y}{c} \rfloor)$

Particionamento de Espaço: Quadtree

Quadtree é uma outra estratégia que utiliza uma árvove para **dividir o espaço** de colisão ricusivamente em quadrantes, possibilitando verificar colisão apenas entre objetos que estão no mesmo quadrante



1. **Divisão do espaço**: A quadtree começa com um nó raiz que representa toda a área do jogo.

2. **Subdivisão recursiva**: Quando um nó contém mais de um número máximo predefinido de objetos (capacidade), o nó se divide em quatro quadrantes iguais: noroeste, nordeste, sudoeste e sudeste.

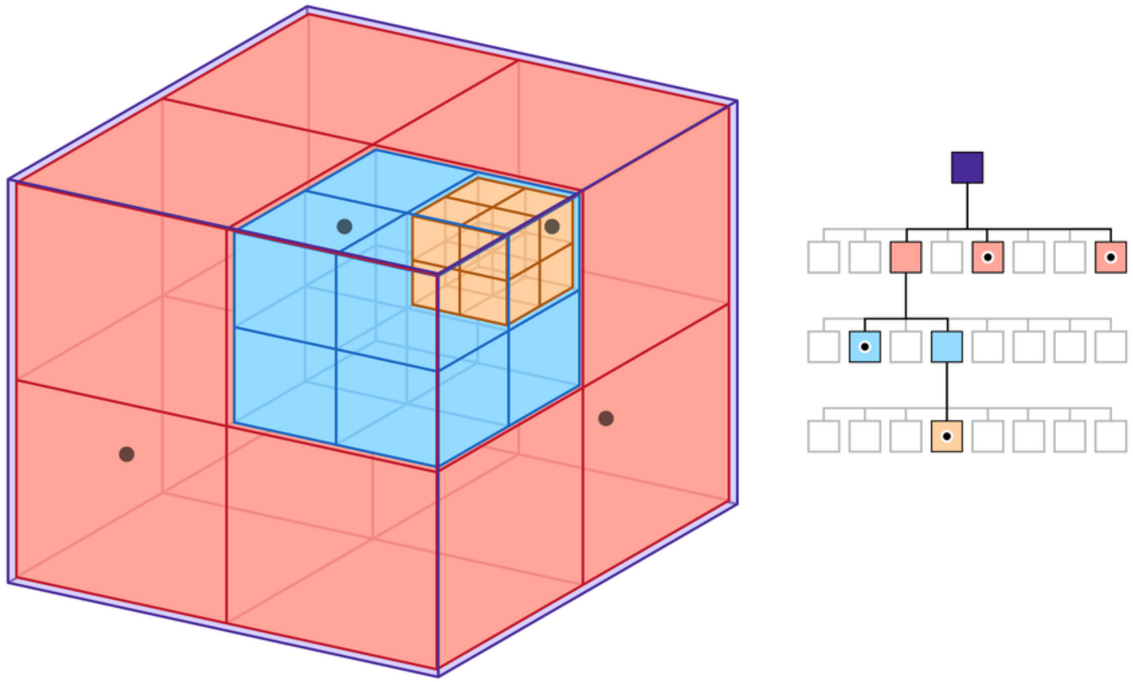
3. **Inserção de objetos**: Os objetos são inseridos nos quadrantes com os quais eles se sobrepõem. Se um objeto cruza uma borda entre quadrantes, ele pode ser colocado no nó pai ou em múltiplos quadrantes.

4. **Busca eficiente**: Para verificar colisões, basta consultar os quadrantes específicos onde um objeto está localizado, em vez de verificar contra todos os objetos do jogo.

https://jimkang.com/quadtreevis/

Particionamento de Espaço: Octree

Octree é a versão em 3D da quadtree. Ao invés de dividir o espaço em 4 quadrantes, ela o divide em 8 octantes iguais. Sendo assim, cada nó da árvore tem exatamente 8 filhos, ao invés de de 4.



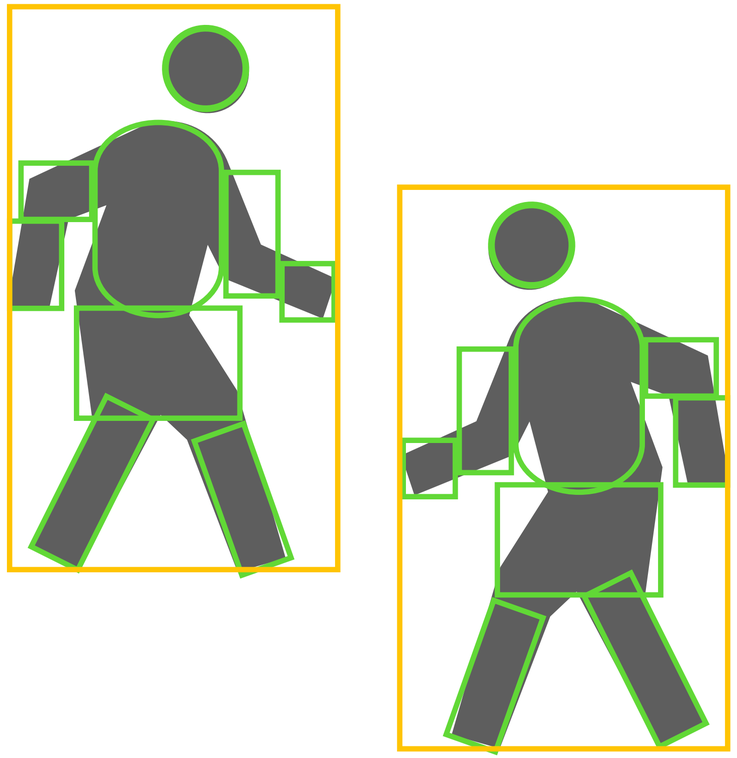
https://threejs.org/examples/games_fps.html

Broad Phase vs. Narrow Phase

Quando os objetos do jogo possuem geometrias complexas (e.g., listas de geometrias), podemos otimizar a detecção de colisão com um algoritmo em duas etapas :

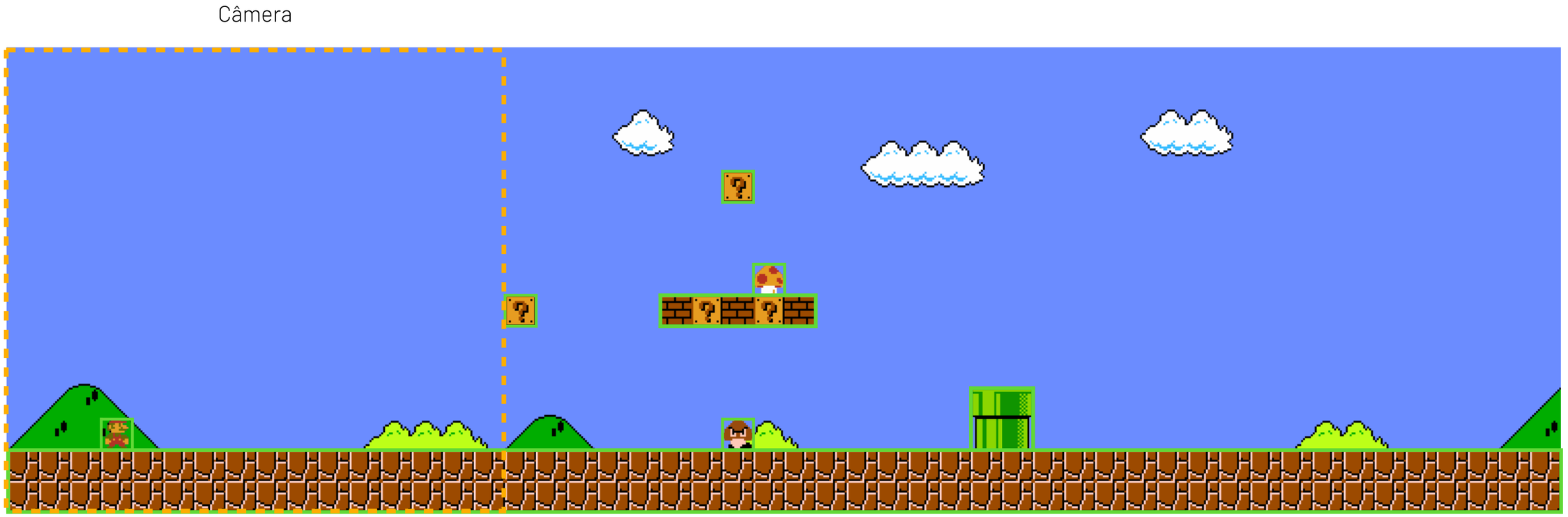
1. **Broad Phrase**: filtrar rapidamente objetos que não podem colidir usando particionamento espacial

2. **Narrow Phase**: Execute testes de colisão precisos apenas em pares potencialmente colidentes



Culling de Região Ativa

O **Culling de Região Ativa** limita a verificação de colisões e atualizações de lógica apenas aos objetos próximos ao jogador ou à câmera.



https://threejs.org/examples/games_fps.html

Próxima aula

A9: Gráficos 2D

- Sprites/Spritesheets
- Animações
- Câmeras 2D
- Tilemaps