1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA

dei25 departamento de engenharia informática 1995 – 2020

Computação Gráfica

André Perrotta (avperrotta@dei.uc.pt)

Hugo Amaro (hamaro@dei.uc.pt)

T_07:

Game Logics: fundamentos de movimento

Objetivos da aula

- Entender conceitos básicos de física e lógica utilizados em videojogos.
- Fazer uma relação entre estes conceitos e as necessidades do projeto.

Tópicos abordados

- Física
  - Movimento
    - Linear
    - Acelerado
    - “easing”
  - Força
    - Impulso (pular)
    - Cair/quicar
- Colisões
  - paredes
  - AABB – axis aligned bounding box
  - Outras (visão geral)
- Organização
  - Máquina de estados

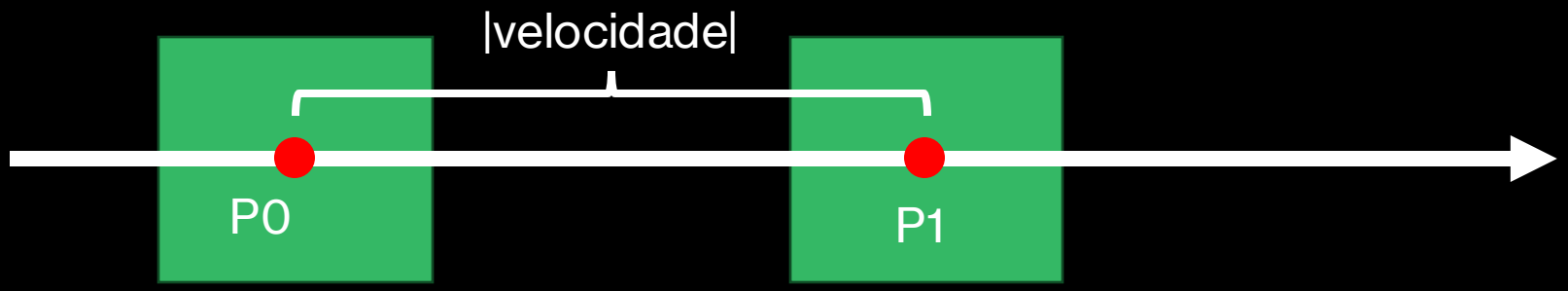
Metodologia

- Para esta aula funcionar, temos de olhar não só para o conceito e quando necessário para a matemática, mas também para o código das implementações.
- O código para esta aula está no Ucstudent

- CG_LEI_2024_T07_gameLogics_src.

Movimento linear

- O movimento mais simples e fácil que podemos dar à um elemento do nosso game é o linear:
  - Posição nova = posição anterior + vetor velocidade

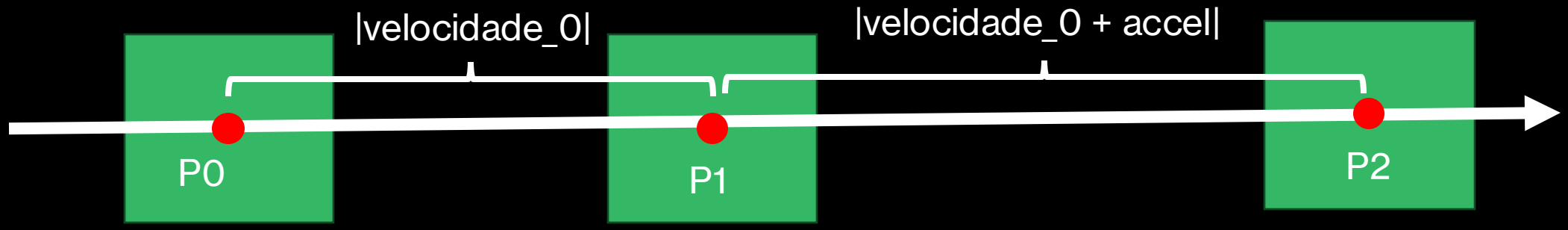


Movimento linear

- Exemplos:
  - Atari2600 – seaquest
  - Atari2600 - pitfall

Movimento acelerado

- Ao tentar simular veículos (carros, aviões, etc.), temos de utilizar uma aproximação melhor para a física do movimento, adicionando a aceleração, que é a variação da velocidade. Isto permite aumentar ou diminuir a velocidade (frear) de forma continua.
  - Vetor velocidade novo = vetor velocidade anterior + vetor aceleração.
  - Posição nova = posição anterior + velocidade nova



Movimento acelerado

- Exemplos:
  - Atari2600 – enduro
  - Atari2600 – river raid
  - Asteroids

Movimento easing

- Elementos da cena que precisam de um movimento autônomo (automático) ou algum tipo de animação, normalmente utilizam uma estratégia de “easing” para “suavizar” o movimento. É muito utilizado também em efeitos de câmera, para contrariar rotações ou relocações abruptas.
- O easing não utiliza simulação de física, mas um algoritmo concebido para esta finalidade.
- A base do algoritmo é o ”target”. O target pode ser uma rotação (ângulo), velocidade, posição, tamanho, etc. O que quisermos, o algoritmo é polivalente. E um valor de “easing”, que deve ser um valor Real entre 0. e 1.

```
Se(!chegou){
        Valor novo += (target – atual)*easing
        se(|distância(target, atual)| < menor unidade do valor){
                chegou = true
                novo = target
        }
}
```

Movimento easing

- Exemplos:
  - É um princípio básico das animações, aparece em praticamente toda e qualquer animação 2D ou 3D.

Força

- Sempre que queremos que os elementos do nosso jogo tenham um comportamento dinâmico (movimento) “similar” ao que teriam no mundo real, temos de utilizar as equações da Física clássica para implementar o movimento:
  - Força = massa * aceleração
  - Ou
  - Aceleração = Força/massa

- A Força da gravidade é constante em módulo e sentido. Para fazer objetos “cair”, tenho de utilizá-la!

Impulso

- Impulso é uma Força aplicada durante um tempo determinado. Sempre que queremos mudar o estado atual de um corpo, temos de lhe dar aceleração, para isso precisamos atuar com uma Força durante um período de tempo.
- Nos games, é assim que fazemos um elemento (por exemplo) “pular”:
  - Aceleração = (força do pulo nos instantes iniciais – força da gravidade)/massa
  - Velocidade += aceleração
  - Posição += velocidade
  - Após os instantes iniciais do pulo, fazemos que “força do pulo” = 0

Força: impulso

- Exemplos:
  - Mario
  - Atari2600 pitfall

Quantidade de movimento

- Para fazer os objetos “quicarem” após cairem e baterem no chão, tenho de utilizar a lei da conservação da quantidade de movimento:
  - Conservação da quantidade do movimento:
    - massa*vel_inicial = coeficiente_elástico*mass*velocidade*final
    - Ou, simplificando:
      - Velocidade_nova = coefciente_elástico*velocidade_inicial
      - O coenficiente elástico pode ser qualquer valor Real. Se ele for negativo, a velocidade resultante após a colisão será na direção oposta à inicial.

Quantidade de movimento: ”bounce”

```
void Movement::bounce()
{
    avatarVelMod += Fg / avatarMass;
    avatarPos.y += avatarVelMod;

    if (avatarPos.y - avatarSize * 0.5 <= -gh() * 0.5)
    {
        avatarPos.y = -gh() * 0.5 + avatarSize * 0.5;
        avatarVelMod *= -bounciness;
    }
}
```

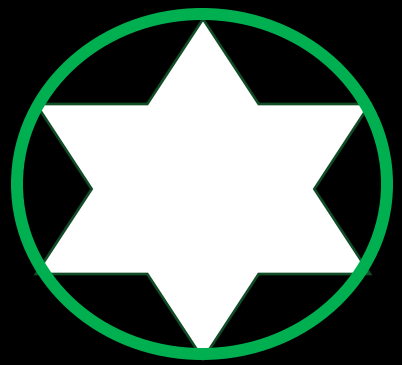
CG_LEI_2024_PL08_gameLogics_src.zip

Colisões

- Nos videojogos, para detetar a interação entre os elementos, muitas vezes é necessário calcular se os elementos estão encostados ou sobrepostos. Neste contexto esta tarefa é a “deteção de colisão”. Em Inglês: collision detection
- Collision detection é uma área enorme no universo de CG, e pode ser mais ou menos complexa/sofisticada dependendo dos objetivos.
- Em videojogos (usuais) utiliza-se simplificações.
  - Precisa funcionar em tempo-real
- Em aplicações para teste e simulação em contexto industrial, utilizam-se implementações mais sofisticadas.
  - Não precisa funcionar em tempo-real
- Em filme/CGI, também é possível utilizar deteção de colisão realista, pois não precisa funcionar em tempo-real.

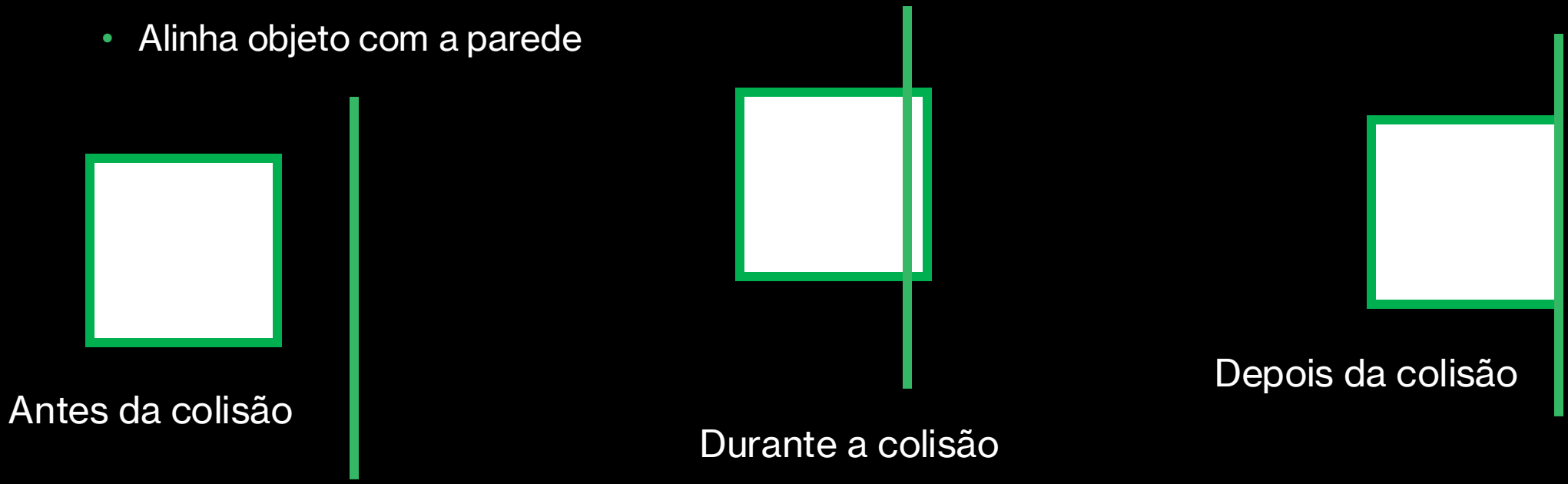
Colisões (videojogos)

- Nos videojogos, é usual utilizarmos simplificações para a geometria de colisões dos objetos.
- Essa geometria é chamada de “collider”



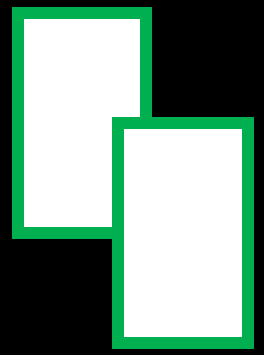
Colisões

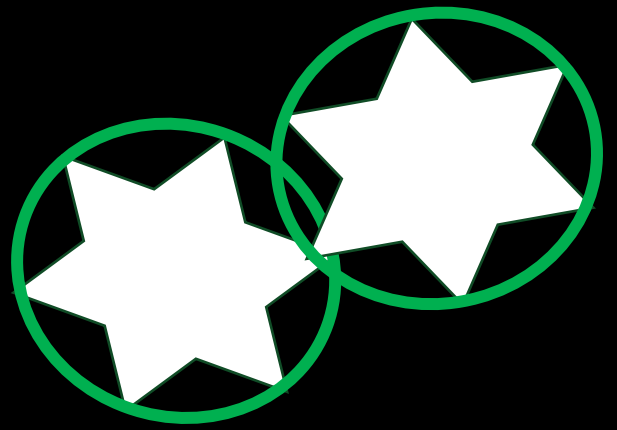
- A colisão mais simples é a de um “collider” com uma parede:
  - If(collider side >= parede)
    - Collision = true
    - Alinha objeto com a parede



Colisões

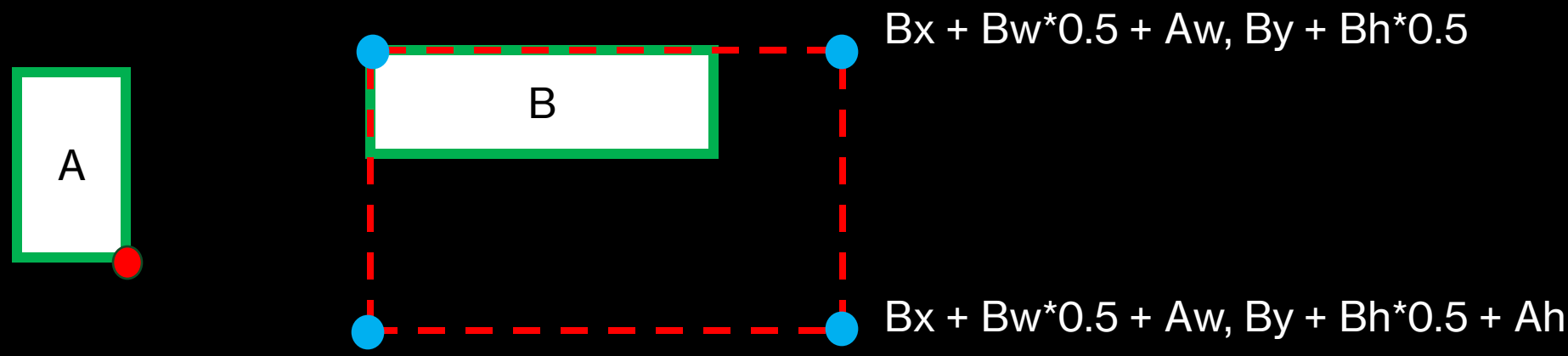
- Quando a colisão acontece entre objetos (colliders), existem várias possíveis estratégias. As mais simples são:
  - Axis aligned bounding box (AABB)
  - Circle/sphere collision





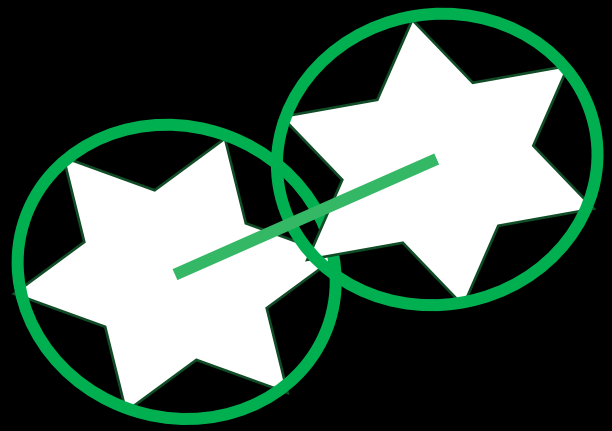
Colisões: AABB

- Na colisão do tipo AABB, basta testar se um dos vértces do objeto A está dentro do volume criado pela união dos objetos A e B.



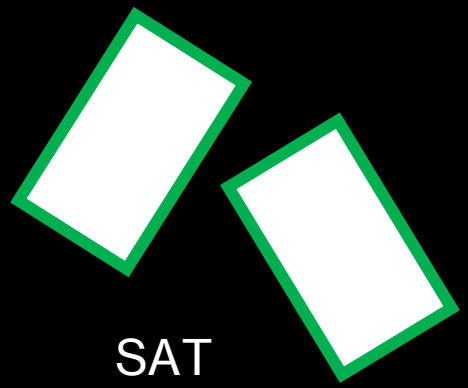
Colisões: sphere/circle

If(dist(A, B) <= rA+rB)
Collision = true



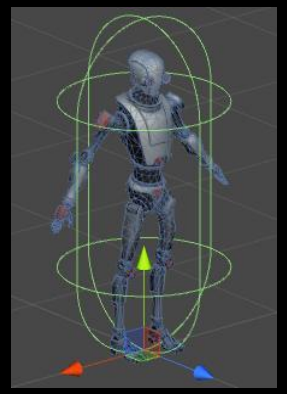
Colisões: outras

- Dependendo da complexidade do jogo, outras estratégias devem ser utilizadas:
  - Separating axis theorem (SAT)
  - Capsule Collider
  - Situações caso-a-caso (por exemplo, objetos não convexos)



SAT

capsule



Máquina de estados

- Um videojogo usualmente possui 2 estágios: setup, loop.
- O loop deve realizar 3 tarefas em sequência:
  - 1 obtem comandos do utilizador
  - 2 atualiza o mundo
  - 3 desenha o mundo

- Felizmente, o OpenFrameworks possui esta estrutura pronta para utilizarmos!

Máquina de estados

- No que toca a parte dos loops, tenho de ter uma estratégia para atualizar e desenhar o mundo para diversos momentos distintos no jogo. Para isso utilizamos uma “máquina de estados”.
- O jogo conta com uma variável “estado” e dependendo do seu valor, diferentes ações serão tomadas.
- Por exemplo:
  - Estado = 0 -> nada é desenhado, algumas varáveis são configuradas e o estado passa para estado = 1
  - Estado = 1 -> uma animação/vídeo é desenhada/atualizada. Quando a animação chega ao fim, o estado passa para estado = 2
  - Estado = 2 -> início do jogo/primeira fase. Jogador pode controlar o avatar e interagir com um objeto. Quando esta interação acontece o estado passa para estado = 3
  - Etc...

Dúvidas?

- Já têm as ferramentas necessárias para implementar a grande maioria dos jogos clássicos dos anos 80/90.

- Certamente aparecerão desafios para além destes exemplos, é importante que vocês tentem encontrar soluções.