DCC192

2025/2

UFMG

Desenvolvimento de Jogos Digitais

A7: Física I - Objetos Rígidos

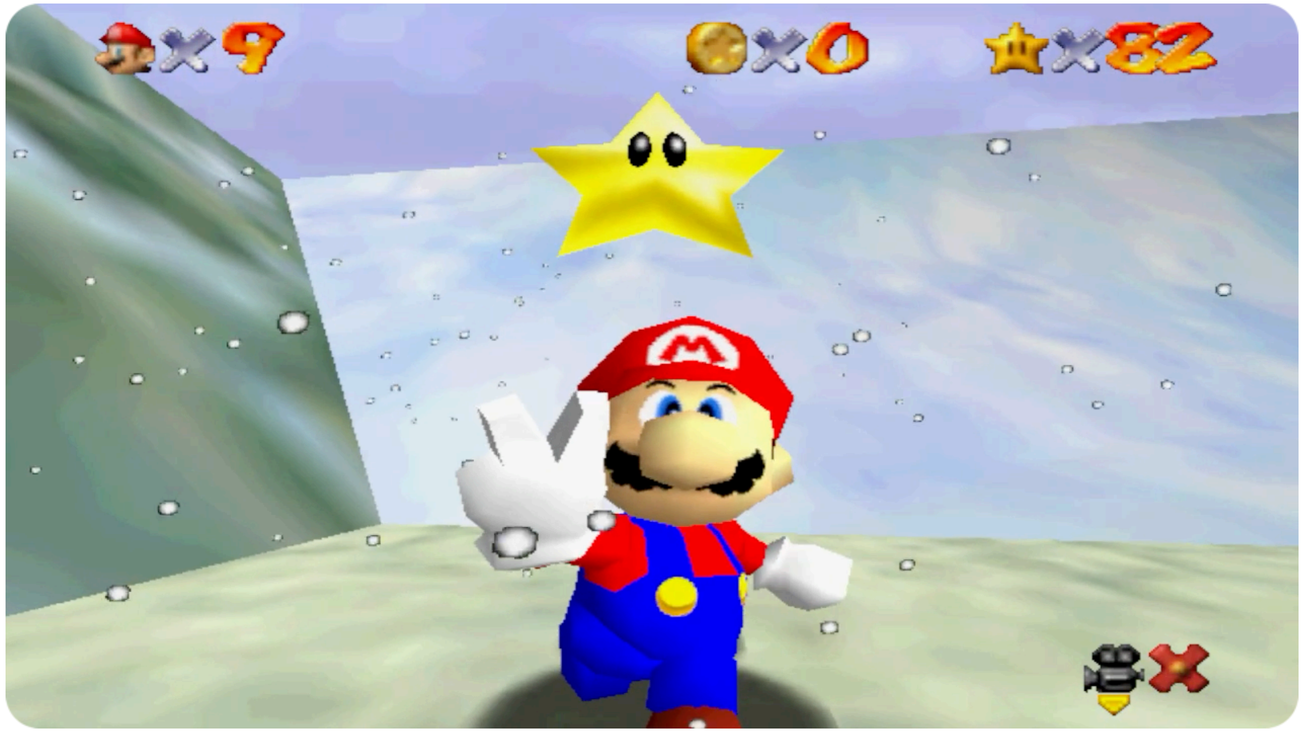
Prof. Lucas N. Ferreira

Plano de Aula

- Física em Jogos Digitais
- Movimentação de Objetos Rígidos
  - Método de Euler Semi-Implícito
- Aceleração da gravidade
- Atrito
- Resistência do Meio

Física em Jogos Digitais

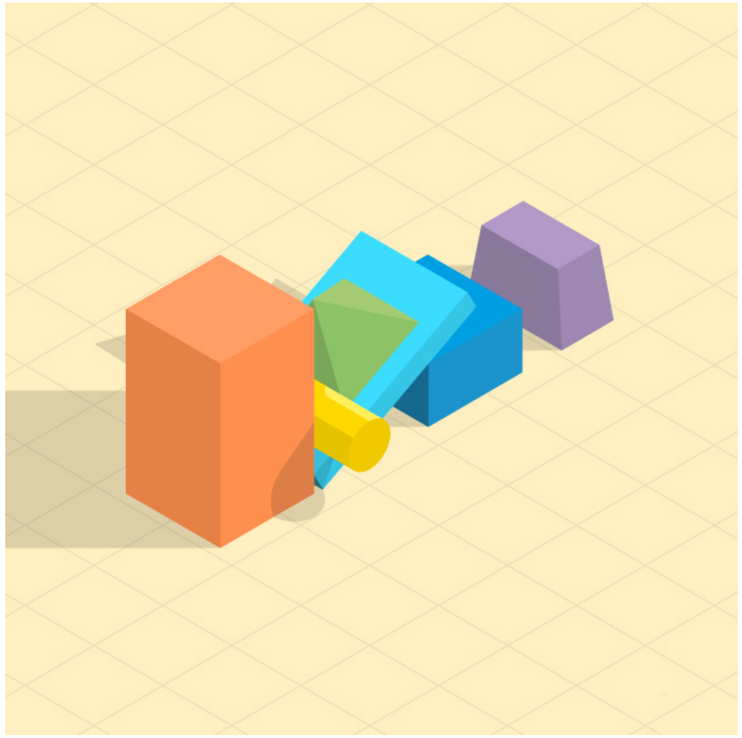
O principal uso de física em jogos é mover objetos segundo as leis da mecânica clássica, ou seja, por meio de **forças** aplicadas pelo jogador ou por outros objetos do jogo. Por exemplo:



- Andar e Correr
- Pular
- Planar (resistência do ar menor)
- Deslizar (em morros inclinados)
- Vento (atuando em uma direção)
- Atrito diferente quando andando no casco
- ...

Objetos Rígidos

Em jogos, geralmente assumimos que os objetos são **Objetos Rígidos,** ou seja, sólidos que não sofrem deformação. As propriedades físicas de objetos rígidos são:

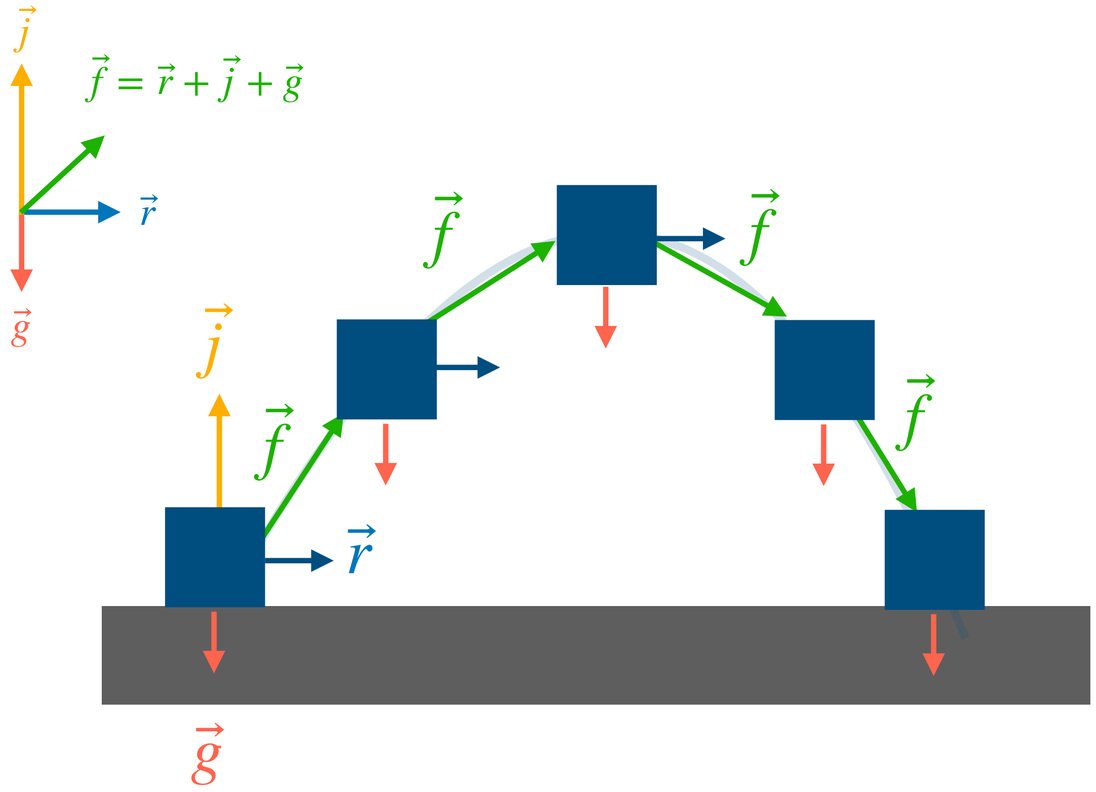


- **Massa** (escalar): quantidade de matéria no corpo
- **Posição** (vetor): localização no espaço (2D ou 3D)
- **Velocidade** (vetor): taxa de variação de posição
- **Aceleração** (vetor): taxa de variação de velocidade

Apesar de objetos rígidos não existirem na vida real, são excelentes simplificações para simulações de objetos em jogos.

Forças em Objetos Rígidos

Uma **força** é um vetor que causa um objeto (rígido) a **acelerar**. Em jogos, múltiplas forças $\vec{f}_i$ podem atual em um mesmo objeto ao mesmo tempo (correr $\vec{r}$, pular $\vec{j}$, gravidade $\vec{g}$, ...)



Segunda Lei de Newton:

Força é igual a massa vezes a aceleração

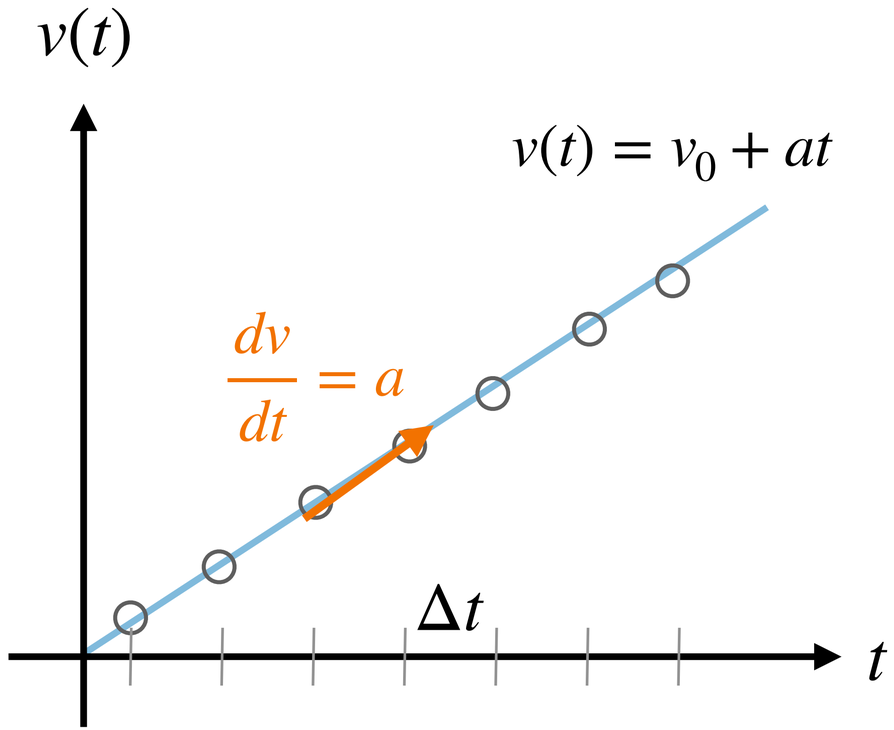
$$\vec{f} = m\vec{a} \longrightarrow \vec{a} = \frac{\vec{f}}{m} \longrightarrow \boxed{\vec{a} = \frac{\sum \vec{f}_i}{m}}$$

Para considerar múltiplas forças, basta somá-las!

- Como temos a aceleração $\vec{a}$, podemos calcular a velocidade $\vec{v}$ e a posição $\vec{s}$ do objeto.
- Porém não estamos em MUV! Então temos que usar cálculo (numérico)!

Movimentação de Objetos Rígidos: Velocidade

Precisamos aproximar a função velocidade $v(t)$ e de posição $s(t)$ em instantes discretos de tempo (quadros do jogo), separados por um intervalo de tempo Δt (*delta time*):



Objetos em jogos possuem aceleração dinâmica, mas a visualização do caso MRUV é mais simples

Derivada aproximada considerando dois quadros consecutivos t e $t + \Delta t$:

$$\frac{dv}{dt} \approx \frac{v(t + \Delta t) - v(t)}{\Delta t}$$

Isolando $v(t + \Delta t)$:

$$v(t + \Delta t) \approx v(t) + \frac{dv}{dt} \cdot \Delta t$$

Substituindo $\frac{dv}{dt} = a$:

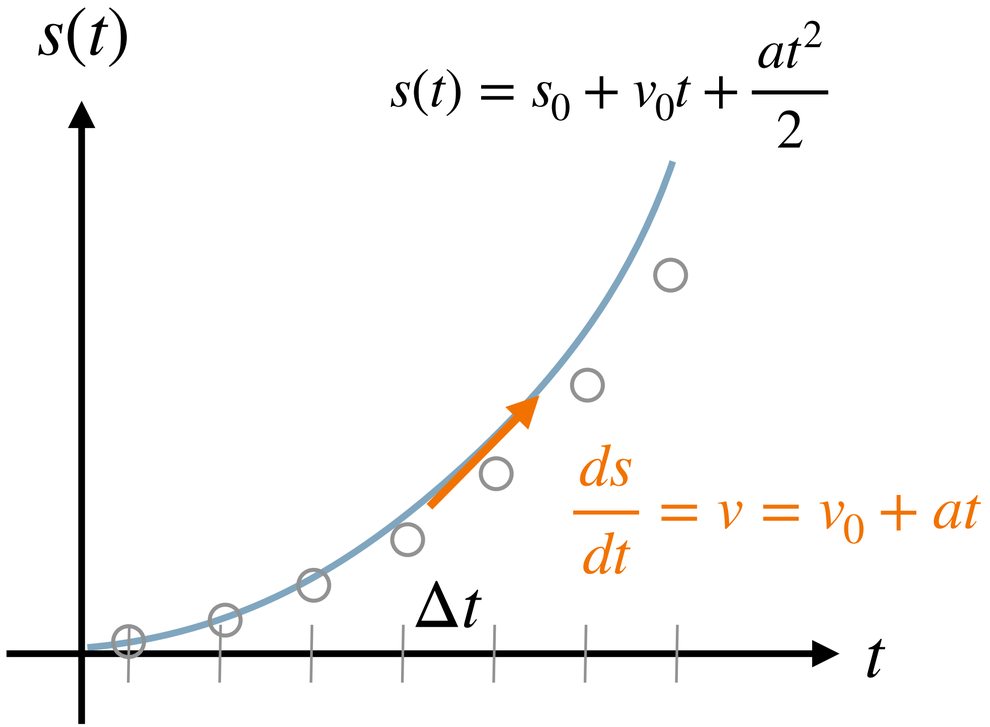
$$v(t + \Delta t) \approx v(t) + a \cdot \Delta t$$

Renomeando as variáveis:

$$v' \leftarrow v + a \cdot \Delta t$$

Movimentação de Objetos Rígidos: Posição

Precisamos aproximar a função velocidade $v(t)$ e de posição $s(t)$ em instantes discretos de tempo (quadros do jogo), separados por um intervalo de tempo Δt (*delta time*):



Objetos em jogos possuem aceleração dinâmica, mas a visualização do caso MRUV é mais simples

Derivada aproximada considerando dois quadros consecutivos t e $t + \Delta t$:

$$\frac{ds}{dt} \approx \frac{s(t + \Delta t) - s(t)}{\Delta t}$$

Isolando $s(t + \Delta t)$:

$$s(t + \Delta t) \approx s(t) + \frac{ds}{dt} \cdot \Delta t$$

Substituindo $\frac{ds}{dt} = v$:

$$s(t + \Delta t) \approx s(t) + v \cdot \Delta t$$

Renomeando as variáveis:

$$s' \leftarrow s + v \cdot \Delta t$$

Esse método de aproximaçao é conhecido como **Método de Euler!**

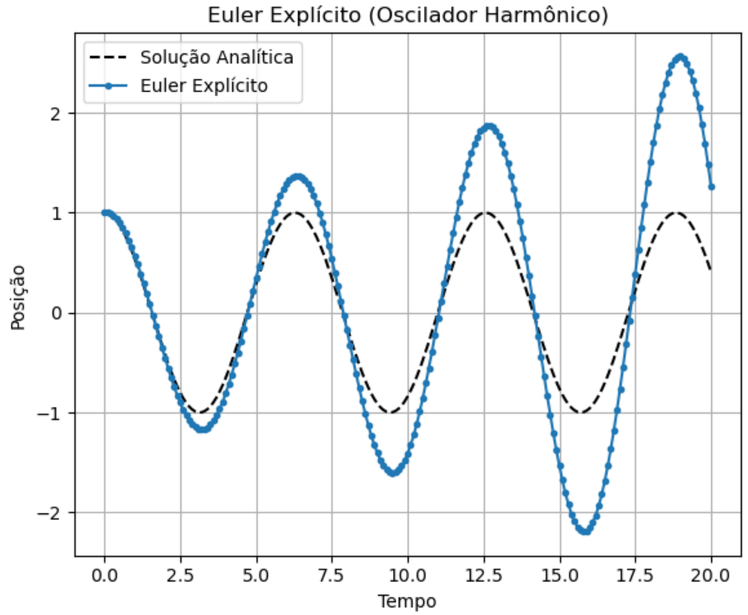
Método de Euler

Existem duas variações básicas do **Método de Euler**:

- **Método de Euler Explícito**

1. $\overrightarrow{s'} = \vec{s} + \vec{v}\Delta t$
2. $\overrightarrow{v'} = \vec{v} + \vec{a}\Delta t$

Primeiro atualiza posição, depois velocidade

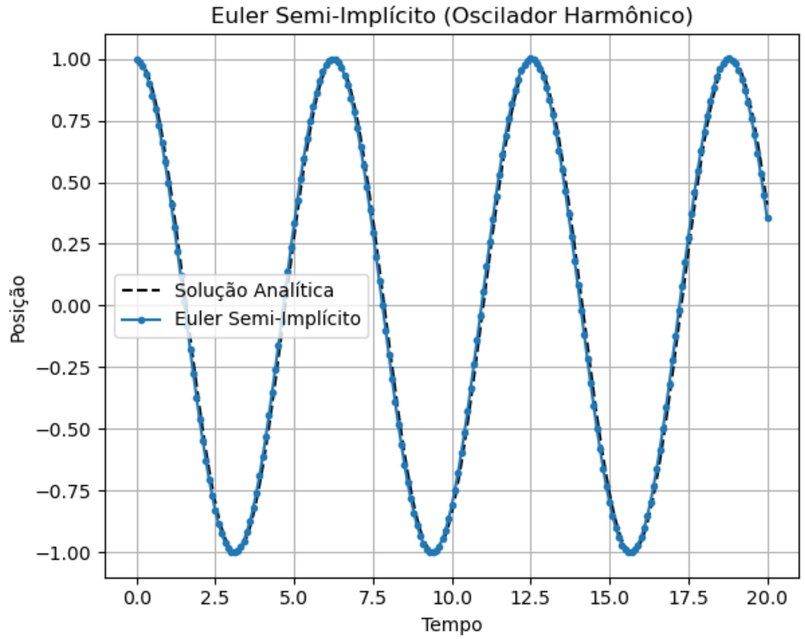


O sistema ganha energia ao longo do tempo!

- **Método de Euler Semi-Implícito**

1. $\overrightarrow{v'} = \vec{v} + \vec{a}\Delta t$
2. $\overrightarrow{s'} = \vec{s} + \overrightarrow{v'}\Delta t$

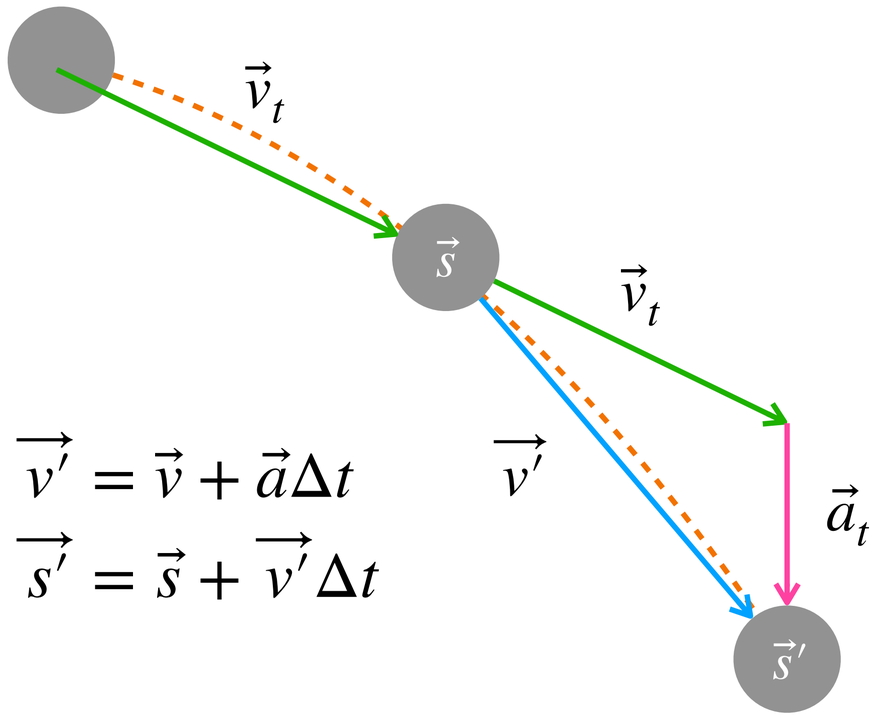
Primeiro atualiza velocidade, depois posição



Mais preciso! Por isso, uma das opções mais comuns para implementação de objetos rígidos em jogos!

Método de Euler Semi-Implícito

Geometricamente, as curvas dos movimentos contínuos reais são aproximados por uma sequência de retas:

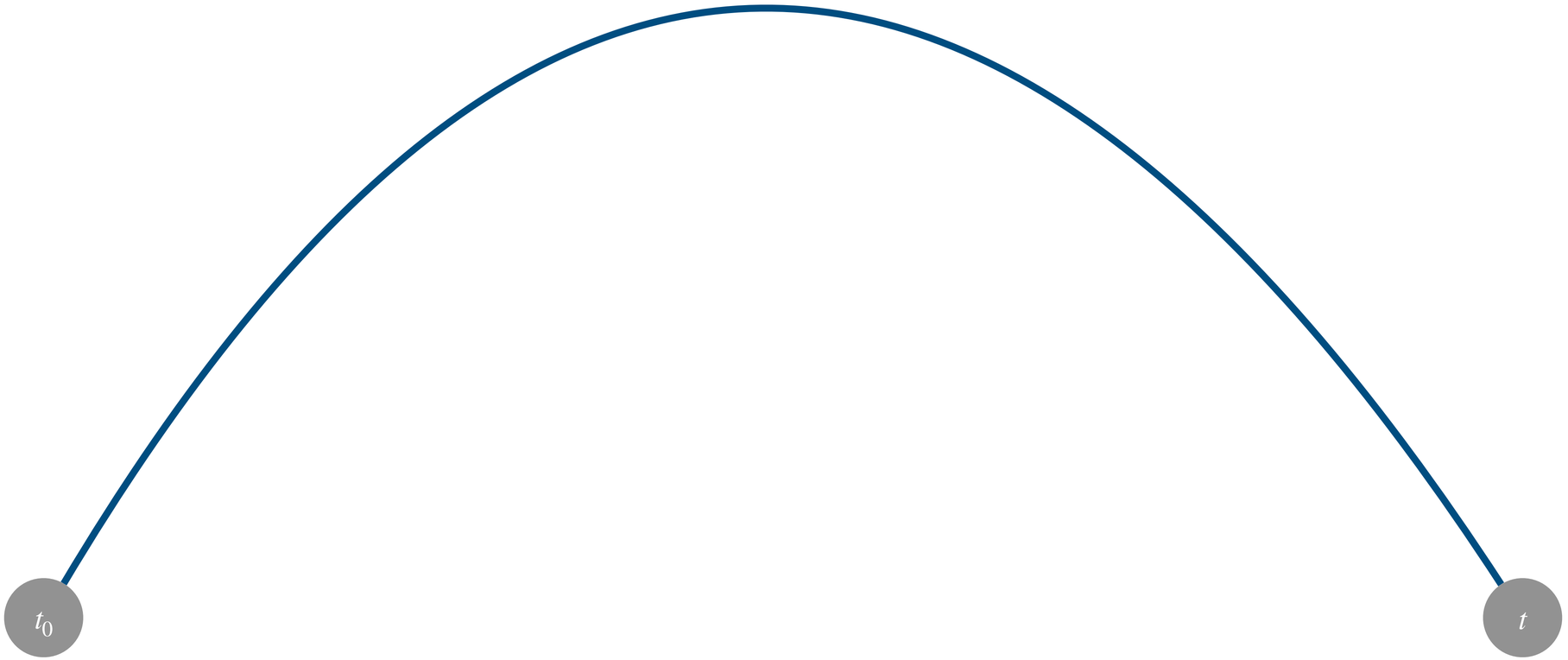



```
void Update(const float deltaTime) {
  mVelocity += mAccelaration * dt;
  mPosition += mVelocity * dt;
  mAcceleration.Set(.0, .0);
}
```

- O intervalo de tempo Δt define a magnitude da aceleração $\vec{a}_t$
- Quanto menor o Δt, melhor a aproximação do **movimento real**.

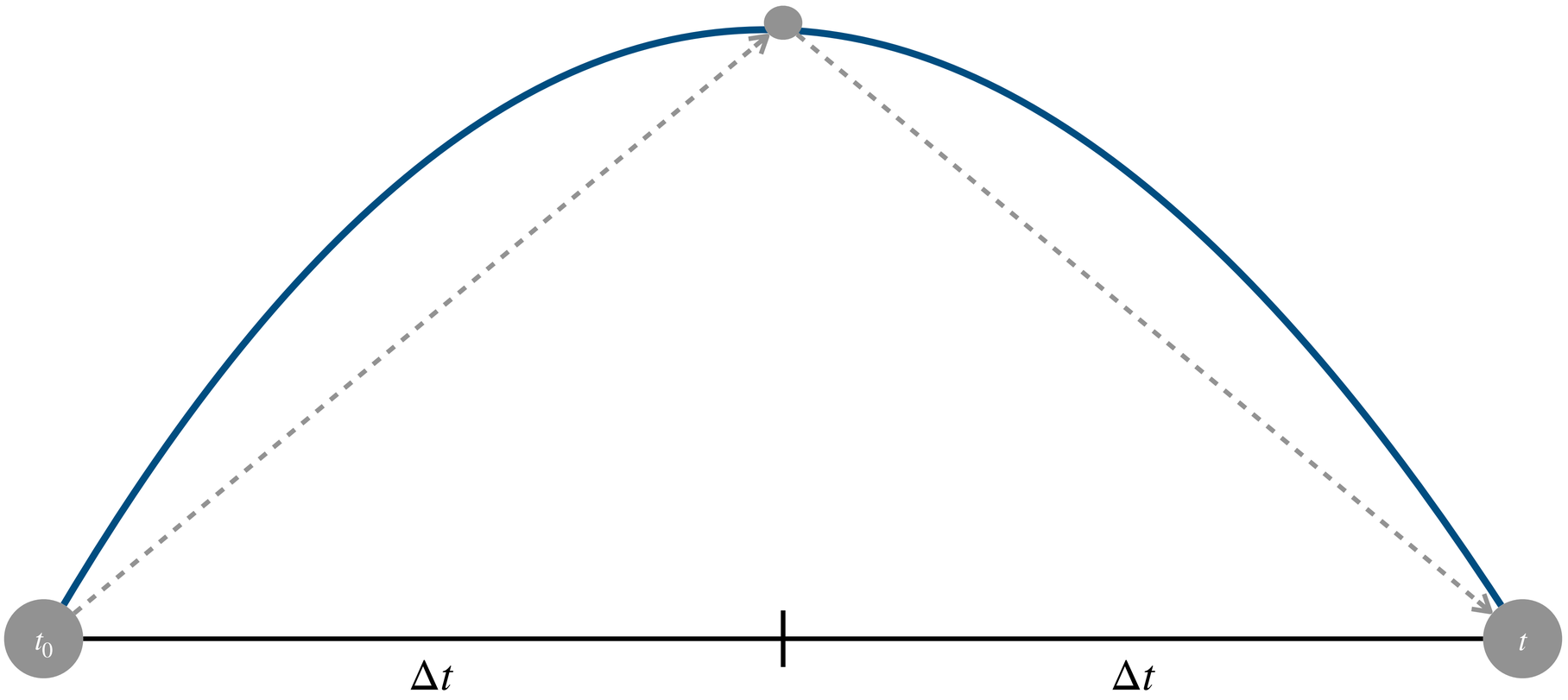
Impacto do tamanho do delta time

Quanto menor o Δt, melhor a aproximação do **movimento real**, ou seja, mais suave será o movimento.



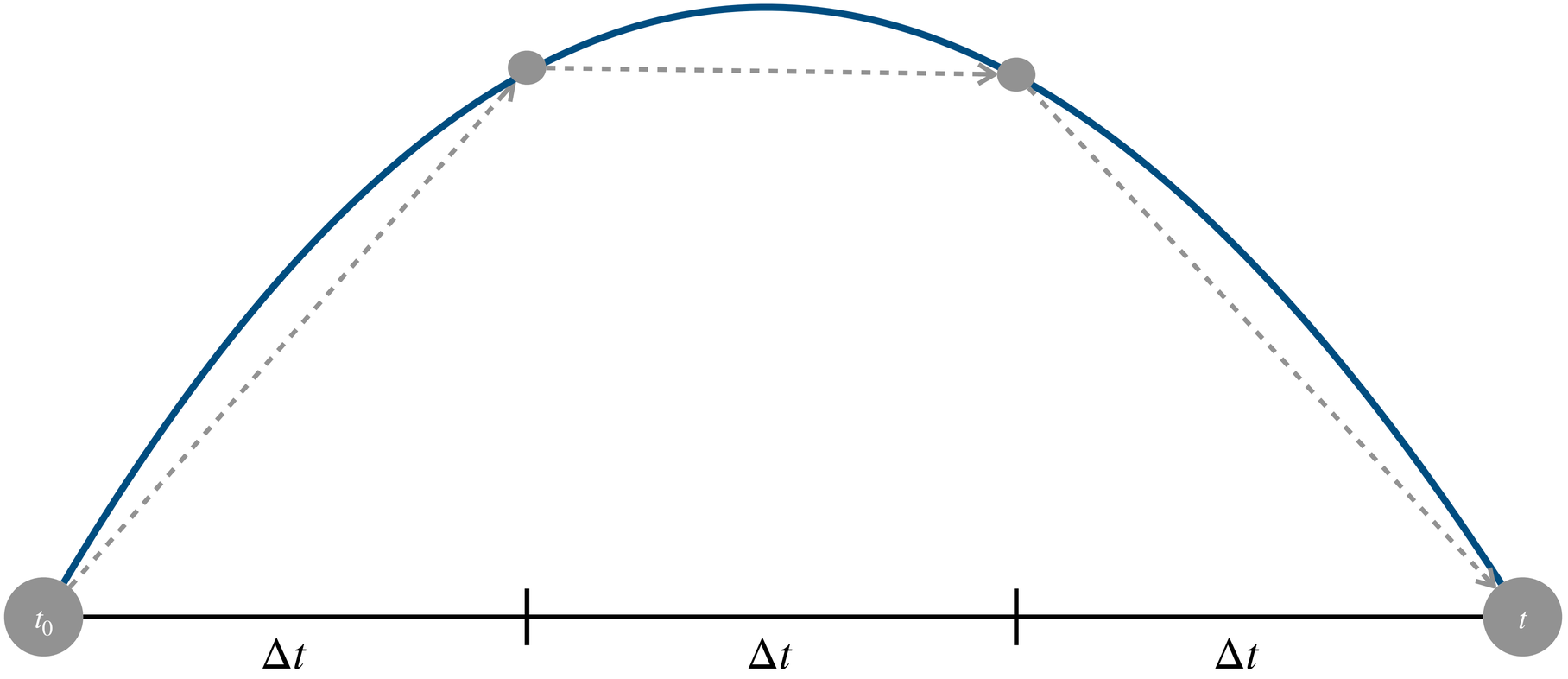
Impacto do tamanho do delta time

Quanto menor o Δt, melhor a aproximação do **movimento real**, ou seja, mais suave será o movimento.



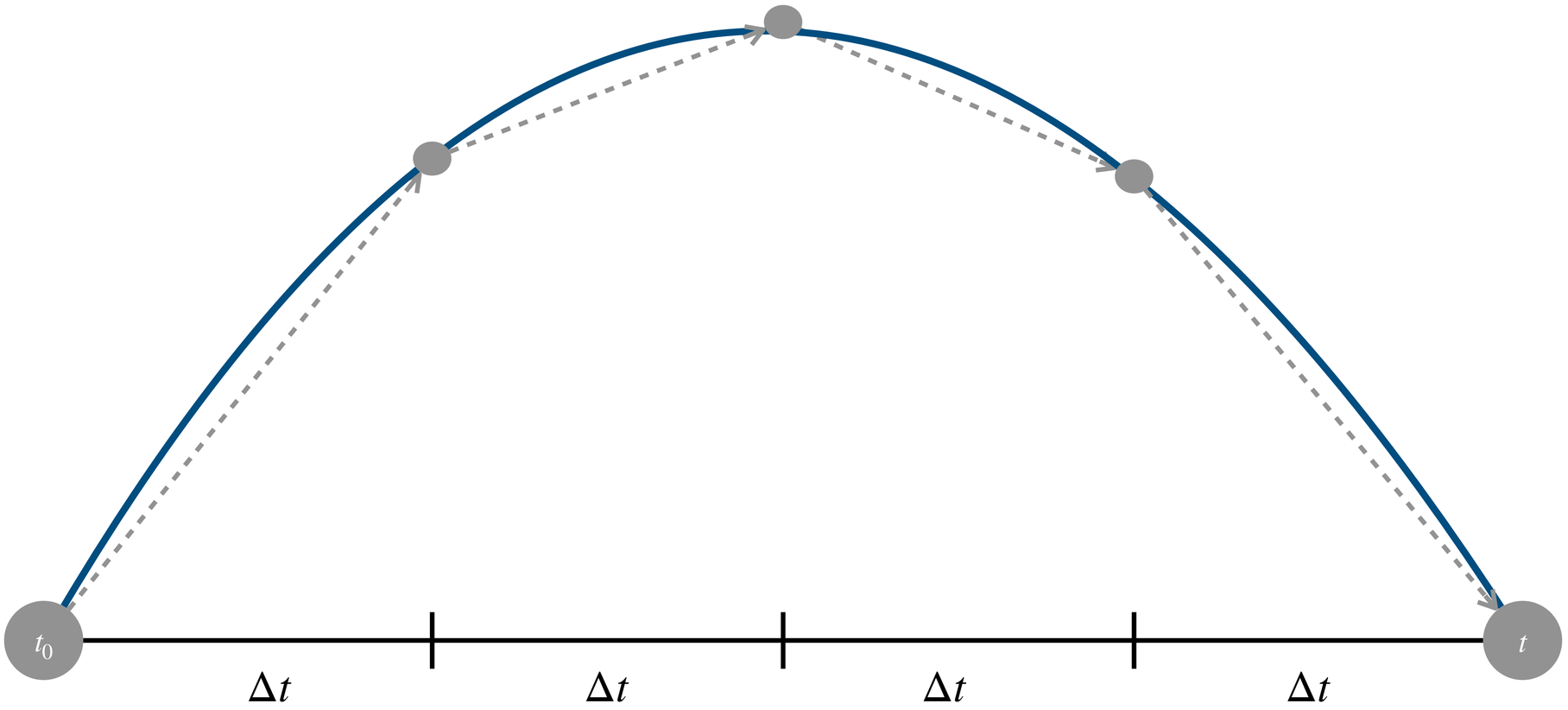
Impacto do tamanho do delta time

Quanto menor o Δt, melhor a aproximação do **movimento real**, ou seja, mais suave será o movimento.



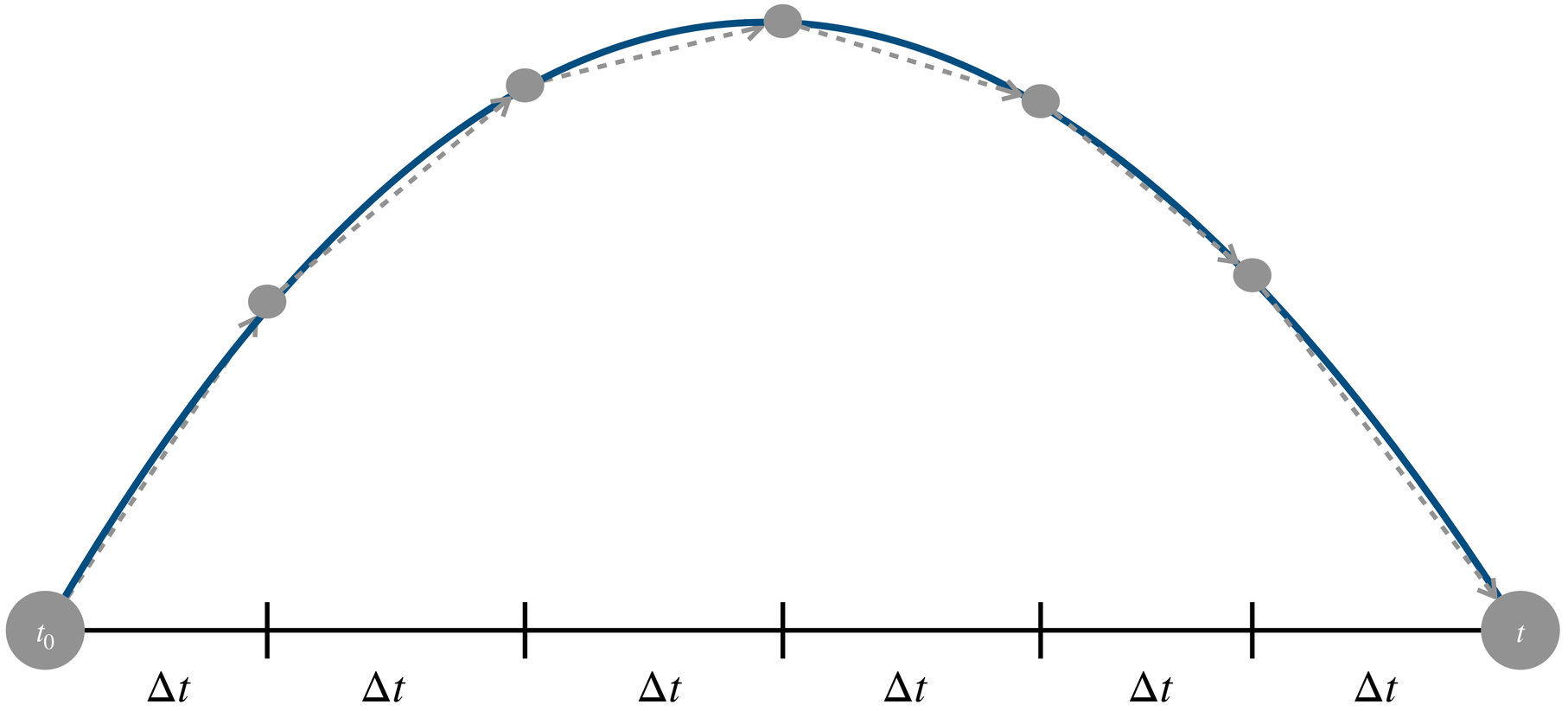
Impacto do tamanho do delta time

Quanto menor o Δt, melhor a aproximação do **movimento real**, ou seja, mais suave será o movimento.



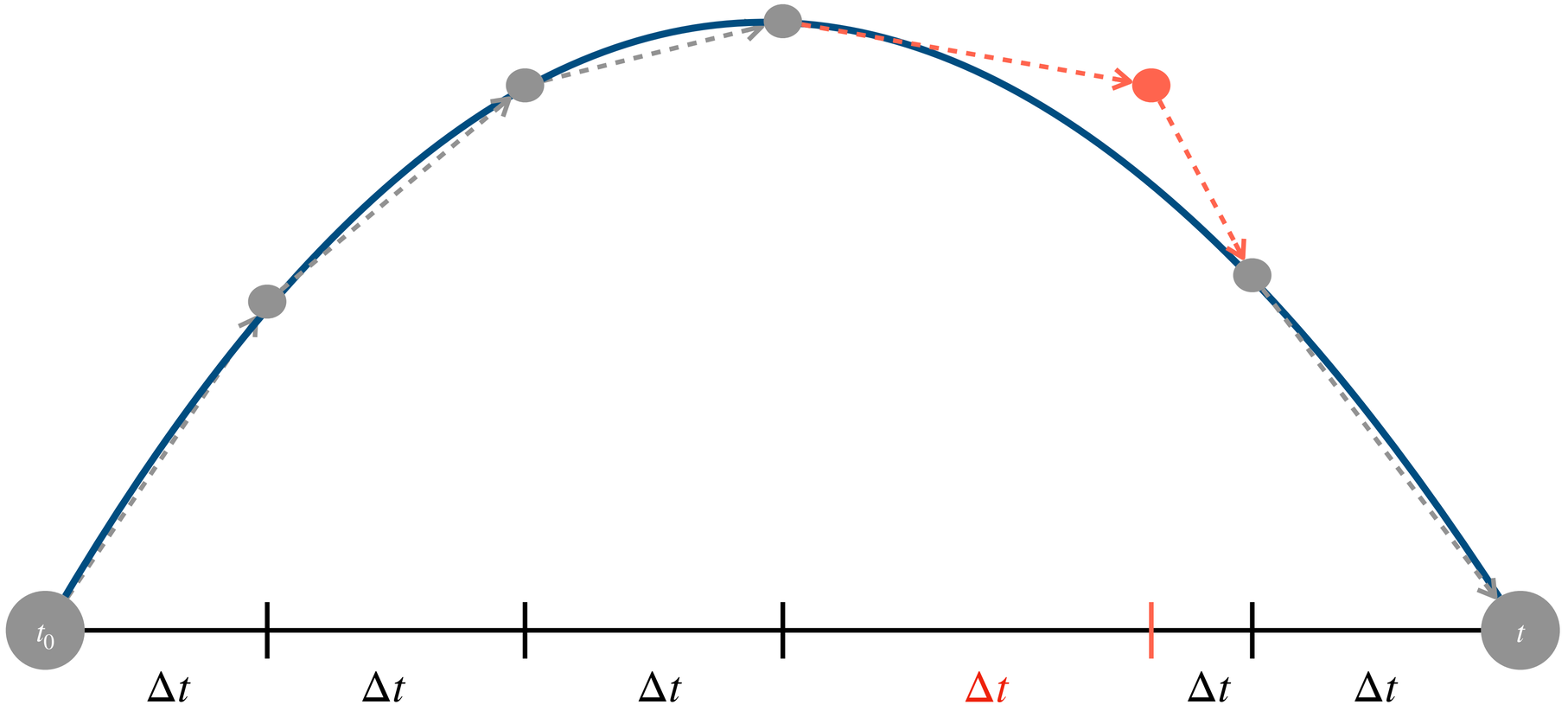
Impacto do tamanho do delta time

Quanto menor o Δt, melhor a aproximação do **movimento real**, ou seja, mais suave será o movimento.



Impacto na variação do delta time

Como mencionados em aulas anteriores, se o delta time Δt for variável entre quadros, a simulação física pode ficar instável!



Implementando Objetos Rígidos

Uma boa forma de implementar Objetos Rígidos é utilizando um componente que pode ser adicionado à lista de componentes dos objetos que terão movimento do jogo:

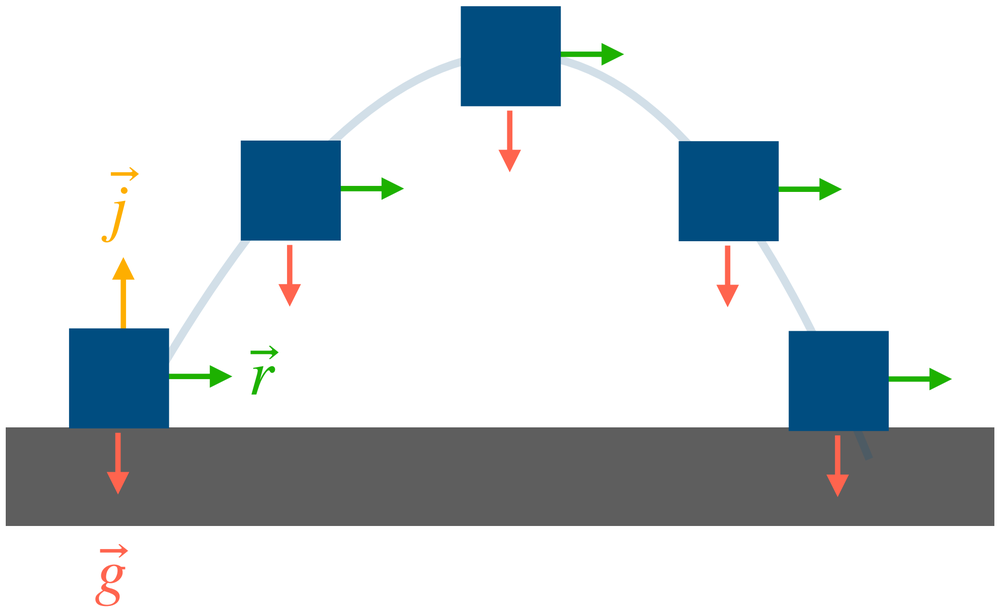
```
class RigidBody : public Component
{
public:
  RigidBody(class Actor* owner, float mass);
  void ApplyForce(const Vector2 &force);
  void Update(float deltaTime) override;

private:

  // Physical properties
  float mMass;
  Vector2 mVelocity;
  Vector2 mAcceleration;
};
```

```
void Update(const float deltaTime) {
  mVelocity += mAccelaration * dt;
  mPosition += mVelocity * dt;
  mAcceleration.Set(.0, .0);
}
```

```
void ApplyForce(const Vector2 &force) {
  mAcceleration += force * (1.f/mMass);
}
```

Aceleração da Gravidade

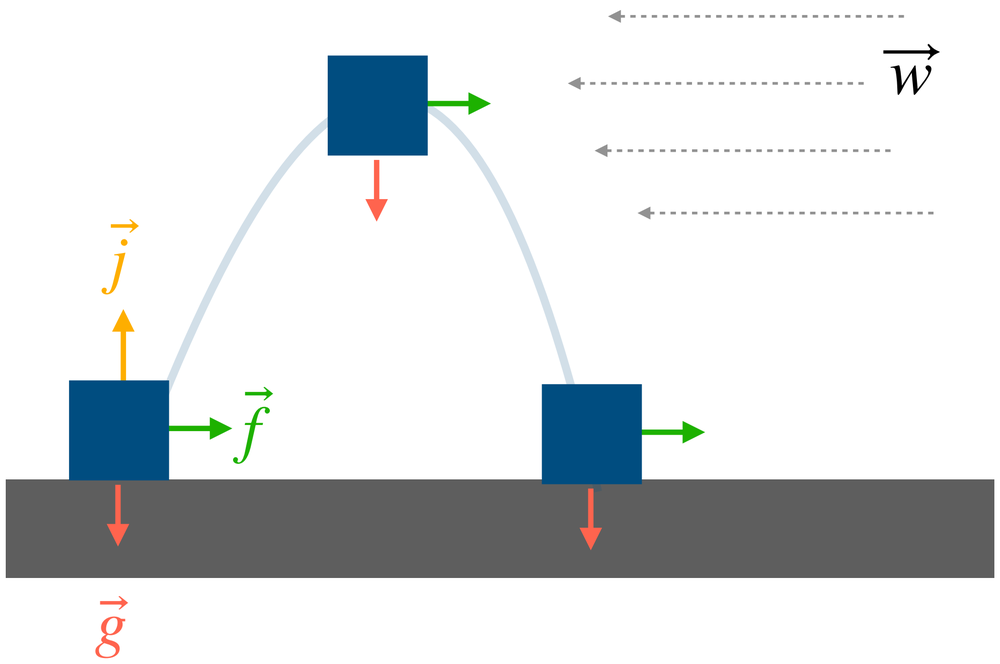
Para implementar uma **Força Peso** $\vec{g}$ nos objetos, basta aplicar uma força constante para baixo a cada atualização do componente RigibBody:




```
Vector2 g = Vector2(0.f, 9.8f);

RigidBody::Update(float dt) {
  ApplyForce(mMass * g);
  mVelocity += mAccelaration * dt;
  mPosition += position * dt;
  mAcceleration.Set(0f, 0f);
}

RigidBody::ApplyForce(Vector2 f) {
  mAccelaration += f * 1f/mMass;
}
```

Simulação de Vento

De forma similar, para simular uma força causada por **vento,** que atua nos objetos em uma determinada direção, basta aplicar uma força constante $\vec{w}$ nessa direção:

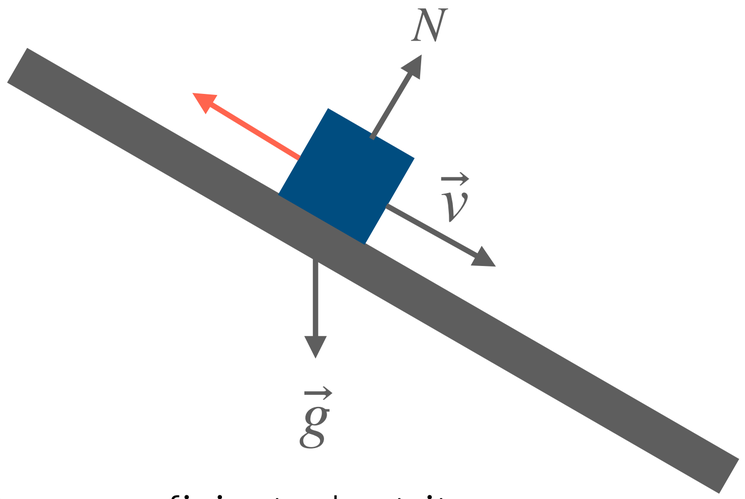



```
Vector2 g = Vector2(0.f, 9.8f);
Vector2 w = Vector2(-10.0f, 0.0f);

RigidBody::Update(float dt) {
  ApplyForce(mMass * g);
  ApplyForce(w);

  mVelocity += mAccelaration * dt;
  mPosition += position * dt;
  mAcceleration.Set(0f, 0f);
}
```

Atrito

Para aplicar uma **Força de Atrito**, podemos calcular o inverso da velocidade normalizada e multiplicar por um coeficiente de atrito μ e a magnitude do vetor normal à superfície N:

$$\vec{F}_a = -1\mu||N||\hat{v}$$

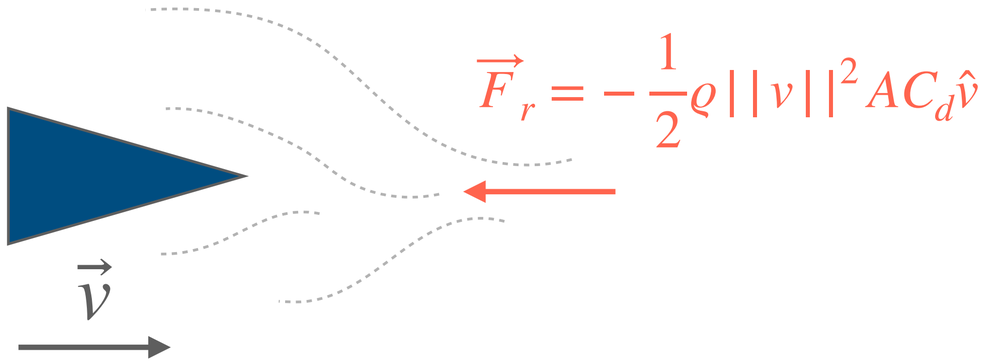


- μ: coeficiente de atrito
- N: é a força normal

```
RigidBody::ApplyFriction(float u, Vector2 N){
  if(mVelocity.Length() <= a)
    return;

  float frictionMag = u * N.Length();
  Vector2 friction = (-1) * mVelocity;
  friction.Normalize();
  friction *= frictionMag;

  ApplyForce(friction);
}
```

Resistência do Meio

A mesma ideia se aplica para parar um objeto que não está em contato com uma superfície, mas sobre **resistência do meio**, como do ar ou de um fluído.



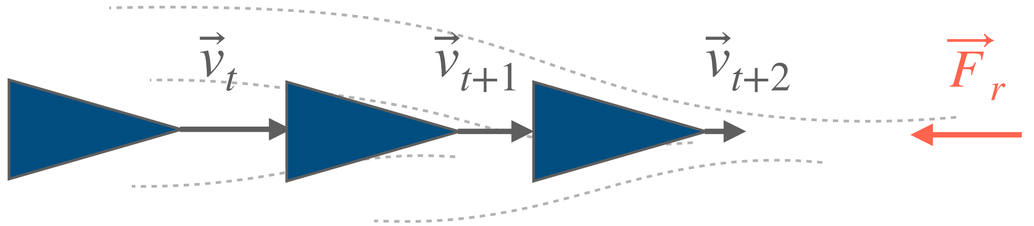
- ϱ: densidade do meio
- $||v||^2$: comprimento do vetor velocidade
- ~~A: área frontal do objeto~~
- ~~C_d: coeficiente de resistência~~
- $\hat{v}$: direção do vetor velocidade

```
RigidBody::ApplyDrag(float rho){
  if(mVelocity.Length() <= MIN_VEL)
    return;

  float vLen = mVelocity.Length();
  float dragLen = rho * vLen * vLen;

  Vector2 drag = (-1) * mVelocity;
  drag.Normalize();
  drag *= dragLen;

  ApplyForce(drag);
}
```

Parando por Completo

Note que a aceleração é zerada a cada quadro, mas e a **velocidade não**, ou seja, o objeto só irá parar por completo quando a aceleração cancelar perfeitamente a velocidade atual:

```
RigidBody::Update(float dt) {
  ApplyDrag(0.1f);
  mVelocity += mAccelaration * dt;
  mPosition += position * dt;
  mAcceleration.Set(0f, 0f);
}
```



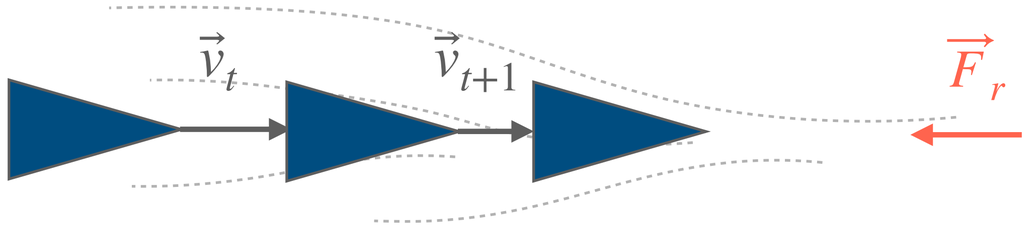
Problemas:

- $\vec{v} = (0,0)$ é extremamente improvável devido às multiplicações por `dt` e pela natureza de ponto flutuante das forças
- O objeto vai chegar a uma velocidade escalar muito baixa (ex. 0.001), mas nunca irá parar por completo

Parando por Completo

Note que a aceleração é zerada a cada quadro, mas e a **velocidade não**, ou seja, o objeto só irá parar por completo quando a aceleração cancelar perfeitamente a velocidade atual:

```
RigidBody::Update(float dt) {
   ApplyDrag(0.1f);
   mVelocity += mAccelaration * dt;

   // Verificar velocidade mínima
   if(mVelocity.Length() < MIN_VEL) {
      mVelocity.Set(.0f, .0f);
   }

   mPosition += position * dt;
   mAcceleration.Set(0f, 0f);
}
```



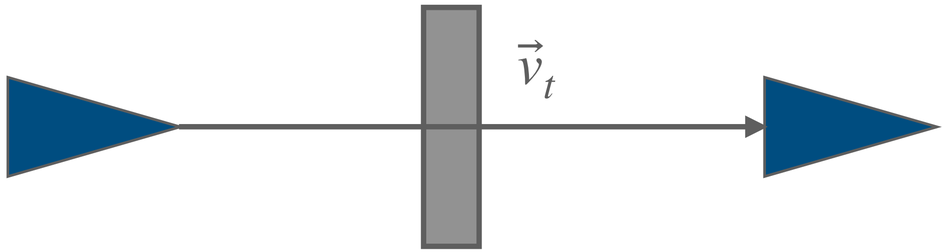
Solução:

- Definir um limiar de velocidade escalar mínima `MIN_VEL`
- Se a velocidade escalar for menor que esse limiar, force ela a ser exatamente zero

Velocidade Máxima

Também é importante definir um limiar de **velocidade máxima**, para tormar a simulação mais controlada e evitar comportamentos inesperados (ex. atravesar paredes)

```
RigidBody::Update(float dt) {
  mVelocity += mAccelaration * dt;
  // Verificar velocidade mínima
  ...
  // Verificar velocidade máxima
  if(mVelocity.Length() > MAX_VEL) {
    mVelocity.Normalize();
    mVelocity *= MAX_VEL;
  }

  mPosition += position * dt;
  mAcceleration.Set(0f, 0f);
}
```



Solução:

- Definir um limiar de velocidade escalar máxima MAX_VEL
- Se a velocidade escalar for maior que esse limiar, force ela a ser exatamente MAX_VEL

Próxima aula

A8: Física II - Detecção de Colisão

- Geometrias de colisão
  - Falsos Positivos
- Detecção de colisão
  - Circunferência vs. Circunferência
  - AABB vs. AABB
- Resolução de Colisão
  - AABB vs. AABB