Motor de Jogos e Arquitetura

Arquitetura por trás de uma *game engine*

Slides por:
Gustavo Ferreira Ceccon (gustavo.ceccon@usp.br)

Este material é uma criação do
Time de Ensino de Desenvolvimento de Jogos Eletrônicos (TEDJE)
Filiado ao grupo de cultura e extensão Fellowship of the Game (FoG), vinculado ao ICMC - USP

Objetivos

- Conceitos básicos e as partes de um motor de jogos
  - Core, Physics, Graphics, Audio e Networking
- Por onde começar
  - Paradigmas e linguagens de programação
  - Plataformas (editor e build)
- Arquitetura de um jogo
  - Classes e ferramentas essenciais

Objetivos

- Estrutura: jogo, cena e game object
- Modelos de programação
  - Herança e composição
- Gerenciamento de memória e processamento
  - Alocação e bom uso
- Design patterns

Objetivos

- Game loop
  - Tipos de game loops
  - Interpolação e extrapolação
  - Paralelismo
- Formas de input

Índice

1. Introdução

1. Introdução

O que é um jogo? (OH NO, NOT AGAIN!!)

1. Introdução

Soft Real-Time Interactive Agent-Based Computer Simulations

1. Introdução

- Computer Simulations
  - Simulação de um mundo virtual
  - Modelos matemáticos e físicos do mundo real
  - Equilíbrio entre realismo e fantasia

Tom Clancy's The Division (2016)

Need for Speed (2015)

1. Introdução

- ➔ Interactive Agent-Based
  - Objetos geralmente modelados como atores que têm impacto no jogo
  - Geralmente orientado à objetos, ou seja, tem características e funções (reativos em relação ao input)

L.A. Noire (2011)

1. Introdução

- Real-Time
  - 60 FPS = 0,0166.. ms; 30 FPS = 0,03.. ms (NTSC)
  - 24 FPS = 0,04166.. ms
  - Tempo limitado para processar o input e mostrar o resultado das ações (interactive)

The Witcher 3 (2015)

1. Introdução

- Soft System
  - Recuperável no caso de fps drop, por exemplo
  - Ao contrário de Hard Systems, que podem ser sistemas críticos

1. Introdução

- ➔ Além disso, usam simulação discreta
  - ◆ A física e toda matemática funciona baseada no tempo passado entre frames
  - ◆ Representação mais fácil dado o contexto (computação)
    - Uso extensivo de ponto flutuante, o que gera erros de aproximação

1. Introdução

- O que é um motor de jogos (game engine)?
  - Estrutura fundamental, base de todo jogo, podendo conter partes específicas de gêneros de jogos
  - Contém os módulos essenciais, como gráficos e áudio

1. Introdução

- ➔ O que oferece?
  - Interface com o programador e designer (editor)
  - Funções básicas como renderizar mesh, tocar sons, aplicar transformações etc., além de estruturar básicas que representam os objetos
  - Exportação para múltiplas plataformas (geralmente), além de editores (alguns casos)

1. Introdução

- Por que estudar?
  - Funcionamento do hardware e software, além do conhecimento de como funciona por trás do game design
  - Aplicação de diversas áreas da computação, aprendidas num curso de Ciências da Computação

1. Introdução

- Vantagens
  - Modularização, um código mais organizado e independente
  - Reaproveitamento, podendo usar em múltiplos jogos
  - Flexível, fácil mudança do código do jogo e adaptação
  - Atender múltiplas plataformas, útil hoje em dia já que temos um grande número de usuários jogando em diferentes consoles, sistemas operacionais etc.

1. Introdução

- Desvantagens
  - Ficar preso à engine e o que ela oferece, levando à gambiarras muitas vezes
  - Não extensível, podendo não atender todas as necessidades, tornando difícil desenvolvimento

1. Introdução

- ➔ Por onde começar?
  - Escolha plataformas, tanto do editor (se existir) e de exportação
  - Escolha de paradigma e de linguagem, além de quais bibliotecas externas e ferramentas de desenvolvimento (version control, IDE)
  - Estruturação e arquitetura da engine, além de que área cobre a sua engine
  - Bottom-up development vs. Top-down development

1. Introdução

- ➔ História breve
  - Arcades e consoles eram hardware specific
  - Ao poucos foram aproveitando código comum entre jogos similares (Quake e outros FPS)
  - O mercado de engines começou a crescer (Unreal e Source)
  - Unreal 4 e novos modelos de negócio, open-source e porcentagem de lucro

1. Introdução

- Exemplos
  - Quake Family (Doom, Quake, Medal of Honor)
  - Unreal Family (Unreal Tournament e Gears of War)
    - Atualmente uma das mais usadas pelas AAA
  - Source Engine (muitos jogos da Valve)
  - Unity (muitos jogos indies)

2. Arquitetura e Estrutura

2. Arquitetura

Exemplo completo

2. Arquitetura

- ➔ Muitas vezes separadas em módulos ou camadas
  - Existe uma certa dificuldade de separar os módulos, porque existe uma grande quantidade de intersecções
    - Camadas de dependência, nível de abstração e de proximidade com hardware
  - Genericamente falando: graphics, audio, physics, networking, além do core, que é o fundamento para o jogo em si
  - Core pode conter coisas específicas de plataforma e de gênero

2. Arquitetura

- Estruturação geral de uma engine (baixo nível)
  - Game/World/Window
  - Scene/Level
  - Entity/Actor/Game Object
- Geralmente usam um modelo hierárquico na implementação, implementando um árvore ou grafo (Scene Graph), o que facilita algumas operações como transformação

2. Arquitetura

- Game/World/Window
  - Cuida da inicialização e término, em algumas plataformas existe um trabalho exaustivo
    - Bibliotecas externas podem ser inicializadas junto
  - Engloba as cenas (uma ou mais delas) e oferece diversas funções para manipulamento delas e da janela
  - Cuida do input (keyboard, mouse, joystick)

2. Arquitetura

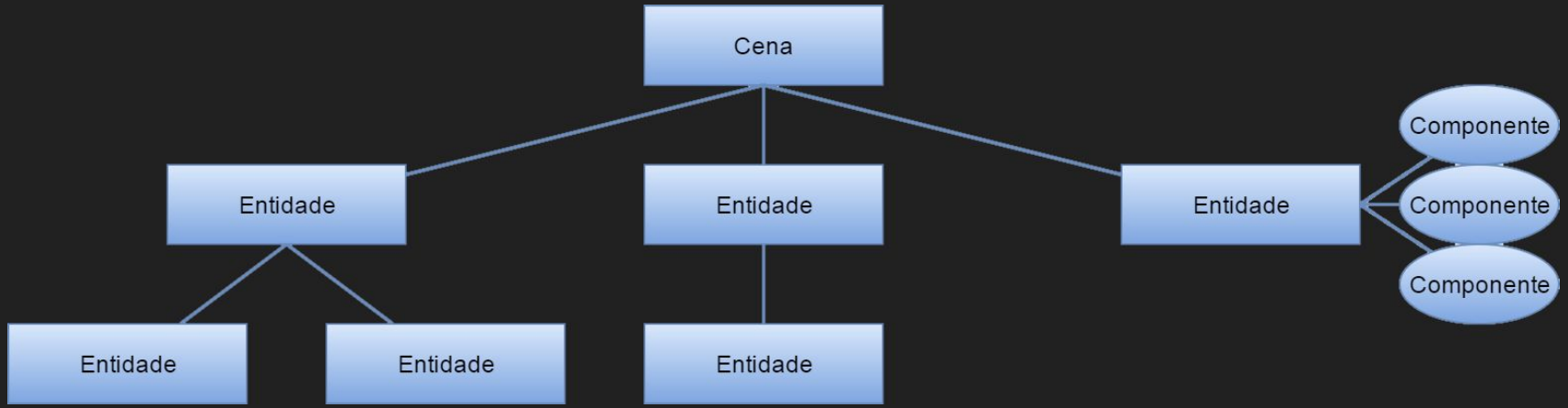
Exemplo: Win32 API

Exemplo: Haze

2. Arquitetura e Estrutura

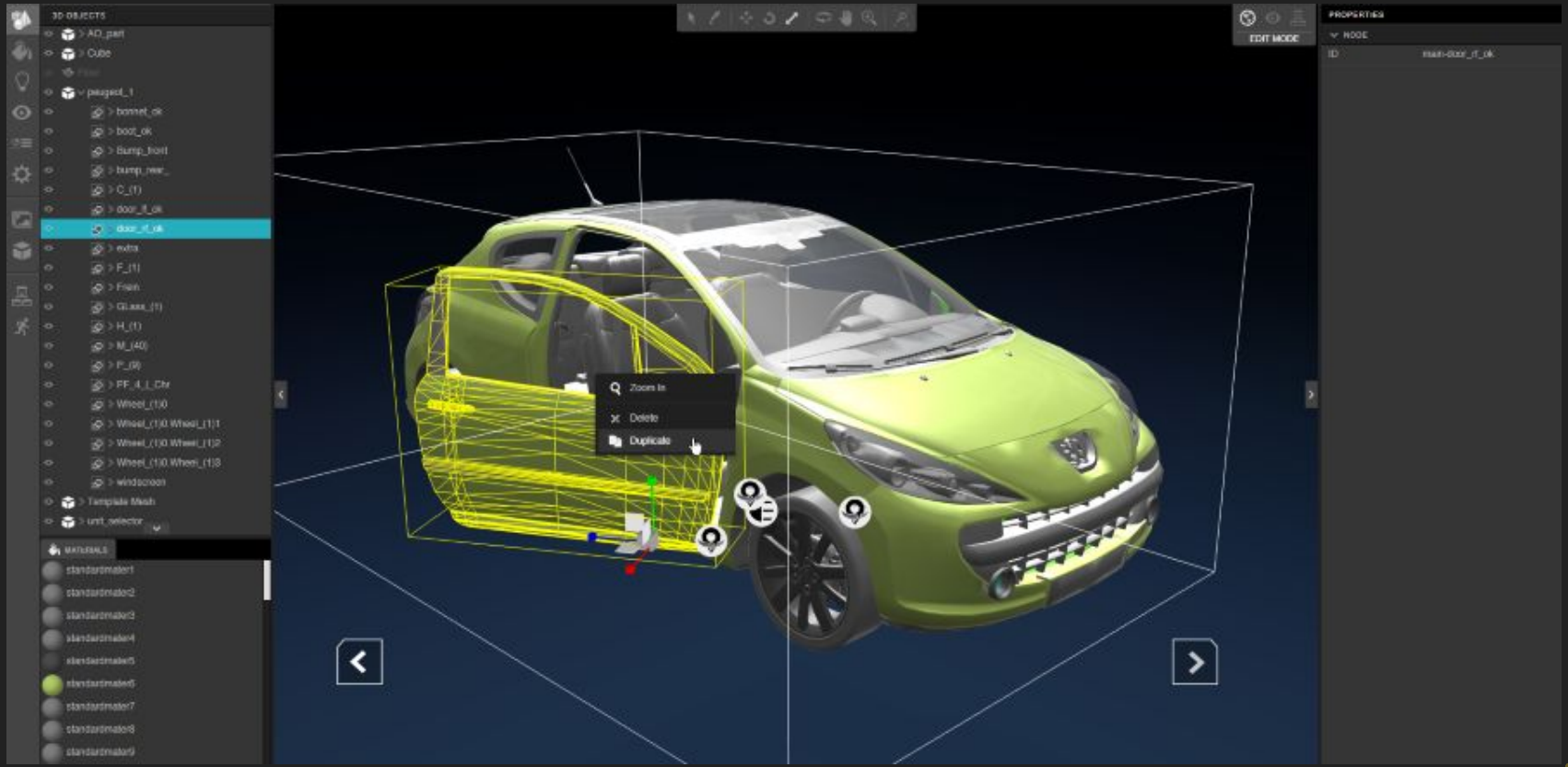
- Scene/Level
  - Cuida dos objetos, execução dos scripts e do level em si
  - Representa o level e está atrelado ao game design
    - Pode ser parte dele, quando o universo é muito grande (GTA V) ou ele inteiro (fase do Mario)
  - Implementa alguma estrutura para guardar os objetos, podendo ser uma árvore, por exemplo

2. Arquitetura e Estrutura



2. Arquitetura e Estrutura

- Hierarquia em árvore
  - Útil para aplicar transformações relativas e globais
  - Usado principalmente na construção do level, pois facilita o posicionamento e interação
  - Jeito intuitivo de mexer com objetos



CL3VER Software

2. Arquitetura e Estrutura

- ➔ Entity/Actor/Game Object
  - ◆ Representam os objetos (desde props até jogador)
  - ◆ Estão atrelados à execução do jogo e ao game loop, contém comportamentos, se houver algum
  - ◆ Existem duas formas principais para criar um game object: herança e composição

2. Arquitetura e Estrutura

➔ Como codificar esses objetos?

- Eles têm atributos, mecânicas e comportamento
- Tem coisas em comum e comunicam entre si
- Dependendo do paradigma escolhido, a implementação fica diferente

2. Arquitetura e Estrutura

- Programação imperativa
  - Simples e direto, sem muito problema na implementação
  - Eficiente, porque é amigável com arquitetura de Von Neumann
  - Data-driven, como jogos geralmente são explorados
  - Pode levar à um código grande e pouco legível
  - Uso de ponteiro de funções pode levar a bugs

2. Arquitetura e Estrutura

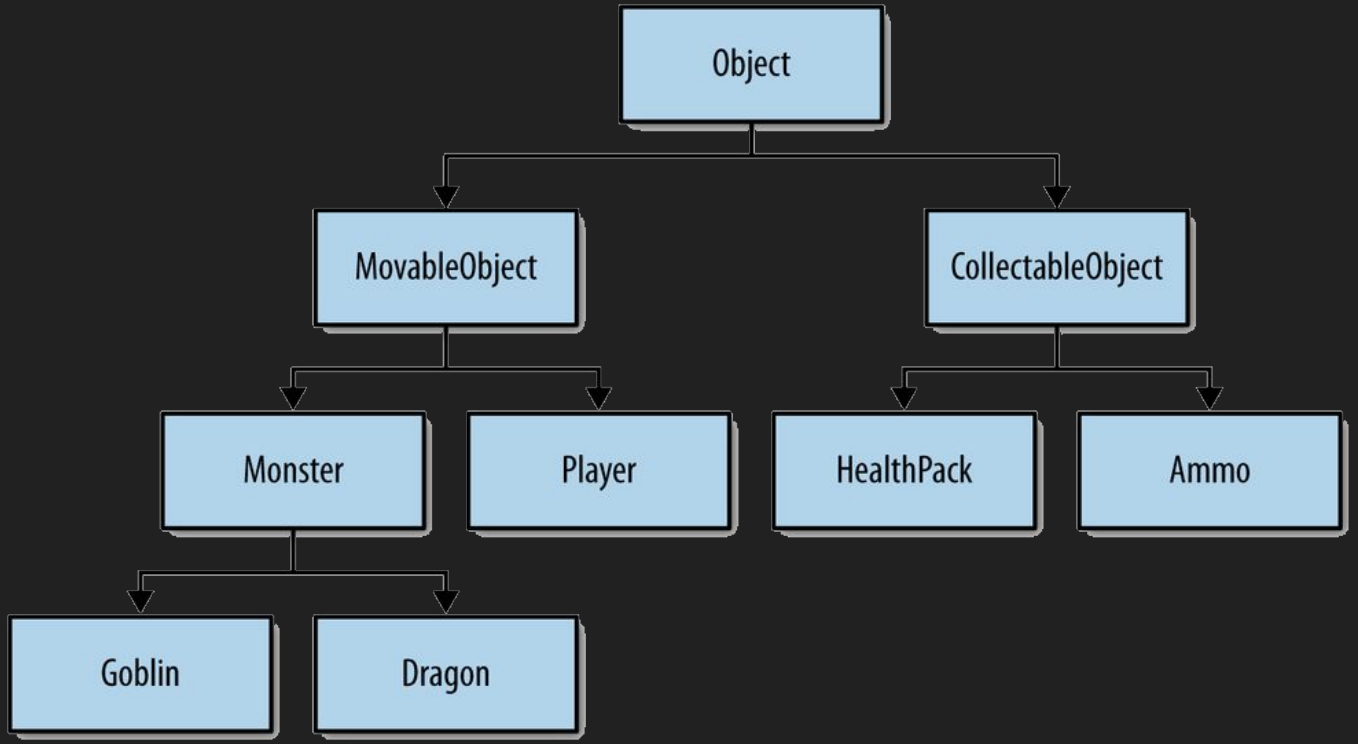
- Programação orientada a objeto
  - Classes cobrem tanto dados quanto comportamento
  - Pode se fazer uso de herança, polimorfismo etc.
  - Bom reaproveitamento de código e extensível
  - Pode ser overkill, coisas que você não precisa realmente
  - Número de classes pode subir exponencialmente, muita generalização pode aumentar a carga de trabalho

2. Arquitetura e Estrutura

- Herança
  - Habilidade de herdar membros e métodos de uma classe pai
  - Útil quando a entidade é generalizada, podendo estender ela para criar os objetos do jogo que tem mesmos atributos e comportamentos
  - Pode ficar complicado separar as diferenças e juntar as semelhanças nos nós da árvore de herança

2. Arquitetura e Estrutura

- Herança
  - Quando há a necessidade de juntar as semelhanças, as funções sobem na árvore, entupindo de funcionalidade as classes pai
    - Classes filho começam a implementar coisas inúteis
  - Quando há necessidade de separar as diferenças, as funções descem na árvore, tornando o código complexo e difícil de entender
    - Classes pai começam a ficar irrelevantes



Árvore de herança

2. Arquitetura e Estrutura

- Composição
  - Adicionar pequenos comportamentos e atributos comuns em cada objeto invés de herdá-los
  - Cada script representa um componente e cada objeto contém um vetor de componentes
  - É possível representar todos scripts como uma matriz também
  - Pode ser overkill para jogos pequenos o suficiente

2. Arquitetura e Estrutura

- ➔ Composição
  - ◆ A dependência entre componentes e objetos pode complicar a execução dos scripts
    - ● Se um script depende de outro script, isso pode quebrar o paralelismo, uma das vantagens de usar composição
    - ● A comunicação entre objetos e scripts fica pesada e juntar scripts pode quebrar a ideia de composição
  - ◆ Nem sempre é trivial separar as funcionalidades

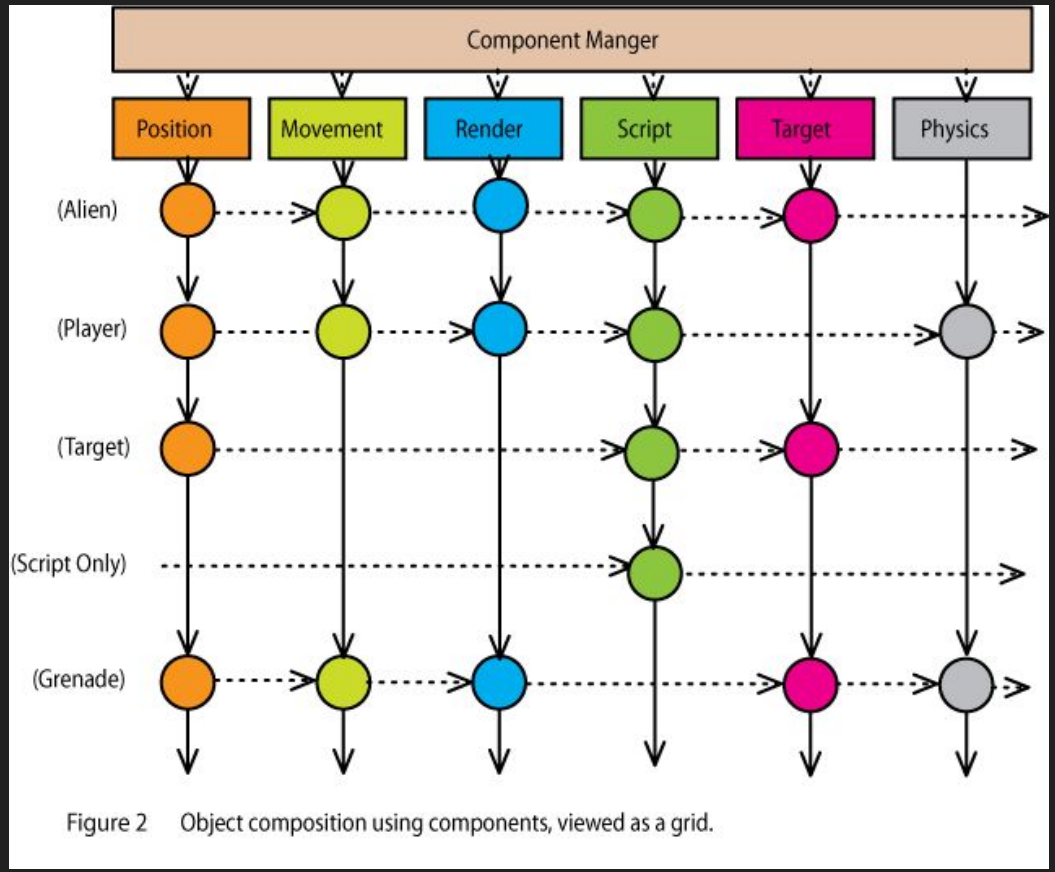


Figure 2 Object composition using components, viewed as a grid.

Matriz esparsa de componentes

2. Arquitetura e Estrutura

- Melhor de dois mundos (híbrido)
  - Usar pouca herança (árvore pequena) e o suficiente de composição (para as funcionalidades) para facilitar o desenvolvimento

3. Gerenciamento de Memória

3. Gerenciamento de Memória

- Como jogos são sistema de tempo real, é necessário aproveitar cada milésimo de segundo possível
  - Isso só é possível paralelizando as tarefas e otimizando o uso de memória e processamento
- Gerenciamento de memória tem um papel importante, já que o custo de alocação e desalocação é alto
- Também implica no aproveitamento da cache

3. Gerenciamento de Memória

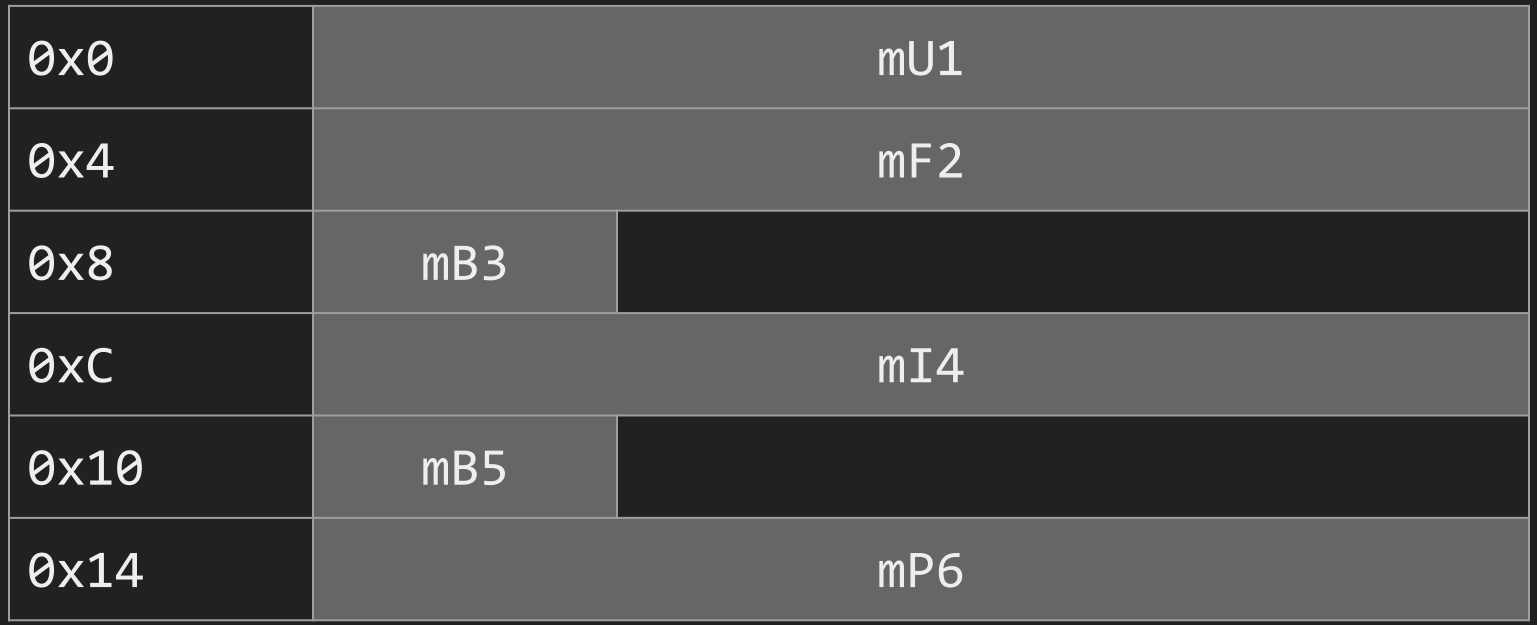
- Estruturas de dados em C e classes em C++ trabalham de formas diferentes
  - Classes têm um ponteiro (vpointer) para uma tabela (vtable) das funções virtuais
    - Funções virtuais, são funções “abstratas” que podem ser sobrescritas
  - Isso modifica o tamanho esperado da instância

3. Gerenciamento de Memória

- Alinhamento é como os dados de uma estrutura estão organizados na memória
  - Em geral eles estão na ordem da declaração, o problema é aproveitamento de espaço
  - Quando temos variáveis pequenas que não podem ser agrupadas com variáveis vizinhas, criamos um buraco
  - Alguns compiladores têm otimização

3. Gerenciamento de Memória

```
struct InefficientPacking {
    U32 mU1; // 32 bits
    F32 mF2; // 32 bits
    U8 mB3; // 8 bits
    I32 mI4; // 32 bits
    bool mB5; // 8 bits
    char* mP6; // 32 bits
};
```



Alinhamento do InefficientPacking

3. Gerenciamento de Memória

```
struct MoreEfficientPacking {
    U32 mU1; // 32 bits (4-byte aligned)
    F32 mF2; // 32 bits (4-byte aligned)
    I32 mI4; // 32 bits (4-byte aligned)
    char* mP6; // 32 bits (4-byte aligned)
    U8 mB3; // 8 bits (1-byte aligned)
    bool mB5; // 8 bits (1-byte aligned)
};
```

0x0	mU1		
0x4	mF2		
0x8	mI4		
0xC	mP6		
0x10	mB3	mB5	

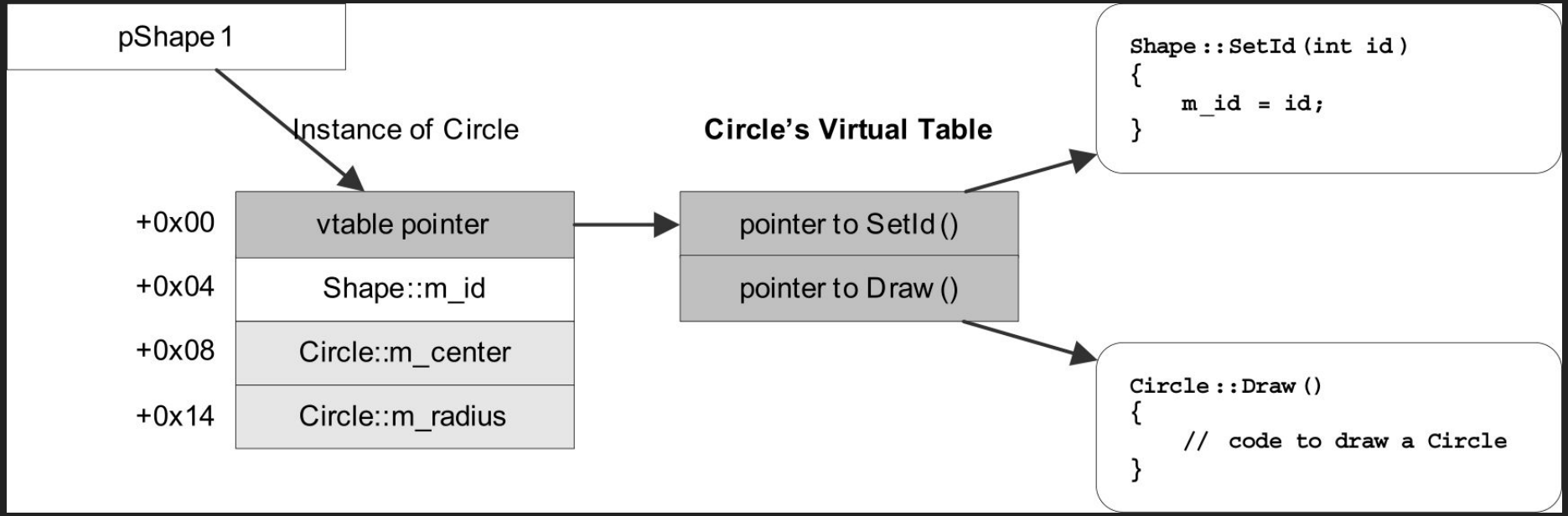
Alinhamento do MoreEfficientPacking

3. Gerenciamento de Memória

```
class Shape {
public:
    virtual void SetId(int id) { m_id = id; }
    int GetId() const { return m_id; }
    virtual void Draw() = 0; // pure virtual – no impl.
private:
    int m_id;
};
```

3. Gerenciamento de Memória

```
class Circle : public Shape {
public:
    void SetCenter(const Vector3& c) { m_center=c; }
    Vector3 GetCenter() const { return m_center; }
    void SetRadius(float r) { m_radius = r; }
    float GetRadius() const { return m_radius; }
virtual void Draw() { // Code to draw a circle }
private:
    Vector3 m_center; float m_radius;
};
```



Organização de Circle

3. Gerenciamento de Memória

- ➔ Alocação de memória é custoso, o sistema operacional faz o que é possível para tornar menos custoso e aproveitar melhor a memória
- ➔ Nem sempre ele faz um trabalho bom (obrigado Windows), então o programador fica encarregado de implementar sua própria classe de gerenciamento de memória

3. Gerenciamento de Memória

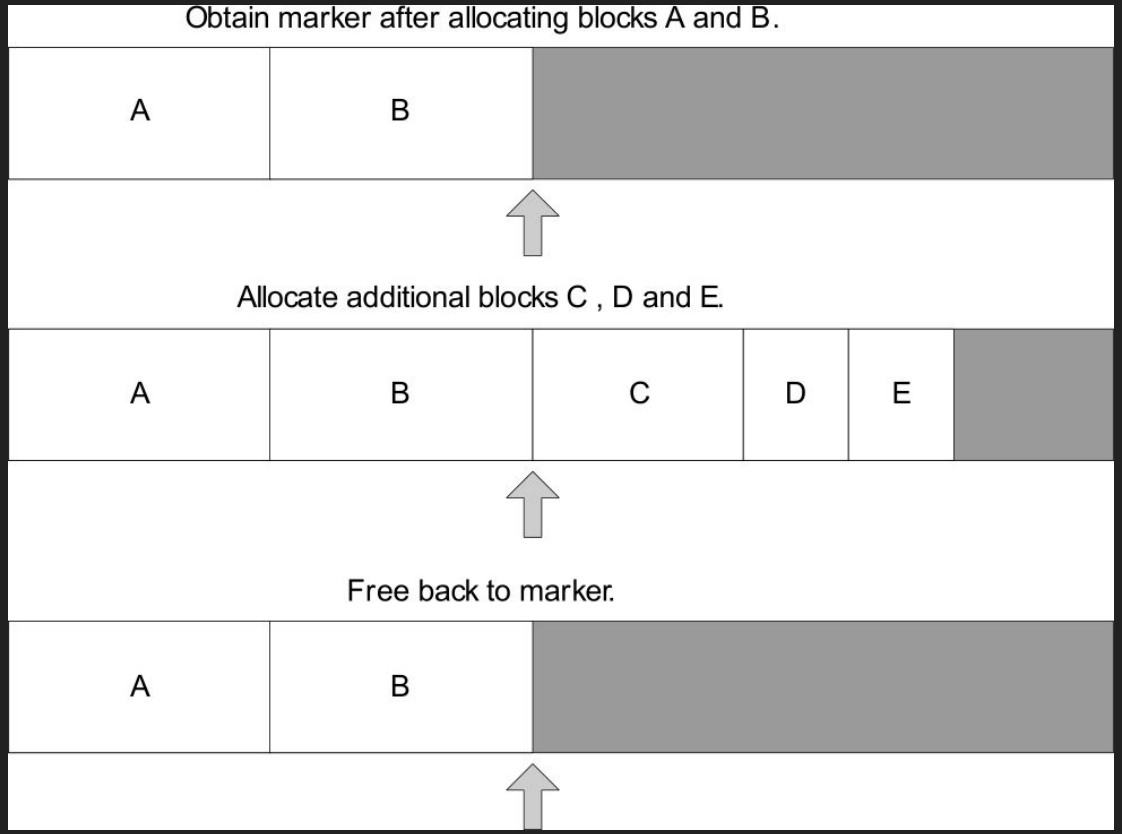
- Um das classes é a Stack Allocator
  - Colocamos cada espaço alocado de memória um seguido do outro
  - Quando necessário liberar memória, fazemos tudo de uma vez
  - Não podemos liberar a memória a qualquer momento, precisa ser na mesma ordem
    - Isso significa que as coisas devem ser feitas no início e no fim de cada bloco de código

3. Gerenciamento de Memória

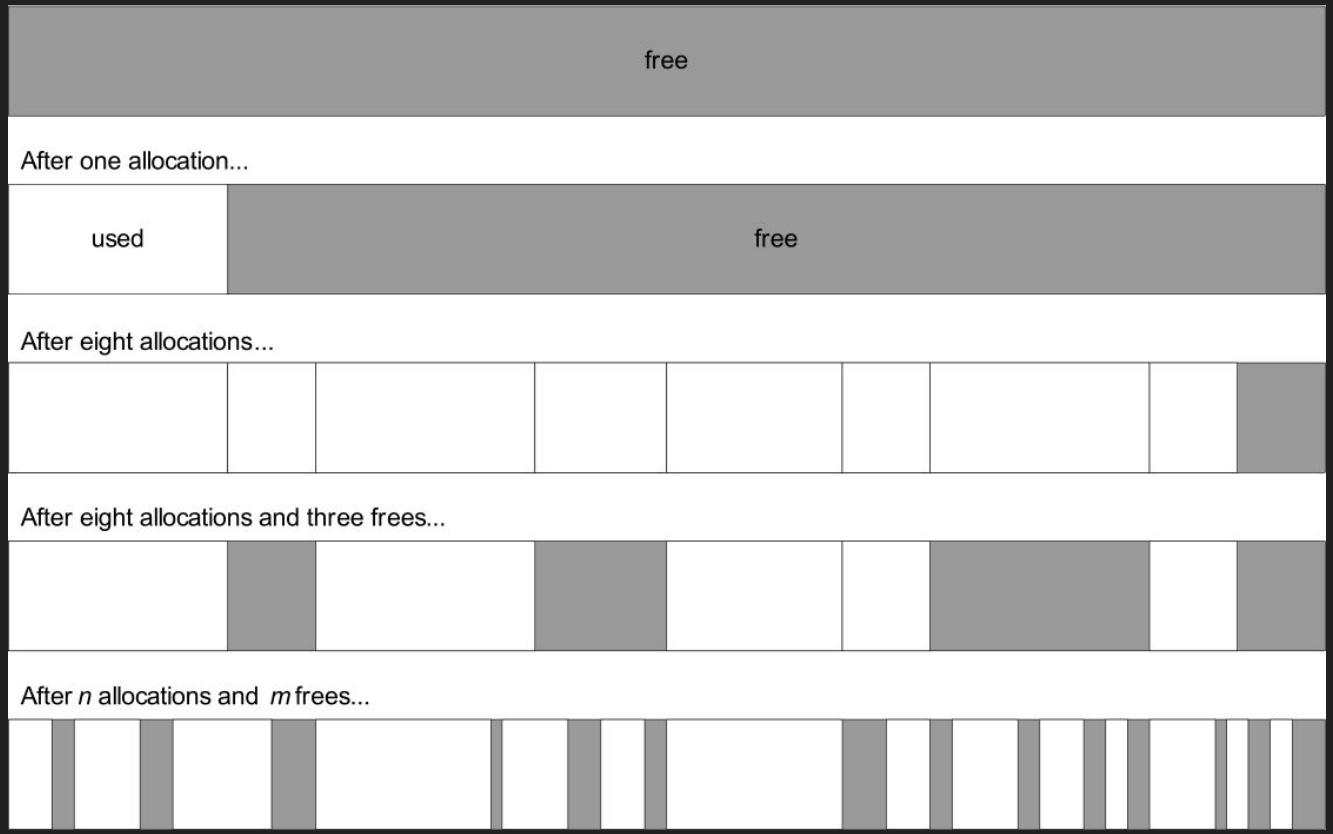
- Uma das classes mais comuns é a Pool Allocator
  - Deixamos um pedaço grande da memória alocada e marcamos blocos dela como livres ou não
  - Quando requisitada memória, escolhemos um bloco na lista de blocos livres e devolvemos
  - Tem problemas com tamanhos diferentes e não múltiplo
    - Múltiplos Pool Allocators? Um para cada tamanho
    - Blocos de tamanho variado? Problema na realocação

3. Gerenciamento de Memória

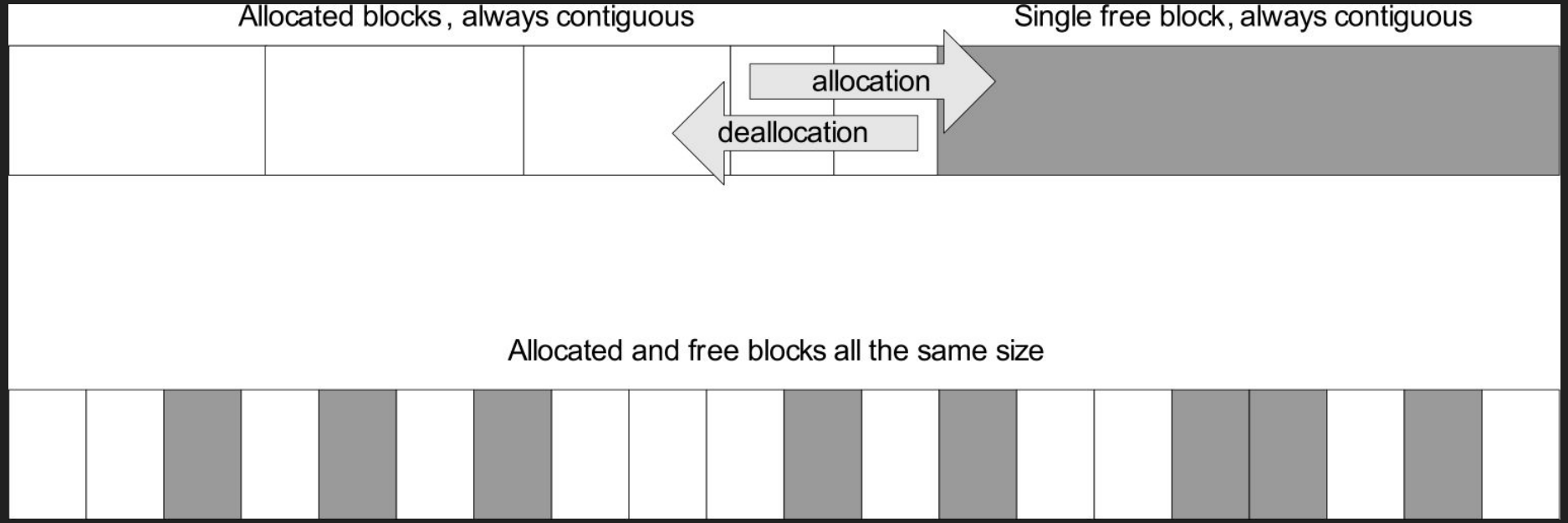
- ➔ Fragmentação acontece frequentemente, quando acaba sobrando um espaço de memória que não conseguimos alocar para nada
- ➔ Solução é desfragmentar a memória, o que é muito mais custoso, então fazemos com sabedoria
- ➔ O Garbage Collector do Java e do C# funciona parecido, só que você tem pouco controle sobre eles



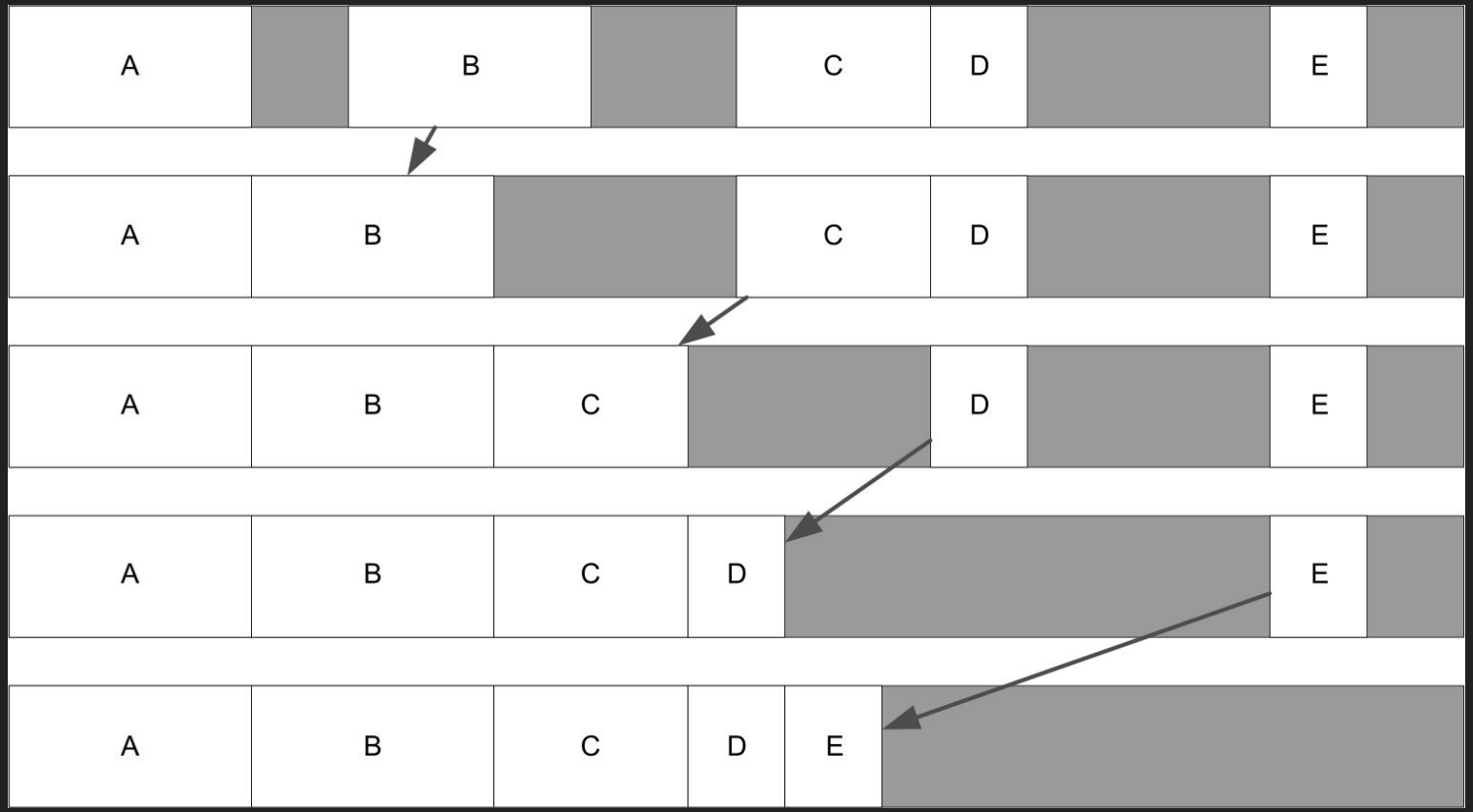
Stack Allocator



Fragmentação



Fragmentação nos casos de Stack e Pool Allocators



Desfragmentação

3. Gerenciamento de Memória

- Tudo isso é para otimizar o caching do processador
- Processador gosta de acessar blocos contínuos da memória, e quando fazemos isso, evitamos cache-miss
  - Um grande boost de velocidade
- Uso de vetores pode auxiliar muito

3. Gerenciamento de Memória

- Object-Centric Architecture (AOS - Array Of Structures)
  - Instâncias de objetos contém as propriedades e temos uma coleção de objetos
  - Mais legível e amigável com programador
- Property-Centric Architecture (SOA - Structure Of Arrays)
  - Temos uma coleção de propriedades e cada objeto tem um id para as propriedades
  - Menos legível e amigável com a máquina

Objeto 1		Objeto 2		
HP	Mana	HP	Mana	Mago
100	10	50	200	Object

HP		Mana		Mago
Objeto 1	Objeto 2	Objeto 1	Objeto 2	Objeto 2
100	50	10	200	Object

3. Gerenciamento de Memória

- O disco rígido também sofre seus problemas, por exemplo, fragmentação
  - Um jogo contém muitos arquivos de arte, música, som, texto etc., isso ocupa muito espaço
  - Muitos arquivos também podem acabar fragmentando o disco
  - Além da compressão de cada asset, compactamos todos eles em zip, por exemplo

3. Gerenciamento de Memória

- ➔ Assim diminuímos o desperdício de memória, porém aumentamos o custo de processamento, para descompactar esses assets
- ➔ Por isso organizamos em diferentes packs, podendo seguir a linha de game design e similaridade
  - ◆ Um level tem um zip próprio com os assets daquele level, por exemplo

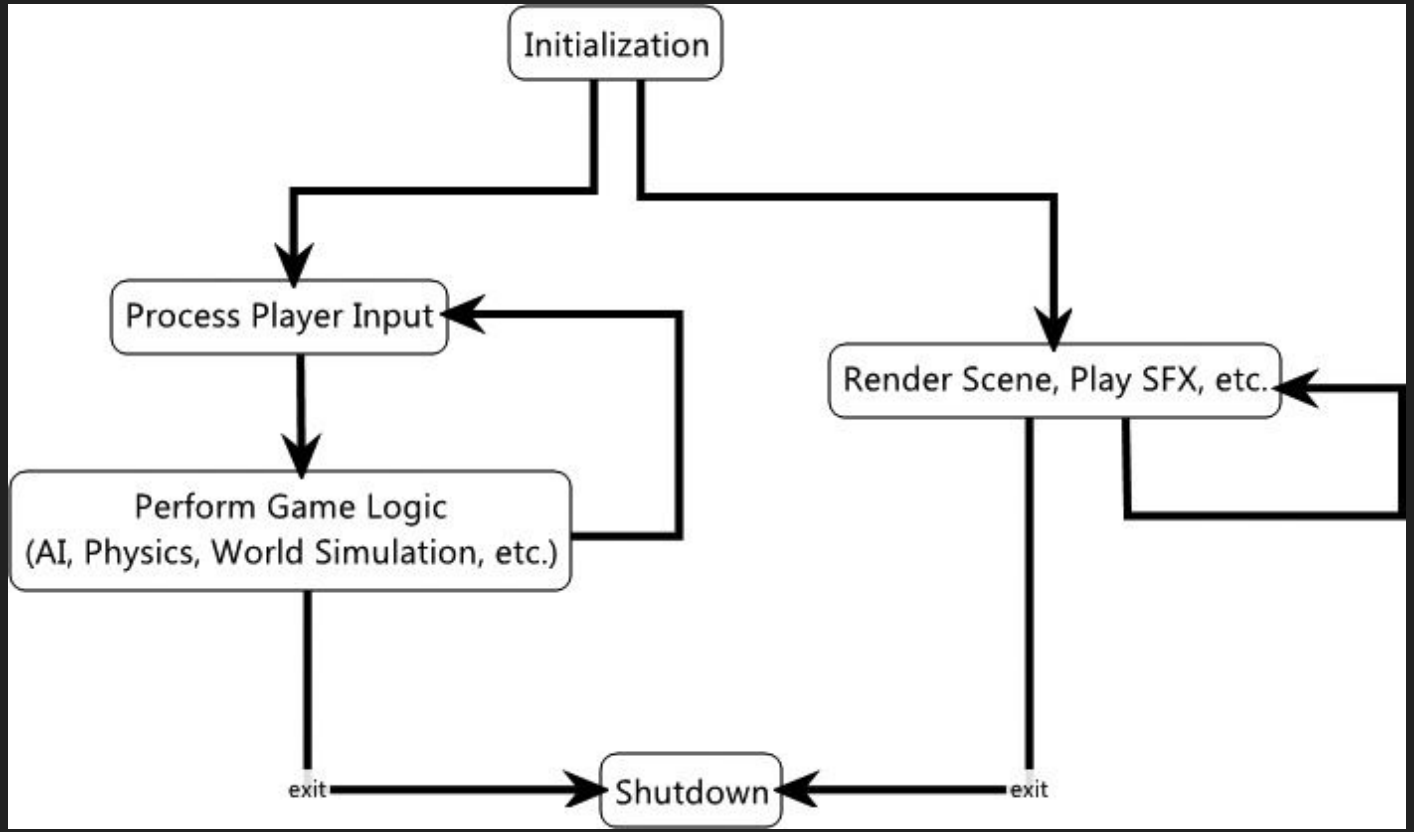
4. Gerenciamento de Processo

4. Gerenciamento de Processo

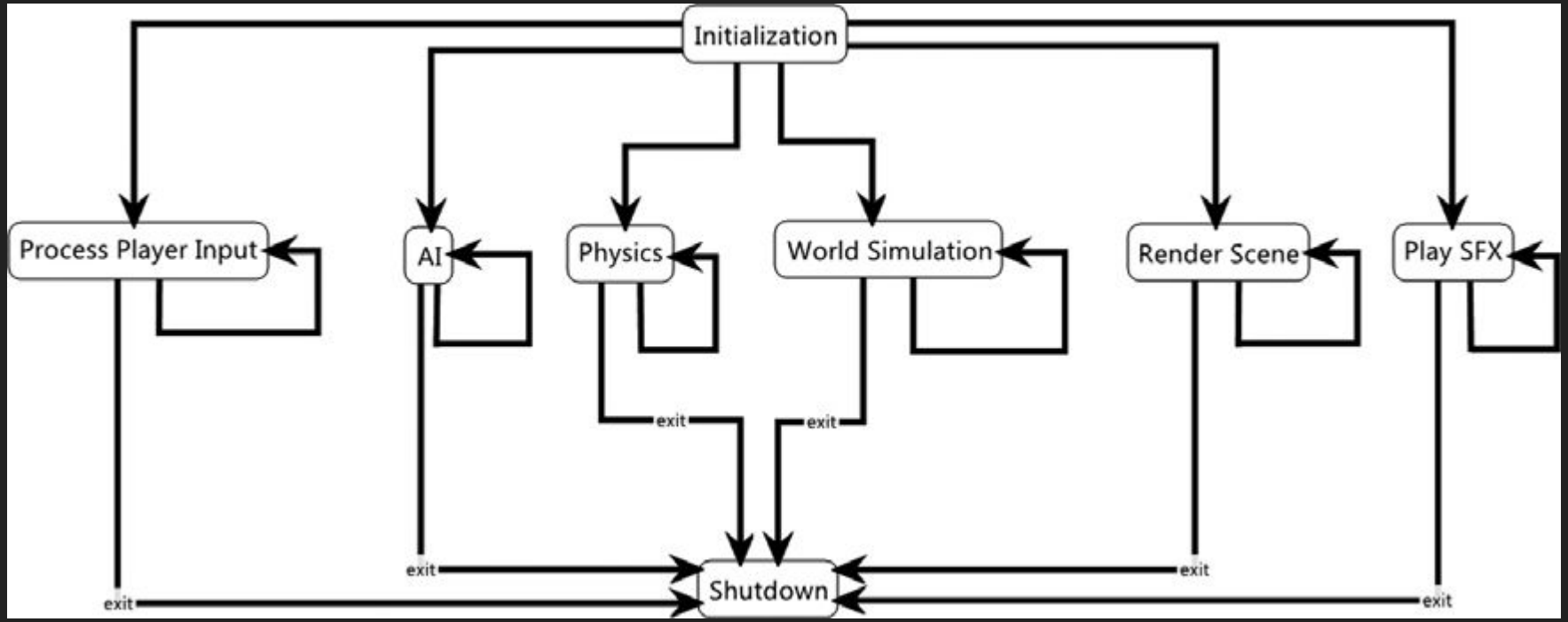
- Além de melhorar o aproveitamento de memória, precisamos melhorar o aproveitamento de processamento
- Temos muitas tarefas para cumprir, num pequeno espaço de tempo, como vamos dividir essas tarefas?
  - Paralelismo
- Mais fácil falar do que fazer, existem diversos problemas de concorrência

4. Gerenciamento de Processo

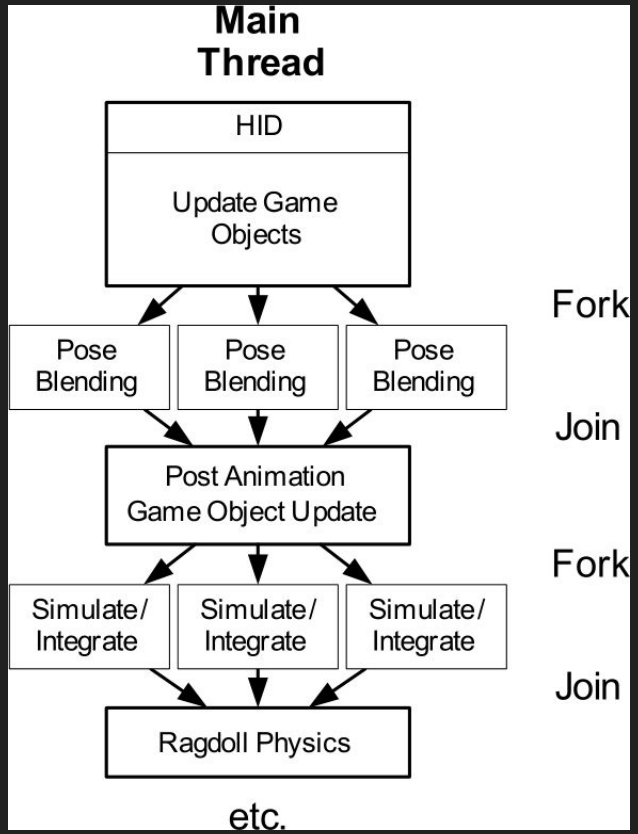
- Opções:
  - Paralelizar cada pedaço da engine (áudio, gráficos, física, IA, entidades etc.)
  - Paralelizar cada tarefa dentro de cada pedaço
  - Paralelizar tudo
- Problemas:
  - Dependência de dados
  - Memória compartilhada



Multithread simples



Multithread cooperativo


Main Thread
HID
Update Game Objects
Fork
Pose Blending
Pose Blending
Pose Blending
Join
Post Animation Game Object Update
Fork
Simulate/ Integrate
Simulate/ Integrate
Simulate/ Integrate
Join
Ragdoll Physics
etc.

Fork e Join

4. Gerenciamento de Processo

- Job System
  - Sistema de threads onde é fácil criar e rodar processos simples (tarefas) em threads separadas
  - Ele cuida de criar, manter (Thread Pool) e priorizar as threads

5. Design Patterns

5. Design Patterns

- Problemas comuns de programação resolvidos, como restrição de acesso, de criação, comportamento etc.
- Não se deve forçar um padrão sobre um problema, deve entender as aplicações e quando usar
- Geralmente voltados a problemas de programação orientada a objeto, devido a herança, escopo, polimorfismo
- Tipos: criação, estrutural, comportamental, concorrente

5. Design Patterns

- Padrões de projetos
  - Singleton (criação)
  - Observer (comportamental)
  - Flyweight (estrutural)
  - Prototype (criação)

5. Design Patterns

- Singletone
  - Exemplo: um game manager que contém informações sensíveis e que devem ser únicas. Ele controla coisas dentro de um jogo e múltiplas instâncias podem causar problemas

5. Design Patterns

- ➔ Solução
  - ◆ Deixar o construtor como privado
  - ◆ Criar uma função global/estática (acessível) que retorna a instância única
    - A instância pode ser criada sempre no início
    - A instância pode ser criada na primeira chamada
      - ○ Lazy initialization (outro padrão!)

5. Design Patterns

- Observer
  - Exemplo: uma HUD precisa saber se o jogador perdeu o não para mostrar a tela de game over, porém não é uma boa ideia chamar uma função a partir do player

5. Design Patterns

- ➔ Solução
  - Criar um evento de callback na HUD
  - Adicionar um listener do script da HUD ao observer do game manager
  - Quando o jogador perder, o game manager ativa o observer / evento, chamando a função de callback

5. Design Patterns

- Flyweight
  - Exemplo: precisamos renderizar um conjunto de objetos iguais (árvores) com o mesmo modelo e em posições diferentes

5. Design Patterns

- ➔ Solução
  - Compartilhamos o mesmo objeto de modelo e textura para a árvore e criamos apenas objetos com a referência ao mesmo
  - Criamos novas instâncias apenas para a posição da árvore

5. Design Patterns

- Prototype e Factory
  - Exemplo: precisamos criar um spawner de monstros que tem o mesmo comportamento e atributos

5. Design Patterns

- ➔ Solução 1 (Prototype)
  - ◆ Criamos um prefab, ou seja, um modelo do monstro
  - ◆ Criamos uma classe spawner que recria vários clones desse objeto
    - Shallow cloning
    - Deep cloning

5. Design Patterns

- Solução 2 (Factory)
  - Criamos uma fábrica, que sabe fazer apenas monstros
  - Cada vez que criarmos um monstro, pedimos para fábrica instanciar um novo objeto e recebemos uma entidade abstrata
  - Abstract Factory é mesma coisa, só que aceita implementação de diferentes fábricas

6. Game Loop

6. Game Loop

- Jogos eletrônicos são simulações de um mundo virtual, além disso sabemos que eles são programas de tempo real
  - Portanto, jogos estão diretamente entrelaçados com a noção de tempo
- Frames Per Second (FPS) é uma medida de quantos quadros conseguimos renderizar por segundo, mas por baixo é muito mais que isso

6. Game Loop

- Temos que mostrar pelo menos 24 frames por segundo, porém também temos que lidar com monte de outras coisas
  - Audio, Input, AI, Networking etc.
- Todo frame temos iterações de processamento dessas coisas e o laço dessas iterações se chama Game Loop
  - A ordem e quantidade de processamento dedicado depende da escolha do game loop e da arquitetura do jogo

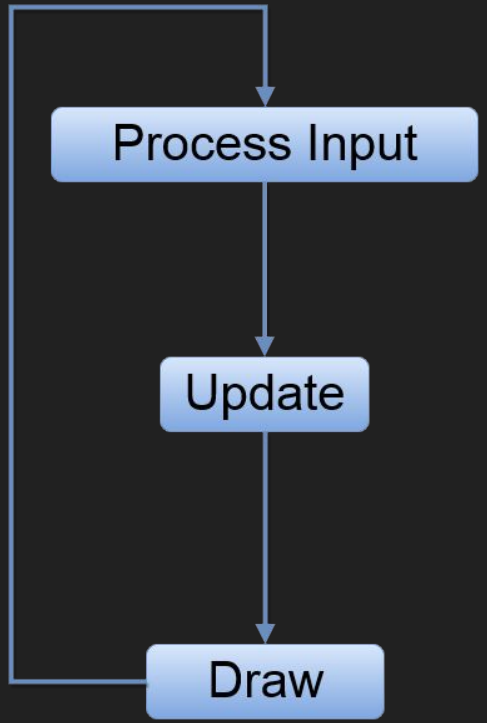
6. Game Loop

- Vamos tentar montar o game loop:
- Objetivo:
  - Renderizar frames, que atendam expectativas do jogador
- Problemas:
  - O que processar?
  - Quanto processar?
  - Em que ordem processar?

Game Loop

- Tipos
  - Simples: CPU-*dependent*
  - Simples com dt: *CPU-independent*
  - Simples com dt fixo: CPU rápida simulando CPU-*dependent*
  - *Catch-up* simples: atualiza de acordo com o tempo de *render*
  - *Catch-up* com extrapolação: atualiza de acordo com o tempo de *render* e extrapola o restante
  - Frame skipping

Game Loop - Simples



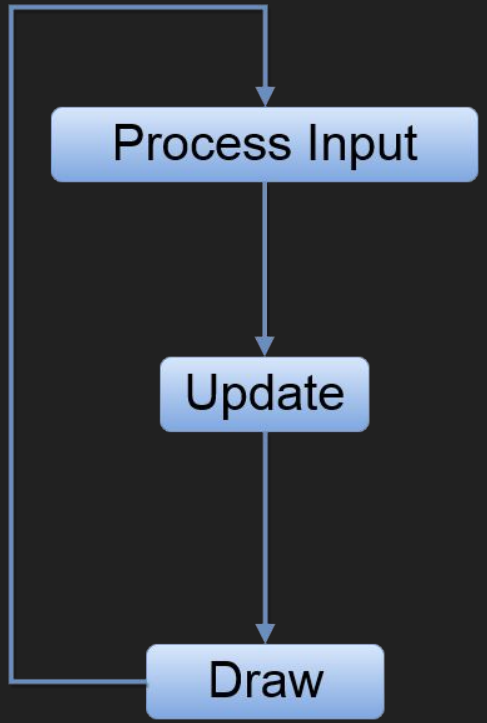
Game Loop - Simples

```
while (!done)
{
    input();  //atualiza estados
    update(); //sem param.
    draw();   //sem param.
}
```

Game Loop

- Tipos
  - Simples: CPU-*dependent*
  - Simples com dt: *CPU-independent*
  - Simples com dt fixo: CPU rápida simulando CPU-*dependent*
  - *Catch-up* simples: atualiza de acordo com o tempo de *render*
  - *Catch-up* com extrapolação: atualiza de acordo com o tempo de *render* e extrapola o restante
  - Frame skipping

Game Loop - Simples com *dt*



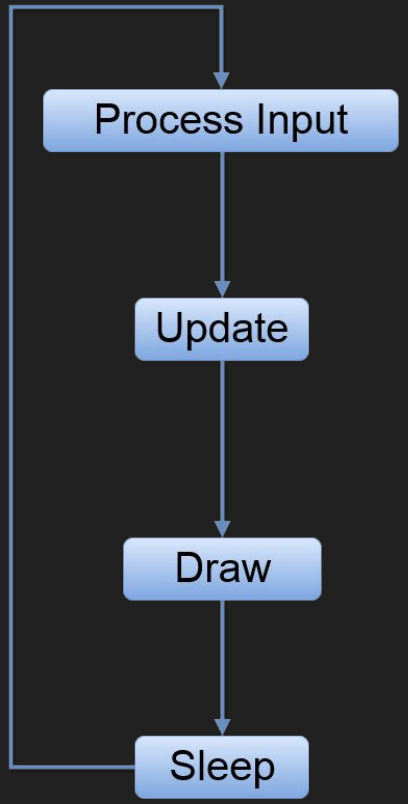
Game Loop - Simples com *dt*

```
lastTime = now();
while (!done)
{
    current = now();
    dt = current - last;
    last = current;
    input();  //atualiza estados
    update(dt); //passa param. Física baseada em dt
    //Método de integração
    draw();   //sem param.
}
```

Game Loop

- Tipos
  - Simples: CPU-*dependent*
  - Simples com dt: *CPU-independent*
  - Simples com dt fixo: CPU rápida simulando CPU-*dependent*
  - *Catch-up* simples: atualiza de acordo com o tempo de *render*
  - *Catch-up* com extrapolação: atualiza de acordo com o tempo de *render* e extrapola o restante
  - Frame skipping

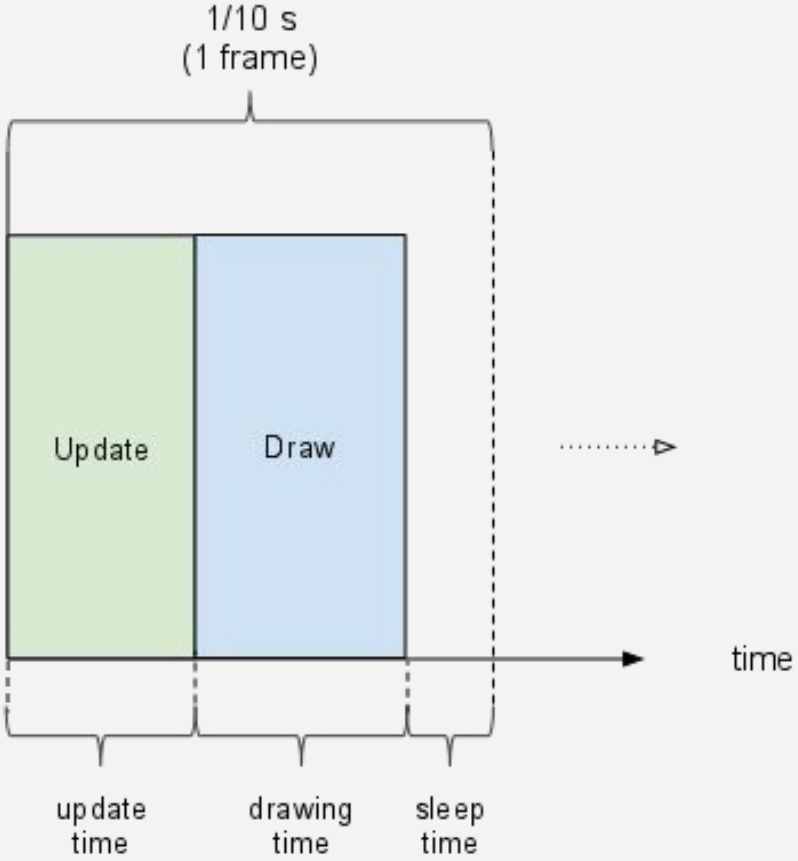
Game Loop - Simples com *dt* fixo



Game Loop - Simples com *dt* fixo

```
while (!done)
{
    start = now();
    input();
    update();
    draw();
    sleep(dt - (now() - start));
    //dt é fixo. now-start é o tempo do loop.
}
```

Game Loop


1/10 s
(1 frame)
Update
Draw
time
update time
drawing time
sleep time

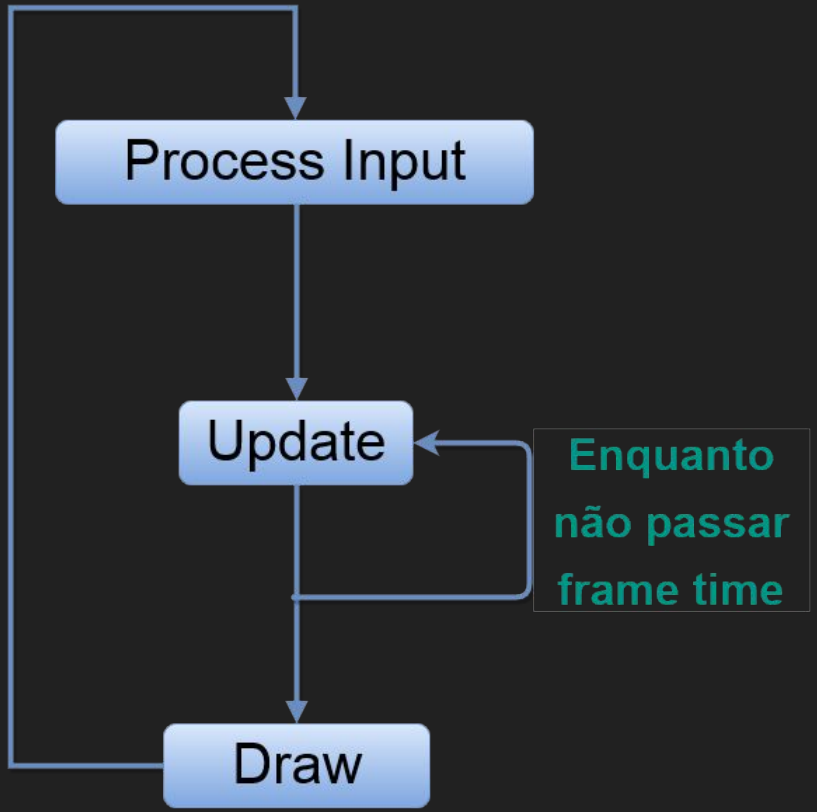
Game Loop

- Tipos
  - Simples: CPU-*dependent*
  - Simples com dt: *CPU-independent*
  - Simples com dt fixo: CPU rápida simulando CPU-*dependent*
  - *Catch-up* simples: atualiza de acordo com o tempo de *render*
  - *Catch-up* com extrapolação: atualiza de acordo com o tempo de *render* e extrapola o restante
  - Frame skipping

Game Loop

- Em alguns casos, podemos ter CPUs mais rápidas que GPUs.
- Neste caso, o Update será mais rápido que o Draw.
- Frame time > Update time
  - UPS ≠ FPS
- Para solucionar o problema, utilizamos catch-up

Game Loop - Catch-up simples



Game Loop - Catch-up simples

```
lastTime = now()
while (!done)
{
    currentTime = now()
    frameTime = currentTime - lastTime;
    lastTime = currentTime;
    while(frameTime > 0) \\Catch-up
    {
        delta = min(frameTime, dt);\\ Menor entre fixo e o restante
        update(delta);
        frameTime -= delta;
    }
    draw();
}
```

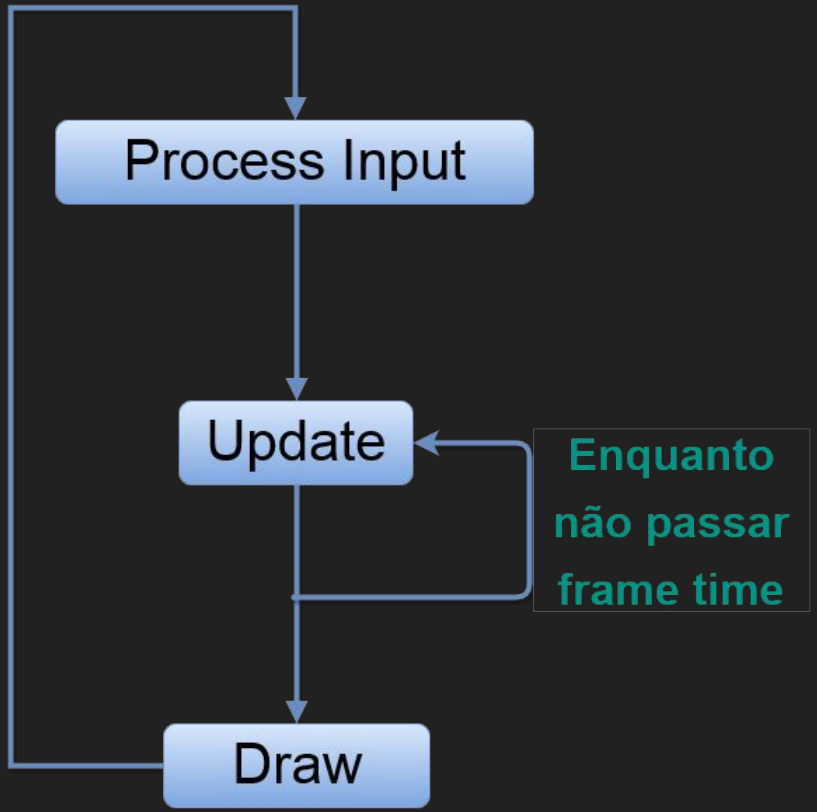
Game Loop

- Tipos
  - Simples: CPU-*dependent*
  - Simples com dt: *CPU-independent*
  - Simples com dt fixo: CPU rápida simulando CPU-*dependent*
  - *Catch-up* simples: atualiza de acordo com o tempo de *render*
  - *Catch-up* com extrapolação: atualiza de acordo com o tempo de *render* e extrapola o restante
  - Frame skipping

Game Loop

- *Catch-up* com extrapolação: atualiza de acordo com o tempo de *render* e extrapola o restante
  - Se um *draw* precisa ocorrer antes de um *update* terminar
    - O resultado entre os *updates* é extrapolado
- Interpolação: um ponto entre dois pontos conhecidos
  - $P' = (1 - \alpha)*P_0 + \alpha*P \quad =0 <= \alpha <= 1$
- Extrapolação: interpolação entre um ponto conhecido e uma previsão

Game Loop - Catch-up com extrapolação



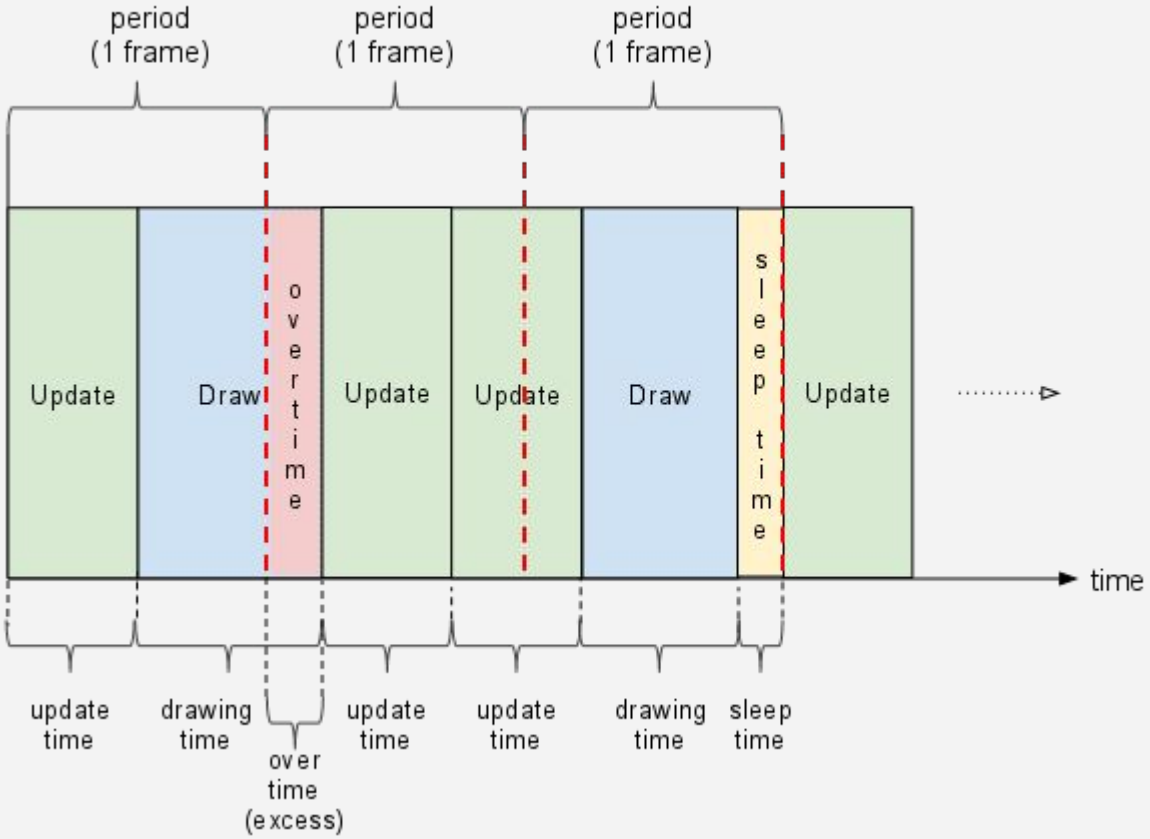
Game Loop - Catch-up com extrapolação

```
lastTime = now()
accumulator = 0;
while (!done)
{
      currentTime = now()
      frameTime = currentTime - lastTime;
      lastTime = currentTime;
      accumulator += frameTime;
      while(accumulator >= dt) \\Catch-up
      {
            update(dt);\\Fixo
            accumulator -= dt;
      }
      alpha = accumulator/dt;
      draw(alpha);
      //state = (1-alpha)*previous + alpha*current;
}
```

Game Loop

- Tipos
  - Simples: CPU-*dependent*
  - Simples com dt: *CPU-independent*
  - Simples com dt fixo: CPU rápida simulando CPU-*dependent*
  - *Catch-up* simples: atualiza de acordo com o tempo de *render*
  - *Catch-up* com extrapolação: atualiza de acordo com o tempo de *render* e extrapola o restante
  - Frame skipping

Game Loop


period
(1 frame)
period
(1 frame)
period
(1 frame)
Update
Draw
overtime
Update
Update
Draw
sleep time
Update
time
update time
drawing time
over time (excess)
update time
update time
drawing time
sleep time

Game Loop - frame skipping

```
MAX_FPS = 50;
MAX_FRAME_SKIPS = 5;
FRAME_PERIOD = 1000 / MAX_FPS;
beginTime;      // the time when the cycle begun
timeDiff;       // the time it took for the cycle to execute
sleepTime;      // ms to sleep (<0 if we're behind)
framesSkipped;    // number of frames being skipped
sleepTime = 0;
```

Game Loop - frame skipping

```
while (!done) {
     beginTime = now();
     framesSkipped = 0;    // resetting the frames skipped
     update(); // update game state
     this.gamePanel.render(canvas); //render state to the screen
     timeDiff = now() - beginTime;  // calculate how long did the cycle take
     sleepTime = FRAME_PERIOD - timeDiff;// calculate sleep time
          if (sleepTime > 0) { // if sleepTime > 0 we're OK
               sleep(sleepTime); // send the thread to sleep for a short period
     }
     while (sleepTime < 0 && framesSkipped < MAX_FRAME_SKIPS) {// we need to catch up
          update(); // update without rendering
          sleepTime += FRAME_PERIOD; // add frame period to check if in next frame
          framesSkipped++;
     }
}
```

6. Game Loop

Exemplo Unity

7. Input

7. Input

- ➔ Como receber eventos e processar eventos?
  - ◆ O sistema operacional oferece diferentes formas de suporte aos eventos, geralmente uma das duas formas:
    - Callbacks e event listeners
    - Event polling e input states

7. Input

- Callbacks e event listeners
  - Implementação de funções especializadas em receber certos parâmetros
    - Keyboard, mouse, joystick
  - O sistema operacional chama todas as funções implementadas “cadastradas”
  - Sistemas operacionais mobile geralmente implementam essa arquitetura (Android, Java ME)

7. Input

Exemplo Android

7. Input

- Event polling e input states
  - São armazenadas mensagens ou eventos pelo sistema operacional, gerado pelas interrupções
  - Quando necessário processar o evento, é dado um poll que retorna a lista de eventos que aconteceu desde o último poll
  - Também é possível implementar um buffer de estados que é dado um reset todo poll
    - Acessível através de funções e membros estáticos

7. Input

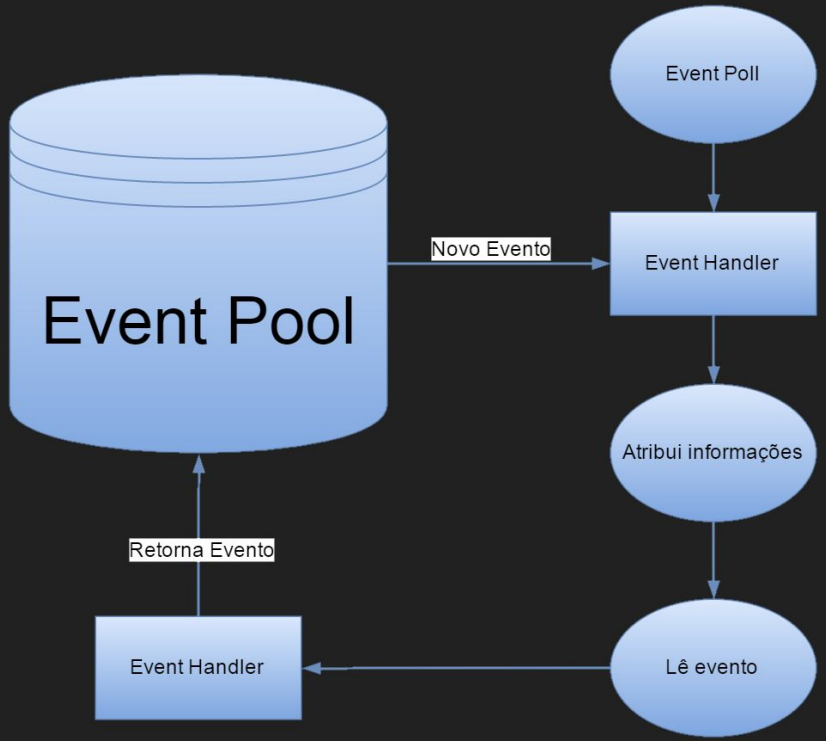
Exemplo SDL

Exemplo Unity

7. Input

- Event pool
  - Pool ≠ Poll
  - Parecido com thread pool, você cria uma piscina de objetos que representam eventos, para diminuir a quantidade de alocação toda vez que processar os eventos

7. Input


Event Poll
Event Pool
Novo Evento
Event Handler
Atribui informações
Retorna Evento
Event Handler
Lê evento

Dúvidas?

Referências

Referências

[1] Jason Gregory-Game Engine Architecture-A K Peters (2009)
[2] Game Coding Complete, Fourth Edition (2012) - Mike McShaffry, David Graham
[3] David H. Eberly 3D Game Engine Architecture Engineering Real-Time Applications with Wild Magic The Morgan Kaufmann Series in Interactive 3D Technology 2004
[4] http://gameprogrammingpatterns.com/
[5] http://gafferongames.com/
[6] http://docs.unity3d.com/Manual/index.html
[7] http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/
[8] https://en.wikipedia.org/wiki/Software_design_pattern
[9] https://www.youtube.com/user/BSVino/videos
[10] https://www.youtube.com/user/thebennybox/videos
[11] https://www.youtube.com/user/GameEngineArchitects/videos
[12] https://www.youtube.com/user/Cercopithecan/videos
[13] http://www.glfw.org/docs/latest/input_guide.html
[14] http://lazyfoo.net/tutorials/SDL/index.php
[15]
[16]
[17]