Contents

Acervo IDJ

Este documento foi criado para ajudar alunos da disciplina de Introdução ao Desenvolvimento de Jogos, da UnB, com os jogos desenvolvidos durante o semestre. Especificamente, isso é um acervo de técnicas implementadas por alunos e monitores de semestres anteriores, com a intenção de:

1. Propor soluções para problemas comuns no desenvolvimento dos trabalhos finais.
2. Centralizar ideias que podem ser de interesse para projetos variados, oferecendo formas de implementá-las.
3. ~~Tentar dominar o mundo~~

De forma alguma o objetivo desse acervo é *limitar* a imaginação dos alunos, ou privá-los de buscar suas próprias soluções - pelo contrário, são apenas sugestões que buscam minimizar o retrabalho com problemas previamente encontrados. Busquem suas proprias saídas, caso desejem, e **contribuições são sempre bem-vindas** .

Para organizar essa coleção, oferecemos abaixo uma tabelinha de conteúdo, com uma descrição resumida de cada tópico. Os links levam para os guias mais detalhados de implementação.

Tópicos aqui abordados podem ser vistos em maior detalhe na bibliografia sugerida ao final do arquivo.

Happy Coding!

Topicos

- Arquivos de Configuração: Maneiras de separar (e organizar) dados e recursos de jogo da implementação de sua engine.
- IA: Implementações de mecanismos de Inteligência Artificial para GameObjects, com variados propósitos e níveis de complexidade.
- Interpolação (Lerping): Técnica extremamente popular e versátil para tornar mais naturais movimentos e transições de diversas naturezas.
- Som Multicanal: Exploração da parte sonora da engine para gerar um efeito de música contextual/dinâmica.
- Event Queueing: Uma forma de organizar a comunicação entre sistemas - incremento arquitetural que facilita a implementação de outras técnicas (e torna a engine mais escalável)
- Sobrecarga de operadores: ...

Bibliografia

- Game Programming Patterns

Inteligência Artificial

Um tópico recorrente no meio do desenvolvimento de jogos é a inteligência artifical, ou IA. Praticamente qualquer jogo faz uso de alguma técnica que simule inteligência, sendo este um conhecimento obrigatório para qualquer um que queira se aventurar no mundo do game development.

- **Objetivo**: Codificar uma IA compativel com o jogo em desenvolvimento - seja para inimigos, aliados, minions, ou o que seja.
- **Proposta**: Estudar diversas implementacoes de IA (em C++), com niveis variados de complexidade, analisando suas forcas e fraquezas.

Um pouco de História

Entender o porquê as coisas foram de um jeito um dia nos ajuda a entender o porquê delas serem como são atualmente. E com IA não é diferente. Inteligência artifical é uma área presente junto aos jogos desde sua concepção. Temos o famoso PONG como exemplo de uma simples IA. Seu comportamento pode ser visualizado neste vídeo.

Como pode ser observado, o oponente do jogador tenta rebater a bola com todas as suas forças, mas falha em conseguir em certos momentos. É uma IA simples, mas que já leva em conta um importante conceito quando lidamos com inteligências para oponentes e inimigos controlados pelo computador:

A inteligência artificial perfeita não é aquela que é, de fato, *perfeita*.

A frase acima pode parecer sem sentido, mas motiva o estudo da IA não somente nos jogos, mas no campo como um todo. Buscamos tornar as inteligências em nossos jogos o mais *humanas* possível (na grande maioria das vezes), e os seres humanos não são perfeitos, pois cometem erros.

Trabalhando a imaginação

Figure 1: banana

Tomemos por exemplo um jogo com uma ideia simples: um bravo aventureiro deve desbravar uma masmorra para encontrar um tesouro perdido. O jogador, controlando este aventureiro, adentra um labirinto com a recompensa ao seu fim. Porém o labirinto está cheio de monstros! Dado este problema e a frase que mencionada acima, como devemos implementar a IA dos monstros?

Fosse essa IA *perfeita*, talvez ela não seria tão complexa. Na simulação de computador que compõe o jogo, o programa já tem as informações de onde o jogador está, como é o labirinto e onde está o monstro. Quisesse apenas que o monstro chegasse até o jogador e o eliminasse, usariamos um algoritmo de pathfinding e pronto. O aventureiro começa a explorar a masmorra, e em algum tempo o monstro chega até ele e o elimina. Daí questionamos: isso é divertido? Isso é o que de fato queremos?

A resposta, normalmente, é não. Pois não queremos a IA *perfeita*, queremos uma IA mais *humana*, ou seja, não onisciente.

Para tal, pensamos no que queremos que o monstro tenha conhecimento. Quais sentidos ele possui? Ele vê através das paredes? Ele se movimenta mesmo quando não percebe ameaças ao redor? São muitas variáveis a se levar em conta, e quanto mais realista e *humana* você quiser que seja sua IA, mais terão de ser pensadas. Não se amedronte, no entanto! A tarefa não é trivial, mas o resultado é recompensador para o programador.

Vamos primeiro estabelecer as características do nosso monstro:

- Nosso monstro não enxerga bem, vendo a apenas 3 metros de distância.
- No entanto, sua audição é aguçada, e ele consegue ouvir passos a até 30 metros, mesmo através de paredes!
- Enquanto não pressentir uma ameaça, nosso monstro vai apenas ficar parado, com uma pequena chance de se mover alguns metros de sua posição.
- Ao perceber que o aventureiro está próximo, ele começará a seguir seus sentidos, andando pelo labirinto até chegar próximo ao jogador.

Parece mais complexo, não é mesmo? Talvez a pergunta seja como codificar isso agora, e como o fazer está mais a frente neste documento. No momento, perceba apenas como a inteligência artificial que queremos não é aquela que executa com maestria o que a foi proposta, e sim aquela que possui *falhas*. Especialmente para inimigos isso é uma verdade, pois uma IA que trabalha contra o jogador e é *perfeita* não dá espaço para a diversão no jogo. Mas aí já estariamos entrando em campos do Game Design. Vamos voltar ao código por enquanto!

Raquete Inimiga

Vamos para nosso primeiro código! Vamos tentar planejar e codificar uma IA simples, no caso, uma IA que poderia ser usada para um jogo como PONG. Mas antes, uma observação: Assume-se que certos métodos e funcões já foram construídos, de forma que possamos focar apenas nas IAs neste documento. De toda forma, ao fim de cada exemplo, mostraremos como encaixar sua IA no jogo proposto.

Preparados? Vamos começar a pensar na **Raquete Inimiga**:

Event Queueing

A engine desenvolvida durante o semestre se baseia numa arquitetura comumente referida como **Entity-Component-System Architecture** (ou **Pattern**), que é basicamente a divisão lógica do programa em 3 partes, que são (isso mesmo):

- **Entity :** Nosso GameObject, essencialmente um container de components.
- **Component :** Dados sem comportamento - uma struct que guarda informações acerca de um aspecto de nossa Entity.
- **System :** Uma classe responsável por uma etapa específica do frame update da engine (combate, física, renderização, som, etc.).

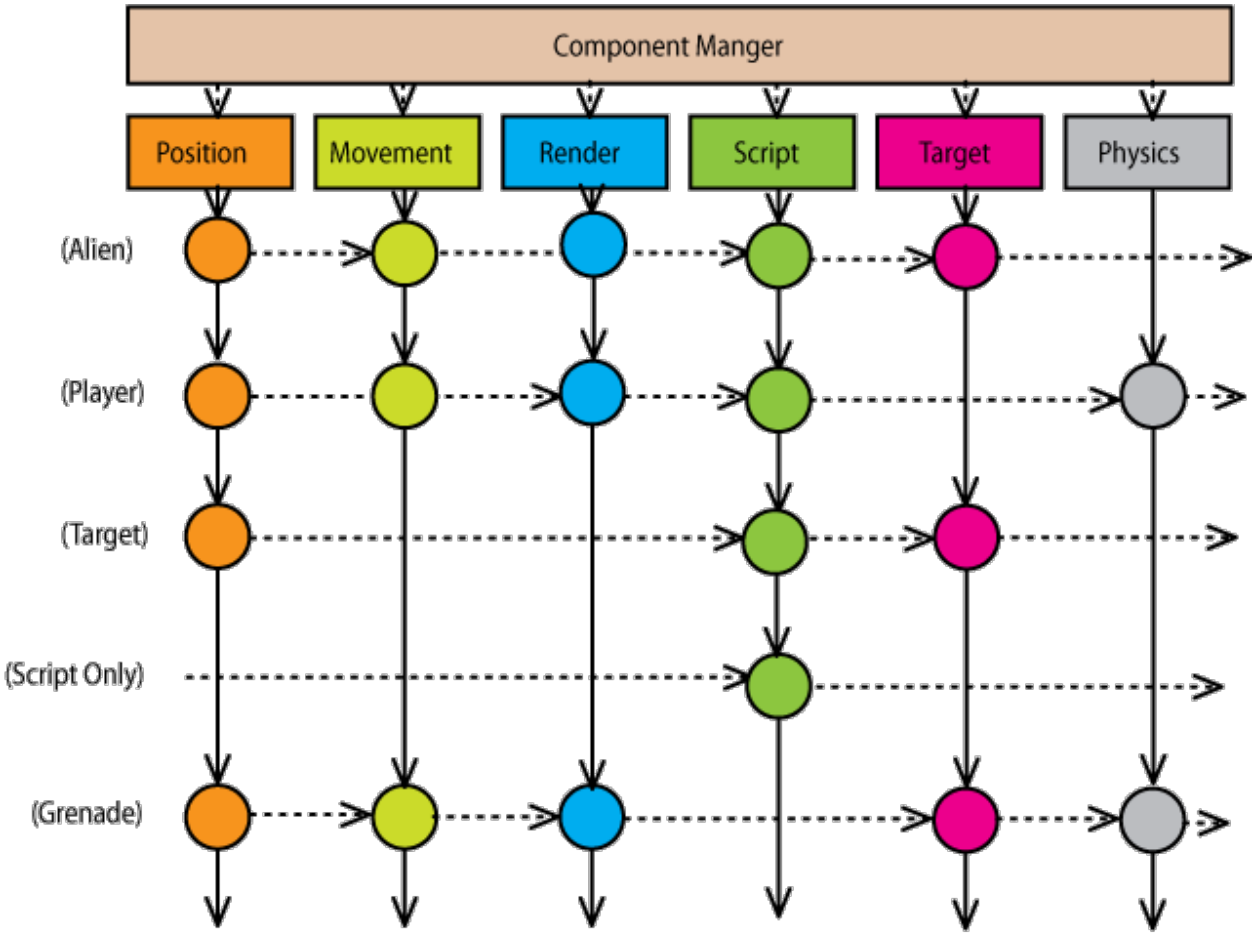

Figure 2:

A forma como esse padrão é implementado varia das formas mais selvagens de engine para engine - são comuns implementações em que entidades basicamente *"não existem"* - são apenas `ints` que se referem a uma grande tabela de componentes (ou seja, torna-se uma **chave**). Da mesma forma, componentes podem ou não adquirir comportamento, caso necessário, e sistemas podem ou não conhecer uns aos outros. Nesse tópico, vamos propor uma forma de implementação, que você deve seguir ou não conforme as necessidades do seu jogo.

- **Objetivo :** Aprimorar a estrutura da nossa engine para que seja facilmente extensível - **especialmente** em termos do comportamento dos elementos de jogo.
- **Proposta :** Isolar todo o comportamento do nosso estado de jogo em sistemas, que não se conhecem, e comunicam-se através de **eventos**.

Eventos

Eventos - nesse contexto específico - são um meio para um sistema reportar ao jogo um acontecimento que chama por alguma ação, mas que não é sua responsabilidade. Por exemplo:

- `Collision` detecta que um projétil colidiu com o player (isso, naturalmente, presume que ambos tenham colisores)
- O HP do player deve ser decrementado, baseado na quantidade de dano indicada pelo componente `Damage` ou `Bullet` do projétil
- **Como?**

`Collision` não sabe fazer isso (e nem tem porque saber), e menos ainda saberia `Health` como detectar a colisão entre eles. Precisamos de alguém que conheça ambos, para que receba uma mensagem de `Collision` e a repasse para `Health` . . . **State!**

```
    public class PlayState {
        ...
        void handle(GameEvent e);
    };
```

Isso evita que embaralhemos os sistemas (o que também é um jeito de resolver o problema, eu acho), e torna o mecanismo da engine fundamentalmente mais simples, como veremos em breve.

Ok, mas o que exatamente eu coloco em um evento?

Qualquer acontecimento que:

1. Não seja previsto no update loop
    - O loop prevê coisas como: calcular velocidades, mover entidades com velocidade, renderizar entidades com sprites, detectar colisões, etc.
2. Envolva mais de um sistema
    - O sistema de colisão, por exemplo, detecta colisões de objetos com o collision map, mas nem por isso é obrigado a lançar um evento: isso geraria uma situação boba, onde o sistema notifica o estado de jogo de um acontecimento que ele mesmo vai tratar - puro custo computacional.

Seguindo essa lógica, durante o Loop de Update, vários eventos surgirão, mas não necessariamente devemos interromper o loop por conta disso - precisamos de um lugar para *guardá-los*.

Ok, mas como eu faço issoooo?

Estrutura

Crie uma struct/classe GameEvent - o nome Event é muito comum entre frameworks, para os mais diversos fins. A própria SDL usa para eventos de input, por exemplo. Caso você adote muito a prática do `using namespace`, isso é uma péssima ideia.

```
// GameEvents.hpp
struct GameEvent {
    GameEventCode code() { return this->code; };
    private GameEventCode code;
};
```

Ok, State já sabe como receber eventos (caso você tenha adicionado aquele método ali em cima), mas ainda não sabe pra quem mandar, e nem onde armazená-los.

```
// State.hpp
public class State {
```

```
    private:
        ...
        std::queue<GameEvent> queue; // Aha!

        // Relaciona o código do evento dado com o ÍNDICE do sistema responsável
        std::unordered_map<GameEventCode, int> handlers;

    public:

        // "Notifica" um evento
        void notify(GameEvent);

        // "Resolve" um evento
        void handle(GameEvent);
};

// State.hpp

// Adiciona novo evento na fila
void State::notify(GameEvent e) {
    queue.push(e);
}

// Handle: notifica o sistema para qual o evento se torna relevante
void State::handle(GameEvent e) {
    if ((auto sys = systems[handlers[e.code]]) != nullptr) {
        sys->handle(e);
    }
}
```

Ok, State sabe como mandar eventos para sistemas, mas como ela sabe **para quem** mandar?

```
    // System.hpp
    public class System {
        // Retorna quais eventos esse sistema é capaz de tratar
        std::vector<GameEventCode> handles() = 0;

        // Retorna se foi possivel processar o evento.
        bool handle(GameEvent);
    };

    // Agora ficou fácil!

    // PlayState.cpp
    PlayState::addSystem(System* sys){
        ...
        for (auto code : sys.code()){
            this.handlers[code] = systems.size();
        }
        systems.push_back(sys);
    };
```

Feito tudo isso, parte do seu main loop pode ser colapsada em poucas linhas:

```
    for(auto& sys : systems) sys.update(dt);

    while (!queue.empty())
        if (!handle(queue.pop()))
            { /* lide com o erro da forma como preferir */ }
```

Isso não é incrível?

Aplicação

Tá... mas o que eu faço com tudo isso?? Meu jogo meio que não mudou...

Crie um trilhão de eventos. Eu coloquei só dois aqui, os outros 999.999.999.998 são contigo (lembre de adicionar os campos que forem relevantes em cada um)

```
// GameEvent.h
enum GameEventCodes {
    Damage,
    PlaySound,
    Effect,
    ...
};

struct Damage : GameEvent {
    int amt;
    int target;
    Damage(int amt, int target){
        this->code   = Damage;
        this->target = target;
    };
};

struct PlaySound : GameEvent {
    std::string file;
    Damage(std::string file){
        this->code = PlaySound;
        this->file = file;
    };
};

...
```

Uma nota sobre o código acima: Em **C++** (ao contrário de **C**), **herança de structs é possivel** - O código acima compila em **C++11**. Pode ser preferível a classes, por motivos de performance e minimização de alocação dinâmica, exceto caso seja necessário armazenar uma quantidade monstra de dados e métodos dentro deles. Alternativamente, pode-ser usar introspecção ou metaprogramação para manipular os eventos, ao invés de uma enum.

Arquivos de Configurações / Asset Management

Quando falamos em Game Engine Development, precisamos focar bastante no significado de **Engine** para esse contexto: Estamos desenvolvendo um programa que é essencialmente uma plataforma - *configurando* essa plataforma, geramos um jogo.

Predomina sempre a noção de separar um programa em **dados** e **implementação** - no nosso, caso, a implementação é a tal da **Engine**, e este tópico visa apresentar formas de organizar então os nossos **dados**.

- **Objetivo**: Construir uma estrutura de dados de tal forma que todos os dados do nosso jogo (fases, entidades, músicas, imagens, padrões de IA, configurações gráficas, etc...), estejam externamente definidos
- **Proposta**: Empregar uma biblioteca de parsing de alguma linguagem de markup, e construir nosso programa de forma a receber esses dados dinamicamente. Esse processo ocorrerá em algumas etapas:
  - Escolher a linguagem
  - Escolher a biblioteca
  - Definir as tags que vão nesses arquivos, baseando-se nas estruturas que a engine exige.

Um exemplo simples para ilustrar o que procuramos:

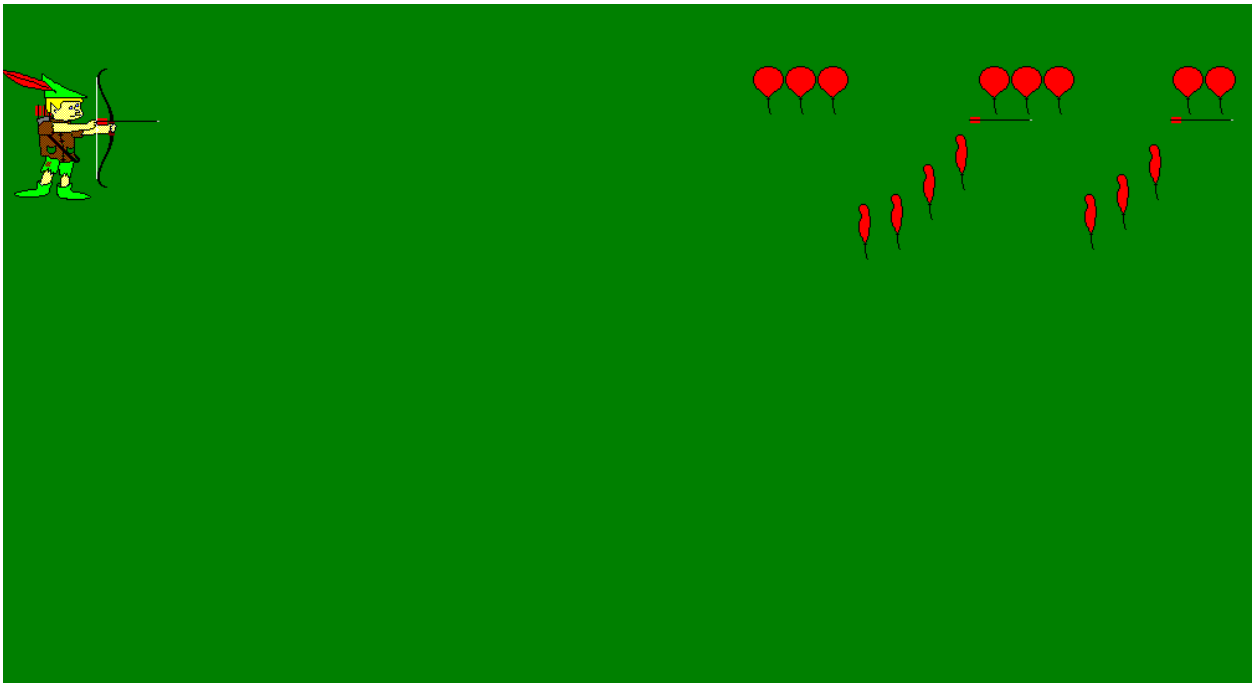

Figure 3: O melhor jogo da história

Suponha que toda vez que clicarmos o mouse, queremos colocar, numa fase, um balão voando pra cima, que é uma **entidade** que possui os seguintes componentes:

- *Position*: Recebe `x` e `y`. Assumimos que, caso o valor seja nulo, utilizaremos a posição do mouse.
- *Speed*: Recebe `x` e `y`, que quando nulos, assumiremos 0.
- *Sprite*: Recebe `image` (string), `centered` (bool, default: false), `width` e `height` (int) - caso nulos, calculamos a partir da imagem (`SDL_QueryTexture`)
- *Hitbox*: Recebe `width` e `height` - quando nulos, utilizaremos os valores da `sprite` dessa entidade, caso exista; do contrário, 0.

*(obs.: por conta dessas inferências, a ordem **importa**)*

Isso fica convenientemente modelado no seguinte YAML:

```yaml
entity:
    position:
    speed:
      y: 0.5
    sprite:
      file: "./img/bloon.png"
    hitbox:
```

ou, em XML:

```xml
<entity>
    <position/>
    <speed y="0.5"/>
    <sprite file="./img/bloon.png"/>
    <hitbox/>
</entity>
```

Para o Player, dependendo da arquitetura desenvolvida, podemos até fazer algo mais interessante:

```xml
<entity>
    <position />
    <player/>
    <input>
        <bind key="spacebar"   command="shoot">
        <bind key="arrow_up"   command="move_up">
        <bind key="arrow_down" command="move_down">
    </input>
    <sprite file="./img/mr_bowman_himself.png"/>
</entity>
```

Longe do ideal, mas é um começo.

Não faz nenhum sentido guardar dados em C++ (~~mas quem acredita sempre alcança, não desista dos seus sonhos~~), portanto precisamos escolher uma linguagem apropriada para essa tarefa. C++ também não é uma linguagem que predispõe de parseres competentes por padrão, portanto convém incluirmos uma biblioteca para isso. Por simplicidade, escolhemos apenas as opções de integração mais conveniente (header-only, em geral) - seguem algumas sugestões:

1. **YAML:** Menos verbosa de todas, mais simples e legível. Configurar a yaml-cpp, no entanto, exige compilação da biblioteca e modificação o makefile para ligá-la.
2. **TOML:** Simples, eficaz, talvez um pouco prolixa - funciona muito bem para tabelas, mas ofusca um pouco estruturas altamente *nestadas*. Pode ser adotada trivialmente usando tinytoml.
3. **JSON:** Escolha bastante comum para objetos complexos - utilizada por vários editores de texto modernos para arquivos de configurações. json11 é uma opção de biblioteca bastante simples.
4. **XML:** Verbosa, porém carrega ridiculamente rápido através da pugixml - cuja incorporação é extremamente simples e intuitiva.

Importante: Para jogos pesados (resource-intensive), esses formatos podem ser inviáveis.

- Guardar dados em texto gera um desperdício monstruoso de espaço. Fora isso, cada tag XML vazia gasta o dobro (duas vezes, 2x, 200%) de bytes que a mesma tag em YAML, por exemplo. Pensa...
- Idealmente, para esses projetos, alguma forma de serialização de dados é imprescindível.
- Para jogos indie / leves / com poucos recursos, podemos permanecer com esses formatos pela conveniência de ter uma linguagem intermediária legível.

Normalmente, essas bibliotecas utilizam a seguinte estratégia: montar uma árvore de nós (DOM), cujos valores (que equivalem aos valores reais no arquivo), são objetos, que devem ser **castados** para o tipo desejado, normalmente, através de um método providenciado - a exemplo, pugixml:

```
using namespace pugi;      // Legibilidade
using namespace Component; // Caso seus componentes estejam num namespace

Entity& entity(xml_node data){
  if(data && data.name() == "entity"){
    e = Entity.new();
    // Ordem de componentes não importa: normalmente é melhor, exceto se você tiver uma quantidade obs
    if (comp = data.child("position")){
      e.addComponent( Position(
        comp.attribute("x") ? comp.attribute("x").as_float() : Input::GetMouse.x(),
        comp.attribute("y") ? comp.attribute("y").as_float() : Input::GetMouse.y()));
    }

    if (comp = data.child("speed")){
      e.addComponent( Speed(
        comp.attribute("x") ? comp.attribute("x").as_float() : 0.0,
        comp.attribute("y") ? comp.attribute("y").as_float() : 0.0));
    }

    if (comp = data.child("sprite")){
      e.addComponent( Sprite(
        comp.attribute("file") ? comp.attribute("file").value() : Input::GetMouse.x(),
        comp.attribute("x")    ? comp.attribute("x").as_int()   : 0,
        comp.attribute("y")    ? comp.attribute("y").as_int()   : 0));
    }

    if (comp = data.child("hitbox")){
      e.addComponent( Hitbox(
          comp.attribute("width")  ?
            comp.attribute("width").as_float() :
              ((r = e.getComponent("render")) ?
                r.width :
                0.0),
          comp.attribute("width")  ?
            comp.attribute("width").as_float() :
              ((r = e.getComponent("render")) ?
                r.width :
                0.0)));
    }

    return e;
  }
  std::cout << "[xml] entidade não chama \"entity\" - não sei o que fazer" << std::endl;
  return null;
}
```

O excerto acima foi construído baseado nas regras estipuladas mais acima, para valores padrão - caso seja conveniente, você pode lançar uma exception, ou tratar esse erro da forma como achar mais adequado. Uma outra implementação possível seria construir um loop que itera sobre os nós, preenchendo quaisquer componentes conforme necessário.

```
for(auto& comp : data.children()){
  switch (comp.name()){
    case "position":
      e.addComponent(Position(
```

```
          comp.attribute("x") ? comp.attribute("x").as_float() : Input::GetMouse.x(),
          comp.attribute("y") ? comp.attribute("y").as_float() : Input::GetMouse.y()));
        break;

      case "speed":
        e.addComponent(Speed(
          comp.attribute("x") ? comp.attribute("x").as_float() : 0.0,
          comp.attribute("y") ? comp.attribute("y").as_float() : 0.0));
        break;

      case "sprite":
        e.addComponent(Sprite(
          comp.attribute("file") ? comp.attribute("file").value() : Input::GetMouse.x(),
          comp.attribute("x")    ? comp.attribute("x").as_int()   : 0,
          comp.attribute("y")    ? comp.attribute("y").as_int()   : 0));
        break;

      case "hitbox":
        e.addComponent( Hitbox(
            comp.attribute("width")  ?
              comp.attribute("width").as_float() :
                ((r = e.getComponent("render")) ?
                  r.width :
                  0.0),
            comp.attribute("width")  ?
              comp.attribute("width").as_float() :
                ((r = e.getComponent("render")) ?
                  r.width :
                  0.0)));
        break;
    }
  }
```

Config File

Figure 4: holy shit

Perceba que estamos focando na construção de um elemento de jogo, mas podemos extrapolar essa estrutura para os mais diversos fins. Um deles, bastante útil, é criar uma classe estática (ou struct, ou o que quer que seja, a vida é tua) para concentrar **configurações de jogo**, como:

- Resolução da tela

- FPS
- Dificuldade (?)
- Debug Mode
- Configurações gráficas (mais relevante para engines 3D, mas ainda assim dá pra escolher, entre outras coisas, o nível de antialiasing de sprites)
- Volume global

```
typedef struct CONFIG {
  int  screen_width;
  int  screen_height;
  int  fps;
  bool debug;

  int  volume_music;
  int  volume_sfx;
} config;
```

Desde que seus parâmetros sejam refletidos da mesma maneira no arquivo (config.yml, config.xml etc.), construí-la deve ser trivial. Mas espera... se cada classe do meu jogo precisa saber se construir a partir de um xml_node, meu código vai ficar uma bagunça! Que desastre!!!

Pois é... é hora de dar uma arrumada na casa.

Namespacing

Por conveniência, você pode separar um `namespace` para guardar funções `f : xml_node -> <tipo desejado>`, que podem inclusive chamarem-se conforme necessário. Ser quisermos uma função que *carregue a fase inteira*, definimos várias outras, que construam partes menores, como `Entity`, `Background`, `Music`... o que for conveniente.

Idealmente, queremos encapsular **tudo** que seja relativo a essa linguagem escolhida (e, portanto, **todo** processo de construção) dentro desse namespace, tanto por higiene de código como por flexibilidade - torna-se muito mais fácil modificar a forma de armazenamento caso ela esteja totalmente localizada em um lugar. Para tornar nossa Play completamente agnóstica à linguagem, podemos (devemos) criar uma função que constrói a fase recebendo uma string. Um protótipo desse `namespace` seria algo assim:

```
namespace build {
  using namespace pugi;

  Level* level(std::string file){
    /* A função mega blaster power ultra sayajin que constrói tudo pra um dado arquivo.
      - abre o arquivo
      - faz parsing, gerando um DOM
      - chama os demais métodos
      - trata erros sumários de leitura (erros especificos ficam para cada função
      - caso você utilize exceptions:
          - executar um bloco try-catch tratando ou relançando exceções.
      - retorna Level construída (ou nullptr)
      */
  };

  // Um Logger simples, local e meio inútil
  void log(std::string err){
    std::cout << "[build] erro: " << err << std::endl;
  }
```

```
  Entity* entity(xml_node data){
    /* code */
  };

  Music* music(xml_node data){
    /* code */
  };

  // Asset Preloading
  Resources* resources(xml_node data){
    /* code */
  }

  Configs* configs(xml_node data){
    /* code */
  }

  Tilemap* tilemap(xml_node data){
    /* code */
   }
   ...
};
```

Exemplos de Modelagem

Som Multicanal : Supondo uma engine capaz de tocar quatro canais

- YAML

```yaml
music:
  files: [
    main_menu_drums.ogg,
    main_menu_flute.ogg,
    main_menu_synth.ogg,
    main_menu_bass.ogg
  ]
  mixes: [
    [127, 0,    0,    0   ],  # Just drums
    [127, 127,  127,  127 ],  # Full band
    [0,   100,  60,   100 ],  # Soft
  ]
```

- XML

```xml
<music>
  <files>
    <track filename="main_menu_drums.ogg"/>
    <track filename="main_menu_flute.ogg"/>
    <track filename="main_menu_synth.ogg"/>
    <track filename="main_menu_bass.ogg"/>
  </files>
  <mixes>
    <mix 1="127"  2="0"   3="0"   4="0"   /> <!--| Just drums -->
    <mix 1="127"  2="127" 3="127" 4="127" /> <!--| Full band  -->
```

```
    <mix 1="0"    2="100" 3="60"  4="100" /> <!--| Soft       -->
  </mixes>
</music>
```

Background Multi-Camadas : Supondo uma engine onde `scale` é um fator de ampliação de imagem, `parallax` é um fator de atenuação de movimento, e `offset` é um fator de deslocamento inicial (em pixels).

- YAML

```yaml
screen:
  main: 3
  layers:
    - file: "./assets/img/woods/layer0.png"
      scale_x: "0.75"
      scale_y: "0.75"
      parallax_x: "0.15"
      parallax_y: "0.0"
      offset_x: "700"
      offset_y: "-300"
    - file: "./assets/img/woods/layer1.png"
      scale_x: "0.75"
      scale_y: "0.5"
      parallax_x: "0.1"
      parallax_y: "0.8"
      offset_x: "0"
      offset_y: "0"
    - file: "./assets/img/woods/layer1.png"
      scale_x: "1.1"
      scale_y: "1.1"
      parallax_x: "1.0"
      parallax_y: "1.0"
      offset_x: "0"
      offset_y: "0"
```

- XML

```xml
<screen main="2">
  <layer tiled="false" file="../img/screen_layers/cidade_1.png"
    parallax_x="0.15" scale_x="0.75"  offset_x="700"
    parallax_y="0"    scale_y="0.75"  offset_y="-300" />
  <layer tiled="false" file="../img/screen_layers/cidade_2.png"
    parallax_x="0.1"  scale_x="0.75"  offset_x="0"
    parallax_y="0"    scale_y="0.5"   offset_y="0"    />
  <layer tiled="false" file="../img/screen_layers/cidade_3.png"
    parallax_x="0"    scale_x="1.0"   offset_x="0"
    parallax_y="0"    scale_y="1.0"   offset_y="0"    />
</screen>
```

Modelagem Estática

Seguindo o padrão proposto, você provavelmente conseguirá modelar suas fases com razoável tranquilidade. Porém, inevitavelmente, você começará a se repetir. Sendo esse o caso, convém que você separe seus recursos em duas partes:

- Um catálogo **global** de recursos (como se fossem macros).

- Uma receita que utiliza (instancia) elementos desse catálogo.

Por exemplo, se eu sei que *tartaruguinhas vermelhas* tem uma `velocidade`, uma `posição`, um `spritesheet`, uma `hitbox`, uma `hurtbox`, uma `health` e estados determinados, eu não preciso digitar tudo cada vez que for instanciar uma: posso dizer

```
  <entity name="tartaruguinha">
    <speed x="10"/>
    <position/>
    <sprite file="./assets/img/enemies/tartaruguinha_vermelinha_fofinha.png">
    <hitbox  w="120" h="260"/>
    <hurtbox w="120" h="260"/>
    <health  max="1" regenerates="true"/>

    <!--| Caso você opte por implementar uma FSM -->
    <states>

    </states>
  </entity>
```

E instanciá-la trivialmente

```
<level>
  ...
  <entities>
    <entity player refid="supermario"/>
    <entity player refid="superluigi"/>

    <entity refid="tartaruguinha"/>
    <entity refid="tartaruguinha"/>
    <entity refid="goomba"/>
    ...
  </entities>
  ...
</level>
```

Tópicos Relacionados:

- **Editor de Fase :** A necessidade dele se torna dolorosamente óbvia quando precisamos manipular um arquivo de 12 mil linhas (e acredite, esse dia chega mais rápido do que parece) para modificar a posição de um objeto na fase. Para a implementação de um editor de fase, torna-se imprescindível que tenhamos uma forma dedicada de armazenar dados.

Lerping

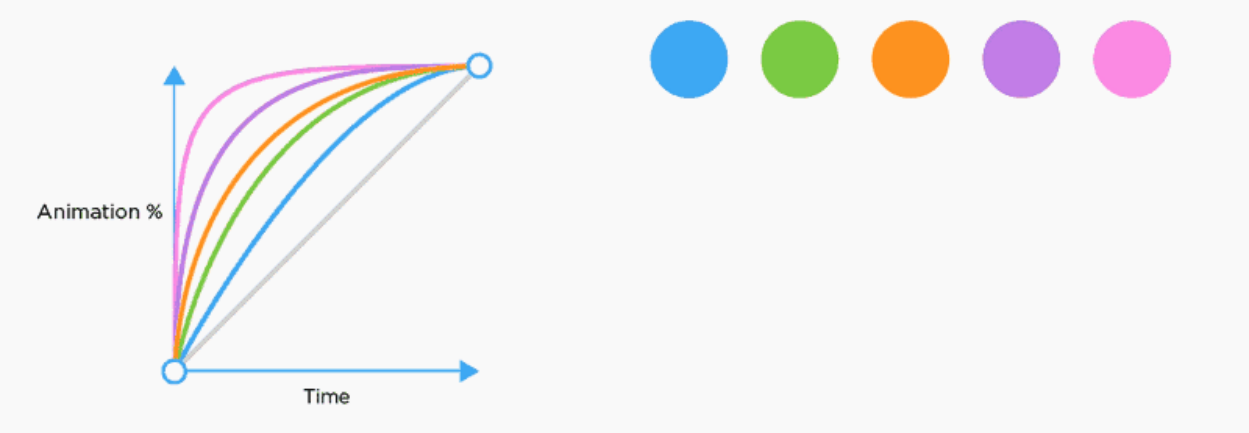

Figure 5:

- **Esse post** - origem do gif acima - contém diversos exemplos visuais diferentes que ajudam a compreensão dos assuntos desse tópico (que podem soar abstratos por si só), e explica como as técnicas abordadas são utilizadas no contexto de web design. Leitura recomendadíssima!

Lerping (Interpolação Linear) é uma tecnica amplamente utilizada pela indústria para gerar transições e movimentos mais naturais e flexíveis de forma simples e eficaz. Se baseia num principio simples: **dados dois valores, parametrizar um valor intermediário**, que usaremos para parametrizar qualquer grandeza parametrizável. Entendeu?

(disclaimer: parte deste capítulo - a parte mais matemática - vai soar um pouco teórica, mas a implementação em si é simplíssima)

- **Objetivo :** Obter geratrizes genéricas para valores intermediários entre dois extremos, para gerar **animações bonitinhas** - a coisa mais importante em todo e qualquer jogo, indiscutivelmente.
- **Proposta :** Criar uma biblioteca de funções **interpoladoras** simples que possam ser combinadas entre si (caso necessario) para gerar padrões transitivos interessantes, da forma mais versátil possível.

Essa palestra (que, apesar do nome imponente e de um pouco de matemática, é razoavelmente simples) ilustra perfeitamente o que vamos desenvolver, e serviu de base conceitual para a maioria dos conceitos a seguir. Recomendo fortemente que veja essa palestra antes de continuar: não só é uma excelente **motivação**, como também mostra na **prática** o que isso e capaz de fazer.

O Básico

Vamos estabelecer alguns conceitos antes de comecar:

- **Lerping** - Interpolar **linearmente** dois valores. definiremos uma função genérica `lerp` para dar cabo disso: ela recebe dois valores de um tipo `T` e um parâmetro `s` para calcular um valor intermediário.

```
// Lerping parametrico, por definição
template<class T> T lerp(T a, T b, float s){
  return a*(1-s) + b*s;
}
```

- **Transform** (ou Transformação) - Função *f:R->R* (ou seja, float -> float) que parametriza uma variável no intervalo [0,1], mantendo suas extremidades. Ou seja, f(0) = 0 e f(1) = 1. Para implementá-las, utilizaremos **functores**

(`structs` com operador `()`), que são uma forma bastante utilizada em `c++` de definir um *tipo* que contém comportamento.

- *(para que o comportamento de uma transformação seja interessante, precisamos também que, dentro desse intervalo, f(t) != t)*
- Atenção: Declarar esse tipo é assustador, mas so acontece uma vez.

```
// Nosso tipo transform
typedef struct TRANSFORM {
TRANSFORM(float (*f)(float)) : proc(f) {};         // Construtor
float (*proc)(float);                              // Função
      float operator() (float x) { return (*proc)(x); }; // Operador ()
} transform;
```

Legenda: temos um **construtor**, que recebe uma função `float->float` como parâmetro, uma função (`proc`), para guardar nosso comportamento, e um operador `()`, para chamarmos `proc` de forma legível.

Convem separar um namespace `lerp` para conter essas definições - isso evita colisões de nome - e podemos chamá-las através do comando `using namespace lerp`, caso desejemos. Para que possamos usar tipos genéricos parametrizados, o melhor (senão único) caminho é certamente **sobrecarregar operadores**.

Combinados, esses blocos de código são bastante poderosos: eles nos permitem utilizar **qualquer** `transform` (que, como veremos, são ridiculamente simples de implementar) para interpolar **qualquer** tipo. Note que `s` pode estar fora do intervalo `[0,1]`, porem isso dificilmente faria sentido.

Ok mas cade as animações bonitinhas?

Transformações

Calma, agora vamos implementar nossas transformações:

```
  namespace lerp {
    ...
    // não Faz Nada
    transform None    ([](float x){ return x;           });

    // Comeca lento, termina acelerado (bruscamente)
    transform Square  ([](float x){ return x*x;         });

    // Comeca rapido, termina suave (desacelerado)
    transform EaseOut ([](float x){ return 1-(1-x)*(1-x) });
  }
```

Nao se deixe confundir pela notação - a única coisa que estamos fazendo é declarar variaveis que na verdade são funções. C++ oferece a notação `[capture](args){ body };` para definir uma função anônima, que pode ser associada a uma variável e chamada sucessivas vezes - isso me dá a oportunidade de apresentar a minha linha **favorita** de C++:

```
  [](){}; // Função que não captura nada, não recebe nada e não retorna nada
```

O pequeno truque é: `c++` automaticamente casta lambdas (funções anônimas) para ponteiros para funções.

ANYWAY, como usar isso??

1. **Transforme** o parâmetro `s` utilizando qualquer combinação que quiser.
2. **Interpole** as variáveis usando o valor transformado.

```
  using namespace lerp;

  Vec2 a(0.f, 0.f);
```

```
Vec2 b(1.f, 1.f);
float s = 0.2;

Vec2 eased = lerp(a, b, EaseOut(s));
Vec2 sqrd  = lerp(a, b, Square(s));
```

O exemplo acima ilustra de maneira bastante simples nosso mecanismo interpolativo. Para experimentar com ele, você pode, por exemplo: - Adicionar uma entidade avulsa em GameState - Passar como parâmetros dois `Vec2` (`a` e `b`) para essa função, junto com o tempo corrente de um `Timer` dividido pelo proprio tempo máximo (para que fique *normalizado*, ou seja, entre 0 e 1), e associar uma `transform` ao mecanismo. - Associar o vetor a posição de algum componente gráfico, como `Face`, a cada frame. - Associar uma tecla ao método `Timer::reset`, fazendo com que a `Face` volte ao estado original. - Substituir a `transform` escolhida e observar a mudança de comportamento.

Ok, como podemos sofisticar esses movimentos?

Possibilidades

1. Podemos operá-los:

   ```
   // Interpolação media entre linear e quatratica
   Vec2 avg = lerp(a, b, (Square(s) + None(s)) / 2.0);
   ```

2. Podemos combininá-los, da seguinte forma:

   ```
   Vec2 combined     = lerp(a, b, Square(EaseOut(s)));
   Vec2 doublesquare = lerp(a, b, Square(Square(s)));
   Vec2 triplesquare = lerp(a, b, Square(Square(Square(s))));
   ...
   ```

- Mas poxa, combinar essas funções gera um monte de parênteses, e parênteses são chatos ~~exceto se você programar em lisp~~. Como resolver isso?
  - Podemos sobrecarregar o operador `>>` de forma **idêntica** ao outro, de forma que `Square(Square(Square(s)))` vira `Square >> Square >> Square(s)`.

3. Podemos **interpolar transformações**
   - Temos os functores `Square(x)` (começa suave, termina brusco) e `EaseOut(x)` (começa brusco, termina suave).
   - Queremos construir uma transformação que *comece e termine suave* - ou seja, queremos uma transformação que **comece** `Square(x)` e **termine** `EaseOut(x)`.
   - `lerp(a, b, lerp(Square(s), EaseOut(s), s))`
   - Conforme `s` evolui, a transformação gradualmente deixa de ser `Square(x)` e passa a ser `EaseOut(x)`

Exemplos de Uso

- *Som*: Podemos interpolar o volume de um canal entre 0 e 155, usando `lerp(0, 155, Transform(x))`, gerando efeito de fade.
- Novamente, coloque um `Timer` pra rodar, controlando o comprimento do fade, chame getTime, *normalize* (divida pelo tempo total), transforme e interpole!
- *Transparencia*: Podemos interpolar o canal alpha de uma textura entre 0 e 1 usando `lerp(0, 1, Transform(x))`
- Timer, getTime, normalize, transforme, interpole!
- *Cor*: Sobrecarregue operadores para RGB, que você provavelmente guarda em uma struct.
- Timer, getTime... você entendeu.
- *Mapa*: Voce pode interpolar a opacidade do proximo comodo em funcao da distancia entre o player e ela, gerando um efeito dos comodos brotarem do escuro.

O video recomendado oferece outras transformações e fórmulas que não foram feitas aqui, além de explicar a origem de algumas dessas formulações.

Sobrecarga de Operadores

Sobrecarga de Operadores (Operator Overloading) é um recurso hoje comum a várias linguagens, que permite ao usuário redefinir o significado de um operador (como +, −, `[]` e até mesmo `typecast`) baseado nos seus operandos - em geral, structs e classes definidas pelo proprio usuário. Isso categoriza uma forma de *sobrecarga* - daí o nome da técnica. Esse recurso tem o potencial de tornar o programa muito mais expressivo, ou deixá-lo igualmente mais confuso.

- **Objetivo:** Aumentar a legibilidade do código, fazendo sua manutenção mais fácil e prática.
- **Proposta:** Implementar sobrecargas para operações comuns, frequentes e/ou complexas.

  (nota: caso você deseje entender um pouco melhor sobre esse mecanismo, leia a referência citada abaixo e [essa resposta no stack overflow][#stackoverflow], que ilustra de forma bem clara como implementá-la)

Prólogo: Os Operadores

Segundo a referência oficial, estes são os operadores que temos à nossa disposição:

```
+ - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= <=> && || ++
-- , ->* -> () []
```

Perceba que a maioria deles já presume um significado, como `->` sendo o acessor de referencia, ou `[]` sendo o acessor de índice/chave de um container. ~~Infelizmente,~~ Nada te impede de sair sobrecarregando tudo indiscriminadamente - mas, como dito, isso tem efeitos colaterais: seu código pode se tornar desnecessariamente ofuscado e convoluído. Imagina só se cada vez que você ler `Vec2 >>= Vec2`, tiver que abrir a classe Vec2 e descobrir que isso na verdade significa *Produto Cruzado entre as Ortonormalizações dos Vetores*...

Pois é.

Exemplo 1: Básico

Um exemplo simples, comum e muito pouco chocante (embora imprescindivel) é implementar operaçoes aritméticas comuns entre Vec2:

```
Vec2 Vec2::operator+(Vec2 const &b) const {
    return Vec2(x+b.x, y+b.y);
}
```

Uma vez definida essa operação, podemos chamá-la trivialmente:

```
Vec2 b(2, 4), c(1, 1);
Vec2 a = b + c;
std::cout << "(" << a.x << ", " << a.y << " )" << std::endl; // (3, 5)
```

Basicamente todo jogo precisa de uma biblioteca de geometria (seja ela bidimensional ou tridimensional ~~ou 11-dimensional~~), e convém bastante que tenham sido implementados esses operadores. Do contrário, cada operação de soma, subtração, produto, divisão - e, dependendo da biblioteca desenvolvida, normalização, produto interno, produto cruzado etc... - seria manualmente digitada - ou chamada atraves de um método da classe struct, que torna-se menos claro e distinto.

Exemplo 2: Algo Muito Prático

Um operador surpreendentemente útil é esse tal de `typecast`. Vamos implementar o typecast de Rect para SDL_Rect:

```
Rect::operator SDL_Rect() const {
    SDL_Rect rec;
    rec.x = floor(x);
    rec.y = floor(y);
    rec.w = ceil(w);
    rec.h = ceil(h);
    return rec;
}
```

Essa conversão seria normalmente feita à mão toda vez que se precisasse chamar alguma função da SDL que receba como argumento um SDL_Rect. Utilizando essa implementação, podemos agora simplemente **passar Rect para uma função da SDL que exiga um SDL_Rect!** Baseado nos dois tipos detectados, o compilador automaticamente chama qualquer typecast aplicável, antes de tratar o conflito como um erro - dessa forma, nosso `Rect` é castado automaticamente - não é mais necessário fazer essa conversão!

Podemos definir outros typecasts, dependendo da necessidade do nosso jogo, como `SDL_Color`.

E isso não é tudo!

A STL (Standart Template Libary) do C++ usa sobrecarga de operadores em vários momentos. - Como o std::vector e o std::unordered_map conseguem usar `[]`? Eles sobrecarregaram o `operator[]`! - Porque podemos fazer acesso ao conteúdo de `std::shared_ptr`, e obtemos um objeto do tipo parametrizado? Isso mesmo! Ele sobrecarrega o `operator*`, e também o `operator->`. Incrível né?

Nota de Precaução

Tome muito cuidado ao sobrecarregar alguns operadores - especialmente esses aqui:

- **<< e >> :** Operadores de input/output formatado para `std::cout` e diversos formatos de arquivos - pode gerar uns erros nefastos se misturados. Por sorte, podemos interferir na precedência das operações utilizando **parênteses**.
- **`()` :** Operador de chamada - um dos mais comuns de se sobrecarregar em structs, gerando a estrutura que chamamos de **Functor** - ocasionalmente um setter. Por ser uma estrutura tão comum e previsivel, dar a ele um significado muito exotico não é boa ideia.
- **<, >, <=, >=, == :** Novamente, são operadores de significados muito estabelecidos, e não é boa ideia mudar isso, embora possível.

Exemplos de Uso

- Lerping: O tópico sugere o uso do operador >> para chamadas de transformações como meio de evitar parênteses. No ambiente funcional, isso é comum, e é reflexo de uma técinca nomeada **composição** de funções.

Bibliografia

- Cpp Reference

Som Multicanal

Motivação

Uma forma de incrementar a imersão do jogador no mundo do game é através da musica. Com poucas exceções, games comerciais implementam música na forma de um arquivo simples, reproduzido conforme apropriado. Há formas de tornar a musica mais dinâmica e responsiva a elementos do jogo, porém, e isso pode ser especialmente útil para jogos que:

- Se baseiem em alguma mecânica que envolva música
- Carreguem alguma informação de jogo atraves da música
- Busquem construir uma ambientação sonora mais profunda.

O exemplo mais clássico de um jogo com musica multicanal seria o Super Mario World (ja ouviu falar?) - qualquer um que o tenha jogado lembra claramente da percussão que aparece quando o Mario se une ao Yoshi. Bastante simples, certo? Nada alem de uma faixa de percussão em reprodução simultânea à música principal. Espera... a nossa engine ainda não permite isso.

Plataforma

A SDL_Mixer nos traz uma limitação: apenas um canal de musica pode ocorrer simultaneamente. A idéia, então, é implementarmos Music de forma semelhante a Sound: cada camada da nossa música torna-se um Chunk, e Music contem então um int que indica quantos canais ela necessita - cada qual carrega o respectivo chunk. Play e Stop manipulam todos os canais simultaneamente (para garantir sincronia). **Cuidado com o volume** de reproducao do canais - a SDL_Mixer não oferece nenhum compressor nativo, ou seja, caso haja volume demais, havera clipping, e reproduzir vários canais ao mesmo tempo propicia esse tipo de acidente. O volume de um chunk pode variar entre 0 e 128, e podemos modificá-lo utilizando a funcão Mix_VolumeChunk.

Implementação

A formula é essa: garantindo que a musica esteja renderizada em multiplos arquivos (canal por canal), Music carrega cada um desses arquivos como chunks, seta volumes, e chama Play. Nada muito emocionante até agora. A mágica acontece quando voce modifica os volumes dos canais dinamicamente. Para tornar isso viável, sugerimos o seguinte:

1. Adicione a Music um `unsigned int channels`, que guarda a quantidade de canais que temos, crie um `std::vector<int>` para guardar o índice dos canais que a música esta tocando (já que chamaremos Mix_PlayChannel com canal -1).

2. Faça um `typedef std::vector<unsigned int> volumestate` - esse tipo representa uma *mix* da musica (um conjunto de volumes, um para cada canal, que pode ser mapeado para mudar como a musica soa). Talvez voce queira sobrecarregar algum operador no processo (?).
    - Alternativamente, voce pode preferir fazer um `typedef int volumestates[x]` (ou `typedef struct vs { std::vector<int> mix } volumestate`, segundo essa resposta no stackoverflow), caso saiba previamente o número fixo de camadas de que a música consiste. Music guarda um `std::vector<volumestate>` que deve ser inicializado no construtor (bear with me).
3. Crie um método Music::SetVolumes(unsigned int), que acessa seu array de volumes, recupera uma mix (caso exista), e aplica aos seus proprios canais (por isso um array de array) usando Mix_VolumeChannel.

Agora experimente criar dois volumestates: um com a *mix* padrão da música para a fase, outro idêntico porém com um canal mutado (volume 0). Torna-se extremamente conveniente, criar atalhos no teclado (ctrl-shift-1, ctrl-shift-2, por exemplo) para alternar entre eles - voce pode passar o proprio numero da tecla para o metodo. Isso pode ser feito acessando InputManager, seja na própria Music ou em GameState.

Caso tudo tenha ocorrido bem, ao apertar os atalhos, voce estaria alternando entre dois volumes, de forma **brusca** (como o exemplo do mario e yoshi). Falta agora definir eventos de jogo que disparem essa transição (**triggers**).

Se quiser **viajar mais ainda na maionese**, você pode construir uma função que incrementa o volume de um instrumento específico, e chamá-la a cada inimigo derrotado ou moeda adquirida. Isso é feito em De Blob, onde cada prédio que o player tinge traz mais à tona um dos instrumentos da música. Naturalmente, por conta do custo, isso é ideia para uma mecânica-chave do jogo - não mais um recurso periférico.

Sinergia

Esse mecanismo pode ser ainda aprimorado usando:

- Lerping: Utilizando funcoes de lerping, podemos tornar **gradual** a transicao entre volumes (*fades*), que gera um efeito aprimorado de imersão. Para isso:
- Adicionamos um `Timer fade` à nossa classe, junto com um float indicando o tempo de fade desejado (que pode fazer parte da struct volumestate, caso seja a opção escolhida - com isso, cada padrão de volume pode ter seu próprio tempo de fade).
- Botamos o timer pra rodar.
- Chamamos alguma função de Lerping com o valor do nosso timer dividido pelo tempo total, para interpolarmos entre 0 e 1. Multiplicamos esse valor pela diferença entre os volumes inicial e final, e, finalmente, somamos o resultado ao volume inicial.
- Note que aqui, novamente, a própria forma de interpolação pode fazer parte da struct, mas a utilidade disso é questionável.
- Arquivos de Configuracao: Caso exista uma organizacao das fases e demais recursos de jogo em termo de arquivos externos, convem bastante adicionar, junto ao arquivo da fase, um catalogo de padroes de volumes.
- Triggers: Com um mecanismo de triggers, podemos automatizar a transicao de volume em termos do cenario. Funciona bem com Eventos