

# Table of Contents

16 Games in C++ .....	2
Configuração do Ambiente .....	5
Requisitos do Sistema .....	8
Instalação do SFML .....	13
Instalação do CMake .....	21
Configuração da IDE .....	30
Configuração Final do Ambiente .....	40
Compilação .....	49
Guia de Compilação .....	59
Compilar e Executar Jogos .....	69
Executando os Jogos .....	81
Estrutura do Projeto .....	92
Jogos .....	102
Tetris .....	109
Doodle Jump .....	122
Arkanoid .....	145
Snake .....	161
Minesweeper .....	176
Troubleshooting .....	188

# 16 Games in C++

Bem-vindo à documentação do projeto **16 Games in C++!** 🎮

Este projeto é uma coleção de 16 jogos clássicos implementados em C++ utilizando a biblioteca SFML (Simple and Fast Multimedia Library). Cada jogo é completo e funcional, perfeito para aprender conceitos de programação de jogos e C++.

## Sobre o Projeto

O projeto **16 Games in C++** foi criado com o objetivo de demonstrar diferentes técnicas de programação de jogos, desde jogos simples como Tetris até jogos mais complexos como Chess e Asteroids. Todos os jogos utilizam:

- **C++17** como linguagem de programação
- **SFML 2.5+** para gráficos, áudio e entrada
- **CMake** para build system
- Estrutura modular e código limpo

## Jogos Incluídos

1. **Tetris** - O clássico jogo de blocos
2. **Doodle Jump** - Pule o mais alto possível
3. **Arkanoid** - Quebre todos os blocos
4. **Snake** - A serpente clássica
5. **Minesweeper** - Campo minado
6. **Fifteen Puzzle** - Quebra-cabeça numérico
7. **Racing (Top Down)** - Corrida vista de cima
8. **Outrun** - Corrida em perspectiva
9. **Xonix** - Conquiste território
10. **Bejeweled** - Combine joias
11. **NetWalk** - Conecte os canos
12. **Mahjong Solitaire** - Paciência mahjong

13. **Tron** - Batalha de luz
14. **Chess** - Xadrez completo
15. **Volleyball** - Vôlei arcade
16. **Asteroids** - Destrua os asteroides



## Início Rápido

Para começar rapidamente:

1. Configure o ambiente ([Configuração do Ambiente](#))
2. Clone o repositório
3. Execute o script de setup: `./setup.sh`
4. Compile e jogue: `make all_games`



## Estrutura da Documentação

Esta documentação está organizada nas seguintes seções:

- **Configuração do Ambiente** ([Configuração do Ambiente](#)) - Como instalar e configurar tudo que você precisa
- **Compilação** ([Compilação](#)) - Guias de build e execução
- **Estrutura do Projeto** ([Estrutura do Projeto](#)) - Como o código está organizado
- **Jogos** ([Jogos](#)) - Documentação específica de cada jogo
- **Troubleshooting** ([Troubleshooting](#)) - Soluções para problemas comuns



## Requisitos do Sistema

- **Sistema Operacional:** Linux, macOS ou Windows
- **Compilador:** GCC 7+ ou Clang 6+ com suporte a C++17
- **CMake:** 3.10 ou superior
- **SFML:** 2.5 ou superior



## Contribuindo

Este projeto é open source! Você pode:

- Reportar bugs
- Sugerir melhorias
- Contribuir com código
- Melhorar a documentação



## Licença

Este projeto é distribuído sob a licença MIT. Veja o arquivo LICENSE para mais detalhes.

**Pronto para começar?** Vá para Configuração do Ambiente ([Configuração do Ambiente](#)) e configure seu sistema para executar os jogos!

# Configuração do Ambiente

Este guia te ajudará a configurar completamente o ambiente de desenvolvimento para executar os 16 jogos em C++. 🛠️

## Visão Geral

Para executar este projeto, você precisará instalar e configurar:

1. **Requisitos do Sistema** ([Requisitos do Sistema](#)) - Verificar compatibilidade
2. **SFML** ([Instalação do SFML](#)) - Biblioteca gráfica principal
3. **CMake** ([Instalação do CMake](#)) - Sistema de build
4. **IDE** ([Configuração da IDE](#)) - Ambiente de desenvolvimento (opcional)
5. **Configuração Final** ([Configuração Final do Ambiente](#)) - Testes e validação







## Setup Automático (Recomendado)

O projeto inclui um script de configuração automática que verifica e configura tudo para você:

```
# Clone o repositório
git clone <repository-url>
cd 16Games-in-Cpp

# Execute o script de setup
chmod +x setup.sh
./setup.sh
```

O script `setup.sh` irá:

-  Verificar se o SFML está instalado
-  Verificar se o CMake está disponível
-  Verificar se há um compilador C++ válido
-  Configurar o projeto com CMake
-  Compilar um jogo de teste
-  Mostrar instruções de uso

## Setup Manual

Se preferir instalar manualmente ou se o script automático falhar, siga os guias específicos:

### 1. Requisitos do Sistema

Primeiro, verifique se seu sistema atende aos requisitos mínimos ([Requisitos do Sistema](#)).

### 2. Instalar SFML

A biblioteca SFML é essencial para gráficos, áudio e entrada. Veja o guia de instalação do SFML ([Instalação do SFML](#)).

### 3. Instalar CMake

O CMake é usado para gerenciar o build do projeto. Veja o guia de instalação do CMake ([Instalação do CMake](#)).

### 4. Configurar IDE (Opcional)

Para uma melhor experiência de desenvolvimento, configure sua IDE preferida ([Configuração da IDE](#)).

### 5. Configuração Final

Complete a configuração do ambiente ([Configuração Final do Ambiente](#)) e teste tudo.

## Verificação Rápida

Após a instalação, execute estes comandos para verificar se tudo está funcionando:

```
# Verificar SFML
pkg-config --exists sfml-all && echo "✅ SFML OK" || echo "❌ SFML não encontrado"

# Verificar CMake
cmake --version && echo "✅ CMake OK" || echo "❌ CMake não encontrado"

# Verificar compilador
g++ --version && echo "✅ G++ OK" || echo "❌ G++ não encontrado"
```

## Próximos Passos

Após configurar o ambiente:

1. Vá para Compilação ([Compilação](#)) para aprender a compilar os jogos
2. Ou vá direto para Build and Run ([Compilar e Executar Jogos](#)) para executar rapidamente


## ! Problemas?

Se encontrar algum problema durante a configuração:

1. Consulte o Troubleshooting ([Troubleshooting](#))
2. Verifique se seguiu todos os passos corretamente
3. Confirme se seu sistema atende aos requisitos mínimos

**Dica:** O script `setup.sh` é a forma mais rápida e confiável de configurar o ambiente. Use-o sempre que possível!

# Requisitos do Sistema

Antes de instalar o projeto 16 Games in C++, verifique se seu sistema atende aos requisitos mínimos. 

## Sistemas Operacionais Suportados

### Linux (Recomendado)

- **Ubuntu:** 18.04 LTS ou superior
- **Debian:** 10 (Buster) ou superior
- **Fedora:** 30 ou superior
- **Arch Linux:** Versão atual
- **openSUSE:** Leap 15.2 ou superior
- **CentOS/RHEL:** 8 ou superior

### macOS

- **macOS:** 10.14 (Mojave) ou superior
- **Xcode:** 10 ou superior (para compilador)

### Windows

- **Windows:** 10 ou superior
- **Visual Studio:** 2019 ou superior
- **MinGW-w64:** Como alternativa ao Visual Studio

## Compilador C++

### Requisitos Mínimos

- **Suporte ao C++17:** Obrigatório
- **Versões mínimas:**
  - **GCC:** 7.0 ou superior
  - **Clang:** 6.0 ou superior



- **MSVC:** Visual Studio 2019 ou superior

## Verificação do Compilador

### Linux/macOS

```
# Verificar GCC
g++ --version
# Deve mostrar versão 7.0+

# Verificar Clang (se disponível)
clang++ --version
# Deve mostrar versão 6.0+
```

### Windows

```
# Visual Studio
cl
# Deve mostrar MSVC 19.20+

# MinGW
g++ --version
# Deve mostrar versão 7.0+
```

## Dependências Principais

### CMake

- **Versão mínima:** 3.10
- **Recomendada:** 3.16 ou superior

### SFML

- **Versão mínima:** 2.5.0
- **Recomendada:** 2.5.1 ou superior
- **Módulos necessários:**
  - sfml-system
  - sfml-window
  - sfml-graphics

- sfml-audio
- sfml-network (opcional)

## Bibliotecas do Sistema (Linux)

```
# Ubuntu/Debian
sudo apt-get install libsFML-dev cmake build-essential

# Fedora
sudo dnf install SFML-devel cmake gcc-c++

# Arch Linux
sudo pacman -S sfml cmake gcc
```

## Espaço em Disco

### Requisitos de Espaço

- **Código fonte:** ~50 MB
- **Dependências:** ~200 MB (SFML + CMake)
- **Build completo:** ~100 MB
- **Total recomendado:** ~500 MB livre

### Estrutura de Diretórios

```
16Games-in-Cpp/
├── build/           # ~100 MB (arquivos compilados)
├── Writerside/      # ~10 MB (documentação)
├── games/           # ~40 MB (código fonte dos jogos)
└── assets/          # ~5 MB (imagens, sons, fonts)
```

## Hardware Recomendado

### Mínimo

- **CPU:** Dual-core 2.0 GHz
- **RAM:** 4 GB
- **GPU:** Integrada com OpenGL 2.1

- **Resolução:** 1024x768

## Recomendado

- **CPU:** Quad-core 2.5 GHz ou superior
- **RAM:** 8 GB ou superior
- **GPU:** Dedicada com OpenGL 3.3+
- **Resolução:** 1920x1080 ou superior








## Verificação Automática

Use o script de verificação incluído no projeto:

```
# Executar verificação de requisitos
./setup.sh

# Ou verificar manualmente
./check_requirements.sh # Se disponível
```

O script verificará:

-  Sistema operacional compatível
-  Compilador C++ com suporte ao C++17
-  CMake versão adequada
-  SFML instalado e funcional
-  Espaço em disco suficiente



## Problemas Comuns

### Compilador Muito Antigo

```
# Ubuntu 18.04 - instalar GCC mais novo
sudo apt update
sudo apt install gcc-9 g++-9
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-9 90
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-9 90
```

### SFML Não Encontrado

```
# Verificar instalação
pkg-config --exists sfml-all
pkg-config --modversion sfml-all

# Se não encontrado, reinstalar
sudo apt-get install --reinstall libsFML-dev
```

## CMake Muito Antigo

```
# Ubuntu - instalar versão mais nova
wget https://github.com/Kitware/CMake/releases/download/v3.20.0/cmake-3.20.0-
Linux-x86_64.sh
chmod +x cmake-3.20.0-Linux-x86_64.sh
sudo ./cmake-3.20.0-Linux-x86_64.sh --prefix=/usr/local --skip-license
```

## Próximos Passos

Se seu sistema atende aos requisitos:

1. Prosiga para Instalação do SFML ([Instalação do SFML](#))
2. Ou execute o setup automático ([Configuração do Ambiente](#))

Se encontrou problemas:

1. Consulte o Troubleshooting ([Troubleshooting](#))
2. Atualize seu sistema e tente novamente

**Dica:** Na dúvida, execute `./setup.sh` - o script detectará automaticamente se há algum problema com os requisitos!

# Instalação do SFML

O SFML (Simple and Fast Multimedia Library) é a biblioteca principal usada por todos os 16 jogos. Este guia mostra como instalá-la em diferentes sistemas operacionais. 🎮



## O que é SFML?

SFML é uma biblioteca C++ que fornece:

- **Gráficos 2D** - Desenho de sprites, formas, texto
- **Áudio** - Reprodução de sons e música
- **Janelas** - Criação e gerenciamento de janelas
- **Eventos** - Captura de teclado, mouse e joystick
- **Rede** - Comunicação TCP/UDP (não usado neste projeto)



## Linux

### Ubuntu/Debian

```
# Atualizar repositórios
sudo apt update

# Instalar SFML e dependências de desenvolvimento
sudo apt install libsFML-dev

# Verificar instalação
pkg-config --modversion sfml-all
```

### Fedora/CentOS/RHEL

```
# Fedora
sudo dnf install SFML-devel

# CentOS/RHEL (com EPEL)
sudo yum install epel-release
sudo yum install SFML-devel

# Verificar instalação
pkg-config --modversion sfml-all
```

## Arch Linux

```
# Instalar SFML
sudo pacman -S sfml

# Verificar instalação
pkg-config --modversion sfml-all
```

## openSUSE

```
# openSUSE Leap/Tumbleweed
sudo zypper install libsFML2-devel

# Verificar instalação
pkg-config --modversion sfml-all
```

## macOS

### Usando Homebrew (Recomendado)

```
# Instalar Homebrew se não tiver
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

# Instalar SFML
brew install sfml

# Verificar instalação
pkg-config --modversion sfml-all
```

### Usando MacPorts

```
# Instalar SFML
sudo port install sfml

# Verificar instalação
pkg-config --modversion sfml-all
```

## Instalação Manual

1. Baixe SFML do site oficial (<https://www.sfml-dev.org/download.php>)
2. Extraia para `/usr/local/`

3. Configure as variáveis de ambiente:

```
export SFML_ROOT=/usr/local/SFML-2.5.1
export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$SFML_ROOT/lib/pkgconfig
```

## Windows

### Visual Studio (Recomendado)

#### 1. Baixar SFML

- Acesse SFML Downloads (<https://www.sfml-dev.org/download.php>)
- Baixe a versão para Visual Studio (ex: SFML-2.5.1-windows-vc15-64-bit.zip)

#### 2. Extrair e Configurar

```
# Extrair para C:\SFML
# Estrutura deve ficar:
C:\SFML\
├─ bin\
├─ include\
├─ lib\
└─ examples\
```

#### 3. Configurar Projeto Visual Studio

No seu projeto CMake ou Visual Studio:

```
set(SFML_ROOT "C:/SFML")
find_package(SFML 2.5 COMPONENTS system window graphics audio REQUIRED)
```

### MinGW-w64

```
# Usando MSYS2
pacman -S mingw-w64-x86_64-sfml

# Verificar instalação
pkg-config --modversion sfml-all
```

### vcpkg (Alternativa)

```
# Instalar vcpkg
git clone https://github.com/Microsoft/vcpkg.git
```

```
cd vcpkg
.\bootstrap-vcpkg.bat

# Instalar SFML
.\vcpkg install sfml:x64-windows
```

## Compilação desde o Código Fonte

Se os pacotes pré-compilados não funcionarem, compile o SFML:

### Linux/macOS

```
# Baixar código fonte
git clone https://github.com/SFML/SFML.git
cd SFML

# Criar diretório de build
mkdir build
cd build

# Configurar com CMake
cmake .. -DCMAKE_INSTALL_PREFIX=/usr/local

# Compilar
make -j$(nproc)

# Instalar (pode precisar de sudo)
sudo make install

# Atualizar cache de bibliotecas (Linux)
sudo ldconfig
```

### Windows (Visual Studio)

```
# No Developer Command Prompt
git clone https://github.com/SFML/SFML.git
cd SFML
mkdir build
cd build

cmake .. -G "Visual Studio 16 2019" -A x64
cmake --build . --config Release
cmake --install . --prefix C:\SFML
```



## ✓ Verificação da Instalação

### Teste Básico

Crie um arquivo `test_sfml.cpp`:

```
#include <SFML/Graphics.hpp>
#include <iostream>

int main() {
    // Tentar criar uma janela
    sf::RenderWindow window(sf::VideoMode(200, 200), "SFML Test");

    if (!window.isOpen()) {
        std::cout << "✗ Erro ao criar janela SFML\n";
        return 1;
    }

    std::cout << "✓ SFML funcionando corretamente!\n";

    // Fechar imediatamente
    window.close();
    return 0;
}
```

### Compilar Teste

```
# Linux/macOS
g++ -o test_sfml test_sfml.cpp $(pkg-config --cflags --libs sfml-all)

# Windows (MinGW)
g++ -o test_sfml.exe test_sfml.cpp -lsfml-graphics -lsfml-window -lsfml-system

# Executar
./test_sfml
```

### Usando CMake (Recomendado)

Crie um `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.10)
project(TestSFML)

set(CMAKE_CXX_STANDARD 17)
```

```
find_package(PkgConfig REQUIRED)
pkg_check_modules(SFML REQUIRED sfml-all>=2.5)

add_executable(test_sfml test_sfml.cpp)
target_link_libraries(test_sfml ${SFML_LIBRARIES})
target_compile_options(test_sfml PRIVATE ${SFML_CFLAGS})
```

```
mkdir build
cd build
cmake ..
make
./test_sfml
```

## Verificação com pkg-config

```
# Verificar se SFML está disponível
pkg-config --exists sfml-all
echo $? # Deve retornar 0

# Ver versão instalada
pkg-config --modversion sfml-all

# Ver flags de compilação
pkg-config --cflags sfml-all

# Ver flags de link
pkg-config --libs sfml-all

# Informações completas
pkg-config --cflags --libs sfml-all
```

## Solução de Problemas

### Erro: "sfml-all not found"

```
# Verificar se pkg-config está instalado
pkg-config --version

# Listar todos os pacotes disponíveis
pkg-config --list-all | grep -i sfml
```

```
# Verificar caminhos do pkg-config
echo $PKG_CONFIG_PATH

# Linux: SFML pode estar em /usr/lib/pkgconfig/
ls /usr/lib/pkgconfig/ | grep sfml
```

## Erro de Linking

```
# Verificar se bibliotecas estão instaladas
ldconfig -p | grep sfml # Linux
find /usr -name "*sfml*" 2>/dev/null # Linux/macOS

# Windows: verificar PATH
echo %PATH%
```

## Versão Incompatível

```
# Desinstalar versão antiga
sudo apt remove libsFML-dev # Ubuntu
brew uninstall sfml         # macOS

# Limpar cache
sudo apt autoremove
brew cleanup

# Reinstalar versão correta
sudo apt install libsFML-dev
brew install sfml
```

## Próximos Passos

Após instalar o SFML com sucesso:

1. Prosiga para Instalação do CMake ([Instalação do CMake](#))
2. Ou volte para Configuração do Ambiente ([Configuração do Ambiente](#))

Se ainda tiver problemas:

1. Consulte o Troubleshooting ([Troubleshooting](#))
2. Execute o script automático: `./setup.sh`

**Dica:** O comando `pkg-config --cflags --libs sfml-all` mostra exatamente como compilar com SFML. Guarde essa informação!

# Instalação do CMake

O CMake é o sistema de build usado pelo projeto 16 Games in C++. Este guia mostra como instalá-lo em diferentes sistemas operacionais. 🛠️



## O que é CMake?

CMake é uma ferramenta multiplataforma que:

- **Gera arquivos de build** para diferentes sistemas (Make, Visual Studio, Xcode)
- **Gerencia dependências** de forma automática
- **Configura compilação** com diferentes opções
- **Simplifica builds** em múltiplas plataformas



## Versões Suportadas

- **Mínima:** 3.10
- **Recomendada:** 3.16 ou superior
- **Ideal:** 3.20 ou superior (melhor suporte ao C++17)



## Linux

### Ubuntu/Debian

#### Versão dos Repositórios (Mais Simples)

```
# Atualizar repositórios
sudo apt update

# Instalar CMake
sudo apt install cmake

# Verificar versão
cmake --version
```

#### Versão Mais Recente (Recomendado)

```
# Adicionar repositório Kitware (oficial)
wget -O - https://apt.kitware.com/keys/kitware-archive-latest.asc 2>/dev/null
| gpg --dearmor - | sudo tee /etc/apt/trusted.gpg.d/kitware.gpg >/dev/null
```

```
# Ubuntu 20.04
sudo apt-add-repository 'deb https://apt.kitware.com/ubuntu/ focal main'

# Ubuntu 22.04
sudo apt-add-repository 'deb https://apt.kitware.com/ubuntu/ jammy main'

# Instalar
sudo apt update
sudo apt install cmake

# Verificar versão
cmake --version
```

## Fedora/CentOS/RHEL

```
# Fedora
sudo dnf install cmake

# CentOS/RHEL 8+
sudo dnf install cmake

# Verificar versão
cmake --version
```

## Arch Linux

```
# Instalar CMake
sudo pacman -S cmake

# Verificar versão
cmake --version
```

## openSUSE

```
# openSUSE Leap/Tumbleweed
sudo zypper install cmake

# Verificar versão
cmake --version
```



## Usando Homebrew (Recomendado)

```
# Instalar Homebrew se não tiver
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

# Instalar CMake
brew install cmake

# Verificar versão
cmake --version
```

## Usando MacPorts

```
# Instalar CMake
sudo port install cmake

# Verificar versão
cmake --version
```

## Instalação Manual

```
# Baixar binário
curl -L -O https://github.com/Kitware/CMake/releases/download/v3.25.1/cmake-
3.25.1-macos-universal.tar.gz

# Extrair
tar -xzf cmake-3.25.1-macos-universal.tar.gz

# Mover para aplicações
sudo mv cmake-3.25.1-macos-universal/CMake.app /Applications/

# Criar link simbólico
sudo ln -s /Applications/CMake.app/Contents/bin/cmake /usr/local/bin/cmake

# Verificar versão
cmake --version
```

## Windows

### Usando winget (Windows 10/11)

```
# Instalar CMake
winget install Kitware.CMake

# Verificar versão (reiniciar terminal)
cmake --version
```

## Usando Chocolatey

```
# Instalar Chocolatey se não tiver
# Ver: https://chocolatey.org/install

# Instalar CMake
choco install cmake

# Verificar versão
cmake --version
```

## Instalação Manual

1. Acesse CMake Downloads (<https://cmake.org/download/>)
2. Baixe o instalador Windows (.msi)
3. Execute o instalador
4. **Importante:** Marque "Add CMake to system PATH"
5. Verificar instalação:

```
cmake --version
```

## Visual Studio

O Visual Studio 2019/2022 já inclui CMake:

- Está em C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\IDE\CommonExtensions\Microsoft\CMake\CMake\bin\cmake.exe
- Pode ser necessário adicionar ao PATH manualmente



## Compilação desde o Código Fonte

Se precisar de uma versão específica ou os pacotes não funcionarem:

## Linux/macOS



```
# Baixar código fonte
wget https://github.com/Kitware/CMake/releases/download/v3.25.1/cmake-3.25.1.tar.gz
tar -xzf cmake-3.25.1.tar.gz
cd cmake-3.25.1

# Configurar (bootstrap)
./bootstrap --prefix=/usr/local

# Compilar
make -j$(nproc)

# Instalar
sudo make install

# Verificar versão
cmake --version
```

### Compilação Rápida (Sem Bootstrap)

```
# Se já tiver CMake instalado (versão mais antiga)
mkdir build
cd build
cmake ..
make -j$(nproc)
sudo make install
```

## Verificação da Instalação

### Teste Básico

```
# Verificar versão
cmake --version

# Deve mostrar algo como:
# cmake version 3.25.1
```

### Teste de Funcionalidade

Crie um projeto de teste:

**CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.10)
project(TestCMake)

set(CMAKE_CXX_STANDARD 17)

add_executable(test_cmake main.cpp)
```

### main.cpp

```
#include <iostream>
int main() {
    std::cout << "✅ CMake funcionando!\n";
    return 0;
}
```

### Compilar e Testar

```
mkdir build
cd build
cmake ..
make # ou cmake --build .
./test_cmake
```

## Configurações Úteis

### Configurar CMake Globalmente

```
# Criar arquivo de configuração
mkdir -p ~/.cmake
cat > ~/.cmake/CMakeCache.txt << EOF
CMAKE_BUILD_TYPE:STRING=Release
CMAKE_CXX_STANDARD:STRING=17
CMAKE_EXPORT_COMPILE_COMMANDS:BOOL=ON
EOF
```

### Aliases Úteis

Adicione ao seu `.bashrc` ou `.zshrc`:

```
# Aliases para CMake
alias cb='cmake --build .'
alias cc='cmake ..'
```

```
alias cr='cmake .. && make'
alias ctest='ctest --output-on-failure'
```

## Ferramentas Complementares

### CMake GUI (Opcional)

```
# Linux
sudo apt install cmake-qt-gui # Ubuntu/Debian
sudo dnf install cmake-gui    # Fedora

# macOS
brew install --cask cmake

# Windows - incluído no instalador
```

### ccmake (Curses Interface)

```
# Linux - geralmente incluído
ccmake ..

# Navegar com setas, Enter para editar, 'c' para configurar, 'g' para gerar
```

## Verificação de Recursos

```
# Ver todas as opções do CMake
cmake --help

# Ver geradores disponíveis
cmake --help | grep "Generators"

# Informações do sistema
cmake --system-information

# Verificar variáveis disponíveis
cmake --help-variable-list | head -20
```

## Solução de Problemas

### CMake Não Encontrado

```
# Verificar PATH
echo $PATH

# Linux/macOS - onde está o CMake?
which cmake
whereis cmake

# Windows
where cmake
```

## Versão Muito Antiga

```
# Desinstalar versão antiga
sudo apt remove cmake      # Ubuntu
brew uninstall cmake        # macOS

# Instalar versão mais recente (ver seções acima)
```

## Erro de Permissão

```
# Linux - problemas de permissão
sudo chown -R $USER:$USER ~/.cmake
sudo chmod -R 755 ~/.cmake
```

## Conflito de Versões

```
# Ver todas as versões instaladas
ls /usr/bin/cmake*
ls /usr/local/bin/cmake*

# Usar versão específica
/usr/local/bin/cmake --version
```

## CMake no Projeto 16 Games

No nosso projeto, o CMake:

- Detecta SFML automaticamente
- Configura compilação para C++17
- Gerencia assets (copia imagens, sons, etc.)

- Cria **targets** para cada jogo
- Oferece comandos de build e execução

## Comandos Principais

```
# Configurar projeto
cmake ..

# Compilar todos os jogos
make all_games

# Compilar jogo específico
make tetris

# Executar jogo
make run_tetris
```

## Próximos Passos

Após instalar o CMake com sucesso:

1. Prossiga para Configuração da IDE ([Configuração da IDE](#))
2. Ou pule para Configuração Final ([Configuração Final do Ambiente](#))
3. Ou volte para Configuração do Ambiente ([Configuração do Ambiente](#))

Se tiver problemas:

1. Consulte o Troubleshooting ([Troubleshooting](#))
2. Execute o script automático: `./setup.sh`

**Dica:** Use sempre `cmake --build .` em vez de `make` para máxima compatibilidade entre plataformas!

# Configuração da IDE

Embora você possa compilar e executar os jogos apenas com terminal, usar uma IDE melhora significativamente a experiência de desenvolvimento. Este guia mostra como configurar diferentes IDEs para o projeto. 🖥️

## 🎯 IDEs Recomendadas

### Para Iniciantes

- Visual Studio Code (["vs Visual Studio Code" in "Configuração da IDE"](#)) - Leve, extensível, gratuito
- Code::Blocks (["Configuração da IDE"](#)) - Simples, focado em C++

### Para Desenvolvedores Experientes

- CLion (["CLion" in "Configuração da IDE"](#)) - Profissional, JetBrains
- Visual Studio (["Visual Studio" in "Configuração da IDE"](#)) - Windows, Microsoft
- Qt Creator (["Qt Creator" in "Configuração da IDE"](#)) - Multiplataforma, excelente CMake

### Editores Avançados

- Vim/Neovim (["Vim/Neovim" in "Configuração da IDE"](#)) - Para usuários experientes
- Emacs (["Emacs" in "Configuração da IDE"](#)) - Altamente customizável

## vs Visual Studio Code

### Instalação

```
# Linux (Ubuntu/Debian)
sudo apt update
sudo apt install code

# macOS
brew install --cask visual-studio-code

# Windows - baixar do site oficial
# https://code.visualstudio.com/
```

### Extensões Essenciais

Instale essas extensões (Ctrl+Shift+X):

1. **C/C++** (Microsoft) - Suporte básico ao C++
2. **CMake Tools** (Microsoft) - Integração com CMake
3. **CMake** (twxs) - Syntax highlighting para CMake
4. **C++ Intellisense** (austin) - Melhor autocomplete
5. **GitLens** (GitKraken) - Integração Git avançada

## Configuração do Projeto

### 1. Abrir Projeto

```
cd 16Games-in-Cpp
code .
```

### 2. Configurar CMake Tools

Pressione **Ctrl+Shift+P** e digite "CMake: Configure":

```
// .vscode/settings.json
{
  "cmake.configureSettings": {
    "CMAKE_BUILD_TYPE": "Debug",
    "CMAKE_CXX_STANDARD": "17"
  },
  "cmake.buildDirectory": "${workspaceFolder}/build",
  "cmake.generator": "Unix Makefiles",
  "C_Cpp.default.configurationProvider": "ms-vscode.cmake-tools"
}
```

### 3. Configurar Tarefas de Build

```
// .vscode/tasks.json
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "CMake Configure",
      "type": "shell",
      "command": "cmake",
      "args": [ "..", "-DCMAKE_BUILD_TYPE=Debug" ],
      "options": {
        "cwd": "${workspaceFolder}/build"
      }
    }
  ]
}
```

```

    },
    "group": "build"
  },
  {
    "label": "Build All Games",
    "type": "shell",
    "command": "make",
    "args": ["all_games"],
    "options": {
      "cwd": "${workspaceFolder}/build"
    },
    "group": {
      "kind": "build",
      "isDefault": true
    }
  }
]
}

```

#### 4. Configurar Launch (Debug)

```

// .vscode/launch.json
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Debug Tetris",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceFolder}/build/games/tetris/tetris",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}/build/games/tetris",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        }
      ]
    }
  ]
}

```



```
]
}
```



## Configuração

1. **Abrir Projeto:** File → Open → Selecionar pasta do projeto
2. **CMake será detectado automaticamente**
3. **Configurar Build Types:**
  - File → Settings → Build, Execution, Deployment → CMake
  - Adicionar Debug e Release profiles

## Configurações Recomendadas

### CMake Settings

```
Debug Profile:
- Build type: Debug
- CMake options: -DCMAKE_BUILD_TYPE=Debug
- Build directory: build/Debug

Release Profile:
- Build type: Release
- CMake options: -DCMAKE_BUILD_TYPE=Release
- Build directory: build/Release
```

### Run Configurations

CLion criará automaticamente configurações para cada jogo. Você pode:

- **Executar:** Ctrl+Shift+F10
- **Debug:** Shift+F9
- **Configurar:** Run → Edit Configurations

## Plugins Úteis

- **Valgrind Memcheck** - Detecção de vazamentos
- **CPU Usage Indicator** - Monitor de performance

- Rainbow Brackets – Melhor visualização



## Visual Studio (Windows)

### Configuração

1. **Instalar:** Visual Studio Community (gratuito)
2. **Workloads:** Marcar "Desktop development with C++"
3. **Componentes:** Incluir CMake tools

### Abrir Projeto

1. File → Open → CMake...
2. Selecionar `CMakeLists.txt` na raiz do projeto
3. Visual Studio configurará automaticamente

### Configurações CMake

```
// CMakeSettings.json (criado automaticamente)
{
  "configurations": [
    {
      "name": "x64-Debug",
      "generator": "Ninja",
      "configurationType": "Debug",
      "buildRoot": "${projectDir}\\build\\${name}",
      "installRoot": "${projectDir}\\install\\${name}",
      "cmakeCommandArgs": "",
      "buildCommandArgs": "",
      "ctestCommandArgs": ""
    }
  ]
}
```



## Qt Creator

### Instalação

```
# Linux
sudo apt install qtcreator
```

```
# macOS
brew install --cask qt-creator

# Windows - baixar do site Qt
```

## Configuração

1. **Abrir:** File → Open File or Project → CMakeLists.txt
2. **Kit Selection:** Escolher kit apropriado (GCC/Clang)
3. **Build Directory:** Configurar diretório de build

## Vantagens

- Excelente suporte CMake
- Debugger integrado
- Profiler built-in
- Git integration



## Code::Blocks

### Instalação

```
# Linux
sudo apt install codeblocks

# Windows/macOS - baixar do site oficial
```

## Importar Projeto CMake

1. File → Import → Import CMake Project
2. Selecionar CMakeLists.txt
3. Configurar compilador e opções



## Vim/Neovim

### Plugins Recomendados (com vim-plug)

```

" .vimrc ou init.vim
call plug#begin()

" LSP e Completions
Plug 'neoclide/coc.nvim', {'branch': 'release'}
Plug 'clangd/coc-clangd'

" CMake
Plug 'cdelledonne/vim-cmake'
Plug 'vhdirk/vim-cmake'

" Syntax
Plug 'vim-syntastic/syntastic'
Plug 'octol/vim-cpp-enhanced-highlight'

" Git
Plug 'tpope/vim-fugitive'

call plug#end()

```

## Configuração CMake

```

" Keybindings para CMake
nnoremap <leader>cc :CMake<CR>
nnoremap <leader>cb :CMakeBuild<CR>
nnoremap <leader>cr :CMakeRun<CR>

```

## Emacs

### Configuração com use-package

```

;; init.el
(use-package cmake-mode
  :ensure t)

(use-package cmake-ide
  :ensure t
  :config
  (cmake-ide-setup))

(use-package company
  :ensure t
  :config

```

```
(global-company-mode))

(use-package lsp-mode
  :ensure t
  :init
  (setq lsp-keymap-prefix "C-c l")
  :hook ((c++-mode . lsp))
  :commands lsp)
```

## Configuração Geral para Todas IDEs

### Formatação de Código

Crie `.clang-format` na raiz do projeto:

```
BasedOnStyle: Google
IndentWidth: 4
TabWidth: 4
UseTab: Never
ColumnLimit: 100
AllowShortFunctionsOnASingleLine: Empty
```

### EditorConfig

Crie `.editorconfig`:

```
root = true

[*]
charset = utf-8
end_of_line = lf
insert_final_newline = true
trim_trailing_whitespace = true

[*.{cpp,hpp,h}]
indent_style = space
indent_size = 4

[CMakeLists.txt,*.cmake]
indent_style = space
indent_size = 2
```

### Git Ignore para IDEs

Adicione ao `.gitignore`:

```
# IDEs
.vscode/
.idea/
*.user
*.pro.user*
.cproject
.project
.settings/
*.cbp
```

## Comandos Úteis por IDE

### VS Code

- Build: `Ctrl+Shift+P` → "CMake: Build"
- Run: `Ctrl+F5`
- Debug: `F5`
- Terminal: `Ctrl+`` (backtick)

### CLion

- Build: `Ctrl+F9`
- Run: `Shift+F10`
- Debug: `Shift+F9`
- Terminal: `Alt+F12`

### Visual Studio

- Build: `Ctrl+Shift+B`
- Run: `Ctrl+F5`
- Debug: `F5`
- Terminal: View → Terminal

## Dicas Gerais

### Performance

- Use SSD para diretório de build
- Configure RAM adequada (mínimo 8GB)
- Feche programas desnecessários durante compilação

## Produtividade

- Configure atalhos personalizados
- Use templates para novos arquivos
- Configure snippets para código comum
- Use git integration

## Debug

- Configure breakpoints nos pontos críticos
- Use watch variables para monitorar estado
- Ative otimização apenas em release



## Próximos Passos

Após configurar sua IDE:

1. Prossiga para Configuração Final ([Configuração Final do Ambiente](#))
2. Ou teste com Build and Run ([Compilar e Executar Jogos](#))
3. Volte para Configuração do Ambiente ([Configuração do Ambiente](#))

Se tiver problemas:

1. Consulte o Troubleshooting ([Troubleshooting](#))
2. Verifique se SFML e CMake estão funcionando

**Dica:** Se é iniciante, comece com VS Code. Se quer máxima produtividade, use CLion. Para projetos simples, Code::Blocks é suficiente!

# Configuração Final do Ambiente

Este é o último passo da configuração do ambiente. Aqui vamos integrar tudo o que foi instalado e fazer os testes finais. 🏁

## Checklist Pré-Requisitos

Antes de continuar, confirme que você já tem:

- ☒ Sistema compatível ([Requisitos do Sistema](#))
- ☒ SFML instalado ([Instalação do SFML](#))
- ☒ CMake instalado ([Instalação do CMake](#))
- ☒ IDE configurada ([Configuração da IDE](#)) (opcional)

## Setup Automático (Recomendado)

O método mais rápido e confiável:

```
# Clonar o repositório
git clone <repository-url>
cd 16Games-in-Cpp

# Executar script de configuração
chmod +x setup.sh
./setup.sh
```

O script fará:

1. ☒ Verificação de dependências
2. ☒ Configuração do CMake
3. ☒ Compilação de teste
4. ☒ Validação do ambiente

Se o script executar sem erros, **seu ambiente está pronto!** 🎉

## Setup Manual

Se preferir fazer manualmente ou se o script automático falhar:



## 1. Clonar e Preparar Projeto

```
# Clonar repositório
git clone <repository-url>
cd 16Games-in-Cpp

# Criar diretório de build
mkdir -p build
cd build
```

## 2. Configurar CMake

```
# Configurar projeto
cmake ..

# Verificar se não houve erros
echo $? # Deve retornar 0
```

Saída esperada:

```
-- The CXX compiler identification is GNU 9.4.0
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Found PkgConfig: /usr/bin/pkg-config (found version "0.29.2")
-- Checking for modules 'sfml-all>=2.5'
--   Found sfml-all, version 2.5.1
-- Configuring done
-- Generating done
-- Build files have been written to: /path/to/16Games-in-Cpp/build
```

## 3. Teste de Compilação

```
# Compilar um jogo de teste
make tetris

# Verificar se o executável foi criado
ls games/tetris/
# Deve mostrar: tetris (e possivelmente images/)
```

## 4. Teste de Execução

```
# Executar o jogo
cd games/tetris
```

```
./tetris

# Ou usar o target cmake
cd ../../ # voltar para build/
make run_tetris
```

Se o jogo abrir uma janela e funcionar, **tudo está perfeito!** 🎮

## ✅ Validação Completa

### Teste Todos os Componentes

#### 1. Verificar Todas as Dependências

```
# Script de verificação completa
cat > check_all.sh << 'EOF'
#!/bin/bash

echo "🔍 Verificação Completa do Ambiente"
echo "===== "

# Verificar compilador
if g++ --version &>/dev/null; then
    echo "✅ G++ - $(g++ --version | head -n1)"
else
    echo "❌ G++ não encontrado"
    exit 1
fi

# Verificar CMake
if cmake --version &>/dev/null; then
    echo "✅ CMake - $(cmake --version | head -n1)"
else
    echo "❌ CMake não encontrado"
    exit 1
fi

# Verificar SFML
if pkg-config --exists sfml-all; then
    echo "✅ SFML - $(pkg-config --modversion sfml-all)"
else
    echo "❌ SFML não encontrado"
    exit 1
fi
```

```
# Verificar espaço em disco
SPACE=$(df -BG . | tail -1 | awk '{print $4}' | sed 's/G//')
if [ $SPACE -gt 1 ]; then
    echo "✅ Espaço em disco - ${SPACE}GB disponível"
else
    echo "⚠️ Pouco espaço em disco - ${SPACE}GB disponível"
fi

echo ""
echo "🎉 Todos os componentes verificados com sucesso!"
EOF

chmod +x check_all.sh
./check_all.sh
```

## 2. Compilar Todos os Jogos

```
# No diretório build/
make all_games

# Verificar se todos foram compilados
ls games/
# Deve mostrar todos os 16 diretórios de jogos
```

## 3. Teste Rápido de Múltiplos Jogos

```
# Script para testar vários jogos
cat > test_games.sh << 'EOF'
#!/bin/bash

games=("tetris" "snake" "arkanoid" "doodle_jump")

for game in "${games[@]"; do
    echo "🎮 Testando $game..."
    cd "games/$game"
    timeout 3s "./$game" &>/dev/null
    if [ $? -eq 124 ]; then # timeout (esperado)
        echo "✅ $game - OK"
    else
        echo "❌ $game - ERRO"
    fi
    cd "../.."
done
```

EOF

```
chmod +x test_games.sh
./test_games.sh
```

## Configurações Opcionais

### Variáveis de Ambiente Úteis

Adicione ao seu `.bashrc` ou `.zshrc`:

```
# Alias para o projeto 16 Games
alias games-build='cd ~/16Games-in-Cpp/build && make all_games'
alias games-clean='cd ~/16Games-in-Cpp && rm -rf build && mkdir build'
alias games-run='cd ~/16Games-in-Cpp/build'

# Variáveis para desenvolvimento
export GAMES_PROJECT_ROOT="$HOME/16Games-in-Cpp"
export GAMES_BUILD_DIR="$GAMES_PROJECT_ROOT/build"

# Função para executar jogos rapidamente
play_game() {
    if [ -z "$1" ]; then
        echo "Uso: play_game <nome_do_jogo>"
        echo "Jogos disponíveis: tetris, snake, arkanoid, etc."
        return 1
    fi

    cd "$GAMES_BUILD_DIR/games/$1" && ".$1"
}
```

### Configuração de Performance

```
# Para compilação mais rápida
export CMAKE_BUILD_PARALLEL_LEVEL=$(nproc)
export MAKEFLAGS="-j$(nproc)"

# Para debug mais detalhado
export CMAKE_VERBOSE_MAKEFILE=ON
```

## Testando Todos os Jogos

### Script de Teste Completo

```

# Criar script de teste abrangente
cat > full_test.sh << 'EOF'
#!/bin/bash

cd "$(dirname "$0")/build"

echo "🎮 Teste Completo dos 16 Games in C++"
echo "===== "

games=(
    "tetris" "doodle_jump" "arkanoid" "snake" "minesweeper"
    "fifteen_puzzle" "racing" "outrun" "xonix" "bejeweled"
    "netwalk" "mahjong" "tron" "chess" "volleyball" "asteroids"
)

success=0
total=${#games[@]}

for game in "${games[@]"; do
    echo -n "Testando $game... "

    if [ -f "games/$game/$game" ]; then
        echo "✅ Compilado"
        ((success++))
    else
        echo "❌ Não encontrado"
    fi
done

echo ""
echo "📊 Resultado: $success/$total jogos compilados com sucesso"

if [ $success -eq $total ]; then
    echo "🎉 Todos os jogos estão funcionando perfeitamente!"
    echo ""
    echo "📌 Comandos para jogar:"
    echo "    cd build/games/tetris && ./tetris"
    echo "    make run_tetris"
    echo "    # ... e assim por diante"
else
    echo "⚠️ Alguns jogos não foram compilados. Execute 'make all_games' novamente."
fi

```

EOF

```
chmod +x full_test.sh
./full_test.sh
```

## Estrutura Final Esperada

Após a configuração completa, sua estrutura deve estar assim:

```
16Games-in-Cpp/
├── build/                # Arquivos compilados
│   ├── games/           # Executáveis dos jogos
│   │   ├── tetris/
│   │   │   ├── tetris    # Executável
│   │   │   └── images/   # Assets copiados
│   │   ├── snake/
│   │   └── ... (16 jogos)
│   └── CMakeCache.txt    # Cache do CMake
├── 01 Tetris/            # Código fonte
├── 02 Doodle Jump/
├── ... (código dos jogos)
├── CMakeLists.txt        # Configuração CMake
├── setup.sh              # Script de configuração
└── README.md
```

## Comandos de Manutenção

### Limpeza e Reconstrução

```
# Limpar build completo
rm -rf build
mkdir build
cd build
cmake ..
make all_games

# Ou usando script
../setup.sh
```

### Atualizar Dependências

```
# Ubuntu/Debian
sudo apt update && sudo apt upgrade libsFML-dev cmake
```

```
# Fedora
sudo dnf update SFML-devel cmake

# macOS
brew update && brew upgrade sfml cmake
```

## Verificação de Integridade

```
# Verificar arquivos corrompidos
find . -name "*.cpp" -exec g++ -fsyntax-only {} \;

# Verificar links das bibliotecas
ldd build/games/tetris/tetris # Linux
otool -L build/games/tetris/tetris # macOS
```

## ! Troubleshooting Final

### Problema: CMake não encontra SFML

```
# Verificar onde SFML está instalado
find /usr -name "*sfml*" 2>/dev/null

# Definir manualmente se necessário
cmake .. -DSFML_ROOT=/usr/local
```

### Problema: Jogos não executam

```
# Verificar dependências
ldd games/tetris/tetris

# Verificar se assets foram copiados
ls games/tetris/images/

# Executar com debug
gdb games/tetris/tetris
```

### Problema: Performance ruim

```
# Compilar em modo Release
cmake .. -DCMAKE_BUILD_TYPE=Release
make all_games
```





## Próximos Passos

Ambiente configurado com sucesso! Agora você pode:

1. **Jogar:** Execute `make run_tetris` para testar
2. **Compilar:** Vá para Guia de Compilação ([Guia de Compilação](#))
3. **Desenvolver:** Explore a Estrutura do Projeto ([Estrutura do Projeto](#))
4. **Documentar:** Veja os Jogos ([Jogos](#)) individuais

## Parabéns!

Se chegou até aqui com sucesso, você tem:

-  Ambiente completamente configurado
-  Todos os 16 jogos compilados
-  Ferramentas de desenvolvimento prontas
-  Scripts de manutenção configurados

Divirta-se jogando e explorando o código! 🎮

**Dica:** Mantenha os scripts `setup.sh` e `full_test.sh` à mão. Eles serão úteis para manutenção futura do ambiente!



# Compilação

Este guia completo mostra como compilar, executar e gerenciar o build dos 16 jogos em C++. 🛠️

## Visão Geral

O projeto usa **CMake** como sistema de build, que oferece:

- **Detecção automática** de dependências
- **Build multiplataforma** (Linux, macOS, Windows)
- **Gerenciamento de assets** (imagens, sons, fonts)
- **Targets individuais** para cada jogo
- **Comandos simplificados** de execução

## Compilação Rápida

### Primeiro Build

```
# Configurar e compilar tudo
./setup.sh

# Ou manualmente:
mkdir build && cd build
cmake ..
make all_games
```

### Builds Subsequentes

```
cd build
make all_games
```

## Compilação por Jogo

### Jogos Disponíveis

```
# Listar todos os targets disponíveis
make help | grep -E "(tetris|snake|arkanoid)"

# Targets dos jogos:
tetris, doodle_jump, arkanoid, snake, minesweeper
```

```
fifteen_puzzle, racing, outrun, xonix, bejeweled  
netwalk, mahjong, tron, chess, volleyball, asteroids
```

## Compilar Jogo Específico

```
# Compilar apenas o Tetris  
make tetris  
  
# Compilar Snake  
make snake  
  
# Compilar Arkanoid  
make arkanoid
```

## Executar Jogos

```
# Método 1: Target CMake (recomendado)  
make run_tetris  
make run_snake  
make run_arkanoid  
  
# Método 2: Executar diretamente  
cd games/tetris && ./tetris  
cd games/snake && ./snake  
  
# Método 3: A partir do build/  
./games/tetris/tetris  
./games/snake/snake
```

## Opções de Build

### Tipos de Build

#### Debug (Padrão)

```
cmake .. -DCMAKE_BUILD_TYPE=Debug  
make all_games  
  
# Características:  
# - Símbolos de debug incluídos  
# - Otimizações desabilitadas  
# - Assertions habilitadas  
# - Executáveis maiores
```

## Release

```
cmake .. -DCMAKE_BUILD_TYPE=Release
make all_games
```

```
# Características:
# - Máxima otimização (-O3)
# - Sem símbolos de debug
# - Executáveis menores
# - Melhor performance
```

## RelWithDebInfo

```
cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo
make all_games
```

```
# Características:
# - Otimizado + símbolos debug
# - Bom para profiling
# - Compromisso entre debug e performance
```

## Compilação Paralela

```
# Usar todos os cores disponíveis
make -j$(nproc) # Linux/macOS
make -j%NUMBER_OF_PROCESSORS% # Windows

# Ou definir permanentemente
export MAKEFLAGS="-j$(nproc)"
make all_games
```

## Compilação Verbose

```
# Ver comandos completos de compilação
make VERBOSE=1

# Ou configurar permanentemente
cmake .. -DCMAKE_VERBOSE_MAKEFILE=ON
make
```



## Configurações Avançadas

### Compilador Específico

```
# Usar GCC específico
cmake .. -DCMAKE_CXX_COMPILER=g++-9

# Usar Clang
cmake .. -DCMAKE_CXX_COMPILER=clang++

# Windows - Visual Studio
cmake .. -G "Visual Studio 16 2019"
```

## Flags de Compilação Customizadas

```
# Adicionar flags extras
cmake .. -DCMAKE_CXX_FLAGS="-Wall -Wextra -Wpedantic"

# Debug com sanitizers
cmake .. -DCMAKE_BUILD_TYPE=Debug \
        -DCMAKE_CXX_FLAGS="-fsanitize=address -fsanitize=undefined"
```

## Configurar SFML Manualmente

```
# Se SFML não for encontrado automaticamente
cmake .. -DSFML_ROOT=/usr/local/SFML

# Ou especificar bibliotecas
cmake .. -DSFML_LIBRARIES="sfml-system;sfml-window;sfml-graphics;sfml-audio"
```

## Estrutura de Build

### Diretórios Gerados

```
build/
├─ CMakeCache.txt           # Cache do CMake
├─ CMakeFiles/              # Arquivos internos do CMake
├─ Makefile                 # Makefile principal
├─ games/                   # Executáveis e assets
│   └─ tetris/
│       └─ tetris           # Executável
│           └─ images/      # Assets copiados
├─ snake/
│   └─ snake
│       └─ images/
```

```
|   └─ ... (para cada jogo)
└─ cmake_install.cmake    # Script de instalação
```

## Assets Automaticamente Gerenciados

O CMake copia automaticamente:

- `images/` → `build/games/<jogo>/images/`
- `fonts/` → `build/games/<jogo>/fonts/`
- `files/` → `build/games/<jogo>/files/`

## Scripts Úteis

### Build Script Personalizado

```
# Criar script de build personalizado
cat > quick_build.sh << 'EOF'
#!/bin/bash

GAME="$1"
BUILD_TYPE="${2:-Debug}"

if [ -z "$GAME" ]; then
    echo "Uso: $0 <jogo> [Debug|Release]"
    echo "Jogos: tetris, snake, arkanoid, etc."
    echo "Exemplo: $0 tetris Release"
    exit 1
fi

echo "🔨 Compilando $GAME em modo $BUILD_TYPE..."

# Criar/limpar build se necessário
if [ ! -d "build" ]; then
    mkdir build
fi

cd build

# Configurar se necessário
if [ ! -f "CMakeCache.txt" ] || [ "$BUILD_TYPE" != "$(cat CMakeCache.txt |
grep CMAKE_BUILD_TYPE | cut -d'=' -f2)" ]; then
    echo "⚙️ Configurando CMake..."
    cmake .. -DCMAKE_BUILD_TYPE="$BUILD_TYPE"
```

```

fi

# Compilar jogo específico
echo "🔧 Compilando..."
make "$GAME" -j$(nproc)

if [ $? -eq 0 ]; then
    echo "✅ $GAME compilado com sucesso!"
    echo "🎮 Para executar: make run_$GAME"
else
    echo "❌ Erro na compilação!"
    exit 1
fi
EOF

chmod +x quick_build.sh

# Usar o script
./quick_build.sh tetris Debug
./quick_build.sh snake Release

```

## Clean Build Script

```

# Script para limpeza completa
cat > clean_build.sh << 'EOF'
#!/bin/bash

echo "🧹 Limpando build anterior..."
rm -rf build

echo "📁 Criando diretório build..."
mkdir build
cd build

echo "⚙️ Configurando CMake..."
cmake ..

echo "🔧 Compilando todos os jogos..."
make all_games -j$(nproc)

echo "✅ Build limpo concluído!"
EOF

```

```
chmod +x clean_build.sh
./clean_build.sh
```



## Debug e Profiling

### Compilar para Debug

```
# Build com informações de debug
cmake .. -DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_FLAGS="-g3 -O0"
make tetris

# Executar com GDB
gdb ./games/tetris/tetris
```

### Profiling com Valgrind

```
# Compilar com símbolos de debug
cmake .. -DCMAKE_BUILD_TYPE=Debug
make tetris

# Executar com Valgrind
valgrind --tool=memcheck --leak-check=full ./games/tetris/tetris
```

### Análise de Performance

```
# Compilar otimizado com símbolos
cmake .. -DCMAKE_BUILD_TYPE=RelWithDebInfo
make tetris

# Profiling com perf (Linux)
perf record ./games/tetris/tetris
perf report
```



## Troubleshooting de Compilação

### Erro: SFML não encontrado

```
# Verificar instalação
pkg-config --exists sfml-all
pkg-config --modversion sfml-all

# Limpar cache e reconfigurar
```

```
rm CMakeCache.txt
cmake .. -DSFML_ROOT=/usr/local
```

### Erro: Compilador não suporta C++17

```
# Verificar versão do compilador
g++ --version

# Usar compilador mais novo
cmake .. -DCMAKE_CXX_COMPILER=g++-9
```

### Erro: Assets não encontrados

```
# Verificar se assets foram copiados
ls build/games/tetris/images/

# Recompilar para forçar cópia
make clean
make tetris
```

### Erro: "make: command not found"

```
# Linux - instalar build-essential
sudo apt install build-essential

# macOS - instalar Xcode Command Line Tools
xcode-select --install

# Windows - usar cmake --build
cmake --build . --target all_games
```



## Monitoramento de Build

### Tempo de Compilação

```
# Medir tempo total
time make all_games

# Medir por jogo
time make tetris
```

### Uso de Recursos



```
# Monitor durante build
htop # Em outro terminal

# Compilação com limite de CPU
make -j2 all_games # Usar apenas 2 cores
```

## Tamanho dos Executáveis

```
# Ver tamanho de todos os jogos
du -sh games/*/

# Detalhes de um jogo específico
ls -lah games/tetris/tetris
file games/tetris/tetris
```

## Comandos de Referência Rápida

```
# Setup inicial
./setup.sh

# Compilar tudo
make all_games

# Compilar jogo específico
make <nome_do_jogo>

# Executar jogo
make run_<nome_do_jogo>

# Limpar e reconstruir
rm -rf build && mkdir build && cd build && cmake .. && make all_games

# Build otimizado
cmake .. -DCMAKE_BUILD_TYPE=Release && make all_games

# Build paralelo
make -j$(nproc) all_games

# Ver comandos disponíveis
make help
```

## Próximos Passos

Agora que você domina a compilação:

1. Explore Build and Run ([Compilar e Executar Jogos](#)) para execução prática
2. Veja Running Games ([Executando os Jogos](#)) para mais opções de execução
3. Consulte Estrutura do Projeto ([Estrutura do Projeto](#)) para entender o código

Se encontrar problemas:

1. Verifique Troubleshooting ([Troubleshooting](#))
2. Confirme que seguiu Configuração do Ambiente ([Configuração do Ambiente](#))

Dica: Use `make -j$(nproc)` para compilação mais rápida, e `make run_<jogo>` para executar diretamente!

# Guia de Compilação

Este guia explica como compilar os jogos do projeto "16 Games in C++". Vamos abordar desde os conceitos básicos até técnicas avançadas de compilação.

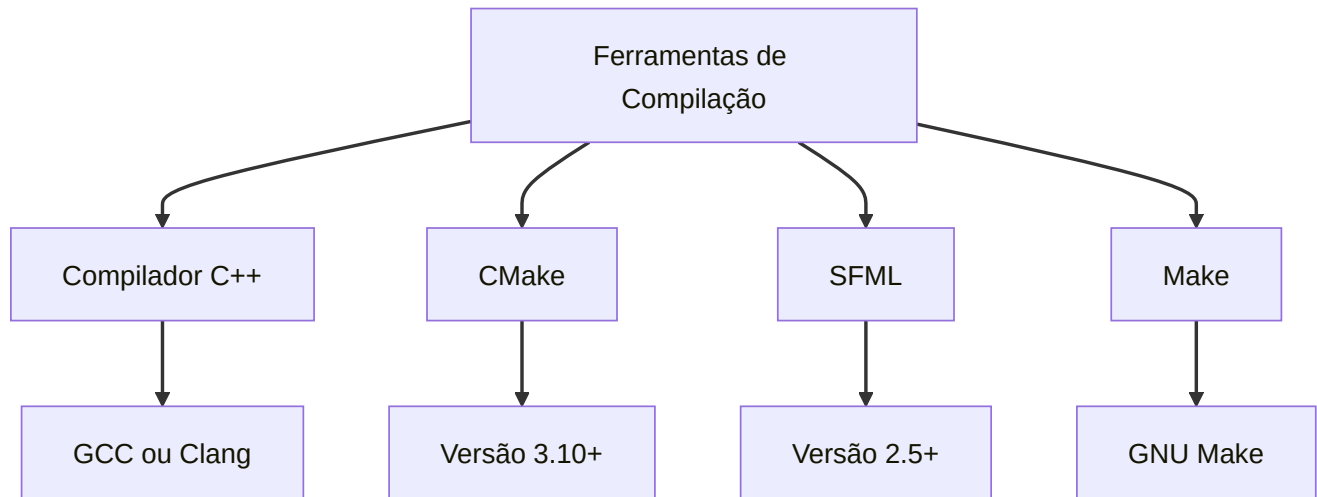
## O que é Compilação

Compilação é o processo de transformar o código fonte que escrevemos (em C++) em um programa executável que o computador consegue entender e executar. É como traduzir um texto do português para uma linguagem que o computador entende.

## Pré-requisitos

Antes de compilar os jogos, você precisa ter algumas ferramentas instaladas no seu sistema:

### Ferramentas Necessárias



### 1. Compilador C++

Um compilador é o programa que converte seu código C++ em código executável.

Opções disponíveis:

- **GCC (GNU Compiler Collection):** O compilador mais comum no Linux
- **Clang:** Compilador alternativo com ótimas mensagens de erro
- **MSVC:** Compilador da Microsoft para Windows

Como verificar se está instalado:

```
g++ --version      # Para GCC
```

```
clang++ --version # Para Clang
```

## 2. CMake

CMake é uma ferramenta que facilita o processo de compilação de projetos complexos. Ele gera automaticamente os arquivos necessários para compilar seu projeto.

**Por que usar CMake:**

- Funciona em diferentes sistemas operacionais
- Gerencia dependências automaticamente
- Configura opções de compilação
- Organiza projetos com múltiplos arquivos

**Como verificar se está instalado:**

```
cmake --version
```

## 3. SFML (Simple and Fast Multimedia Library)

SFML é a biblioteca que usamos para criar os jogos. Ela fornece funcionalidades para:

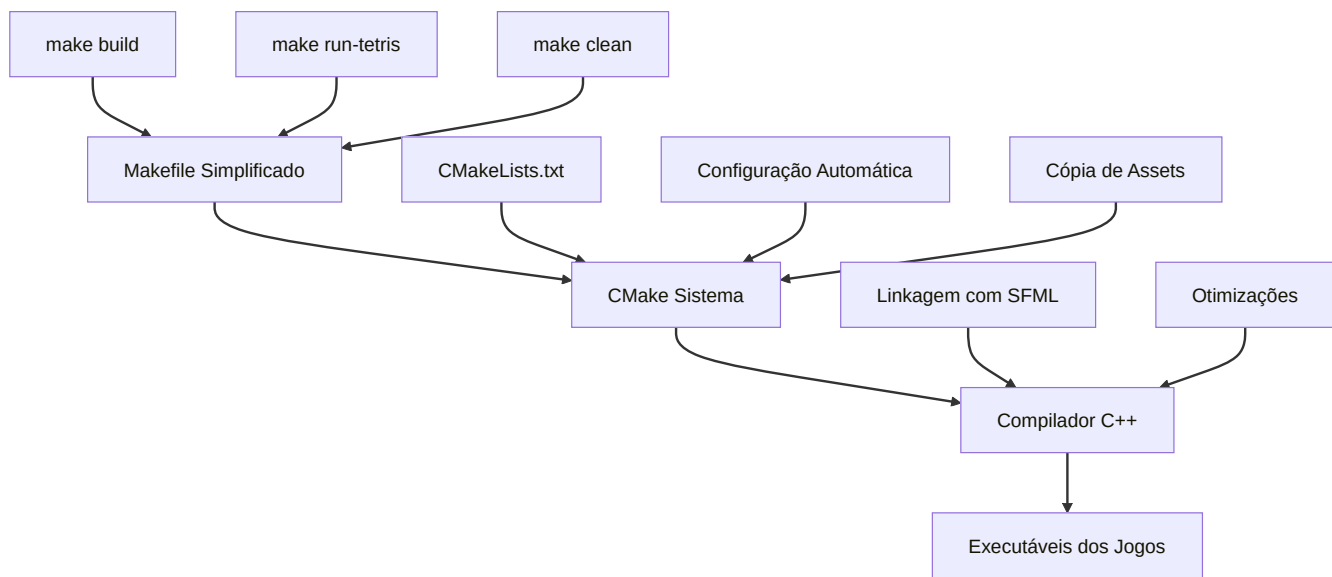
- Criar janelas e interfaces gráficas
- Desenhar sprites e formas
- Reproduzir sons
- Capturar entrada do teclado e mouse

**Como verificar se está instalado:**

```
pkg-config --exists sfml-all && echo "SFML instalado" || echo "SFML não encontrado"
```

## Sistema de Compilação do Projeto

Nosso projeto usa um sistema de compilação em duas camadas:



## Camada 1: Makefile Simplificado

O Makefile fornece comandos fáceis de usar:

```

make setup      # Configuração inicial
make build      # Compilar todos os jogos
make build-tetris # Compilar apenas o Tetris
make run-tetris  # Executar o Tetris
make clean      # Limpar arquivos temporários
  
```

## Camada 2: CMake

O CMake gerencia a compilação real:

- Encontra as bibliotecas necessárias (SFML)
- Configura opções de compilação
- Cria executáveis para cada jogo
- Copia assets (imagens, fontes) para os locais corretos

## Processo de Compilação Passo a Passo

### Configuração Inicial

```

# 1. Executar o script de configuração
./setup.sh
  
```

Este script:

1. Verifica se todas as dependências estão instaladas
2. Cria o diretório `build`
3. Configura o CMake
4. Testa a compilação com o jogo Tetris

## Compilação Individual

Para compilar um jogo específico:

```
# Método 1: Usando Makefile (mais simples)
make build-tetris

# Método 2: Usando CMake diretamente
cd build
make tetris
```

## Compilação de Todos os Jogos

```
# Método 1: Usando Makefile
make build

# Método 2: Usando CMake diretamente
cd build
make all_games
```

## Estrutura de Arquivos de Compilação

Quando você compila um jogo, o CMake organiza os arquivos assim:

```
build/
├── games/
│   ├── tetris/
│   │   ├── tetris          # Executável
│   │   └── images/        # Assets copiados
│   ├── doodle_jump/
│   │   ├── doodle_jump    # Executável
│   │   ├── images/        # Assets copiados
│   │   └── fonts/         # Fontes copiadas
│   └── ...
├── CMakeFiles/             # Arquivos internos do CMake
└── Makefile                # Makefile gerado pelo CMake
```

## Por que Essa Estrutura?

1. **Organização:** Cada jogo tem sua própria pasta
2. **Assets Isolados:** Imagens e recursos ficam junto ao executável
3. **Execução Simples:** Cada jogo pode ser executado de sua própria pasta
4. **Facilita Debug:** Problemas ficam isolados por jogo

## Entendendo o CMakeLists.txt

O arquivo `CMakeLists.txt` é o "manual de instruções" para o CMake. Vamos entender as partes principais:

### Configuração Básica

```
cmake_minimum_required(VERSION 3.10) # Versão mínima do CMake
project(16Games LANGUAGES CXX)       # Nome do projeto e linguagem
set(CMAKE_CXX_STANDARD 17)           # Usar C++17
```

### Encontrar Dependências

```
find_package(PkgConfig REQUIRED)        # Ferramenta para encontrar
bibliotecas
pkg_check_modules(SFML REQUIRED sfml-all) # Encontrar SFML
```

### Função para Criar Jogos

```
function(add_game GAME_NAME GAME_DIR)
    # Encontrar arquivos .cpp e .hpp
    file(GLOB_RECURSE GAME_SOURCES "${GAME_DIR}/*.cpp" "${GAME_DIR}/*.hpp")

    # Criar executável
    add_executable(${GAME_NAME} ${GAME_SOURCES})

    # Linkar com SFML
    target_link_libraries(${GAME_NAME} ${SFML_LIBRARIES})

    # Copiar assets
    # ...
endfunction()
```

## Tipos de Compilação

## Debug vs Release

### Debug Mode (Modo de Depuração):

- Código mais lento
- Inclui informações para debugging
- Facilita encontrar erros

### Release Mode (Modo de Lançamento):

- Código otimizado e mais rápido
- Menor tamanho de arquivo
- Sem informações de debug

```
# Compilar em modo debug (padrão)
cmake ..

# Compilar em modo release
cmake -DCMAKE_BUILD_TYPE=Release ..
```

## Compilação Paralela


Para acelerar a compilação, use múltiplos cores do processador:

```
# Usar 4 cores para compilar
make -j4

# Usar todos os cores disponíveis
make -j$(nproc)
```

## Resolvendo Problemas de Compilação

### Erro: SFML não encontrado

 Erro: Package 'sfml-all' not found

### Solução:

```
# Ubuntu/Debian
sudo apt-get install libsFML-dev
```



```
# Fedora
sudo dnf install SFML-devel

# Arch Linux
sudo pacman -S sfml
```

### Erro: CMake muito antigo

✗ Erro: CMake 3.10 or higher is required

#### Solução:

```
# Ubuntu/Debian
sudo apt-get install cmake

# Ou instalar versão mais nova
sudo snap install cmake --classic
```

### Erro: Compilador não encontrado

✗ Erro: No CMAKE\_CXX\_COMPILER could be found

#### Solução:

```
# Ubuntu/Debian
sudo apt-get install build-essential

# Fedora
sudo dnf group install "Development Tools"
```

### Erro: Arquivo não encontrado durante execução

✗ Erro: Failed to load images/background.png

#### Solução:

- Verificar se os assets foram copiados corretamente
- Executar o jogo do diretório correto:

```
cd build/games/tetris
./tetris
```

## Compilação Customizada

### Adicionando Flags de Compilação

Para adicionar opções especiais de compilação, edite o `CMakeLists.txt`:

```
# Adicionar flags de warning
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra")

# Adicionar otimizações específicas
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -O3 -march=native")
```

### Compilação Condicional

```
# Compilar apenas jogos funcionais
if(EXISTS "${CMAKE_SOURCE_DIR}/${GAME_DIR}/main.cpp")
    add_game(${GAME_NAME} ${GAME_DIR})
endif()
```

## Scripts de Automação

### Script de Teste Rápido

```
#!/bin/bash
echo "Testando compilação de todos os jogos..."
make clean
make build

if [ $? -eq 0 ]; then
    echo "✅ Todos os jogos compilaram com sucesso!"
else
    echo "❌ Erro na compilação"
    exit 1
fi
```

### Script de Deploy

```
#!/bin/bash
echo "Criando pacote de distribuição..."
make clean
cmake -DCMAKE_BUILD_TYPE=Release ..
make all_games
```

```
# Criar arquivo tar com todos os jogos
tar -czf 16games-release.tar.gz build/games/
```

## Otimização de Performance

### Compilação Otimizada

```
# Configurar para máxima performance
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_FLAGS="-O3 -march=native" ..
```

### Profile-Guided Optimization (PGO)

Para jogos com alta performance:

```
# 1. Compilar com instrumentação
cmake -DCMAKE_CXX_FLAGS="-fprofile-generate" ..
make tetris

# 2. Executar para coletar dados
./games/tetris/tetris

# 3. Recompilar com otimizações baseadas no perfil
cmake -DCMAKE_CXX_FLAGS="-fprofile-use" ..
make tetris
```

## Integração com IDEs

### CLion

1. Abrir o diretório do projeto
2. CLion detectará automaticamente o CMakeLists.txt
3. Configurar build profiles (Debug/Release)
4. Usar os targets automáticos para compilar e executar

### Visual Studio Code

1. Instalar extensões: C/C++, CMake Tools
2. Abrir o projeto
3. Configurar kit de compilação

4. Usar paleta de comandos: "CMake: Build"

## **Code::Blocks**

1. Criar projeto vazio
2. Adicionar arquivos fonte
3. Configurar linker para SFML
4. Configurar diretórios de include

## **Conclusão**

A compilação é um processo fundamental no desenvolvimento de jogos. Nosso projeto usa um sistema robusto que:

- **Automatiza** tarefas repetitivas
- **Organiza** arquivos de forma lógica
- **Facilita** a manutenção e debugging
- **Funciona** em diferentes sistemas operacionais

Dominar esses conceitos te permitirá não apenas compilar os jogos existentes, mas também modificá-los, criar novos jogos e entender como projetos maiores são organizados.

### **Próximos passos:**

1. Pratique com diferentes jogos
2. Experimente modificar opções de compilação
3. Tente adicionar novos arquivos a um jogo existente
4. Aprenda a usar ferramentas de debugging como GDB

# Compilar e Executar Jogos

Este guia mostra como compilar e executar os jogos de forma prática e eficiente. Aprenda desde comandos básicos até técnicas avançadas de execução.

## Compilação Rápida

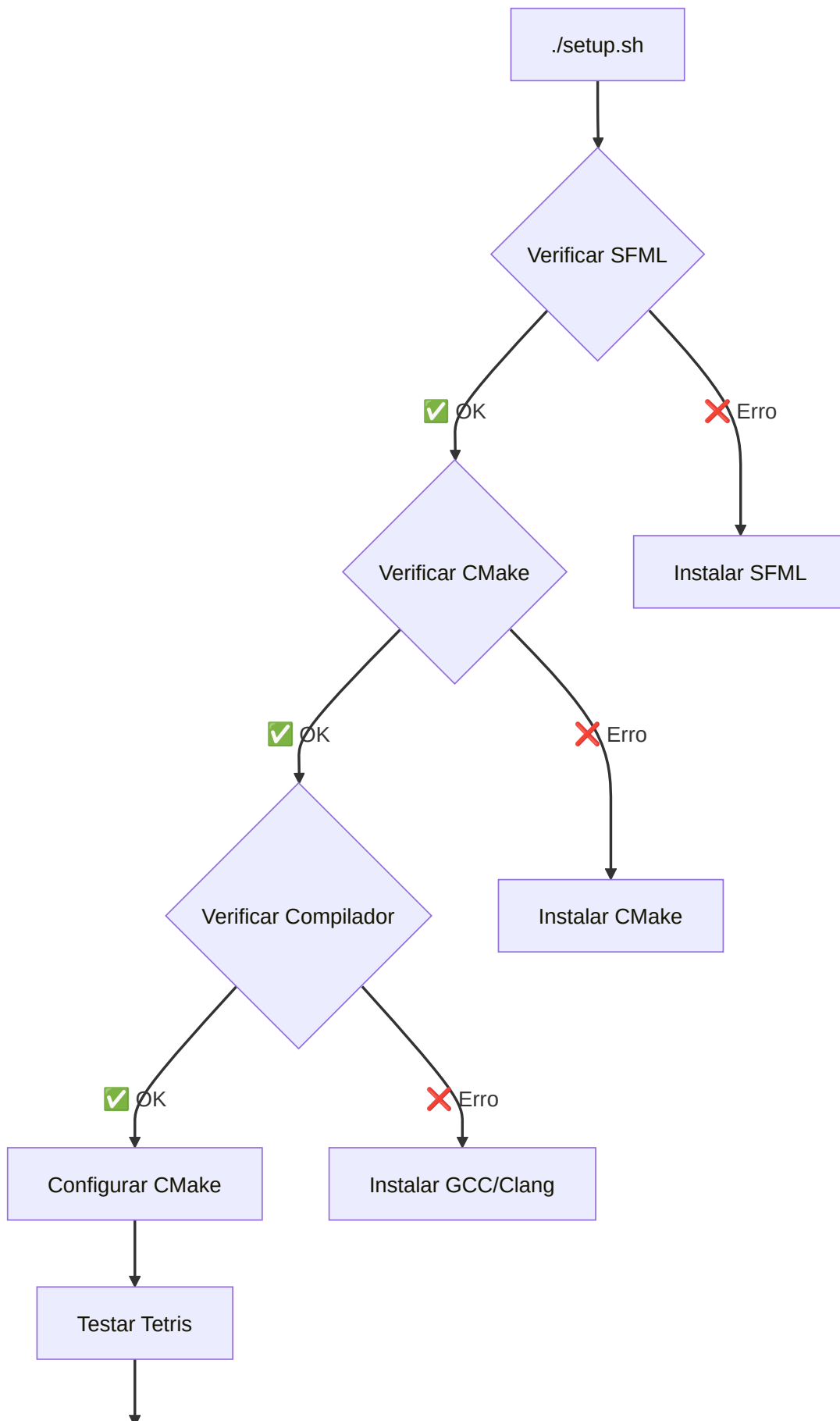
### Configuração Inicial (Apenas Uma Vez)

Antes de compilar qualquer jogo, execute a configuração inicial:

```
# Na pasta raiz do projeto
./setup.sh
```

Este comando:

- Verifica se todas as dependências estão instaladas
- Configura o ambiente de compilação
- Testa a compilação com um jogo simples
- Prepara o sistema para uso



✓ Pronto para usar

## Métodos de Compilação

### Método 1: Makefile Simplificado (Recomendado)

O método mais fácil para iniciantes:

```
# Compilar todos os jogos
make build

# Compilar um jogo específico
make build-tetris
make build-doodle_jump
make build-snake

# Executar diretamente
make run-tetris
make run-doodle_jump
make run-snake
```

### Método 2: CMake Direto

Para usuários mais avançados:

```
# Entrar na pasta de build
cd build

# Compilar todos os jogos
make all_games

# Compilar um jogo específico
make tetris
make doodle_jump
make snake

# Executar um jogo
make run_tetris
make run_doodle_jump
make run_snake
```

### Método 3: Execução Direta

Após compilar, você pode executar diretamente:

```
# Navegar para o jogo
cd build/games/tetris

# Executar
./tetris
```

## Compilação por Categoria

### Jogos de Puzzle

```
# Tetris - O clássico jogo de blocos
make build-tetris && make run-tetris

# Fifteen Puzzle - Quebra-cabeça numérico
make build-fifteen_puzzle && make run-fifteen_puzzle

# Minesweeper - Campo minado
make build-minesweeper && make run-minesweeper

# NetWalk - Conectar tubulações
make build-netwalk && make run-netwalk

# Mahjong Solitaire - Paciência com peças
make build-mahjong && make run-mahjong
```

### Jogos de Ação

```
# Doodle Jump - Pular nas plataformas
make build-doodle_jump && make run-doodle_jump

# Snake - A serpente clássica
make build-snake && make run-snake

# Arkanoid - Quebrar blocos com a bola
make build-arkanoid && make run-arkanoid

# Asteroids - Destruir asteroides no espaço
make build-asteroids && make run-asteroids
```

### Jogos de Estratégia



```
# Chess - Xadrez com IA
make build-chess && make run-chess

# Bejeweled - Combinar joias
make build-bejeweled && make run-bejeweled

# Xonix - Conquistar território
make build-xonix && make run-xonix
```

## Verificação de Compilação

### Status dos Jogos

Nem todos os jogos podem estar funcionais. Vamos verificar o status:

```
# Tentar compilar todos e ver quais funcionam
make build 2>&1 | grep -E "(✅|❌|Error|error)"
```

### Teste Individual

Para testar se um jogo específico funciona:

```
# Testar Tetris
echo "Testando Tetris..."
if make build-tetris; then
    echo "✅ Tetris compilou com sucesso!"
    make run-tetris
else
    echo "❌ Erro na compilação do Tetris"
fi
```

### Script de Teste Automatizado

Crie um script para testar todos os jogos:

```
#!/bin/bash
# Salve como test_all_games.sh

games=("tetris" "doodle_jump" "arkanoid" "snake" "minesweeper"
      "fifteen_puzzle" "racing" "xonix" "bejeweled" "netwalk")

echo "Testando compilação de todos os jogos..."
echo "=====
```

```
for game in "${games[@]}"; do
    echo -n "Testando $game... "
    if make build-$game &>/dev/null; then
        echo "✅"
    else
        echo "❌"
    fi
done
```

## Execução Avançada

### Execução com Parâmetros

Alguns jogos podem aceitar parâmetros de linha de comando:

```
# Executar com resolução específica (se suportado)
cd build/games/tetris
./tetris --width=800 --height=600

# Executar em modo fullscreen (se suportado)
./tetris --fullscreen
```

### Execução com Debugging

Para investigar problemas:

```
# Executar com GDB (debugger)
cd build/games/tetris
gdb ./tetris

# Comandos no GDB:
# (gdb) run           # Executar o programa
# (gdb) bt            # Ver stack trace se houver crash
# (gdb) quit          # Sair do GDB
```

### Execução com Profiling

Para medir performance:

```
# Medir tempo de execução
cd build/games/tetris
time ./tetris
```

```
# Profiling detalhado com valgrind
valgrind --tool=callgrind ./tetris
```

## Gerenciamento de Assets

### Verificar Assets

Os jogos dependem de arquivos de imagem e som. Para verificar se estão corretos:

```
# Verificar se as imagens foram copiadas
ls build/games/tetris/images/

# Verificar se as fontes foram copiadas (Doodle Jump)
ls build/games/doodle_jump/fonts/
```

### Recopiar Assets

Se algum asset não foi copiado corretamente:

```
# Limpar build e recompilar
make clean
make build-tetris
```

### Assets Customizados

Para usar seus próprios assets:

1. Substituir arquivos na pasta original:

```
# Exemplo: trocar a imagem de fundo do Tetris
cp minha_imagem.png "01 Tetris/images/background.png"
```

2. Recompilar o jogo:

```
make clean
make build-tetris
```

## Compilação em Lote

### Compilar Jogos Funcionais

Script para compilar apenas jogos que funcionam:

```
#!/bin/bash
# Lista de jogos que compilam sem erro
working_games=("tetris" "doodle_jump" "arkanoid" "snake" "minesweeper")

echo "Compilando jogos funcionais..."
for game in "${working_games[@]}; do
    echo "Compilando $game..."
    make build-$game
done

echo "✅ Jogos funcionais compilados!"
```

## Compilação Paralela

Para acelerar a compilação usando múltiplos cores:

```
# Usar 4 cores
cd build
make -j4 all_games

# Usar todos os cores disponíveis
make -j$(nproc) all_games
```

## Resolução de Problemas

### Problemas Comuns de Compilação

**Erro: SFML não encontrado**

```
error: Package 'sfml-all' not found
```

**Solução:**

```
# Ubuntu/Debian
sudo apt-get install libsFML-dev

# Reconfigurar
make clean
./setup.sh
```

**Erro: Arquivos não encontrados na execução**

```
Failed to load images/background.png
```

## Soluções:

### 1. Verificar diretório de execução:

```
# Sempre executar da pasta do jogo
cd build/games/tetris
./tetris
```

### 2. Recompilar para recopiar assets:

```
make clean
make build-tetris
```

## Erro: Permissão negada

```
bash: ./tetris: Permission denied
```

## Solução:

```
# Dar permissão de execução
chmod +x build/games/tetris/tetris
```

## Debugging de Problemas

### Verificar Dependências

```
# Verificar se o executável foi linkado corretamente
ldd build/games/tetris/tetris

# Deve mostrar libsFML-* nas dependências
```

### Verificar Assets

```
# Verificar se todos os assets estão presentes
find build/games/tetris/ -name "*.png" -o -name "*.jpg" -o -name "*.ttf"
```

### Log de Execução

```
# Executar com output detalhado
cd build/games/tetris
./tetris 2>&1 | tee tetris.log
```

```
# Verificar o log
cat tetris.log
```

## Automação com Scripts

### Script Completo de Build e Test

```
#!/bin/bash
# build_and_test.sh

set -e # Parar se houver erro

echo "🎮 Build e Test Automatizado"
echo "===== "

# Configuração inicial
echo "1 Configurando ambiente..."
./setup.sh

# Lista de jogos para testar
games=("tetris" "doodle_jump" "snake" "arkanoid" "minesweeper")

echo "2 Compilando jogos..."
for game in "${games[@]}; do
    echo "    Compilando $game..."
    make build-$game
done

echo "3 Testando execução (5 segundos cada)..."
for game in "${games[@]}; do
    echo "    Testando $game..."
    cd build/games/$game
    timeout 5s ./$game || echo "    (Fechado automaticamente)"
    cd - > /dev/null
done

echo "✅ Build e teste concluídos!"
```

### Script de Limpeza

```
#!/bin/bash
# clean_all.sh
```

```
echo "🧹 Limpando arquivos de build..."

# Remover diretório build
rm -rf build

# Remover arquivos temporários
find . -name "*.o" -delete
find . -name "*.so" -delete
find . -name "core" -delete

echo "✅ Limpeza concluída!"
```

## Performance e Otimização

### Compilação Otimizada

```
# Compilar em modo Release (otimizado)
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make all_games
```

### Medição de Performance

```
# Medir tempo de compilação
time make all_games

# Medir uso de memória durante compilação
/usr/bin/time -v make tetris
```

### Compilação Incremental

Para desenvolvimento, compile apenas o que mudou:

```
# Depois de modificar código, apenas:
cd build
make tetris # Só recompila se necessário
```

## Integração com Editor/IDE

### Visual Studio Code

Configurar tasks.json para compilação rápida:

```

{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Build Tetris",
      "type": "shell",
      "command": "make",
      "args": ["build-tetris"],
      "group": "build"
    },
    {
      "label": "Run Tetris",
      "type": "shell",
      "command": "make",
      "args": ["run-tetris"],
      "group": "test"
    }
  ]
}

```

## CLion

1. Abrir projeto (CLion detecta CMakeLists.txt automaticamente)
2. Configurar targets de build
3. Usar botões de Run/Debug diretamente na IDE

## Conclusão

Compilar e executar jogos é um processo que envolve várias etapas, mas com os métodos corretos torna-se simples e eficiente. O projeto oferece múltiplas formas de abordar a compilação:

- **Para iniciantes:** Use `make build` e `make run-nome_do_jogo`
- **Para desenvolvimento:** Use compilação incremental e debugging
- **Para distribuição:** Use compilação otimizada em modo Release

Dominar essas técnicas permitirá que você não apenas execute os jogos existentes, mas também modifique e crie novos jogos com confiança.



# Executando os Jogos

Este guia completo mostra como executar cada um dos 16 jogos, incluindo controles, objetivos e dicas para cada jogo.

## Formas de Executar

### Método Rápido (Recomendado)

```
# Compilar e executar em um comando
make run-tetris
make run-doodle_jump
make run-snake
make run-arkanoid
```

### Método Manual

```
# 1. Compilar primeiro
make build-tetris

# 2. Navegar para a pasta do jogo
cd build/games/tetris

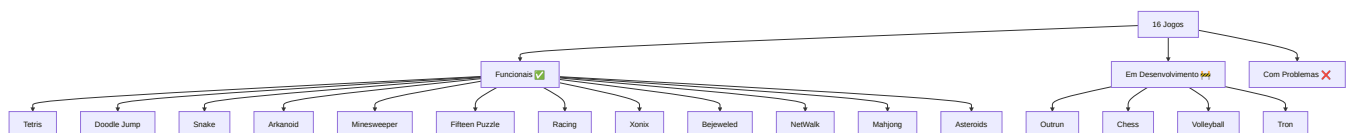
# 3. Executar
./tetris
```

### Método Direto (Após compilação)

```
# Se já compilou antes, pode executar diretamente
cd build/games/tetris && ./tetris
```

## Jogos Disponíveis

### Status de Funcionamento



## Guia Individual dos Jogos

### 1. Tetris

#### Como executar:

```
make run-tetris
```

#### Controles:

- **Setas Esquerda/Direita:** Mover peça
- **Seta Baixo:** Acelerar queda
- **Seta Cima:** Rotacionar peça
- **Espaço:** Drop instantâneo

**Objetivo:** Completar linhas horizontais para eliminá-las. O jogo acelera conforme você progride.

#### Dicas:

- Deixe espaços para peças longas (I-tetromino)
- Não acumule peças muito alto
- Use o drop instantâneo para pontuação extra

## 2. Doodle Jump

#### Como executar:

```
make run-doodle_jump
```

#### Controles:

- **Setas Esquerda/Direita:** Mover personagem
- **Espaço:** Iniciar jogo (no menu)
- **R:** Reiniciar (game over)
- **M:** Voltar ao menu

**Objetivo:** Pule de plataforma em plataforma, alcançando a maior altura possível.

#### Dicas:

- Use o movimento horizontal para alcançar plataformas distantes
- O personagem "atravessa" a tela (sai de um lado, aparece do outro)
- Quanto mais alto, mais pontos você ganha

### 3. Snake

Como executar:

```
make run-snake
```

Controles:

- **Setas:** Direção da serpente
- **Não pode** voltar na direção contrária

**Objetivo:** Comer comida para crescer, evitando colidir com as bordas ou com o próprio corpo.

Dicas:

- Planeje sua rota com antecedência
- Evite criar "armadilhas" para si mesmo
- Use as bordas da tela estrategicamente

### 4. Arkanoid

Como executar:

```
make run-arkanoid
```

Controles:

- **Setas Esquerda/Direita:** Mover raquete
- **Espaço:** Iniciar jogo/Lançar bola

**Objetivo:** Quebrar todos os blocos rebatendo a bola com a raquete.

Dicas:

- Mantenha a bola sempre em movimento
- Use os ângulos da raquete para controlar a direção
- Mire nos cantos dos blocos para ângulos interessantes

### 5. Minesweeper (Campo Minado)

Como executar:

```
make run-minesweeper
```

**Controles:**

- **Click Esquerdo:** Revelar célula
- **Click Direito:** Marcar/desmarcar bandeira
- **(Pode variar dependendo da implementação)**

**Objetivo:** Revelar todas as células sem bombas, usando números como pistas.

**Dicas:**

- Números indicam quantas bombas estão adjacentes
- Comece pelos cantos e bordas
- Use bandeiras para marcar bombas conhecidas

## 6. Fifteen Puzzle

**Como executar:**

```
make run-fifteen_puzzle
```

**Controles:**

- **Setas:** Mover peças
- **Mouse:** Clicar em peças adjacentes ao espaço vazio

**Objetivo:** Organizar números de 1 a 15 em ordem, deixando o espaço vazio no canto inferior direito.

**Dicas:**

- Resolva linha por linha, de cima para baixo
- Use o espaço vazio estrategicamente
- Últimas duas linhas requerem técnica especial

## 7. Racing (Top Down)

**Como executar:**

```
make run-racing
```

**Controles:**

- **Setas:** Direção do carro
- **Acelerar/Freiar** (dependendo da implementação)

**Objetivo:** Completar voltas na pista no menor tempo possível.

**Dicas:**

- Freie antes das curvas
- Use a pista inteira nas curvas
- Mantenha velocidade constante nas retas

## 8. Xonix

**Como executar:**

```
make run-xonix
```

**Controles:**

- **Setas:** Mover personagem

**Objetivo:** Conquistar território desenhando linhas, evitando inimigos.

**Dicas:**

- Faça movimentos rápidos para evitar inimigos
- Conquiste pequenas áreas por vez
- Cuidado com inimigos que seguem sua trilha

## 9. Bejeweled

**Como executar:**

```
make run-bejeweled
```

**Controles:**

- **Mouse:** Selecionar e trocar joias
- **Setas:** Navegar (se implementado)

**Objetivo:** Formar grupos de 3 ou mais joias iguais para eliminá-las.

**Dicas:**

- Procure por grupos de 4 ou 5 para power-ups
- Planeje várias jogadas em sequência
- Observe oportunidades de cascata

## 10. NetWalk

**Como executar:**

```
make run-netwalk
```

**Controles:**

- **Mouse:** Clicar para rotacionar peças
- **Setas:** Navegar pelo grid

**Objetivo:** Conectar todas as peças de tubulação para formar uma rede completa.

**Dicas:**

- Comece pelas peças de canto (menos opções)
- Identifique o caminho principal primeiro
- Rotacione peças sistematicamente

## 11. Mahjong Solitaire

**Como executar:**

```
make run-mahjong
```

**Controles:**

- **Mouse:** Selecionar peças
- **Scroll:** Rotacionar visualização (se 3D)

**Objetivo:** Remover todas as peças combinando pares iguais que estejam livres.

**Dicas:**

- Uma peça está "livre" se não há peças em cima ou dos lados
- Procure por peças únicas primeiro
- Planeje para não bloquear peças necessárias

## 12. Asteroids

Como executar:

```
make run-asteroids
```

Controles:

- **Setas:** Rotacionar e acelerar nave
- **Espaço:** Atirar
- **Shift/Ctrl:** Escudo/Hiper-espaço (se implementado)

**Objetivo:** Destruir todos os asteroides sem ser atingido.

Dicas:

- Asteroides grandes se dividem em menores
- Use o impulso com cuidado (sem atrito no espaço)
- Atenção às bordas da tela (wraparound)

## Jogos em Desenvolvimento

### 13. Outrun

Status: 🚧 Em desenvolvimento **Problema comum:** Renderização 3D complexa

### 14. Chess

Status: 🚧 Em desenvolvimento **Problema comum:** IA e validação de movimentos

### 15. Volleyball

Status: 🚧 Em desenvolvimento **Problema comum:** Física da bola e multiplayer

### 16. Tron

Status: 🚧 Em desenvolvimento **Problema comum:** Trail rendering e IA

## Solução de Problemas na Execução

## Jogo não inicia

```
# Verificar se foi compilado
ls build/games/tetris/tetris

# Verificar permissões
chmod +x build/games/tetris/tetris

# Tentar executar com debug
cd build/games/tetris
gdb ./tetris
```

## Erro "Failed to load image"

```
# Verificar se assets estão presentes
ls build/games/tetris/images/

# Recompilar para recopiar assets
make clean
make build-tetris
```

## Jogo executa mas tela preta

### Possíveis causas:

- Assets não encontrados
- Problema com drivers gráficos
- Resolução incompatível

### Soluções:

```
# Verificar se SFML funciona
pkg-config --exists sfml-all && echo "OK" || echo "Problema"

# Testar com jogo mais simples
make run-snake # Snake usa menos recursos gráficos
```

## Performance ruim

```
# Compilar em modo otimizado
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
```



```
make tetris

# Verificar uso de recursos
top -p $(pgrep tetris)
```

## Scripts Úteis

### Testar Todos os Jogos

```
#!/bin/bash
# test_all.sh

games=("tetris" "doodle_jump" "snake" "arkanoid" "minesweeper"
      "fifteen_puzzle" "racing" "xonix" "bejeweled" "netwalk"
      "mahjong" "asteroids")

for game in "${games[@]}; do
    echo "Testando $game..."
    if make run-$game &>/dev/null & then
        PID=$!
        sleep 3
        kill $PID 2>/dev/null
        echo "✅ $game funciona"
    else
        echo "❌ $game tem problemas"
    fi
done
```

### Menu Interativo

```
#!/bin/bash
# game_menu.sh

echo "🎮 Seletor de Jogos"
echo "====="
echo "1) Tetris"
echo "2) Doodle Jump"
echo "3) Snake"
echo "4) Arkanoid"
echo "5) Minesweeper"
echo "0) Sair"

read -p "Escolha um jogo (0-5): " choice
```

```
case $choice in
    1) make run-tetris ;;
    2) make run-doodle_jump ;;
    3) make run-snake ;;
    4) make run-arkanoid ;;
    5) make run-minesweeper ;;
    0) echo "Tchau!" ;;
    *) echo "Opção inválida" ;;
esac
```

## Dicas Gerais de Execução

### Controles Universais

A maioria dos jogos usa:

- **ESC:** Sair do jogo
- **Enter:** Confirmar/Pausar
- **Espaço:** Ação principal
- **Setas:** Navegação/Movimento

### Resolução de Tela

Alguns jogos podem ter resolução fixa. Se a janela ficar muito pequena ou grande:

#### 1. Modificar código fonte (avançado):

```
// Procurar por linhas como:
RenderWindow window(VideoMode(800, 600), "Nome do Jogo");
```

#### 2. Usar modo janela:

- A maioria dos jogos abre em janela
- Pode ser possível redimensionar manualmente

### Performance

Para melhor performance:

- Feche outros programas

- Use modo Release quando compilar
- Verifique drivers de vídeo atualizados

## Salvamento

A maioria dos jogos não salva progresso automaticamente:

- High scores podem ser perdidos ao fechar
- Anote suas melhores pontuações manualmente
- Alguns jogos podem criar arquivos de save

## Conclusão

Executar os jogos é a parte mais divertida do projeto! Cada jogo oferece uma experiência única e demonstra diferentes conceitos de programação de jogos. Use este guia como referência para:

- **Descobrir** novos jogos para jogar
- **Aprender** diferentes mecânicas de jogo
- **Solucionar** problemas de execução
- **Comparar** implementações diferentes

Divirta-se explorando todos os 16 jogos e descobrindo suas mecânicas e segredos!

# Estrutura do Projeto

Este guia explica como o projeto "16 Games in C++" está organizado, facilitando a navegação, compreensão e modificação do código.

## Visão Geral da Estrutura

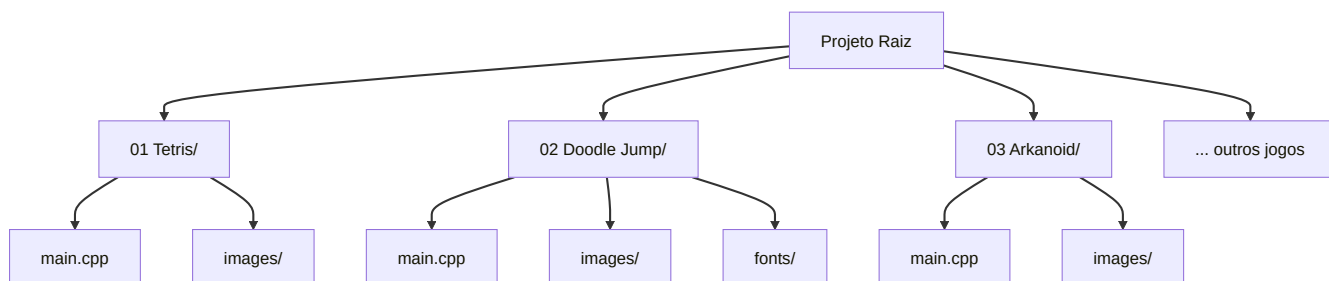
O projeto segue uma organização lógica que separa jogos, documentação, scripts e configurações:

```
16Games-in-Cpp/  
├── 01 Tetris/           # Jogo 1: Tetris  
├── 02 Doodle Jump/     # Jogo 2: Doodle Jump  
├── 03 Arkanoid/         # Jogo 3: Arkanoid  
├── ...                 # Jogos 4-16  
├── Writerside/          # Documentação  
├── scripts/             # Scripts de automação  
├── build/               # Arquivos compilados (gerado)  
├── CMakeLists.txt       # Configuração de build  
├── Makefile             # Comandos simplificados  
├── setup.sh             # Script de configuração  
└── README.md            # Informações básicas
```

## Estrutura Detalhada

### Diretórios dos Jogos

Cada jogo tem sua própria pasta seguindo o padrão `NN Nome/`:



### Estrutura Típica de um Jogo

```
01 Tetris/  
├── main.cpp             # Código principal do jogo  
├── images/              # Assets visuais  
│   ├── tiles.png        # Sprites dos blocos  
│   ├── background.png   # Imagem de fundo  
│   └── ...  
└── fonts/               # Fontes (quando necessário)
```

```
|   └─ arial.ttf
└─ files/           # Outros recursos (sons, configs)
    └─ config.txt
```

## Arquivos de Configuração

### CMakeLists.txt

O arquivo principal de configuração do sistema de build:

```
# Configuração básica
cmake_minimum_required(VERSION 3.10)
project(16Games LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 17)

# Função para adicionar jogos automaticamente
function(add_game GAME_NAME GAME_DIR)
    # Encontra arquivos fonte
    # Configura compilação
    # Copia assets
    # Cria targets de execução
endfunction()

# Lista de todos os jogos
add_game(tetris "01 Tetris")
add_game(doodle_jump "02 Doodle Jump")
# ... outros jogos
```

### Makefile

Interface simplificada para comandos comuns:

```
# Comandos básicos
setup:    # Configuração inicial
build:    # Compilar todos os jogos
clean:    # Limpar arquivos temporários
test:     # Testar compilação

# Comandos específicos por jogo
build-%:  # Compilar jogo específico
run-%:    # Executar jogo específico
```

## Diretório de Build

Quando você compila o projeto, é criada a estrutura `build/`:

```

build/
├── games/                                # Executáveis organizados
│   ├── tetris/
│   │   ├── tetris                        # Executável
│   │   └── images/                      # Assets copiados
│   ├── doodle_jump/
│   │   ├── doodle_jump                 # Executável
│   │   ├── images/                    # Assets copiados
│   │   └── fonts/                     # Fontes copiadas
│   └── ...
├── CMakeFiles/                          # Arquivos internos do CMake
├── Makefile                             # Makefile gerado
└── cmake_install.cmake                  # Script de instalação

```

## Organização por Categoria

### Jogos de Puzzle

```

Puzzle Games/
├── 01 Tetris/                            # Blocos que caem
├── 05 Minesweeper/                      # Campo minado
├── 06 Fifteen-Puzzle/                   # Quebra-cabeça deslizante
├── 11 NetWalk/                          # Conectar tubulações
└── 12 Mahjong Solitaire/                # Paciência com peças

```

### Jogos de Ação

```

Action Games/
├── 02 Doodle Jump/                      # Pular plataformas
├── 03 Arkanoid/                         # Quebrar blocos
├── 04 Snake/                           # Serpente clássica
├── 09 Xonix/                           # Conquistar território
└── 16 Asteroids/                       # Nave espacial

```

### Jogos de Corrida/EspORTE

```

Racing/Sports Games/
├── 07 Racing (Top Down)/                # Corrida vista de cima
├── 08 Outrun/                          # Corrida 3D
└── 15 Volleyball/                      # Vôlei multiplayer

```

### Jogos de Estratégia

```
Strategy Games/  
├─ 10 Bejeweled/           # Combinar joias  
├─ 13 Tron/                # Batalha de motos  
└─ 14 Chess/               # Xadrez com IA
```

## Padrões de Código

### Estrutura Típica do main.cpp

```
#include <SFML/Graphics.hpp>  
#include <iostream>  
// ... outras includes  
  
using namespace sf;  
  
// 1. Estruturas e variáveis globais  
struct GameData {  
    // dados do jogo  
};  
  
// 2. Funções auxiliares  
void initializeGame() {  
    // inicialização  
}  
  
void updateGame() {  
    // lógica do jogo  
}  
  
void renderGame() {  
    // desenhar na tela  
}  
  
// 3. Função principal  
int main() {  
    // Configuração inicial  
    RenderWindow window(VideoMode(800, 600), "Nome do Jogo");  
  
    // Loop principal  
    while (window.isOpen()) {  
        // Processar eventos  
        Event event;  
        while (window.pollEvent(event)) {
```

```

        // tratar entrada do usuário
    }

    // Atualizar lógica
    updateGame();

    // Renderizar
    window.clear();
    renderGame();
    window.display();
}

return 0;
}

```

## Convenções de Nomenclatura

### Arquivos

- **Executáveis:** Nome do jogo em minúsculas com underscore
  - tetris, doodle\_jump, fifteen\_puzzle
- **Diretórios:** Número + nome em maiúsculas
  - 01 Tetris, 12 Mahjong Solitaire
- **Assets:** Nomes descritivos em minúsculas
  - background.png, player\_sprite.png, block\_01.png

### Código

- **Variáveis:** camelCase
  - gameState, playerPosition, currentScore
- **Constantes:** UPPER\_CASE
  - SCREEN\_WIDTH, MAX\_PLAYERS, GRAVITY\_FORCE
- **Funções:** camelCase com verbos
  - initializeGame(), handleInput(), drawSprites()

## Sistema de Assets

### Organização de Imagens



```

images/
├── sprites/                # Personagens e objetos
│   ├── player.png
│   ├── enemy_01.png
│   └── powerup.png
├── backgrounds/           # Fundos
│   ├── menu_bg.png
│   └── game_bg.png
├── ui/                    # Interface
│   ├── button.png
│   ├── health_bar.png
│   └── score_panel.png
└── tiles/                 # Elementos de cenário
    ├── wall.png
    ├── floor.png
    └── platform.png

```

## Gerenciamento de Assets

O CMake automaticamente copia assets durante a compilação:

```

# Copiar imagens
if(EXISTS "${CMAKE_SOURCE_DIR}/${GAME_DIR}/images")
    file(COPY "${CMAKE_SOURCE_DIR}/${GAME_DIR}/images/"
         DESTINATION "${CMAKE_BINARY_DIR}/games/${GAME_NAME}/images")
endif()

# Copiar fontes
if(EXISTS "${CMAKE_SOURCE_DIR}/${GAME_DIR}/fonts")
    file(COPY "${CMAKE_SOURCE_DIR}/${GAME_DIR}/fonts/"
         DESTINATION "${CMAKE_BINARY_DIR}/games/${GAME_NAME}/fonts")
endif()

```

## Scripts e Automação

### Scripts Principais

```

scripts/
├── main.sh                # Script principal de automação
├── push_remote_repo.sh    # Deploy/publicação
└── unzip_writerside.sh    # Processamento de documentação

```

### Scripts na Raiz

```
./
├── setup.sh           # Configuração inicial do ambiente
├── fix_games.sh       # Correção de problemas comuns
└── test_games.sh      # Teste automatizado dos jogos
```

#### Exemplo: setup.sh

```
#!/bin/bash
echo "🎮 Configurando ambiente para 16 Games in C++"

# Verificar dependências
echo "📋 Verificando dependências..."
check_sfml() { ... }
check_cmake() { ... }

# Configurar build
echo "🔧 Configurando projeto..."
mkdir -p build
cd build && cmake ..

# Testar compilação
echo "🎯 Testando compilação..."
make tetris
```

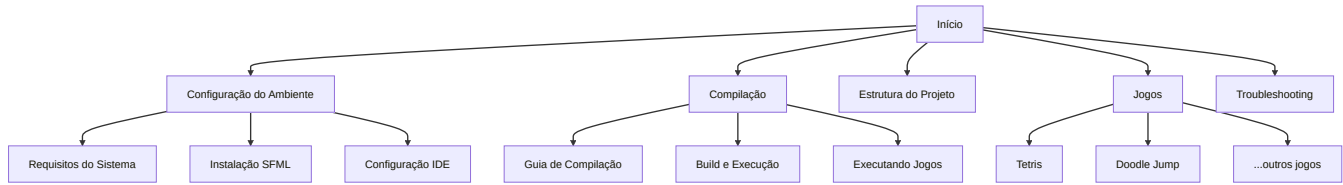
## Documentação

### Estrutura do Writerside

```
Writerside/
├── cfg/           # Configurações do Writerside
├── images/        # Imagens da documentação
├── topics/        # Arquivos de documentação
│   ├── inicio.md  # Página inicial
│   ├── system-requirements.md
│   ├── compilation-guide.md
│   ├── tetris.md  # Tutorial do Tetris
│   ├── doodle-jump.md  # Tutorial do Doodle Jump
│   └── ...
├── gc.tree        # Estrutura de navegação
└── writerside.cfg # Configuração principal
```

### Organização de Tópicos

A documentação está organizada hierarquicamente:



## Boas Práticas

### Organização de Código

#### 1. Separação de Responsabilidades

- Um arquivo main.cpp por jogo
- Funções específicas para cada aspecto (input, update, render)
- Estruturas de dados bem definidas

#### 2. Gerenciamento de Recursos

- Assets organizados por tipo
- Verificação de carregamento de arquivos
- Liberação adequada de memória

#### 3. Configuração Flexível

- Constantes para valores importantes
- Separação entre lógica e configuração
- Facilidade de modificação

## Extensibilidade

Para adicionar um novo jogo:

1. Criar diretório seguindo o padrão NN Nome/
2. Implementar main.cpp com estrutura padrão
3. Adicionar assets na pasta images/
4. Registrar no CMakeLists.txt:

```
add_game(novo_jogo "17 Novo Jogo")
```

## 5. Atualizar lista no Makefile se necessário

## Manutenção

### 1. Backup Regular

- Versionar código com Git
- Backup de assets importantes
- Documentar mudanças significativas

### 2. Testes Regulares

- Verificar compilação de todos os jogos
- Testar funcionalidade básica
- Validar assets e dependências

### 3. Documentação Atualizada

- Manter tutoriais sincronizados com código
- Documentar mudanças na estrutura
- Atualizar guias de instalação

## Navegação Eficiente

### Comandos de Terminal

```
# Navegar rapidamente para um jogo
cd "01 Tetris"           # Usar aspas por causa do espaço
cd build/games/tetris    # Executável compilado

# Encontrar arquivos específicos
find . -name "*.png"     # Todas as imagens
find . -name "main.cpp"  # Todos os arquivos principais
grep -r "SFML" .         # Procurar por SFML no código
```

### Atalhos Úteis

```
# Aliases para .bashrc ou .zshrc
alias games='cd ~/path/to/16Games-in-Cpp'
alias buildgames='cd ~/path/to/16Games-in-Cpp/build'
alias rungames='cd ~/path/to/16Games-in-Cpp && make run- '
```

```
# Funções úteis
runGame() {
    cd ~/path/to/16Games-in-Cpp
    make run-$1
}
# Uso: runGame tetris
```

## Conclusão

A estrutura do projeto "16 Games in C++" foi projetada para ser:

- **Intuitiva:** Fácil de navegar e entender
- **Escalável:** Fácil de adicionar novos jogos
- **Mantível:** Código e assets bem organizados
- **Flexível:** Suporta diferentes tipos de jogos
- **Automatizada:** Build e deploy simplificados

Compreender essa estrutura é fundamental para:

- **Modificar** jogos existentes
- **Criar** novos jogos
- **Contribuir** para o projeto
- **Usar** como base para outros projetos

Use este guia como referência para navegar eficientemente pelo projeto e aproveitar ao máximo sua organização.

# Jogos

Conheça todos os 16 jogos incluídos neste projeto, cada um com suas mecânicas únicas e implementações interessantes! 🎮

## Visão Geral

Este projeto inclui 16 jogos clássicos, cada um demonstrando diferentes conceitos de programação de jogos:

- Física e movimento
- Detecção de colisão
- Inteligência artificial (IA básica)
- Geração procedural
- Estados de jogo
- Interface de usuário
- Gerenciamento de assets

## Lista Completa de Jogos

### Jogos de Puzzle

1. Tetris ([Tetris](#)) - O clássico jogo de blocos que caem
2. - Quebra-cabeça numérico deslizante
3. Minesweeper ([Minesweeper](#)) - Campo minado clássico
4. - Conecte os canos para criar uma rede
5. - Paciência com peças do mahjong

### Jogos de Ação/Arcade

6. Doodle Jump ([Doodle Jump](#)) - Pule o mais alto possível
7. Snake ([Snake](#)) - A serpente clássica que cresce
8. Arkanoid ([Arkanoid](#)) - Quebre todos os blocos com a bola
9. - Combine joias para pontos

10. - Conquiste território evitando inimigos

### **Jogos de Corrida**

11. - Corrida vista de cima

12. - Corrida em perspectiva 3D

### **Jogos de Combate/Estratégia**

13. - Destrua asteroides no espaço

14. - Batalha de motos de luz

15. - Xadrez completo com IA

16. - Vôlei arcade multiplayer

### **Status dos Jogos**

Jogo	Compilação	Execução	Complexidade	Conceitos Principais
Tetris	✓	✓	★ ★ ★	Estados, Rotação, Grid
Doodle Jump	✓	✓	★ ★	Física, Câmera, Procedural
Arkanoid	✓	✓	★ ★	Colisão, Física da Bola
Snake	✓	✓	★ ★	Lista, Crescimento
Minesweeper	✓	✓	★ ★	Grid, Recursão
Fifteen Puzzle	✓	✓	★ ★	Algoritmos, Shuffling
Racing	✓	✓	★ ★ ★	Movimento, Colisão
Xonix	✓	✓	★ ★ ★	Flood Fill, Territory
Bejeweled	✓	✓	★ ★ ★	Match-3, Animações
NetWalk	✓	✓	★ ★ ★	Conectividade, Rotação
Mahjong	✓	✓	★ ★ ★	3D Stacking, Matching
Asteroids	✓	✓	★ ★ ★ ★	Vetores, Rotação
Outrun	✗	-	★ ★ ★ ★	Pseudo-3D, Sprites
Chess	✗	-	★ ★ ★ ★ ★	IA, Validação de Movimentos
Volleyball	✗	-	★ ★ ★	Multiplayer, Física
Tron	✗	-	★ ★	Trail Rendering, IA

**Legenda de Complexidade:**

- ★ = Muito Simples
- ★ ★ = Simples



- ★★★★★ = Intermediário
- ★★★★★★ = Avançado
- ★★★★★★★★ = Muito Avançado

## Como Jogar

### Execução Rápida

```
# Compilar todos os jogos
make all_games

# Executar um jogo específico
make run_tetris
make run_doodle_jump
make run_snake
```

### Execução Individual

```
# Navegar para o jogo
cd build/games/tetris

# Executar
./tetris
```

## Conceitos por Jogo

### Física e Movimento

- Doodle Jump ([Doodle Jump](#)): Gravidade, impulso, wrapping
- Arkanoid ([Arkanoid](#)): Rebote de bola, colisão angular
- : Movimento vetorial, rotação

### Algoritmos

- Minesweeper ([Minesweeper](#)): Flood fill recursivo
- : Algoritmo de embaralhamento
- : Minimax, avaliação de posição

## Renderização

- Tetris ([Tetris](#)): Grid-based rendering
- : Pseudo-3D com sprites
- : Trail rendering

## Inteligência Artificial

- : IA completa com diferentes níveis
- : IA básica de pathfinding

## Geração Procedural

- Doodle Jump ([Doodle Jump](#)): Plataformas infinitas
- : Geração de pista

## Recursos de Aprendizado

### Para Iniciantes

Comece com estes jogos mais simples:

1. Snake ([Snake](#)) - Conceitos básicos
2. Doodle Jump ([Doodle Jump](#)) - Física simples
3. Minesweeper ([Minesweeper](#)) - Lógica de grid

### Para Intermediários

Avance para estes jogos:

1. Tetris ([Tetris](#)) - Estados complexos
2. Arkanoid ([Arkanoid](#)) - Física de colisão
3. - Match-3 algorithm

### Para Avançados

Desafie-se com:

1. - IA complexa

2. - Matemática vetorial

3. - Renderização 3D

## Estrutura dos Jogos

Cada jogo segue uma estrutura similar:

```
<Jogo>/
├─ main.cpp          # Código principal
├─ images/           # Sprites e texturas
├─ fonts/            # Fontes (quando necessário)
└─ files/            # Assets adicionais
```

### Padrão de Implementação

```
// Estrutura comum dos jogos
int main() {
    // 1. Inicialização
    RenderWindow window(...);
    // Carregar assets

    // 2. Loop principal
    while (window.isOpen()) {
        // 2.1 Eventos
        Event event;
        while (window.pollEvent(event)) {
            // Processar entrada
        }

        // 2.2 Lógica do jogo
        // Atualizar estado

        // 2.3 Renderização
        window.clear();
        // Desenhar elementos
        window.display();
    }

    return 0;
}
```

## Próximos Passos

1. **Escolha um jogo** que te interesse
2. **Leia o tutorial** específico
3. **Execute o jogo** para entender a mecânica
4. **Analise o código** para ver a implementação
5. **Experimente modificações** para aprender



## Dicas de Estudo

### Análise de Código

- Comece lendo a função `main()`
- Identifique o loop principal
- Entenda as estruturas de dados
- Trace o fluxo de execução

### Experimentação

- Modifique valores constantes
- Adicione prints para debug
- Implemente pequenas melhorias
- Teste diferentes cenários

### Progressão

- Domine um jogo antes de passar para outro
- Implemente variações dos jogos
- Combine conceitos de diferentes jogos
- Crie seus próprios jogos

**Escolha seu jogo favorito e comece a explorar!** Cada um oferece uma experiência única de aprendizado.



# Tetris

Este tutorial ensina como criar o jogo Tetris do zero usando C++ e SFML. Vamos começar com conceitos básicos e construir o conhecimento passo a passo, explicando cada parte de forma clara e detalhada.

## O que é Tetris

Imagine um jogo onde peças de diferentes formatos caem do céu como chuva, e você precisa organizá-las de forma inteligente para que se encaixem perfeitamente. É como um quebra-cabeças em movimento, onde:

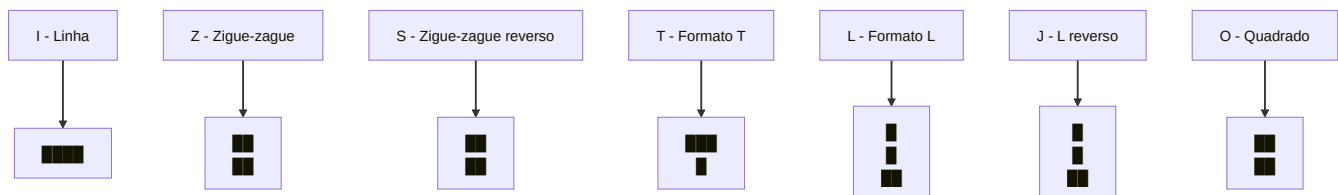
- Peças geométricas (chamadas de "tetrominós") caem de cima para baixo
- Você pode mover as peças para esquerda e direita
- Você pode girar as peças para encaixá-las melhor
- Quando uma linha horizontal fica completamente preenchida, ela desaparece
- O objetivo é durar o máximo de tempo possível sem deixar as peças chegarem ao topo

Este jogo é perfeito para aprender conceitos fundamentais de programação como arrays bidimensionais, rotação de objetos, detecção de colisões e manipulação de dados.

## A Matemática Secreta dos Tetrominós

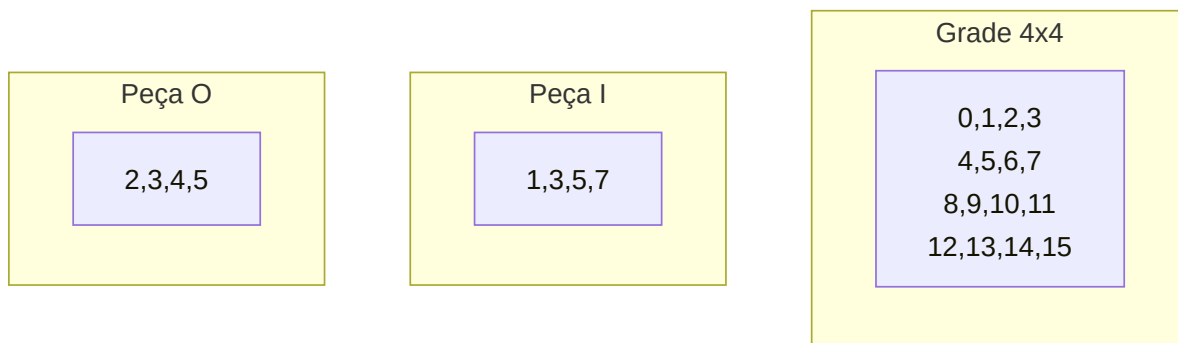
### O que são Tetrominós?

Tetrominós são formas geométricas feitas de exatamente 4 quadrados conectados. Existem apenas 7 formas diferentes possíveis:



### Como Representar Formas com Números

A parte mais inteligente do nosso Tetris é como representamos cada peça usando apenas números. Em vez de desenhar cada forma, usamos um sistema de coordenadas em uma grade 4x4:



Vamos entender como isso funciona:

```
int figures[7][4] = {  
    1, 3, 5, 7, // I - Linha vertical  
    2, 4, 5, 7, // Z - Zigue-zague  
    3, 5, 4, 6, // S - Zigue-zague reverso  
    3, 5, 4, 7, // T - Formato T  
    2, 3, 5, 7, // L - Formato L  
    3, 5, 7, 6, // J - L reverso  
    2, 3, 4, 5, // O - Quadrado  
};
```

### Como converter números em posições:

Para transformar um número em coordenadas (x, y):

```
int x = numero % 2; // Resto da divisão por 2 = coluna  
int y = numero / 2; // Divisão inteira por 2 = linha
```

Por exemplo, o número 5:

- $x = 5 \% 2 = 1$  (coluna 1)
- $y = 5 / 2 = 2$  (linha 2)

Isso significa que a posição 5 está na coluna 1, linha 2 da nossa grade 4x4.

## A Estrutura do Jogo

### O Campo de Jogo - Uma Grade Inteligente

O campo de jogo é como uma folha de papel quadriculado, onde cada quadradinho pode estar vazio (0) ou preenchido com uma cor (1 a 7):

```
const int M = 20; // 20 linhas de altura  
const int N = 10; // 10 colunas de largura
```

```
int field[M][N] = {0}; // Inicializa tudo com 0 (vazio)
```

### Por que 20x10?

- É o tamanho clássico do Tetris original
- Oferece desafio suficiente sem ser impossível
- Permite que as peças tenham espaço para manobrar

## Representando as Peças Ativas

Cada peça tem 4 blocos, então usamos duas estruturas de dados:

```
struct Point {  
    int x, y;    // Coordenadas de cada bloco  
} a[4], b[4];   // a = posição atual, b = posição anterior
```

### Por que duas arrays?

- **a[4]**: Posição atual da peça (onde ela está agora)
- **b[4]**: Posição anterior da peça (onde ela estava antes)
- Se um movimento for inválido, copiamos b para a (desfazemos o movimento)

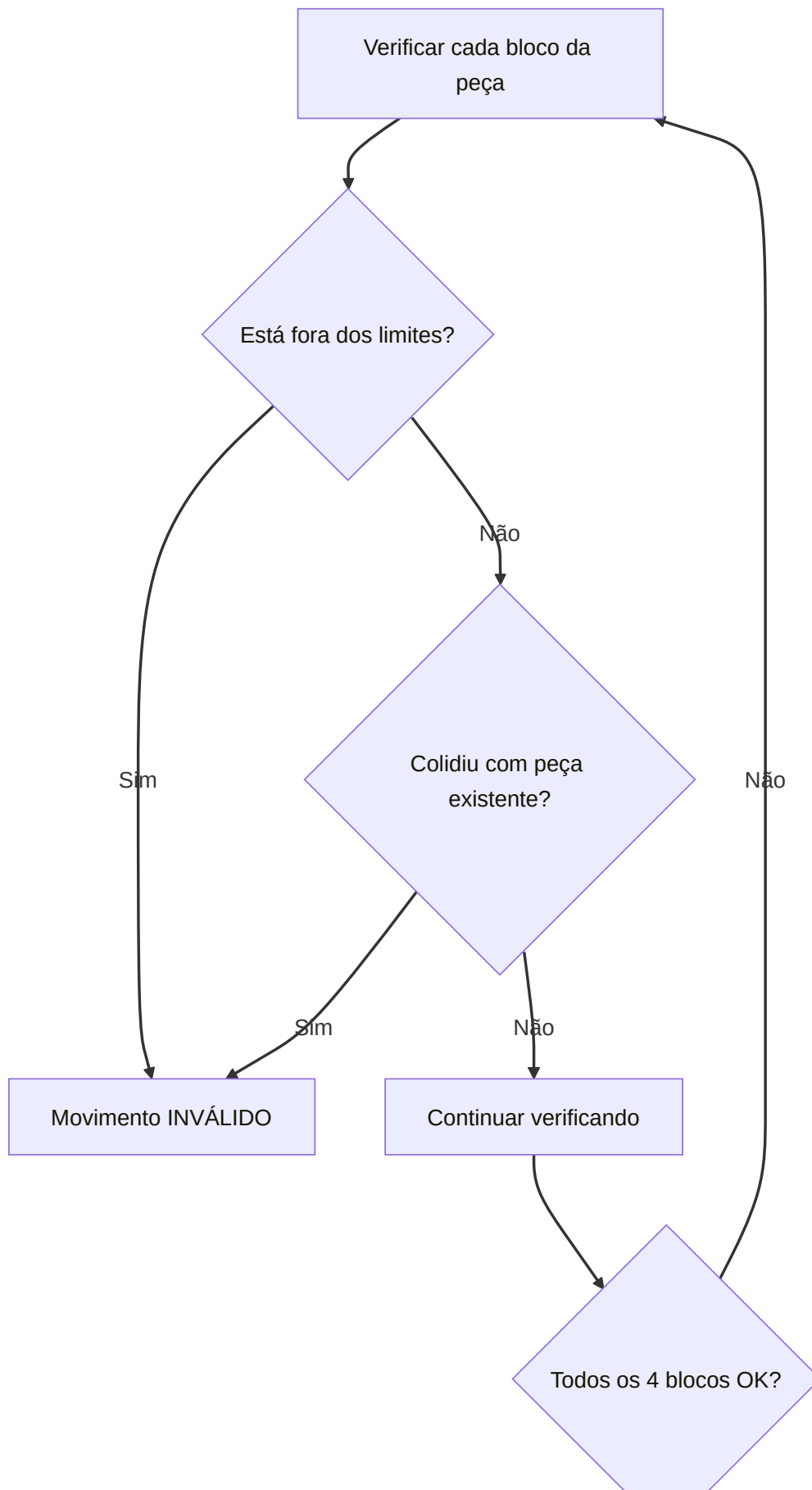
## As Principais Mecânicas do Jogo

### 1. Verificação de Colisões - A Função Mais Importante

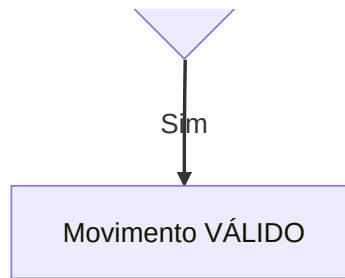
Esta é a função que decide se um movimento é válido ou não:

```
bool check() {  
    for (int i = 0; i < 4; i++) {  
        // Verifica se a peça saiu dos limites da tela  
        if (a[i].x < 0 || a[i].x >= N || a[i].y >= M) return false;  
  
        // Verifica se a peça colidiu com algo já no campo  
        else if (field[a[i].y][a[i].x]) return false;  
    }  
    return true; // Movimento é válido  
}
```

O que esta função verifica:







## 2. Movimento Horizontal - Esquerda e Direita

O movimento horizontal é simples mas usa um truque inteligente:

```
// Salvar posição atual
for (int i = 0; i < 4; i++) {
    b[i] = a[i];          // Guardar posição antiga
    a[i].x += dx;         // Mover para nova posição
}

// Se o movimento for inválido, desfazer
if (!check()) {
    for (int i = 0; i < 4; i++) {
        a[i] = b[i];     // Voltar para posição anterior
    }
}
```

A estratégia "Tentar e Desfazer":

1. Salvar a posição atual em **b**
2. Mover a peça para a nova posição em **a**
3. Verificar se a nova posição é válida
4. Se não for válida, copiar **b** de volta para **a**

## 3. Rotação - A Parte Mais Matemática

A rotação é baseada em matemática de transformação de coordenadas. Giramos cada bloco 90 graus ao redor do segundo bloco da peça:

```
if (rotate) {
    Point p = a[1]; // Centro de rotação (segundo bloco)

    for (int i = 0; i < 4; i++) {
        // Calcular posição relativa ao centro
        int x = a[i].y - p.y;
```

```

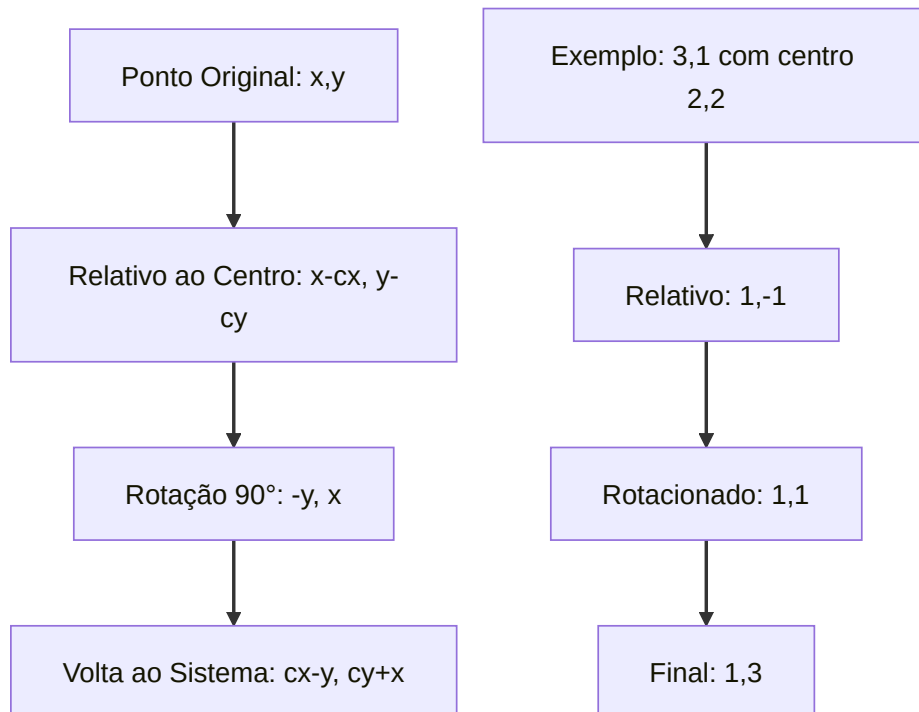
    int y = a[i].x - p.x;

    // Aplicar rotação de 90 graus
    a[i].x = p.x - x;
    a[i].y = p.y + y;
}

// Se a rotação for inválida, desfazer
if (!check()) {
    for (int i = 0; i < 4; i++) {
        a[i] = b[i];
    }
}
}

```

Como funciona a rotação matemática:



#### 4. Queda Automática - O Tick do Jogo

As peças caem sozinhas seguindo um cronômetro:

```

float timer = 0, delay = 0.3; // 0.3 segundos entre cada queda

if (timer > delay) {
    // Salvar posição atual
    for (int i = 0; i < 4; i++) {

```

```

        b[i] = a[i];
        a[i].y += 1;    // Mover para baixo
    }

    // Se não conseguir mover para baixo, fixar a peça
    if (!check()) {
        for (int i = 0; i < 4; i++) {
            field[b[i].y][b[i].x] = colorNum; // Fixar no campo
        }

        // Gerar nova peça
        colorNum = 1 + rand() % 7;
        int n = rand() % 7;
        for (int i = 0; i < 4; i++) {
            a[i].x = figures[n][i] % 2;
            a[i].y = figures[n][i] / 2;
        }
    }

    timer = 0; // Resetar cronômetro
}

```

## A Lógica de Eliminar Linhas

### Detectando Linhas Completas

A parte mais inteligente do Tetris é como removemos linhas completas. Usamos um algoritmo de "compactação":

```

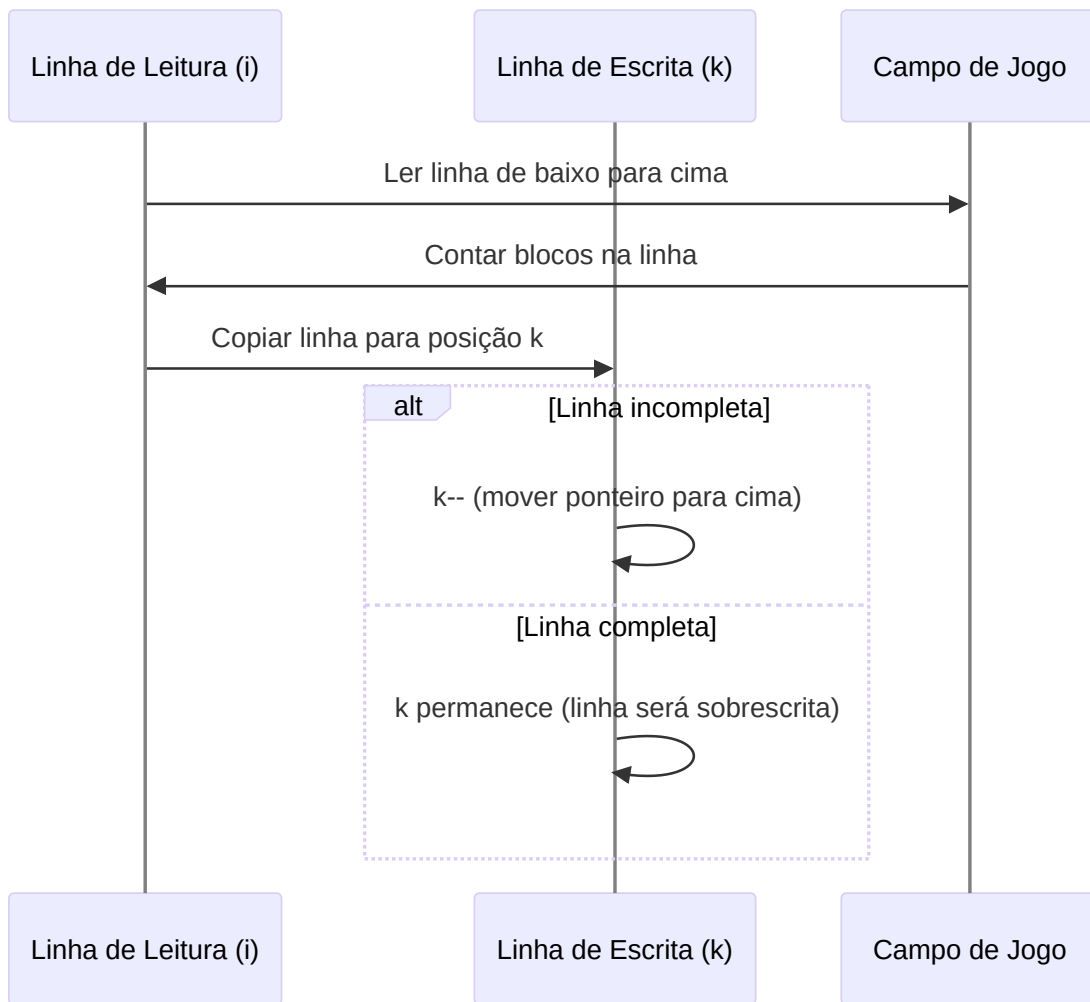
int k = M - 1; // Ponteiro para onde escrever
for (int i = M - 1; i > 0; i--) { // Ler de baixo para cima
    int count = 0;

    // Contar quantos blocos há nesta linha
    for (int j = 0; j < N; j++) {
        if (field[i][j]) count++;
        field[k][j] = field[i][j]; // Copiar linha
    }

    // Se a linha não está completa, manter ela
    if (count < N) k--;
}

```

### Como Funciona o Algoritmo



**Por que isso funciona:**

1. Começamos de baixo para cima
2. Copiamos cada linha para a posição  $k$
3. Se a linha não está completa, movemos  $k$  para cima
4. Se a linha está completa,  $k$  não se move (linha será sobrescrita)
5. Linhas completas "desaparecem" naturalmente

**Exemplo prático:**

Antes:

```

██████████
██████████
██████████
  
```

Depois:

```

██████████
██████████
██████████
  
```



## Gerando Peças Aleatórias

### Como Criar Peças Novas

Quando uma peça é fixada, criamos uma nova peça aleatória:

```
colorNum = 1 + rand() % 7;    // Cor aleatória de 1 a 7
int n = rand() % 7;           // Forma aleatória de 0 a 6

for (int i = 0; i < 4; i++) {
    a[i].x = figures[n][i] % 2; // Posição X do bloco
    a[i].y = figures[n][i] / 2; // Posição Y do bloco
}
```

### Por que Começar no Topo?

As peças novas sempre aparecem no topo da tela porque:

- figures[n][i] representa posições de 0 a 15
- figures[n][i] / 2 dá valores de 0 a 7 (linhas do topo)
- Isso garante que as peças sempre começam visíveis

## O Sistema de Cores

### Cada Peça Tem Sua Cor

```
int colorNum = 1; // Cor da peça atual (1 a 7)
```

Mapeamento de cores:

- 0: Vazio (preto)
- 1-7: Diferentes cores para cada tipo de peça

### Como Desenhar com Cores

```
// Para peças já fixadas no campo
s.setTextureRect(IntRect(field[i][j] * 18, 0, 18, 18));
```

```
// Para a peça atual em movimento  
s.setTextureRect(IntRect(colorNum * 18, 0, 18, 18));
```

Cada cor é uma seção de 18x18 pixels na textura.

## Otimizações Inteligentes

### Por que Usar Arrays de Tamanho Fixo?

```
Point a[4], b[4]; // Sempre exatamente 4 pontos
```

#### Vantagens:

- Cada tetrominó tem exatamente 4 blocos
- Arrays de tamanho fixo são mais rápidos
- Menos uso de memória
- Código mais simples

### Por que Não Usar Vetores Dinâmicos?

Para um jogo simples como Tetris:

- A complexidade extra não vale a pena
- Arrays fixos são mais eficientes
- O código fica mais fácil de entender

## A Matemática do Timing

### Controlando a Velocidade

```
float timer = 0, delay = 0.3;
```

#### Como funciona:

- `timer` acumula o tempo passado
- `delay` define quantos segundos entre cada queda
- Quando `timer > delay`, a peça desce um nível

### Calculando FPS

Se o jogo roda a 60 FPS:

- delay = 0.3 segundos
- Peça cai a cada  $0.3 \times 60 = 18$  frames
- Isso dá tempo suficiente para o jogador pensar e agir

## Detecção de Game Over

### Quando o Jogo Termina?

O jogo termina quando uma nova peça não consegue ser colocada no topo:

```
// Após gerar nova peça
if (!check()) {
    // Game Over - peça não cabe no topo
}
```

Embora o código atual não implemente explicitamente o game over, a lógica está lá: se `check()` retornar false para uma peça recém-criada, significa que o campo está cheio.

## Conceitos Avançados

### Por que Usar Coordenadas Relativas?

Quando geramos peças, usamos coordenadas relativas (0-15) que depois convertemos para coordenadas absolutas da tela. Isso permite:

1. **Fácil mudança de tamanho:** Alterar o tamanho dos blocos não quebra o jogo
2. **Rotação simples:** Matemática de rotação funciona melhor com coordenadas relativas
3. **Reutilização:** O mesmo código serve para diferentes posições na tela

### A Beleza da Simplicidade

O Tetris prova que jogos incríveis podem ter código simples:

- Apenas 154 linhas de código
- Usa conceitos básicos: arrays, loops, condicionais
- Sem classes complexas ou padrões de design rebuscados
- Foca na lógica do jogo, não em arquitetura

## Possíveis Melhorias

### 1. Sistema de Pontuação

```
int score = 0;
int linesCleared = 0;

// Após eliminar linha
score += 100 * level;
linesCleared++;
```

### 2. Níveis de Dificuldade

```
int level = 1;
delay = 0.5 - (level * 0.05); // Fica mais rápido a cada nível
```

### 3. Previsão da Próxima Peça

```
int nextPiece = rand() % 7;
// Mostrar nextPiece na interface
```

### 4. Sistema de Pausa

```
bool paused = false;
if (Keyboard::isKeyPressed(Keyboard::P)) paused = !paused;
```

## Por que este Código Funciona Tão Bem

### 1. Separação Clara de Responsabilidades

- `check()`: Valida movimentos
- Loop principal: Gerencia input e timing
- Algoritmo de linha: Remove linhas completas

### 2. Uso Inteligente de Dados

- Arrays 2D para o campo
- Arrays 1D para peças
- Números simples para representar formas



### 3. Algoritmos Eficientes

- Verificação de colisão:  $O(1)$  por bloco
- Remoção de linhas:  $O(n)$  onde  $n$  é o número de linhas
- Rotação:  $O(1)$  por bloco

### 4. Matemática Simples mas Poderosa

- Módulo e divisão para conversão de coordenadas
- Rotação usando transformação linear
- Timing baseado em acumulação de tempo

## O Tetris Como Ferramenta de Aprendizado

Este jogo é perfeito para iniciantes porque ensina:

1. **Arrays bidimensionais:** O campo de jogo
2. **Estruturas de dados:** Points para representar blocos
3. **Algoritmos:** Detecção de colisão, remoção de linhas
4. **Matemática básica:** Rotação, conversão de coordenadas
5. **Game loop:** Timing, input, renderização
6. **Lógica booleana:** Verificações de validade

Cada conceito é usado de forma prática e imediata, tornando o aprendizado mais efetivo do que estudar teoria abstrata.

## Conclusão

O Tetris é um exemplo perfeito de como um jogo simples pode ensinar conceitos profundos de programação. Sua beleza está na simplicidade elegante: com menos de 200 linhas de código, temos um jogo completo e divertido que demonstra arrays, algoritmos, matemática e lógica de jogos.

O código não é apenas funcional - é educativo. Cada linha tem um propósito claro, cada algoritmo resolve um problema específico, e o resultado final é um jogo que diverte e ensina ao mesmo tempo.

# Doodle Jump

Este tutorial ensina como criar o jogo Doodle Jump do zero usando C++ e SFML. Vamos começar com conceitos básicos e construir o conhecimento passo a passo, explicando cada parte de forma clara e detalhada.

## O que é Doodle Jump

Imagine um jogo onde você controla um pequeno personagem que precisa pular de uma plataforma para outra, tentando subir o mais alto possível. É como pular de degrau em degrau de uma escada infinita, mas com algumas regras especiais:

- O personagem sempre cai devido à gravidade (como na vida real)
- Ele só pode se mover para esquerda e direita
- Quando toca uma plataforma enquanto está caindo, automaticamente pula para cima
- Se cair muito para baixo, o jogo termina
- O objetivo é alcançar a maior altura possível

Este jogo nos permite aprender vários conceitos importantes de programação de jogos de forma simples e divertida.

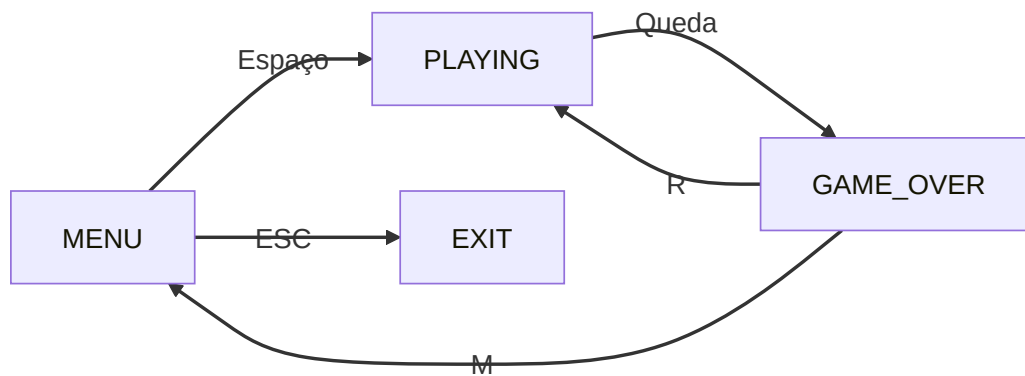
## Como Organizar um Jogo

### Estados do Jogo - Diferentes Telas

Antes de começar a programar, precisamos pensar em como organizar nosso jogo. Todo jogo tem diferentes "telas" ou "estados". Por exemplo:

- **Menu:** A tela inicial onde o jogador decide se quer jogar
- **Jogando:** Quando o jogo está realmente acontecendo
- **Game Over:** Quando o jogador perde e vê sua pontuação

Chamamos isso de "estados do jogo". É como ter diferentes salas em uma casa - você só pode estar em uma sala por vez, mas pode se mover entre elas.



Este diagrama mostra como o jogador navega entre as telas:

- Do **Menu**, apertar Espaço leva para o jogo
- Durante o **Jogo**, se o jogador cair, vai para Game Over
- No **Game Over**, pode apertar R para jogar de novo ou M para voltar ao menu

Para implementar isso no código, usamos algo chamado "enum" - que é uma forma de dar nomes para números:

```
enum GameState {  
    MENU,          // Valor 0 - Tela inicial do jogo  
    PLAYING,       // Valor 1 - Quando estamos jogando  
    GAME_OVER      // Valor 2 - Quando o jogo termina  
};
```

## Guardando Informações - Variáveis e Estruturas

Em qualquer jogo, precisamos guardar informações. Por exemplo, onde está o jogador? Onde estão as plataformas? Qual é a pontuação atual?

### Posição das Plataformas

Para cada plataforma, precisamos saber sua posição na tela. Uma posição tem duas coordenadas: X (horizontal) e Y (vertical). Criamos uma estrutura para isso:

```
struct point {  
    int x, y;    // x = posição horizontal, y = posição vertical  
};
```

Pense nisso como um endereço: "A plataforma está na posição X=100, Y=200".

### Informações do Jogador

Para o jogador, precisamos guardar várias informações importantes:

```
int x = 100, y = 100; // Onde o jogador está na tela
int h = 200;          // Uma altura especial (explicaremos depois)
float dx = 0, dy = 0; // Velocidade do jogador
int score = 0;        // Pontos que o jogador fez
int height = 0;       // Maior altura que o jogador alcançou
```

Vamos entender cada uma:

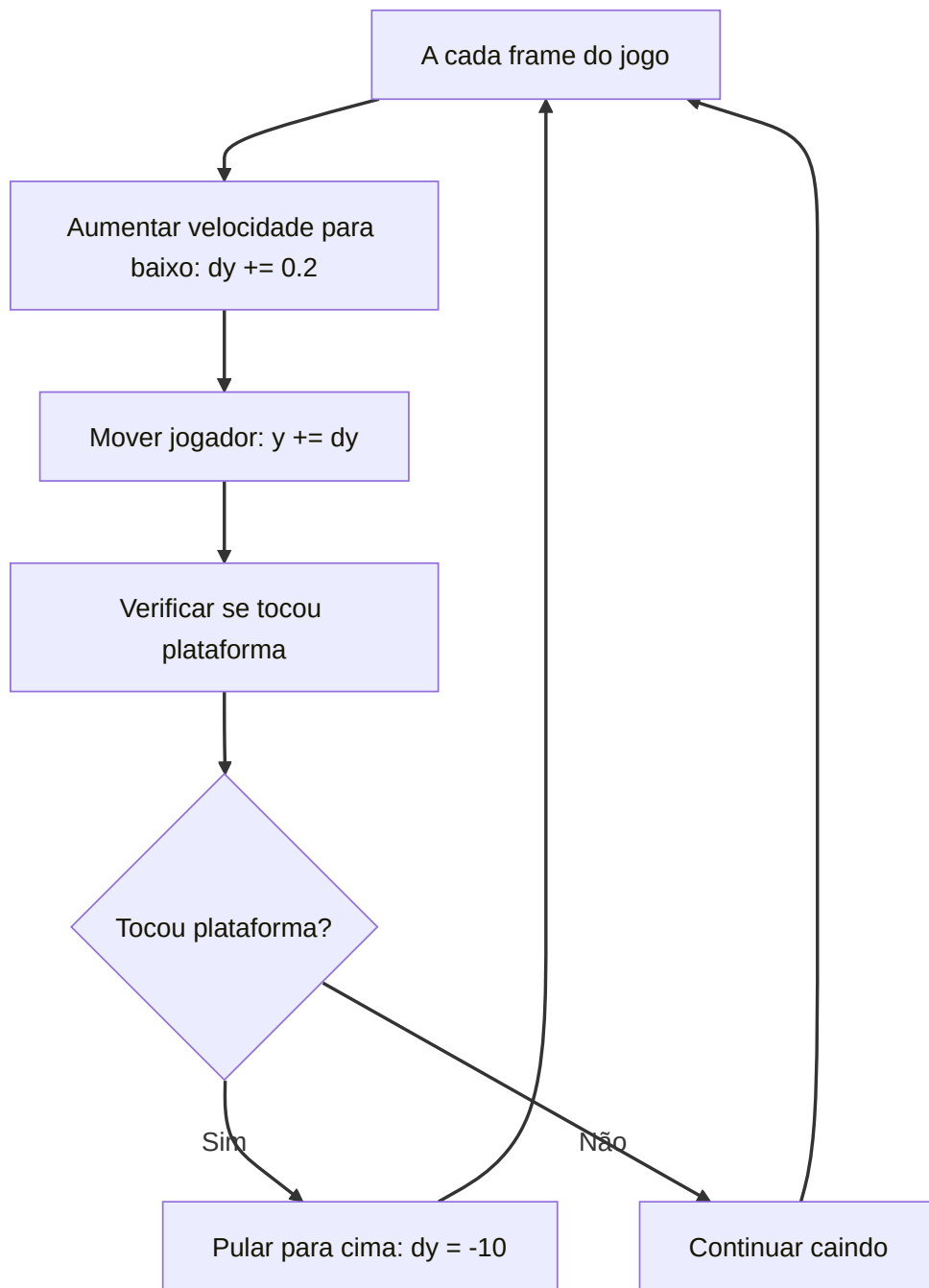
- **x, y:** A posição do jogador na tela (como coordenadas de um mapa)
- **dx, dy:** A velocidade do jogador (dx = velocidade horizontal, dy = velocidade vertical)
- **h:** Uma altura de referência especial que usamos para o sistema de câmera
- **score:** Os pontos que o jogador conquistou
- **height:** A maior altura que o jogador já alcançou no jogo

## As Principais Mecânicas do Jogo

### Como Funciona a Gravidade

Na vida real, quando você pula, a gravidade te puxa para baixo. No nosso jogo, precisamos simular essa gravidade de forma simples.

Imagine a gravidade como uma força que está sempre puxando o jogador para baixo. A cada momento do jogo (a cada "frame"), a gravidade faz o jogador cair um pouco mais rápido.



Vamos entender isso passo a passo:

```
dy += 0.2; // A cada frame, o jogador cai um pouco mais rápido
y += dy;   // Mover o jogador baseado na velocidade atual
```

Como funciona:

- **dy** é a velocidade vertical do jogador
- Quando **dy** é negativo (exemplo: -10), o jogador se move para cima

- Quando **dy** é positivo (exemplo: +5), o jogador se move para baixo
- A gravidade sempre adiciona +0.2 ao **dy**, fazendo o jogador cair mais rápido
- Quando o jogador toca uma plataforma, definimos **dy = -10**, fazendo ele pular para cima

## Movimento Horizontal - Esquerda e Direita

O jogador pode se mover para esquerda e direita usando as setas do teclado. Mas há um truque especial: quando o jogador sai de um lado da tela, ele aparece do outro lado.

Imagine que a tela é como um cilindro - se você andar para a direita e sair da tela, você aparece do lado esquerdo. Isso cria a sensação de um mundo infinito.

```
if (Keyboard::isKeyPressed(Keyboard::Right)) {
    x += 3;                // Move 3 pixels para a direita
    if (x > 400) x = -50;  // Se saiu da direita, aparece na esquerda
}
if (Keyboard::isKeyPressed(Keyboard::Left)) {
    x -= 3;                // Move 3 pixels para a esquerda
    if (x < -50) x = 400;  // Se saiu da esquerda, aparece na direita
}
```

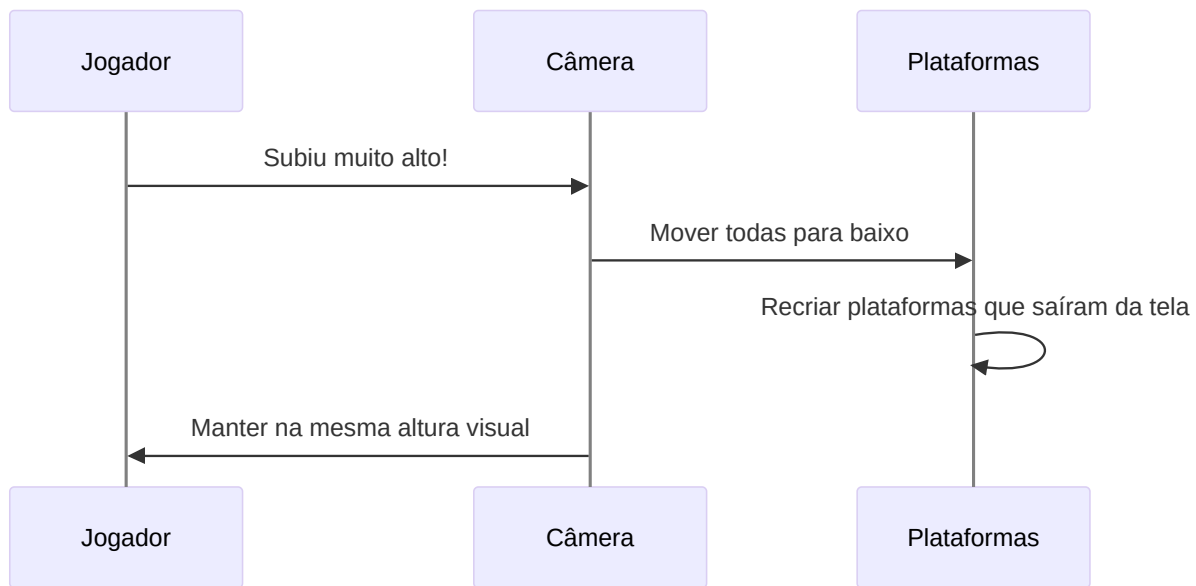
### Por que -50 e 400?

- A tela tem 400 pixels de largura (de 0 a 400)
- Usamos -50 e 400 para criar uma transição suave
- O jogador desaparece gradualmente de um lado antes de aparecer do outro

## O Truque da Câmera - A Parte Mais Inteligente

Esta é a parte mais interessante do jogo. Em vez de fazer o jogador subir na tela quando ele pula alto, fazemos o contrário: mantemos o jogador no mesmo lugar e movemos todo o mundo para baixo!

Imagine que você está em uma esteira rolante que se move para baixo. Você está sempre na mesma posição na esteira, mas o mundo ao seu redor está se movendo.



Como isso funciona no código:

```
if (y < h) { // Se o jogador subiu acima da altura de referência
    int heightGain = h - y; // Calcular quanto subiu
    height += heightGain; // Contar para estatísticas
    score += heightGain / 5; // Dar pontos por subir

    // Mover TODAS as plataformas para baixo
    for (int i = 0; i < 10; i++) {
        plat[i].y = plat[i].y - dy;

        // Se uma plataforma saiu da tela por baixo, criar uma nova no topo
        if (plat[i].y > 533) {
            plat[i].y = -50; // Nova posição no topo
            plat[i].x = rand() % 332; // Posição horizontal aleatória
        }
    }
    y = h; // Colocar o jogador de volta na altura de referência
}
```

Por que fazer assim?

- O jogador sempre fica visível na tela
- Podemos criar plataformas infinitamente
- É mais fácil de programar
- O jogo nunca "acaba" - sempre há mais plataformas aparecendo

## Como Detectar se o Jogador Tocou uma Plataforma

Para saber se o jogador tocou uma plataforma, precisamos verificar se eles estão "se sobrepondo" na tela. É como verificar se dois retângulos estão se tocando.

Mas há uma regra especial: só detectamos a colisão quando o jogador está **caindo** (não quando está subindo). Isso permite que o jogador passe através das plataformas quando está subindo, mas "aterrisse" nelas quando está descendo.

```
for (int i = 0; i < 10; i++) { // Verificar todas as 10 plataformas
    if ((x + 25 > plat[i].x) && // Jogador não está muito à
        esquerda
        (x + 25 < plat[i].x + 68) && // Jogador não está muito à
        direita
        (y + 70 > plat[i].y) && // Jogador não está muito acima
        (y + 70 < plat[i].y + 14) && // Jogador não está muito abaixo
        (dy > 0)) { // Jogador está caindo (não
        subindo)

        dy = -10; // Fazer o jogador pular para cima
        score += 10; // Dar pontos por conseguir pular
    }
}
```

### Entendendo as condições:

- **x + 25:** Usamos x + 25 porque queremos verificar o centro do jogador
- **plat[i].x + 68:** 68 é a largura da plataforma
- **y + 70:** 70 é aproximadamente a altura do jogador
- **plat[i].y + 14:** 14 é a altura da plataforma
- **dy > 0:** Só detecta colisão quando o jogador está caindo

## Por que só quando está caindo?

- Se o jogador está subindo, ele deve passar através da plataforma
- Se o jogador está descendo, ele deve "aterriçar" na plataforma
- Isso evita que o jogador fique "grudado" na plataforma

# A Matemática Por Trás do Jogo



## Calculando a Altura dos Pulos

Vamos descobrir algumas coisas interessantes sobre o nosso jogo usando matemática simples.

No nosso jogo:

- A gravidade adiciona 0.2 à velocidade a cada frame
- Quando o jogador pula, sua velocidade inicial é -10

### Qual é a altura máxima que o jogador pode alcançar?

Podemos calcular isso! Quando o jogador pula, ele começa com velocidade -10 e a gravidade vai diminuindo essa velocidade até chegar a 0 (quando ele para de subir).

$$\begin{aligned}\text{Altura máxima} &= (\text{velocidade inicial})^2 \div (2 \times \text{gravidade}) \\ \text{Altura máxima} &= 10^2 \div (2 \times 0.2) = 100 \div 0.4 = 250 \text{ pixels}\end{aligned}$$

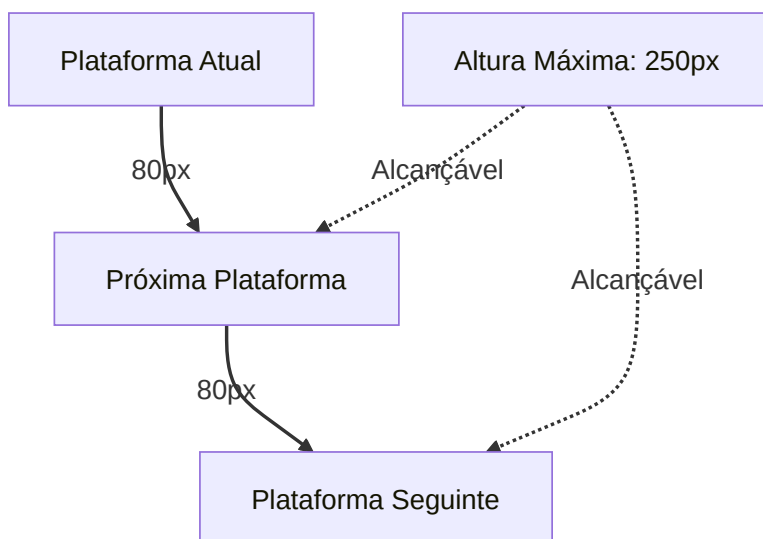
### Quanto tempo o jogador fica no ar?

$$\begin{aligned}\text{Tempo para subir} &= \text{velocidade inicial} \div \text{gravidade} = 10 \div 0.2 = 50 \text{ frames} \\ \text{Tempo total no ar} &= 2 \times \text{tempo para subir} = 100 \text{ frames}\end{aligned}$$

Se o jogo roda a 60 fps, isso significa que cada pulo dura cerca de 1.67 segundos.

## Por que as Plataformas Estão a 80 Pixels de Distância?

As plataformas estão espaçadas de 80 pixels verticalmente. Como o jogador pode pular até 250 pixels de altura, ele sempre consegue alcançar as próximas plataformas. Isso mantém o jogo desafiador mas sempre possível de jogar.



## Como Criar Plataformas Infinitas

Uma das partes mais legais do jogo é que as plataformas nunca acabam. Isso é chamado de "geração procedural" - o computador cria novo conteúdo automaticamente conforme você joga.

## Criando as Primeiras Plataformas

Quando o jogo começa, criamos 10 plataformas:

```
for (int i = 0; i < 10; i++) {  
    plat[i].x = rand() % 332;    // Posição horizontal aleatória  
    plat[i].y = i * 80 + 100;    // Espaçamento vertical de 80 pixels  
}
```

### Por que usar 332?

- Nossa tela tem 400 pixels de largura
- Cada plataforma tem 68 pixels de largura
- Para a plataforma caber completamente na tela:  $400 - 68 = 332$
- Então podemos colocar a plataforma em qualquer posição de 0 a 332

## Reciclando Plataformas

Quando uma plataforma sai da parte de baixo da tela, não a jogamos fora. Em vez disso, a "reciclamos" criando uma nova plataforma no topo:

```
if (plat[i].y > 533) {          // Se a plataforma saiu da tela  
    plat[i].y = -50;            // Colocar no topo da tela  
    plat[i].x = rand() % 332;   // Nova posição horizontal aleatória  
}
```

Isso significa que sempre temos exatamente 10 plataformas na tela, mas elas estão sempre mudando de posição.

## Como Funciona a Pontuação

O jogo tem três formas de ganhar pontos:

### 1. Pontos por Subir

```
score += heightGain / 5; // 1 ponto a cada 5 pixels que subir
```

Conforme você sobe no jogo, ganha pontos automaticamente. Quanto mais alto, mais pontos!

### 2. Pontos por Pular em Plataformas

```
score += 10; // 10 pontos cada vez que toca uma plataforma
```

Cada vez que você consegue pular em uma plataforma, ganha 10 pontos extras.

### 3. Bônus Especiais

```
if (height % 1000 == 0 && height > 0) {  
    score += 500; // 500 pontos a cada 1000 unidades de altura  
}
```

A cada 1000 unidades de altura, você ganha um bônus especial de 500 pontos!

## Melhorando a Aparência do Texto

### Texto com Contorno

Para que o texto seja sempre visível (independente da cor do fundo), criamos uma função especial que desenha um contorno ao redor das letras:

```
void drawTextWithOutline(RenderWindow& window, Text& text, Color outlineColor)  
{  
    Vector2f originalPos = text.getPosition();  
    Color originalColor = text.getFillColor();  
  
    // Desenhar "sombra" do texto em 8 posições ao redor  
    text.setFillColor(outlineColor);  
    for (int dx = -1; dx <= 1; dx++) {  
        for (int dy = -1; dy <= 1; dy++) {  
            if (dx != 0 || dy != 0) {  
                text.setPosition(originalPos.x + dx, originalPos.y + dy);  
                window.draw(text);  
            }  
        }  
    }  
  
    // Desenhar o texto principal por cima  
    text.setFillColor(originalColor);  
    text.setPosition(originalPos);  
    window.draw(text);  
}
```

Esta função desenha o texto em 8 posições ligeiramente diferentes ao redor da posição original, criando um efeito de contorno que torna o texto sempre legível.

## **Por que o Jogo Funciona Bem**

### **Usando Apenas 10 Plataformas**

O jogo só precisa de 10 plataformas ativas ao mesmo tempo. Isso é inteligente porque:

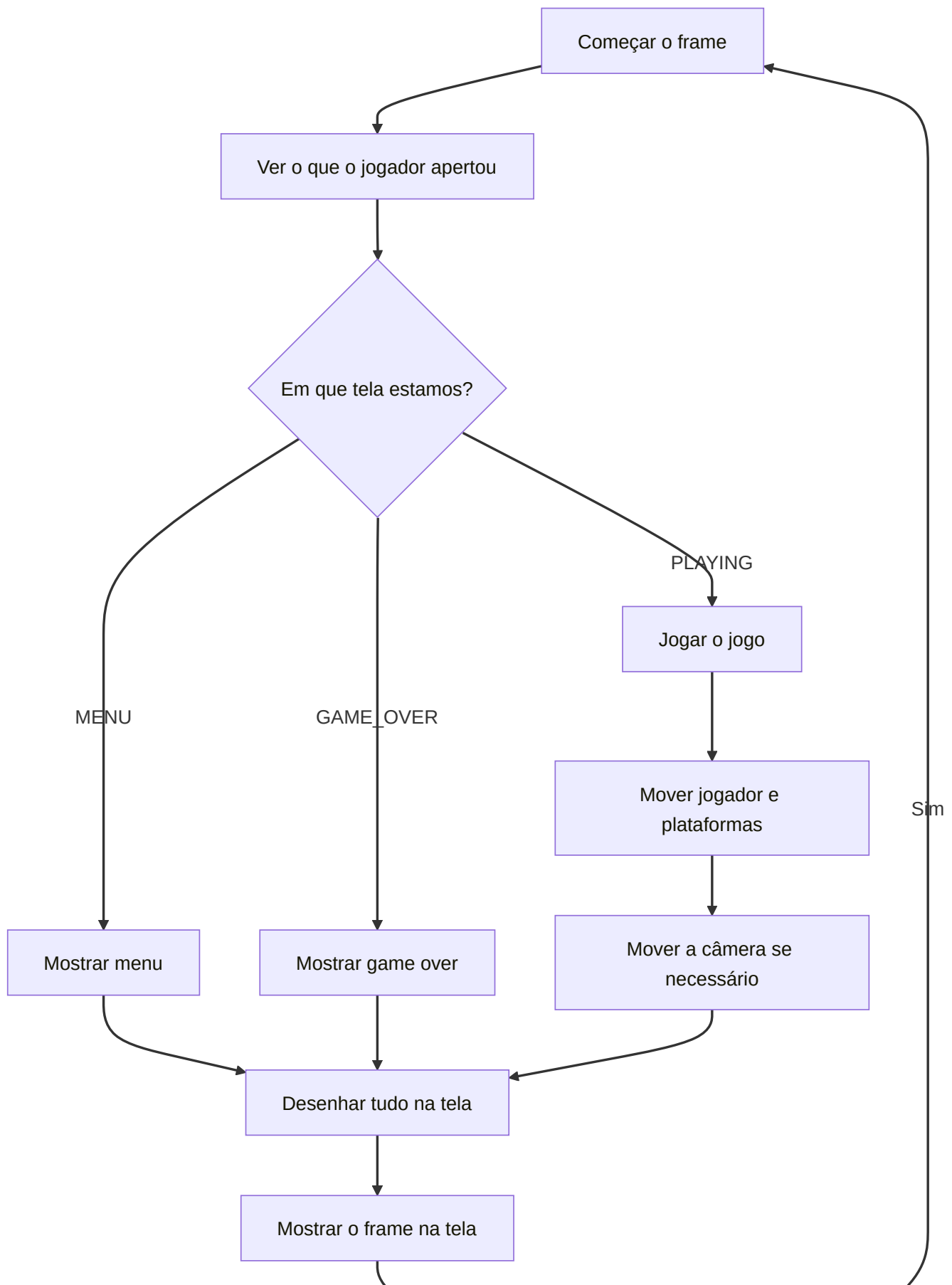
- Economiza memória do computador
- O jogo funciona sempre na mesma velocidade
- É mais fácil de programar e debugar

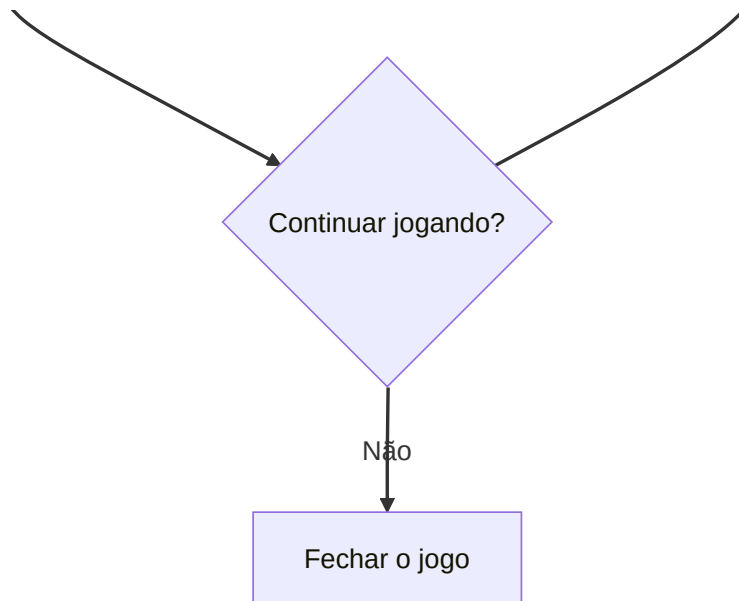
### **Verificações Inteligentes**

Só verificamos colisões quando o jogador está caindo, não quando está subindo. Isso torna o jogo mais rápido e evita problemas.

## **Como o Jogo Funciona - O Loop Principal**

Todo jogo tem um "loop principal" - um ciclo que se repete muitas vezes por segundo. A cada repetição (chamada de "frame"), o jogo:





## O que Cada Estado Faz

- **MENU:** Mostra o título do jogo e espera o jogador apertar Espaço para começar
- **PLAYING:** Roda toda a lógica do jogo (física, colisões, pontuação)
- **GAME\_OVER:** Mostra a pontuação final e permite reiniciar ou voltar ao menu

## Como Executar o Jogo

### Compilar

```
cd build  
make doodle_jump
```

### Jogar

```
make run_doodle_jump
```

### Arquivos Necessários

O jogo precisa destes arquivos para funcionar:

- `images/background.png`: A imagem de fundo
- `images/platform.png`: A imagem da plataforma
- `images/doodle.png`: A imagem do personagem
- `fonts/Carlito-Regular.ttf`: A fonte para os textos

# Ideias para Melhorar o Jogo

## Novas Mecânicas

- Power-ups como pulo duplo ou jetpack
- Plataformas especiais (que se movem, quebram, ou dão super pulo)
- Inimigos para evitar
- Efeitos sonoros e música

## Melhorias Técnicas

- Mais plataformas na tela
- Gráficos mais bonitos
- Animações suaves
- Sistema de save para lembrar da maior pontuação

## Código do Jogo Completo

- Repo (<https://github.com/mrpunkdasilva/16Games-in-Cpp/tree/main/02%20%20Doodle%20Jump>)

## Ver código

```
#include <SFML/Graphics.hpp>
#include <time.h>
#include <iostream>
#include <string>

using namespace sf;

struct point {
    int x, y;
};

enum GameState {
    MENU,
    PLAYING,
    GAME_OVER
};
```

```

// Função para desenhar texto com borda para melhor visibilidade
void drawTextWithOutline(RenderWindow& window, Text& text, Color outlineColor
= Color::Black) {
    Vector2f originalPos = text.getPosition();
    Color originalColor = text.getFillColor();

    // Desenhar sombra/borda em várias posições
    text.setFillColor(outlineColor);
    for (int dx = -1; dx <= 1; dx++) {
        for (int dy = -1; dy <= 1; dy++) {
            if (dx != 0 || dy != 0) {
                text.setPosition(originalPos.x + dx, originalPos.y + dy);
                window.draw(text);
            }
        }
    }

    // Desenhar o texto principal
    text.setFillColor(originalColor);
    text.setPosition(originalPos);
    window.draw(text);
}

int main() {
    srand(time(0));

    RenderWindow app(VideoMode(400, 533), "Doodle Game!");
    app.setFramerateLimit(60);

    Texture t1,t2,t3;

    // Verificação de carregamento de imagens
    if (!t1.loadFromFile("images/background.png")) {
        std::cout << "Erro ao carregar background.png" << std::endl;
        return -1;
    }
    if (!t2.loadFromFile("images/platform.png")) {
        std::cout << "Erro ao carregar platform.png" << std::endl;
        return -1;
    }
    if (!t3.loadFromFile("images/doodle.png")) {
        std::cout << "Erro ao carregar doodle.png" << std::endl;
        return -1;
    }
}

```



```

}

Sprite sBackground(t1), sPlat(t2), sPers(t3);

point plat[20];

// Inicialização das plataformas com espaçamento adequado
for (int i = 0; i < 10; i++) {
    plat[i].x = rand() % 332; // 400 - 68 (largura da plataforma)
    plat[i].y = i * 80 + 100; // Espaçamento vertical adequado
}

// Estado do jogo
GameState gameState = MENU;

// Variáveis do jogo
int x = 100, y = 100, h = 200;
float dx = 0, dy = 0;
int score = 0;
int highScore = 0;
int height = 0; // Altura máxima alcançada

// Configuração de fonte
Font font;
bool fontLoaded = false;

// Carrega fonte local do projeto (incluída para distribuição)
if (font.loadFromFile("fonts/Carlito-Regular.ttf")) {
    fontLoaded = true;
}

// Textos do menu
Text titleText, startText, quitText, instructionsText;
if (fontLoaded) {
    titleText.setFont(font);
    titleText.setCharacterSize(48);
    titleText.setFillColor(Color::Blue); // Azul para contraste com fundo
branco
    titleText.setString("DOODLE JUMP");
    titleText.setPosition(80, 100);

    startText.setFont(font);
    startText.setCharacterSize(32);
    startText.setFillColor(Color::Black); // Preto para contraste

```

```

startText.setString("Press SPACE to Start");
startText.setPosition(70, 250);

quitText.setFont(font);
quitText.setCharacterSize(24);
quitText.setFillColor(Color(64, 64, 64)); // Cinza escuro
quitText.setString("Press ESC to Quit");
quitText.setPosition(120, 350);

instructionsText.setFont(font);
instructionsText.setCharacterSize(20);
instructionsText.setFillColor(Color(0, 100, 0)); // Verde escuro
instructionsText.setString("Use LEFT/RIGHT arrows to move\nJump on
platforms to go higher!");
instructionsText.setPosition(40, 400);
}

// Textos do jogo
Text scoreText, heightText;
if (fontLoaded) {
    scoreText.setFont(font);
    scoreText.setCharacterSize(24);
    scoreText.setFillColor(Color::Black); // Preto para contraste
    scoreText.setPosition(10, 10);

    heightText.setFont(font);
    heightText.setCharacterSize(20);
    heightText.setFillColor(Color(0, 0, 150)); // Azul escuro
    heightText.setPosition(10, 40);
}

// Textos do game over
Text gameOverText, finalScoreText, highScoreText, restartText;
if (fontLoaded) {
    gameOverText.setFont(font);
    gameOverText.setCharacterSize(48);
    gameOverText.setFillColor(Color::Red); // Vermelho fica bom contra
fundo branco
    gameOverText.setString("GAME OVER");
    gameOverText.setPosition(90, 150);

    finalScoreText.setFont(font);
    finalScoreText.setCharacterSize(28);
    finalScoreText.setFillColor(Color::Black); // Preto para contraste

```

```

        finalScoreText.setPosition(100, 220);

        highScoreText.setFont(font);
        highScoreText.setCharacterSize(24);
        highScoreText.setFillColor(Color(150, 100, 0)); // Marrom/dourado
        escuro
        highScoreText.setPosition(80, 260);

        restartText.setFont(font);
        restartText.setCharacterSize(20);
        restartText.setFillColor(Color(0, 100, 100)); // Verde-azul escuro
        restartText.setString("Press R to Restart\nPress M for Menu");
        restartText.setPosition(110, 320);
    }

    while (app.isOpen()) {
        Event e;
        while (app.pollEvent(e)) {
            if (e.type == Event::Closed)
                app.close();

            if (e.type == Event::KeyPressed) {
                switch (gameState) {
                    case MENU:
                        if (e.key.code == Keyboard::Space) {
                            gameState = PLAYING;
                            // Reiniciar variáveis do jogo
                            x = 100;
                            y = 100;
                            h = 200;
                            dy = 0;
                            score = 0;
                            height = 0;

                            // Reiniciar plataformas
                            for (int i = 0; i < 10; i++) {
                                plat[i].x = rand() % 332;
                                plat[i].y = i * 80 + 100;
                            }
                        }
                        else if (e.key.code == Keyboard::Escape) {
                            app.close();
                        }
                    }
                break;
            }
        }
    }

```

```

principal
    case PLAYING:
        // Controles durante o jogo são tratados no loop

        break;

    case GAME_OVER:
        if (e.key.code == Keyboard::R) {
            gameState = PLAYING;
            // Reiniciar variáveis do jogo
            x = 100;
            y = 100;
            h = 200;
            dy = 0;
            score = 0;
            height = 0;

            // Reiniciar plataformas
            for (int i = 0; i < 10; i++) {
                plat[i].x = rand() % 332;
                plat[i].y = i * 80 + 100;
            }
        }
        else if (e.key.code == Keyboard::M) {
            gameState = MENU;
        }
        break;
    }
}

// Lógica do jogo
if (gameState == PLAYING) {
    // Controles horizontais com wrapping (movimento infinito)
    if (Keyboard::isKeyPressed(Keyboard::Right)) {
        x += 3;
        if (x > 400) x = -50; // Sai pela direita, aparece na esquerda
    }
    if (Keyboard::isKeyPressed(Keyboard::Left)) {
        x -= 3;
        if (x < -50) x = 400; // Sai pela esquerda, aparece na direita
    }

    // Aplicar gravidade

```

```

dy += 0.2;
y += dy;

// Se o jogador cair muito baixo, game over
if (y > 600) {
    gameState = GAME_OVER;
    // Atualizar high score
    if (score > highScore) {
        highScore = score;
    }
}

// Lógica de movimento do mundo (câmera)
if (y < h) {
    // Calcular altura alcançada
    int heightGain = h - y;
    height += heightGain;

    // Aumenta o score quando o jogador sobe
    score += heightGain / 5; // Pontos por altura

    // Bônus por altura alcançada
    if (height % 1000 == 0 && height > 0) {
        score += 500; // Bônus a cada 1000 unidades de altura
    }

    // Move todas as plataformas para baixo
    for (int i = 0; i < 10; i++) {
        plat[i].y = plat[i].y - dy;
        // Se a plataforma sair da tela por baixo, reposiciona no
topo
        if (plat[i].y > 533) {
            plat[i].y = -50; // Aparece no topo
            plat[i].x = rand() % 332; // Nova posição horizontal
        }
    }
    y = h; // Mantém o jogador na mesma altura visual
}

// Verificação de colisão corrigida
for (int i = 0; i < 10; i++) {
    // Verifica se o jogador está caindo (dy > 0) e colidindo com
a plataforma
    if ((x + 25 > plat[i].x) && (x + 25 < plat[i].x + 68) &&

```

```

        (y + 70 > plat[i].y) && (y + 70 < plat[i].y + 14) && (dy >
0)) {

        dy = -10; // Faz o jogador pular
        score += 10; // Pontos por pular na plataforma
    }
}

// Desenhando tudo
app.draw(sBackground);

switch (gameState) {
    case MENU:
        if (fontLoaded) {
            drawTextWithOutline(app, titleText, Color::White);
            drawTextWithOutline(app, startText, Color::White);
            drawTextWithOutline(app, quitText, Color::White);
            drawTextWithOutline(app, instructionsText, Color::White);

            // Mostrar high score no menu
            if (highScore > 0) {
                Text menuHighScore;
                menuHighScore.setFont(font);
                menuHighScore.setCharacterSize(20);
                menuHighScore.setFillColor(Color(150, 100, 0)); //
Marrom/dourado escuro
                menuHighScore.setString("High Score: " +
std::to_string(highScore));
                menuHighScore.setPosition(130, 300);
                drawTextWithOutline(app, menuHighScore, Color::White);
            }
        }
        break;

    case PLAYING:
        // Desenhando plataformas
        for (int i = 0; i < 10; i++) {
            sPlat.setPosition(plat[i].x, plat[i].y);
            app.draw(sPlat);
        }

        // Desenhando jogador
        sPers.setPosition(x, y);
        app.draw(sPers);

```

```

        // Desenhando UI do jogo
        if (fontLoaded) {
            scoreText.setString("Score: " + std::to_string(score));
            drawTextWithOutline(app, scoreText, Color::White);

            heightText.setString("Height: " + std::to_string(height));
            drawTextWithOutline(app, heightText, Color::White);
        }
        break;

    case GAME_OVER:
        // Desenhando estado final do jogo (sem movimento)
        for (int i = 0; i < 10; i++) {
            sPlat.setPosition(plat[i].x, plat[i].y);
            app.draw(sPlat);
        }

        sPers.setPosition(x, y);
        app.draw(sPers);

        // Desenhando UI do game over
        if (fontLoaded) {
            drawTextWithOutline(app, gameOverText, Color::White);

            finalScoreText.setString("Final Score: " +
std::to_string(score));
            drawTextWithOutline(app, finalScoreText, Color::White);

            highScoreText.setString("High Score: " +
std::to_string(highScore));
            drawTextWithOutline(app, highScoreText, Color::White);

            drawTextWithOutline(app, restartText, Color::White);
        }
        break;
    }

    app.display();
}

```

```
    return 0;  
}
```

## Conclusão

Parabéns! Você aprendeu como funciona um jogo completo. Doodle Jump pode parecer simples, mas ele ensina conceitos muito importantes:

- **Estados de jogo:** Como organizar diferentes telas
- **Física básica:** Como simular gravidade e movimento
- **Deteção de colisão:** Como saber quando objetos se tocam
- **Geração procedural:** Como criar conteúdo infinito
- **Sistema de câmera:** Como fazer o mundo se mover em vez do jogador

Estes conceitos são usados em jogos muito mais complexos. Agora que você entende como funciona, pode experimentar modificar os valores no código para ver o que acontece, ou até mesmo criar suas próprias mecânicas!



# Arkanoid

Este tutorial ensina como criar o jogo Arkanoid do zero usando C++ e SFML. Vamos começar com conceitos básicos e construir o conhecimento passo a passo, explicando cada parte de forma clara e detalhada.

## O que é Arkanoid

Imagine um jogo onde você controla uma raquete na parte inferior da tela, e precisa usar uma bola para destruir todos os blocos coloridos que estão organizados na parte superior. É como se você estivesse jogando tênis, mas em vez de rebater a bola para o outro lado, você a usa para quebrar tijolos em uma parede:

- Uma bola ricocheia pela tela seguindo leis da física
- Você controla uma raquete que pode se mover para esquerda e direita
- A bola deve rebater na raquete para não cair fora da tela
- Cada bloco destruído dá pontos
- O objetivo é destruir todos os blocos sem deixar a bola cair

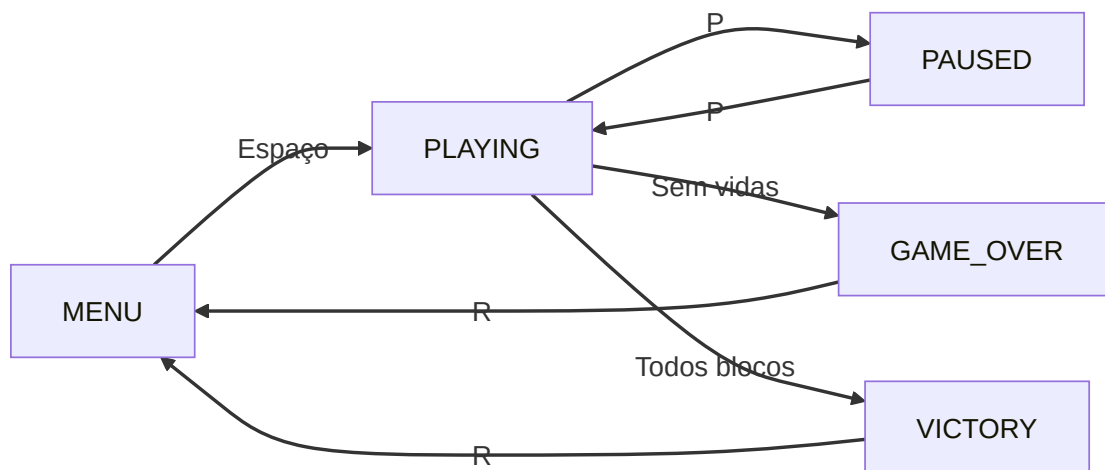
Este jogo nos permite aprender vários conceitos importantes de programação de jogos, incluindo física de colisões, programação orientada a objetos e gerenciamento de estados.

## Como Organizar um Jogo Complexo

### Estados do Jogo - Diferentes Telas

O Arkanoid tem mais estados que jogos simples, pois precisa gerenciar situações como pausa e vitória:

- **Menu:** A tela inicial com instruções
- **Jogando:** Quando o jogo está ativo
- **Pausado:** Quando o jogador pausa o jogo
- **Game Over:** Quando o jogador perde todas as vidas
- **Vitória:** Quando o jogador completa todos os níveis



No código, implementamos isso com um enum mais completo:

```
enum GameState {
    MENU,          // Tela inicial
    PLAYING,       // Jogando ativamente
    PAUSED,        // Jogo pausado
    GAME_OVER,     // Perdeu todas as vidas
    VICTORY        // Completou todos os níveis
};
```

## Programação Orientada a Objetos

Diferente de jogos simples, o Arkanoid usa classes para organizar melhor o código. Cada elemento importante do jogo tem sua própria classe:

### Classe Block - Representando os Blocos

```
class Block {
public:
    Sprite sprite;      // Como o bloco aparece na tela
    bool isDestroyed;   // Se foi destruído ou não
    int points;         // Quantos pontos vale

    Block() : isDestroyed(false), points(10) {} // Construtor

    void destroy() { isDestroyed = true; }      // Marcar como destruído
    FloatRect getBounds() const;               // Área de colisão
    void draw(RenderWindow& window);           // Desenhar na tela
};
```

Por que usar uma classe?

- Cada bloco tem suas próprias propriedades (posição, cor, se foi destruído)
- Podemos ter muitos blocos facilmente
- O código fica mais organizado e reutilizável

### Classe Ball - A Bola Física

```
class Ball {
public:
    Sprite sprite;          // Aparência da bola
    Vector2f velocity;      // Velocidade (direção e rapidez)
    float speed;            // Velocidade base

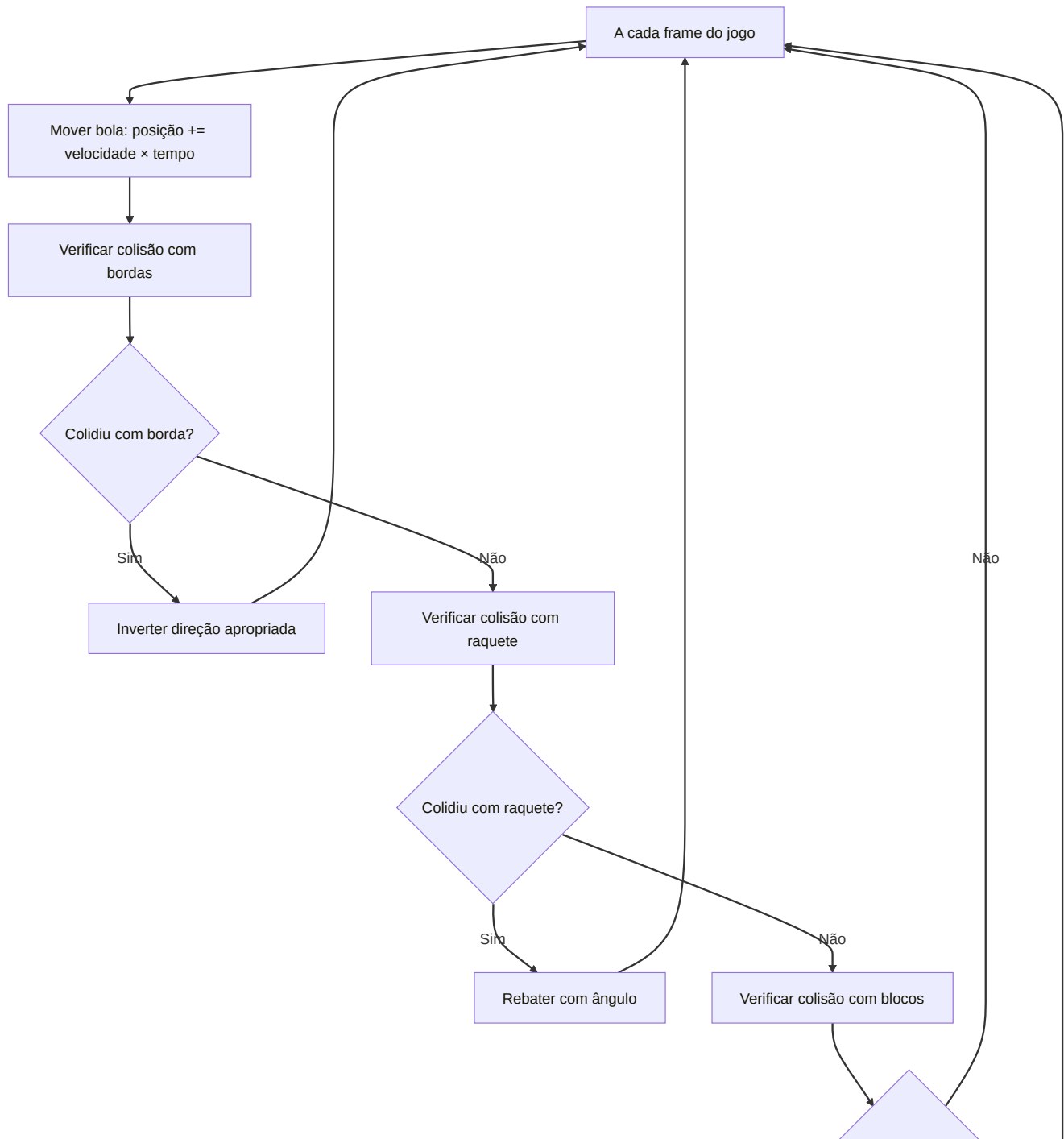
    void update(float deltaTime, const Vector2u& windowSize); // Atualizar
    posição
    void reverseX() { velocity.x = -velocity.x; }              // Inverter
    direção X
    void reverseY() { velocity.y = -velocity.y; }              // Inverter
    direção Y
    bool isOutOfBounds(const Vector2u& windowSize) const;      // Verificar se
    saiu da tela
};
```

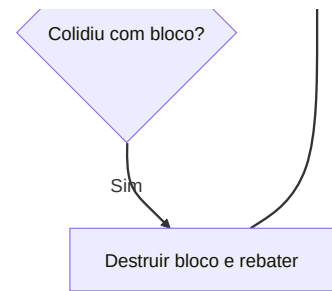
A bola é mais complexa porque precisa simular física realista.

## As Principais Mecânicas do Jogo

### Física da Bola - Movimento e Colisões

A bola do Arkanoid segue leis físicas simples. Ela se move em linha reta até colidir com algo, então muda de direção.





## Movimento Básico

```
void Ball::update(float deltaTime, const Vector2u& windowSize) {  
    Vector2f pos = getPosition();  
    pos += velocity * deltaTime; // Mover baseado na velocidade  
  
    // Colisão com bordas laterais e superior  
    if (pos.x <= 0 || pos.x + getBounds().width >= windowSize.x) {  
        velocity.x = -velocity.x; // Inverter direção horizontal  
    }  
    if (pos.y <= 0) {  
        velocity.y = -velocity.y; // Inverter direção vertical  
    }  
  
    sprite.setPosition(pos);  
}
```

### Entendendo o deltaTime:

- `deltaTime` é o tempo que passou desde o último frame
- Multiplicar por `deltaTime` faz o movimento ser suave independente da velocidade do computador
- Se o jogo roda a 60 FPS, `deltaTime` será aproximadamente 0.0167 segundos

## Controle Inteligente da Raquete

A raquete responde aos comandos do jogador, mas tem limitações realistas:

```
void Paddle::update(float deltaTime, const Vector2u& windowSize) {
    Vector2f pos = getPosition();

    // Movimento baseado nas teclas pressionadas
    if (Keyboard::isKeyPressed(Keyboard::Left) ||
        Keyboard::isKeyPressed(Keyboard::A)) {
        pos.x -= speed * deltaTime; // Mover para esquerda
    }
    if (Keyboard::isKeyPressed(Keyboard::Right) ||
        Keyboard::isKeyPressed(Keyboard::D)) {
        pos.x += speed * deltaTime; // Mover para direita
    }

    // Não permitir sair da tela
    pos.x = std::max(0.0f, std::min(pos.x, (float>windowSize.x -
        getBounds().width));
    sprite.setPosition(pos);
}
```

Por que usar `speed * deltaTime`?

- Garante movimento suave
- A raquete se move na mesma velocidade em qualquer computador
- Permite controle responsivo

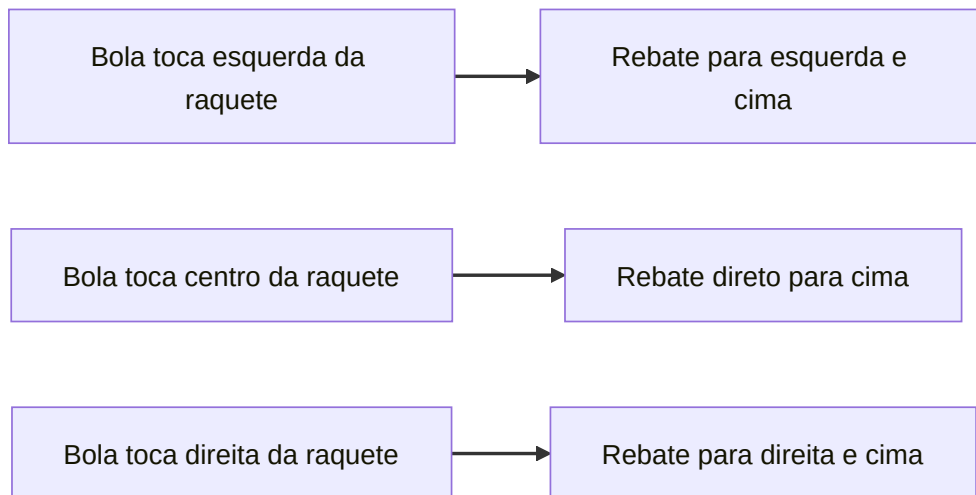
## O Truque do Rebote na Raquete

Uma das partes mais inteligentes do jogo é como a bola rebate na raquete. Não é apenas um rebote simples - o ângulo depende de onde a bola toca a raquete:

```
// Verificar colisão bola-raquete
if (ball.getBounds().intersects(paddle.getBounds())) {
    ball.reverseY(); // Sempre rebater para cima

    // Calcular ângulo baseado na posição da colisão
    float ballCenter = ball.getPosition().x + ball.getBounds().width / 2;
    float paddleCenter = paddle.getPosition().x + paddle.getBounds().width /
2;
    float offset = (ballCenter - paddleCenter) / (paddle.getBounds().width /
2);
```

```
// Ajustar velocidade horizontal baseada no offset
ball.velocity.x = ball.speed * offset * 0.75f;
ball.velocity.y = -std::abs(ball.velocity.y); // Sempre para cima
}
```



**Como funciona o cálculo:**

- **offset** varia de -1 (extrema esquerda) a +1 (extrema direita)
- **offset = 0** significa que a bola tocou o centro da raquete
- A velocidade horizontal é proporcional ao offset
- Multiplicamos por 0.75 para não deixar o rebote muito extremo

## Detecção de Colisão com Blocos

Detectar qual lado do bloco a bola tocou é crucial para um rebote realista:

```
for (auto& block : blocks) {
    if (!block.isDestroyed && ball.getBounds().intersects(block.getBounds()))
    {
        block.destroy(); // Destruir o bloco
        score += block.points; // Adicionar pontos

        // Determinar de que lado a bola colidiu
        Vector2f ballCenter = Vector2f(ball.getPosition().x +
ball.getBounds().width / 2,
ball.getPosition().y +
ball.getBounds().height / 2);
        Vector2f blockCenter = Vector2f(block.getBounds().left +
block.getBounds().width / 2,
block.getBounds().top +
```

```

block.getBounds().height / 2);

    float dx = std::abs(ballCenter.x - blockCenter.x); // Distância
horizontal
    float dy = std::abs(ballCenter.y - blockCenter.y); // Distância
vertical

    if (dx > dy) {
        ball.reverseX(); // Colidiu pela lateral
    } else {
        ball.reverseY(); // Colidiu por cima/baixo
    }

    break; // Só colidir com um bloco por frame
}
}

```

Por que esse método funciona?

- Se `dx > dy`, a bola está mais próxima da lateral do bloco
- Se `dy > dx`, a bola está mais próxima do topo/base do bloco
- Isso determina se devemos inverter a velocidade X ou Y

## Criando o Mundo do Jogo

### Geração Procedural de Blocos

O jogo cria uma grade organizada de blocos coloridos:

```

void createBlocks() {
    blocks.clear();
    const int rows = 8;      // 8 fileiras de blocos
    const int cols = 10;     // 10 colunas de blocos
    const float blockWidth = 60;
    const float blockHeight = 25;
    const float spacing = 5; // Espaço entre blocos

    // Centralizar a grade na tela
    const float startX = (800 - (cols * blockWidth + (cols - 1) * spacing)) /
2;
    const float startY = 50;

    for (int row = 0; row < rows; row++) {

```



```

    for (int col = 0; col < cols; col++) {
        Block block;

        // Calcular posição
        float x = startX + col * (blockWidth + spacing);
        float y = startY + row * (blockHeight + spacing);
        block.setPosition(x, y);

        // Blocos superiores valem mais pontos
        block.points = (rows - row) * 10;

        // Cores diferentes para cada fileira
        Color blockColor = blockColors[row % blockColors.size()];
        block.sprite.setColor(blockColor);

        blocks.push_back(block);
    }
}

```

Por que essa organização?

- **Centralizamos** a grade para ficar visualmente equilibrada
- **Blocos superiores** valem mais pontos (mais difíceis de alcançar)
- **Cores diferentes** tornam o jogo mais atrativo
- **Espaçamento uniforme** cria uma aparência profissional

## Sistema de Vidas e Progressão

O jogo implementa um sistema de vidas realista:

```

// Verificar se bola saiu da tela
if (ball.isOutOfBounds(window.getSize())) {
    lives--; // Perder uma vida
    if (lives <= 0) {
        gameState = GAME_OVER; // Fim de jogo
    } else {
        ball.reset(400, 300); // Reposicionar bola
    }
}

```

E um sistema de progressão por níveis:

```

// Verificar vitória (todos os blocos destruídos)
bool allDestroyed = true;
for (const auto& block : blocks) {
    if (!block.isDestroyed) {
        allDestroyed = false;
        break;
    }
}

if (allDestroyed) {
    level++;
    score += 1000 * level;    // Bônus por completar nível
    if (level <= 3) {        // Máximo 3 níveis
        createBlocks();      // Criar novos blocos
        ball.reset(400, 300); // Reposicionar bola
        ball.speed += 50;    // Aumentar dificuldade
    } else {
        gameState = VICTORY; // Vitória total!
    }
}
}

```

## Texturas Procedurais - Gráficos Sem Arquivos

Uma característica inteligente do jogo é criar gráficos automaticamente se não encontrar arquivos de imagem:

### Criando uma Bola Redonda

```

if (!ballTexture.loadFromFile("images/ball.png")) {
    // Criar textura de bola procedural
    Image ballImage;
    ballImage.create(20, 20, Color::White);

    for (int x = 0; x < 20; x++) {
        for (int y = 0; y < 20; y++) {
            int dx = x - 10; // Distância do centro
            int dy = y - 10;

            // Se está dentro do círculo (raio = 10)
            if (dx*dx + dy*dy <= 100) {
                ballImage.setPixel(x, y, Color::White);
            } else {
                ballImage.setPixel(x, y, Color::Transparent);
            }
        }
    }
}

```

```

        }
    }
}
ballTexture.loadFromImage(ballImage);
}

```

#### Como funciona a matemática:

- $dx*dx + dy*dy$  é a distância ao quadrado do centro
- Se for  $\leq 100$ , está dentro de um círculo de raio 10
- Pixels dentro do círculo ficam brancos, fora ficam transparentes

### Criando um Fundo com Estrelas

```

// Criar fundo procedural com gradiente e estrelas
Image bgImage;
bgImage.create(800, 600, Color::Black);

// Criar um gradiente do azul escuro para preto
for (int y = 0; y < 600; y++) {
    for (int x = 0; x < 800; x++) {
        float factor = (float)y / 600.0f; // 0 no topo, 1 na base
        Uint8 blue = (Uint8)(30 * (1.0f - factor));
        Uint8 green = (Uint8)(10 * (1.0f - factor));
        bgImage.setPixel(x, y, Color(0, green, blue));
    }
}

// Adicionar algumas "estrelas" aleatórias
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> xDist(0, 799);
std::uniform_int_distribution<> yDist(0, 399); // Apenas na parte superior
std::uniform_int_distribution<> brightDist(100, 255);

for (int i = 0; i < 100; i++) {
    int x = xDist(gen);
    int y = yDist(gen);
    Uint8 brightness = brightDist(gen);
    bgImage.setPixel(x, y, Color(brightness, brightness, brightness));
}

```

## Por que fazer isso?

- O jogo funciona mesmo sem arquivos de imagem
- Reduz dependências externas
- Permite personalização fácil
- Ensina como criar gráficos programaticamente

## Sistema de Pontuação Inteligente

O jogo tem múltiplas formas de ganhar pontos:

### 1. Pontos por Bloco Destruído

```
score += block.points; // Cada bloco tem valor diferente
```

Blocos superiores valem mais pontos pois são mais difíceis de alcançar.

### 2. Bônus por Completar Nível

```
score += 1000 * level; // Bônus aumenta a cada nível
```

Completar níveis mais altos dá bônus maiores.

### 3. Sistema de Vidas

```
int lives = 3; // Começar com 3 vidas
```

Cada vida perdida significa uma nova chance, mas a pontuação não reseta.

## Efeitos Visuais Avançados

### Mostrar Pontos Ganhados

Quando um bloco é destruído, o jogo mostra os pontos ganhos:

```
// Mostrar bonus de pontos
lastScoreGain = block.points;
showScoreBonus = true;
effectClock.restart();

// Na renderização:
if (showScoreBonus) {
    bonusText.setString("+" + std::to_string(lastScoreGain));
}
```

```

        bonusText.setPosition(ball.getPosition().x, ball.getPosition().y - 30);
        window.draw(bonusText);
    }

    // Esconder após 1 segundo
    if (showScoreBonus && effectClock.getElapsedTime().asSeconds() > 1.0f) {
        showScoreBonus = false;
    }

```

## Interface de Usuário Dinâmica

```

void updateUI() {
    std::stringstream ss;
    ss << "Score: " << score;
    scoreText.setString(ss.str());

    ss.str("");
    ss << "Lives: " << lives;
    livesText.setString(ss.str());

    ss.str("");
    ss << "Level: " << level;
    levelText.setString(ss.str());
}

```

## A Matemática Por Trás do Jogo

### Velocidade da Bola

A bola tem uma velocidade constante, mas sua direção muda:

```

float speed = 300.0f; // Pixels por segundo
Vector2f velocity = Vector2f(speed, -speed); // Diagonal inicial

```

Calculando a velocidade real:

```

Velocidade diagonal = sqrt(speed2 + speed2) = sqrt(2) × speed
Para speed = 300: velocidade real = 424.26 pixels/segundo

```

### Progressão de Dificuldade

A cada nível, a velocidade da bola aumenta:

```

ball.speed += 50; // +50 pixels/segundo por nível

```

### Velocidades por nível:

- Nível 1: 300 px/s
- Nível 2: 350 px/s
- Nível 3: 400 px/s

### Cálculo de Pontuação Máxima

Com 8 fileiras de 10 blocos cada:

```
// Pontos por fileira (de cima para baixo): 80, 70, 60, 50, 40, 30, 20, 10
// Total por nível: (80+70+60+50+40+30+20+10) × 10 = 3600 pontos
// Bônus por nível: 1000, 2000, 3000
// Pontuação máxima: (3600 × 3) + (1000 + 2000 + 3000) = 16,800 pontos
```

## Por que o Jogo Funciona Bem

### Organização com Classes

Usar classes torna o código:

- **Mais legível:** Cada classe tem uma responsabilidade clara
- **Mais reutilizável:** Podemos criar múltiplos blocos facilmente
- **Mais fácil de debugar:** Problemas ficam isolados em classes específicas
- **Mais expansível:** Podemos adicionar novos tipos de blocos facilmente

### Gerenciamento Inteligente de Memória

```
std::vector<Block> blocks; // Vetor dinâmico de blocos
```

- **Crescimento automático:** O vetor cresce conforme necessário
- **Memória contígua:** Melhor performance para iteração
- **Limpeza automática:** RAII garante que a memória seja liberada

### Estados Bem Definidos

O sistema de estados evita bugs comuns:

- Só processa entrada do jogo quando `gameState == PLAYING`

- Só atualiza física quando necessário
- Interface limpa entre diferentes telas

## Separação de Responsabilidades

```
void handleEvents(); // Só lida com entrada do usuário
void update();       // Só atualiza lógica do jogo
void render();       // Só desenha na tela
```

Esta separação torna o código mais fácil de manter e expandir.

## Extensões Possíveis

O código está estruturado para permitir extensões fáceis:

### Novos Tipos de Blocos

```
class SpecialBlock : public Block {
    int hitCount; // Blocos que precisam de múltiplos hits
    bool hasBonus; // Blocos que dão power-ups
};
```

### Power-ups

```
class PowerUp {
    enum Type { BIGGER_PADDLE, SLOWER_BALL, EXTRA_LIFE };
    // Implementar lógica de power-ups
};
```

### Mais Níveis

```
void createBlocks(int levelNumber) {
    // Criar padrões diferentes para cada nível
    // Blocos mais resistentes em níveis altos
}
```

### Salvamento de Pontuação

```
void saveHighScore(int score) {
    // Salvar em arquivo para persistir entre sessões
}
```

## Conceitos Importantes Aprendidos

Este tutorial ensina:

1. **Programação Orientada a Objetos:** Classes, herança, encapsulamento
2. **Física de Jogos:** Movimento, colisões, rebotes realistas
3. **Gerenciamento de Estados:** Como organizar diferentes telas de um jogo
4. **Geração Procedural:** Criar conteúdo automaticamente
5. **Interface de Usuário:** Feedback visual e informações para o jogador
6. **Otimização:** Como escrever código eficiente para jogos

O Arkanoid é um exemplo perfeito de como jogos aparentemente simples podem ensinar conceitos avançados de programação de forma divertida e prática.



# Snake

Este tutorial ensina como criar o clássico jogo Snake do zero usando C++ e SFML. Vamos construir o conhecimento passo a passo, explicando cada mecânica e conceito de programação envolvido, desde o movimento básico até sistemas avançados de validação.

## O que é Snake

Imagine um jogo onde você controla uma cobra que cresce a cada fruta que come, mas nunca pode tocar em si mesma. É um dos jogos mais simples e viciantes já criados:

- Uma cobra se move continuamente pela tela
- Você controla apenas a direção (cima, baixo, esquerda, direita)
- A cobra cresce cada vez que come uma fruta
- O jogo termina se a cobra colidir consigo mesma ou com as paredes
- O objetivo é conseguir a maior pontuação possível comendo frutas

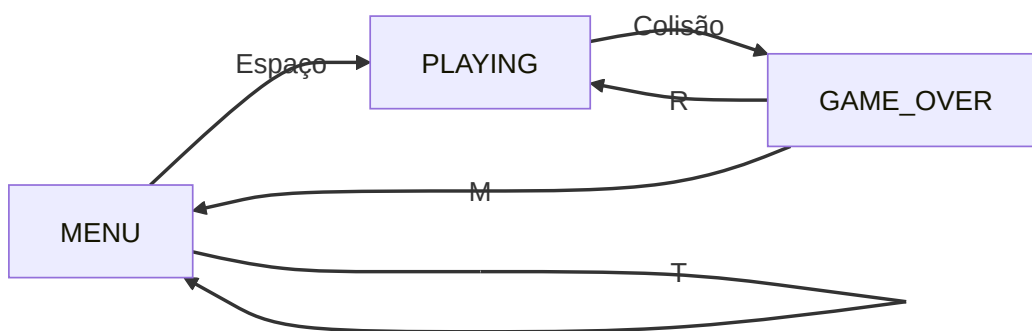
Este jogo nos permite aprender conceitos fundamentais como arrays, lógica de movimento, detecção de colisões e validação de entrada do usuário.

## Como Organizar o Jogo Snake

### Estados do Jogo - Controle de Fluxo

O Snake possui três estados principais que controlam toda a experiência:

- **Menu:** Tela inicial com instruções e configurações
- **Jogando:** Quando o jogo está ativo e a cobra se move
- **Game Over:** Quando o jogador perde, com opções de reiniciar



No código, implementamos isso com um enum simples:

```
enum GameState {  
    MENU,          // Tela inicial  
    PLAYING,       // Jogando ativamente  
    GAME_OVER      // Perdeu o jogo  
};  
  
GameState currentState = MENU; // Começa no menu
```

## Estruturas de Dados - Representando o Mundo do Jogo

### Estrutura da Cobra

```
struct Snake {  
    int x, y;      // Posição na grade  
} s[100];         // Array para até 100 segmentos  
  
int num = 4;      // Número atual de segmentos da cobra  
int dir = 0;      // Direção atual (0=baixo, 1=esquerda, 2=direita, 3=cima)
```

#### Por que usar um array?

- A cobra é uma sequência de segmentos conectados
- Cada segmento segue o movimento do anterior
- Array permite acesso direto e eficiente a qualquer segmento
- Tamanho fixo evita problemas de alocação dinâmica

### Estrutura da Fruta

```
struct Fruit {  
    int x, y;      // Posição na grade  
} f;  
  
int score = 0;     // Pontuação atual  
int highScore = 0; // Maior pontuação já alcançada
```

## Sistema de Grade - Coordenadas Lógicas vs. Visuais

O jogo funciona em duas camadas de coordenadas:

```
int N = 30, M = 20; // Grade lógica: 30x20 células  
int size = 16;      // Tamanho de cada célula em pixels
```

```
int w = size * N;    // Largura da janela: 480 pixels
int h = size * M;    // Altura da janela: 320 pixels
```

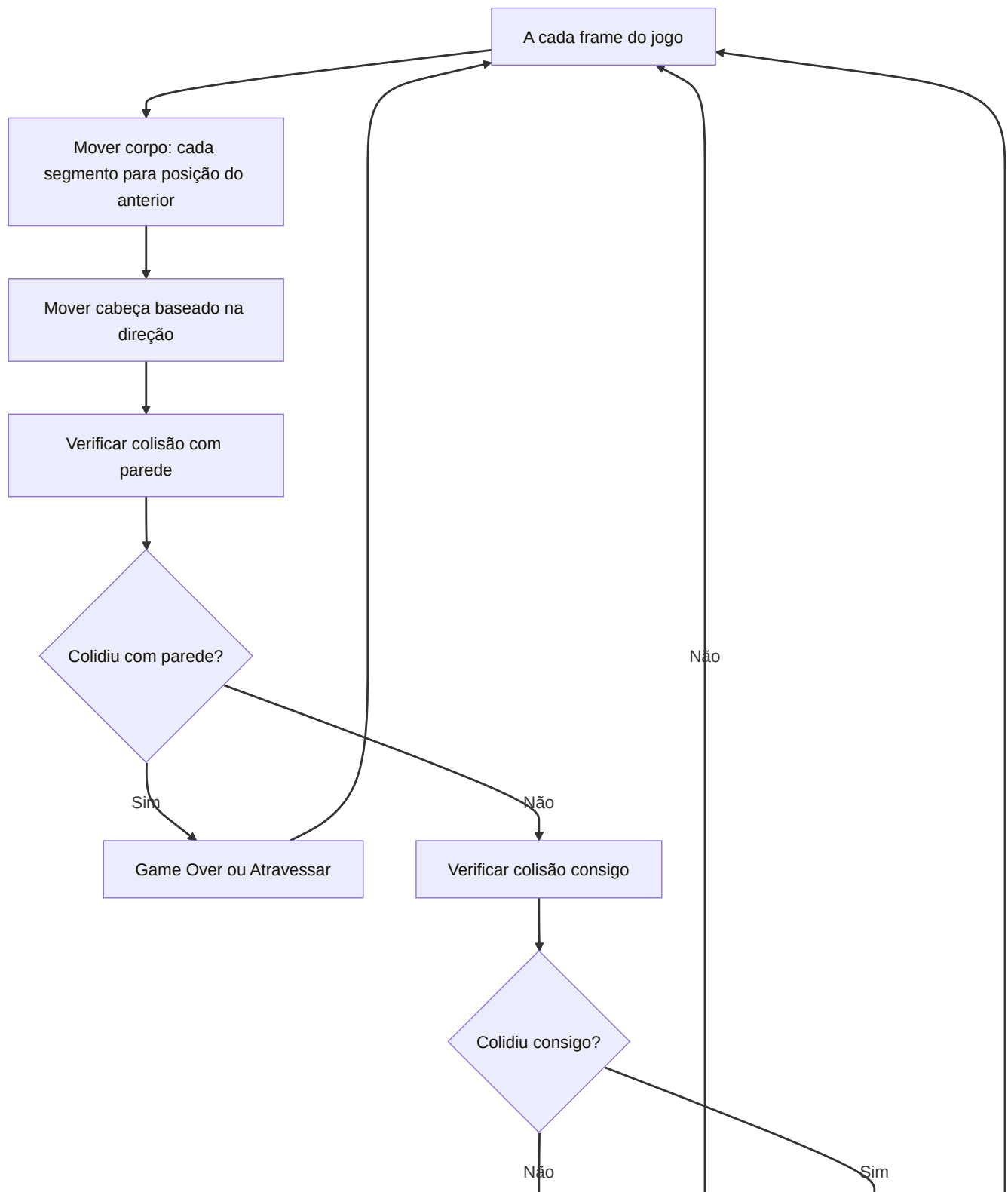
#### Vantagens deste sistema:

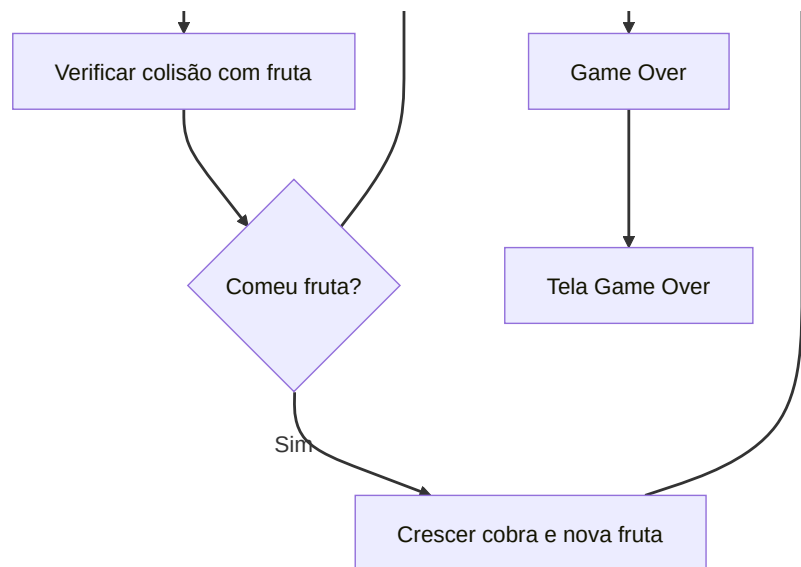
- **Lógica simples:** Posições são números inteiros (0 a 29, 0 a 19)
- **Movimento discreto:** A cobra se move célula por célula
- **Colisões precisas:** Comparação simples de coordenadas inteiras
- **Escalabilidade:** Fácil mudar tamanho do jogo alterando `size`

## As Principais Mecânicas do Jogo

### Movimento da Cobra - A Essência do Snake

O movimento da cobra é o coração do jogo. Cada segmento segue o anterior, criando o efeito de uma cobra se movendo:





## Implementação do Movimento

```

void Tick() {
    // 1. Mover o corpo da cobra (cada segmento segue o anterior)
    for (int i = num - 1; i > 0; --i) {
        s[i].x = s[i-1].x; // Copia posição X do segmento anterior
        s[i].y = s[i-1].y; // Copia posição Y do segmento anterior
    }

    // 2. Mover a cabeça da cobra baseado na direção
    if (dir == 0) s[0].y += 1; // Baixo
    if (dir == 1) s[0].x -= 1; // Esquerda
    if (dir == 2) s[0].x += 1; // Direita
    if (dir == 3) s[0].y -= 1; // Cima

    // 3. Resto da lógica de colisões...
}

```

Por que esse algoritmo funciona?

- **Movimento em cadeia:** Começamos do final (`num-1`) e vamos até o segundo segmento (1)
- **Preservação de posições:** Cada segmento "lembra" onde o anterior estava

- **Cabeça independente:** Só a cabeça (`s[0]`) se move baseada na direção do jogador
- **Efeito visual:** Cria a ilusão de uma cobra deslizando suavemente

## Controle de Direção - Prevenindo Morte Acidental

Um aspecto crucial é impedir que o jogador vá na direção oposta, o que causaria morte instantânea:

```
// Input do jogo com validação
if (currentState == PLAYING) {
    // Validar input para evitar movimento na direção oposta
    if (Keyboard::isKeyPressed(Keyboard::Left) && dir != 2) dir=1;    // ← só
se não estava indo →
    if (Keyboard::isKeyPressed(Keyboard::Right) && dir != 1) dir=2;    // → só
se não estava indo ←
    if (Keyboard::isKeyPressed(Keyboard::Up) && dir != 0) dir=3;        // ↑ só
se não estava indo ↓
    if (Keyboard::isKeyPressed(Keyboard::Down) && dir != 3) dir=0;    // ↓ só
se não estava indo ↑
}
```

### Mapeamento das direções:

- `dir = 0`: Baixo (↓) - **oposto**: Cima (3)
- `dir = 1`: Esquerda (←) - **oposto**: Direita (2)
- `dir = 2`: Direita (→) - **oposto**: Esquerda (1)
- `dir = 3`: Cima (↑) - **oposto**: Baixo (0)

## Sistema de Colisões - Detectando Eventos do Jogo

### Colisão com Parede

```
bool checkWallCollision() {
    return (s[0].x < 0 || s[0].x >= N || s[0].y < 0 || s[0].y >= M);
}

void handleWallCollision() {
    if (wallCollisionEnabled) {
        // Modo clássico: game over ao bater na parede
        currentState = GAME_OVER;
    } else {
        // Modo wrap-around: atravessa para o outro lado
        if (s[0].x >= N) s[0].x = 0;        // Direita → Esquerda
```

```

        if (s[0].x < 0) s[0].x = N - 1;    // Esquerda → Direita
        if (s[0].y >= M) s[0].y = 0;      // Baixo → Cima
        if (s[0].y < 0) s[0].y = M - 1;  // Cima → Baixo
    }
}

```

#### Dois modos de jogo:

- **Clássico:** Colidir com parede = Game Over
- **Wrap-around:** Cobra atravessa as bordas (como Pac-Man)

#### Colisão Consigo Mesmo

```

bool checkSelfCollision() {
    for (int i = 1; i < num; i++) { // Começar do índice 1 (pular a cabeça)
        if (s[0].x == s[i].x && s[0].y == s[i].y) {
            return true; // Cabeça tocou algum segmento do corpo
        }
    }
    return false;
}

```

#### Por que começar do índice 1?

- `s[0]` é a cabeça da cobra
- `s[1]` até `s[num-1]` são os segmentos do corpo
- A cabeça não pode colidir consigo mesma (índice 0)

#### Colisão com Fruta

```

bool checkFruitCollision() {
    return (s[0].x == f.x && s[0].y == f.y);
}

// Quando come fruta
if (checkFruitCollision()) {
    num++;           // Aumenta o tamanho da cobra
    score += 10;     // Adiciona pontos
    if (score > highScore) {
        highScore = score; // Atualiza recorde
    }
}

```

```
spawnNewFruit(); // Gera nova fruta  
}
```

## Geração Inteligente de Frutas

Gerar uma fruta em posição válida é mais complexo do que parece:

```
void spawnNewFruit() {  
    bool validPosition = false;  
    int attempts = 0;  
  
    while (!validPosition && attempts < 100) {  
        f.x = rand() % N; // Posição X aleatória  
        f.y = rand() % M; // Posição Y aleatória  
  
        // Verifica se a fruta não está na cobra  
        validPosition = true;  
        for (int i = 0; i < num; i++) {  
            if (f.x == s[i].x && f.y == s[i].y) {  
                validPosition = false; // Posição ocupada pela cobra  
                break;  
            }  
        }  
        attempts++;  
    }  
  
    // Fallback: se não achou posição em 100 tentativas  
    if (!validPosition) {  
        f.x = rand() % N;  
        f.y = rand() % M;  
    }  
}
```

**Problemas que este algoritmo resolve:**

- **Fruta na cobra:** Impede gerar fruta onde já há segmentos
- **Loop infinito:** Limite de 100 tentativas evita travamento
- **Cobras grandes:** Funciona mesmo quando a cobra ocupa muito espaço
- **Fallback seguro:** Se não achar posição válida, pelo menos gera algo

## Gerenciamento de Estados e Interface



## Sistema de Menus - Múltiplas Interfaces

O jogo precisa de diferentes interfaces para cada estado:

```
void drawMenu(RenderWindow& window, Font& font, bool fontLoaded) {
    if (fontLoaded) {
        // Interface com texto quando fonte está disponível
        Text title("SNAKE GAME", font, 50);
        title.setFillColor(Color::Green);
        title.setPosition(w/2 - 120, h/2 - 150);

        Text instruction("Pressione SPACE para jogar", font, 20);
        instruction.setFillColor(Color::White);
        instruction.setPosition(w/2 - 120, h/2 - 70);

        Text modeText("Modo: " + std::string(wallCollisionEnabled ? "Parede
Mata" : "Atravessa Parede"), font, 18);
        modeText.setFillColor(Color::Cyan);
        modeText.setPosition(w/2 - 90, h/2 - 40);

        window.draw(title);
        window.draw(instruction);
        window.draw(modeText);
    } else {
        // Interface visual sem texto (fallback criativo)
        for (int i = 0; i < 5; i++) {
            RectangleShape segment(Vector2f(20, 20));
            segment.setFillColor(Color::Green);
            segment.setPosition(w/2 - 50 + i * 25, h/2 - 150);
            window.draw(segment); // Desenha "SNAKE" visualmente
        }
    }
}
```

## Controle de Velocidade - Timing do Jogo

```
Clock clock;
float timer = 0, delay = 0.1; // Delay de 100ms entre movimentos

// No loop principal
float time = clock.getElapsedTime().asSeconds();
clock.restart();
timer += time;
```

```

if (currentState == PLAYING) {
    if (timer > delay) {
        timer = 0;
        Tick(); // Executar próximo movimento
    }
}

```

Como funciona o timing:

- **clock.getElapsedTime():** Tempo desde última medição
- **timer:** Acumula tempo até atingir o delay
- **delay = 0.1:** Cobra se move 10 vezes por segundo
- **timer = 0:** Reset para próximo ciclo

### Função de Reset - Começar Nova Partida

```

void resetGame() {
    num = 4;           // Tamanho inicial da cobra
    score = 0;         // Zerar pontuação
    dir = 0;           // Direção inicial (baixo)
    f.x = 10;          // Posição inicial da fruta
    f.y = 10;

    // Resetar posição da cobra (linha horizontal)
    for (int i = 0; i < 4; i++) {
        s[i].x = 4 - i; // x: 4, 3, 2, 1
        s[i].y = 5;     // y: todos na linha 5
    }

    currentState = PLAYING; // Iniciar jogo
}

```

Posicionamento inicial:

- **Cabeça:** `s[0]` na posição (4, 5)
- **Corpo:** `s[1]` (3,5), `s[2]` (2,5), `s[3]` (1,5)
- **Resultado:** Cobra horizontal de 4 segmentos indo para a direita

## Conceitos Avançados de Programação

## Validação de Dados - Prevenindo Bugs

```
void validateGameState() {
    // Verificar se a cabeça está dentro dos limites
    if (s[0].x < 0 || s[0].x >= N || s[0].y < 0 || s[0].y >= M) {
        if (wallCollisionEnabled && currentState != GAME_OVER) {
            // Erro: deveria ter terminado o jogo
            // Útil para debug
        }
    }

    // Verificar colisão com próprio corpo
    for (int i = 1; i < num; i++) {
        if (s[0].x == s[i].x && s[0].y == s[i].y) {
            if (currentState != GAME_OVER) {
                // Erro: deveria ter terminado o jogo por auto-colisão
            }
        }
    }
}
```

### Propósito da validação:

- **Debug:** Identificar bugs na lógica
- **Testes:** Verificar se o jogo está funcionando corretamente
- **Robustez:** Detectar estados inválidos
- **Manutenção:** Facilitar correção de problemas

## Fallback Gráfico - Funcionar Sem Recursos

O jogo funciona mesmo sem arquivos de imagem:

```
// Tentar carregar texturas
Texture t1, t2;
if (!t1.loadFromFile("images/white.png")) {
    // Criar textura procedural branca
    Image whiteImage;
    whiteImage.create(16, 16, Color::White);
    t1.loadFromImage(whiteImage);
}

if (!t2.loadFromFile("images/red.png")) {
```

```

// Criar textura procedural vermelha
Image redImage;
redImage.create(16, 16, Color::Red);
t2.loadFromImage(redImage);
}

```

#### Vantagens:

- **Independência:** Jogo funciona em qualquer ambiente
- **Robustez:** Não quebra por arquivos faltando
- **Desenvolvimento:** Pode testar sem assets
- **Distribuição:** Menos arquivos para gerenciar

### Sistema de Pontuação Visual

Quando não há fonte disponível, criar visualização de pontuação:

```

void drawScore(RenderWindow& window, Font& font, bool fontLoaded) {
    if (!fontLoaded) {
        // Score visual com barras
        int scoreBars = (score / 10) > 15 ? 15 : (score / 10);
        for (int i = 0; i < scoreBars; i++) {
            RectangleShape bar(Vector2f(6, 15));
            bar.setFillColor(Color::White);
            bar.setPosition(10 + i * 8, 10);
            window.draw(bar);
        }

        // High score visual
        int highScoreBars = (highScore / 10) > 15 ? 15 : (highScore / 10);
        for (int i = 0; i < highScoreBars; i++) {
            RectangleShape bar(Vector2f(6, 15));
            bar.setFillColor(Color::Yellow);
            bar.setPosition(w - 130 + i * 8, 10);
            window.draw(bar);
        }
    }
}

```

### Estrutura Completa do Main Loop

```

int main() {
    srand(time(0)); // Seed para números aleatórios

    RenderWindow window(VideoMode(w, h), "Snake Game!");

    // Inicialização de recursos...

    while (window.isOpen()) {
        // 1. Controle de timing
        float time = clock.getElapsedTime().asSeconds();
        clock.restart();
        timer += time;

        // 2. Processar eventos
        Event e;
        while (window.pollEvent(e)) {
            if (e.type == Event::Closed) {
                window.close();
            }

            // Eventos específicos por estado...
        }

        // 3. Lógica do jogo
        if (currentState == PLAYING) {
            // Input com validação
            // Movimento automático com timer
        }

        // 4. Renderização
        window.clear();

        if (currentState == MENU) {
            drawMenu(window, font, fontLoaded);
        }
        else if (currentState == PLAYING) {
            // Desenhar fundo, cobra, fruta, interface
        }
        else if (currentState == GAME_OVER) {
            drawGameOver(window, font, fontLoaded);
        }

        window.display();
    }
}

```

```
}  
  
    return 0;  
}
```

## Conceitos de Programação Aprendidos

### 1. Arrays e Indexação

- Uso de arrays para representar sequências
- Manipulação de índices com cuidado
- Diferença entre tamanho lógico e físico

### 2. Máquinas de Estado

- Enum para representar estados
- Transições controladas entre estados
- Comportamento específico por estado

### 3. Algoritmos de Movimento

- Movimento em cadeia (seguir o líder)
- Coordenadas lógicas vs. físicas
- Controle de timing

### 4. Detecção de Colisões

- Colisão ponto-a-ponto
- Múltiplos tipos de colisão
- Ordem de verificação de colisões

### 5. Validação de Entrada

- Filtrar input inválido
- Prevenir estados inconsistentes
- Interface responsiva e segura

## 6. Geração Procedural

- Algoritmos de spawn inteligente
- Tratamento de casos extremos
- Fallbacks para situações problemáticas

Este jogo Snake demonstra como um conceito simples pode envolver múltiplas técnicas avançadas de programação, desde estruturas de dados básicas até algoritmos de validação robustos.

# Minesweeper

Este tutorial aprofunda a implementação do clássico jogo Campo Minado (Minesweeper) utilizando C++ e a biblioteca SFML. Abordaremos a arquitetura do jogo, as estruturas de dados subjacentes, os algoritmos centrais e a gestão da interface do utilizador de forma técnica e detalhada.

## Visão Geral Técnica

Minesweeper é um jogo de lógica baseado em uma grade, onde o jogador deve deduzir a localização de minas ocultas. A implementação foca na gestão eficiente de estados, manipulação de grades 2D e um algoritmo recursivo de "flood fill" para revelar áreas do tabuleiro.

## Organização do Jogo

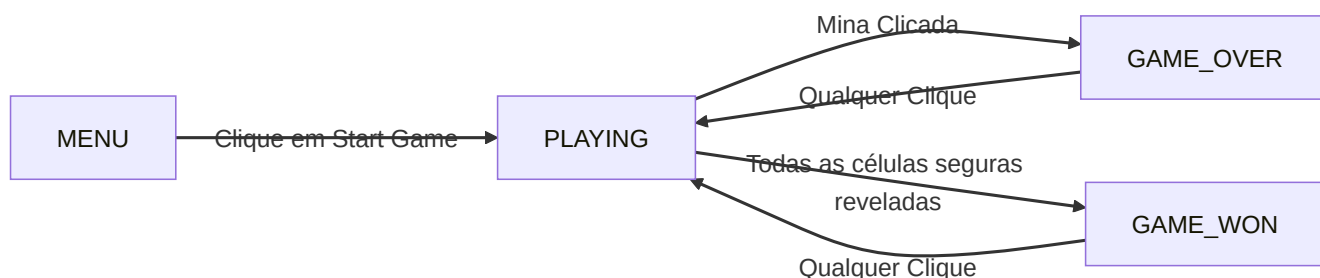
### Estados do Jogo - Uma Máquina de Estados Finitos

O fluxo do jogo é controlado por uma máquina de estados simples, definida por um `enum` e uma variável global `currentGameState`. Isso permite que o programa se comporte de maneira diferente dependendo do contexto (menu, jogo ativo, fim de jogo).

```
enum GameState {  
    MENU,          // Tela inicial, aguardando o jogador iniciar  
    PLAYING,       // Jogo em andamento, processando interações do tabuleiro  
    GAME_OVER,     // Jogo finalizado por derrota (mina detonada)  
    GAME_WON       // Jogo finalizado por vitória (todas as células seguras  
reveladas)  
};
```

```
GameState currentGameState = MENU; // O jogo sempre inicia no estado de menu
```

As transições de estado são acionadas por eventos do utilizador (cliques) ou por condições de jogo (detonar mina, revelar todas as células seguras).



### Estruturas de Dados - Representação do Tabuleiro

O tabuleiro do Minesweeper é modelado por duas matrizes bidimensionais de inteiros, ambas de tamanho `12x12` para acomodar uma borda invisível de células (índices 0 e 11) que simplifica a lógica de



verificação de vizinhos.

- `int grid[12][12]`: Esta matriz armazena o estado *lógico* de cada célula:
  - 0 a 8: Indica o número de minas adjacentes.
  - 9: Representa uma mina.
  - As células na borda (índices 0 e 11) são geralmente inicializadas com 0 e não são exibidas ao jogador.
- `int sgrid[12][12]`: Esta matriz armazena o estado *visível* de cada célula para o jogador:
  - 10: Célula coberta (não revelada).
  - 11: Célula marcada com uma bandeira.
  - 0 a 9: Célula revelada, exibindo o número de minas adjacentes ou uma mina (se detonada).

```
int w = 32; // Tamanho em pixels de cada célula (largura e altura)
int grid[12][12]; // Armazena o layout das minas e contagens
int sgrid[12][12]; // Armazena o que é visível para o jogador
```

## Sistema de Coordenadas

O jogo opera com dois sistemas de coordenadas:

1. **Coordenadas Lógicas (Grid)**: Pares de inteiros `(i, j)` que representam a posição da célula na matriz (ex: `grid[i][j]`). Estas são usadas para toda a lógica do jogo (cálculo de minas, verificação de vizinhos).
2. **Coordenadas Visuais (Pixel)**: Pares de inteiros `(x, y)` em pixels, usadas para renderização na janela SFML. A conversão é feita multiplicando as coordenadas lógicas pelo tamanho da célula (`w`).

```
// Conversão de coordenadas de pixel para grid
int x_grid = pos.x / w;
int y_grid = pos.y / w;

// Conversão de coordenadas de grid para pixel para posicionamento de sprites
s.setPosition(i * w, j * w);
```

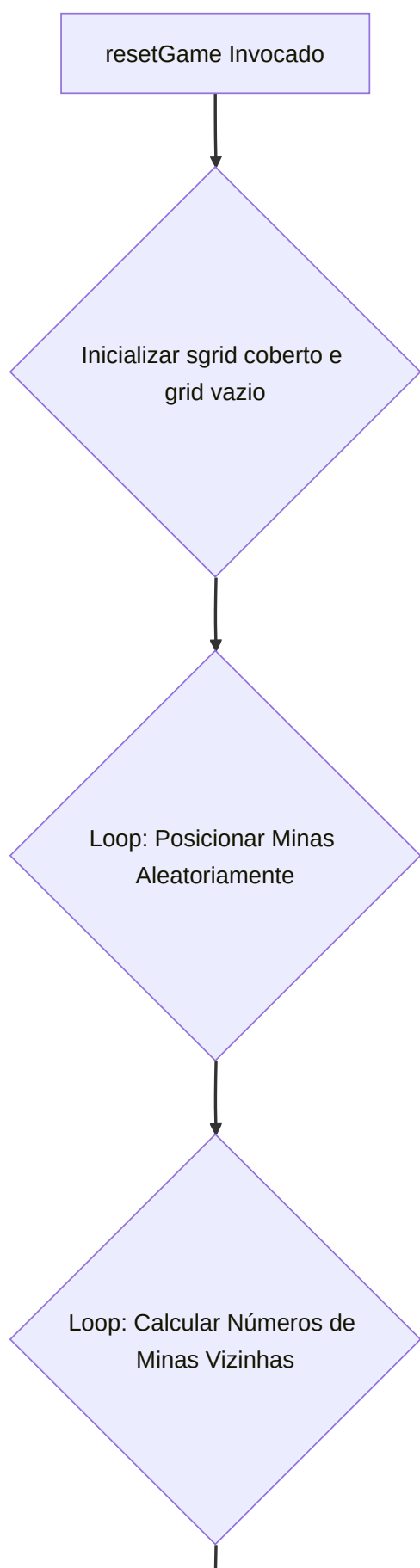
## Mecânicas e Algoritmos Centrais

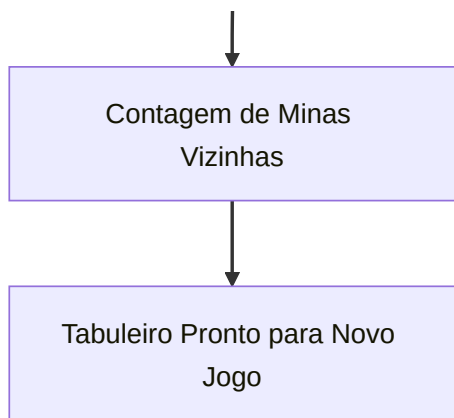
### `resetGame()` - Inicialização e Reconfiguração do Tabuleiro

Esta função é invocada no início de cada nova partida. Seu algoritmo é:

1. **Inicialização de sgrid:** Todas as células visíveis são definidas como 10 (cobertas).
2. **Inicialização de grid:** Todas as células lógicas são definidas como 0 (sem minas).
3. **Posicionamento de Minas:** Um loop duplo itera sobre as células (1,1) a (10,10). Para cada célula, um número aleatório é gerado (`rand() % 5 == 0`). Se a condição for verdadeira (aproximadamente 20% de chance), a célula em `grid[i][j]` é definida como 9 (mina).
4. **Cálculo de Números:** Outro loop duplo itera sobre as células (1,1) a (10,10). Se uma célula *não* contiver uma mina (`grid[i][j] != 9`), ela verifica seus 8 vizinhos (incluindo diagonais). Para cada vizinho que contém uma mina (`grid[vizinho_x][vizinho_y] == 9`), um contador é incrementado. O valor final do contador é atribuído a `grid[i][j]`. A borda invisível (0 e 11) garante que as verificações de vizinhos não saiam dos limites da matriz.

```
void resetGame() {
    // ... inicialização de grids ...
    for (int i = 1; i <= 10; i++) {
        for (int j = 1; j <= 10; j++) {
            if (grid[i][j] == 9) continue; // Ignora minas
            int n = 0;
            for (int dx = -1; dx <= 1; dx++) { // Itera sobre vizinhos
                for (int dy = -1; dy <= 1; dy++) {
                    if (grid[i + dx][j + dy] == 9) n++; // Conta minas
                }
            }
            grid[i][j] = n;
        }
    }
}
```





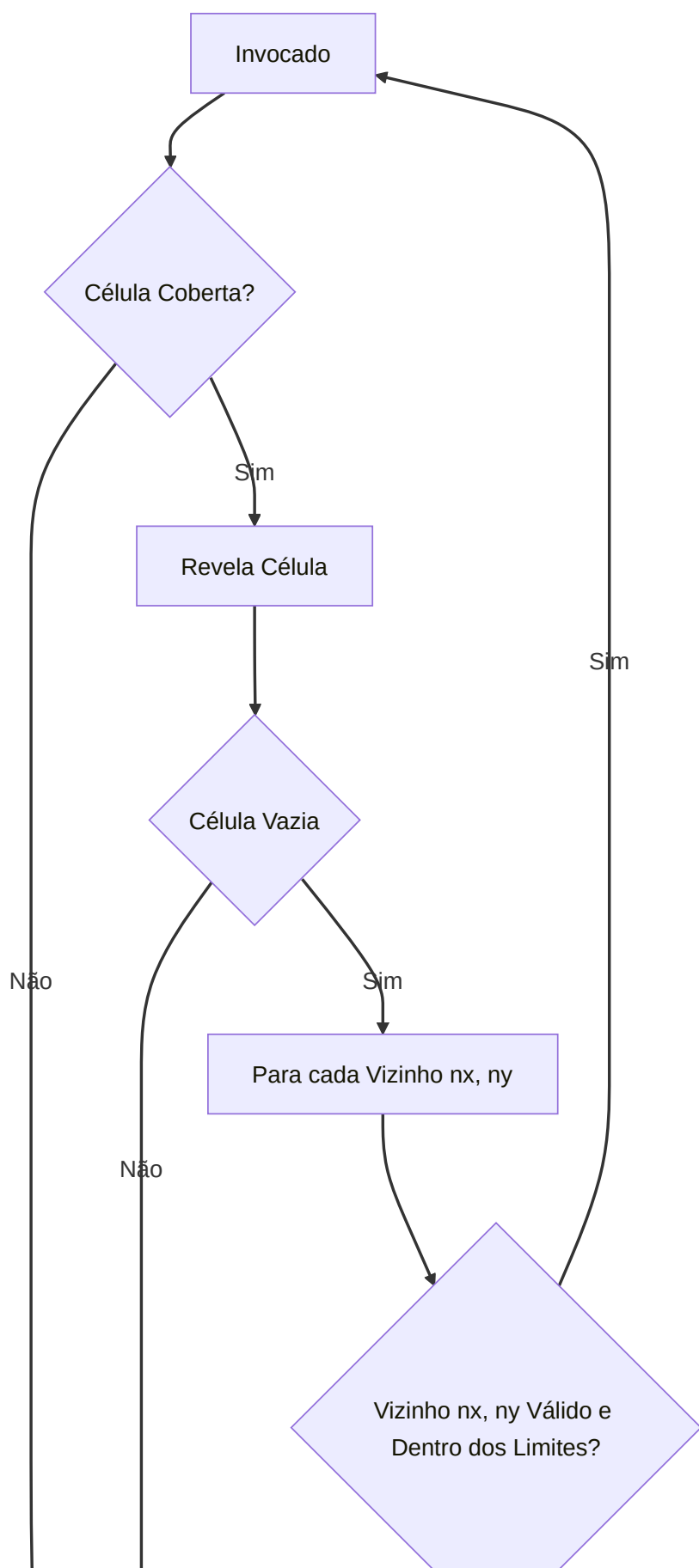
## openCells() - Algoritmo Recursivo de Flood Fill

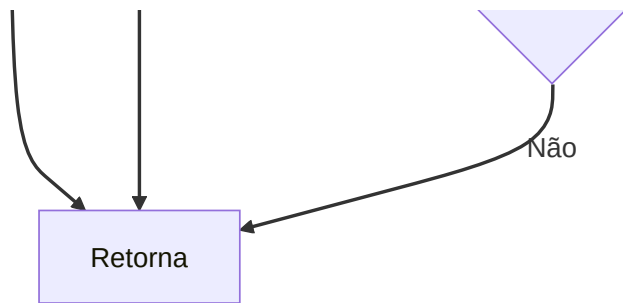
Esta função é a espinha dorsal da mecânica de revelação de células vazias. É uma implementação clássica do algoritmo de *flood fill*:

1. **Condição de Parada:** A recursão para se a célula  $(i, j)$  já não estiver coberta ( $sgrid[i][j] != 10$ ). Isso evita loops infinitos e reprocessamento de células já reveladas.
2. **Revelação:** A célula  $sgrid[i][j]$  é atualizada para o valor correspondente em  $grid[i][j]$ , tornando-a visível.
3. **Propagação (para células vazias):** Se a célula revelada for 0 (vazia, sem minas adjacentes), a função chama a si mesma recursivamente para todos os seus 8 vizinhos (incluindo diagonais). As chamadas recursivas são protegidas por verificações de limites ( $i + dx \geq 1 \ \&\& \ i + dx \leq 10$ , etc.) para garantir que não acessem índices inválidos da matriz.

```

void openCells(int i, int j) {
    if (sgrid[i][j] == 10) { // Se a célula está coberta
        sgrid[i][j] = grid[i][j]; // Revela
        if (grid[i][j] == 0) { // Se for vazia, propaga
            for (int dx = -1; dx <= 1; dx++) {
                for (int dy = -1; dy <= 1; dy++) {
                    // Verifica limites e chama recursivamente
                    if (i + dx >= 1 && i + dx <= 10 && j + dy >= 1 && j + dy
<= 10) {
                        openCells(i + dx, j + dy);
                    }
                }
            }
        }
    }
}
  
```





## Processamento de Entrada do Utilizador

O jogo responde a cliques do mouse, com o comportamento variando de acordo com o `currentGameState`:

- **Estado MENU:** Um clique esquerdo dentro da área do texto "Start Game" transiciona o jogo para o estado `PLAYING` e inicia uma nova partida via `resetGame()`.
- **Estados GAME\_OVER/GAME\_WON:** Qualquer clique esquerdo reinicia o jogo, transicionando para `PLAYING` e chamando `resetGame()`.
- **Estado PLAYING:**
  - **Clique Esquerdo:** Se a célula clicada estiver coberta (`sgrid[x][y] == 10`):
    - Se `grid[x][y] == 9` (mina): O estado muda para `GAME_OVER`. Todas as minas são reveladas em `sgrid`.
    - Se `grid[x][y] == 0` (vazia): `openCells(x, y)` é chamada para iniciar a revelação recursiva.
    - Caso contrário (número 1-8): A célula é simplesmente revelada (`sgrid[x][y] = grid[x][y]`).
  - **Clique Direito:** Alterna o estado da célula entre coberta (10) e bandeira (11), mas apenas se a célula estiver coberta ou já tiver uma bandeira.

## Condições de Vitória e Derrota

- **Derrota:** Detectada imediatamente quando um clique esquerdo revela uma mina (`grid[x][y] == 9`). O `currentGameState` é definido como `GAME_OVER`.
- **Vitória:** Verificada a cada frame no estado `PLAYING`. O jogo é ganho se:
  1. O número de células *cobertas* (`sgrid[i][j] == 10` ou `sgrid[i][j] == 11`) for igual ao número total de minas no tabuleiro.
  2. E o número de minas *corretamente* marcadas com bandeiras (`sgrid[i][j] == 11 && grid[i][j] == 9`) for igual ao número total de minas.

Esta lógica garante que o jogador não apenas revele todas as células seguras, mas também identifique corretamente todas as minas. Se a condição for satisfeita, `currentGameState` é definido

como `GAME_WON`.

## Interface do Utilizador (UI) e Renderização

O SFML é utilizado para desenhar todos os elementos visuais do jogo.

### Uso de Sprite Sheet (images/tiles.jpg)

O arquivo `tiles.jpg` é uma *sprite sheet* contendo todas as imagens para os diferentes estados das células. Cada imagem é um quadrado de `32x32` pixels. A função `s.setTextureRect(IntRect(sgrid[i][j] * w, 0, w, w))` é crucial aqui: ela seleciona a porção correta da sprite sheet (`sgrid[i][j] * w` pixels a partir da esquerda) para desenhar a célula correspondente ao seu estado visível.

- **Índices da Sprite Sheet:**

- `0` a `8`: Sprites para células reveladas com `0` a `8` minas adjacentes.
- `9`: Sprite da mina (exibida ao perder o jogo).
- `10`: Sprite da célula coberta.
- `11`: Sprite da bandeira.

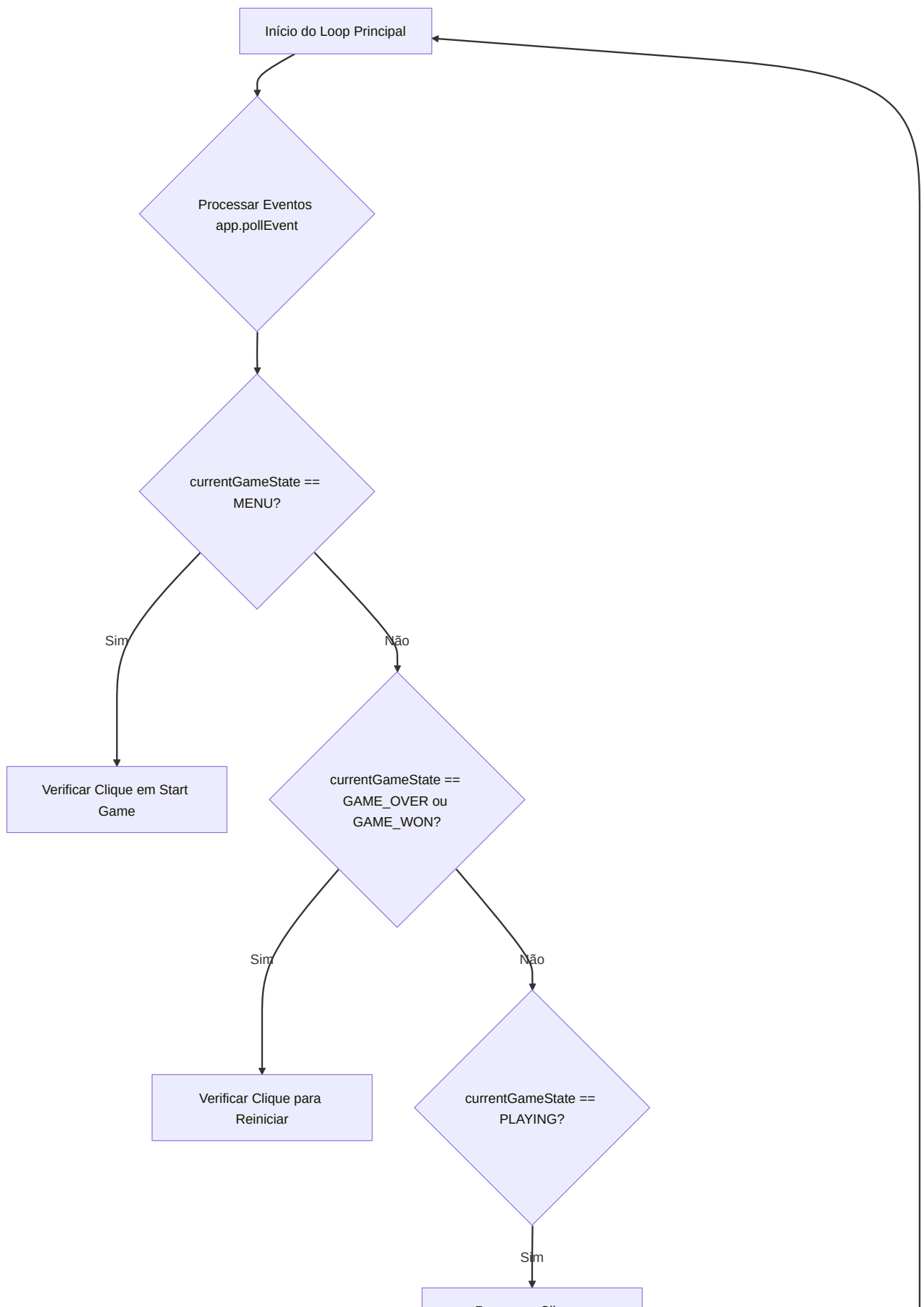
### Renderização de Texto (`sf::Font`, `sf::Text`)

A fonte `Carlito-Regular.ttf` é carregada para renderizar mensagens de status (`GAME OVER`, `YOU WIN!`, `Click to Play Again`) e elementos do menu (`MINESWEEPER`, `Start Game`).

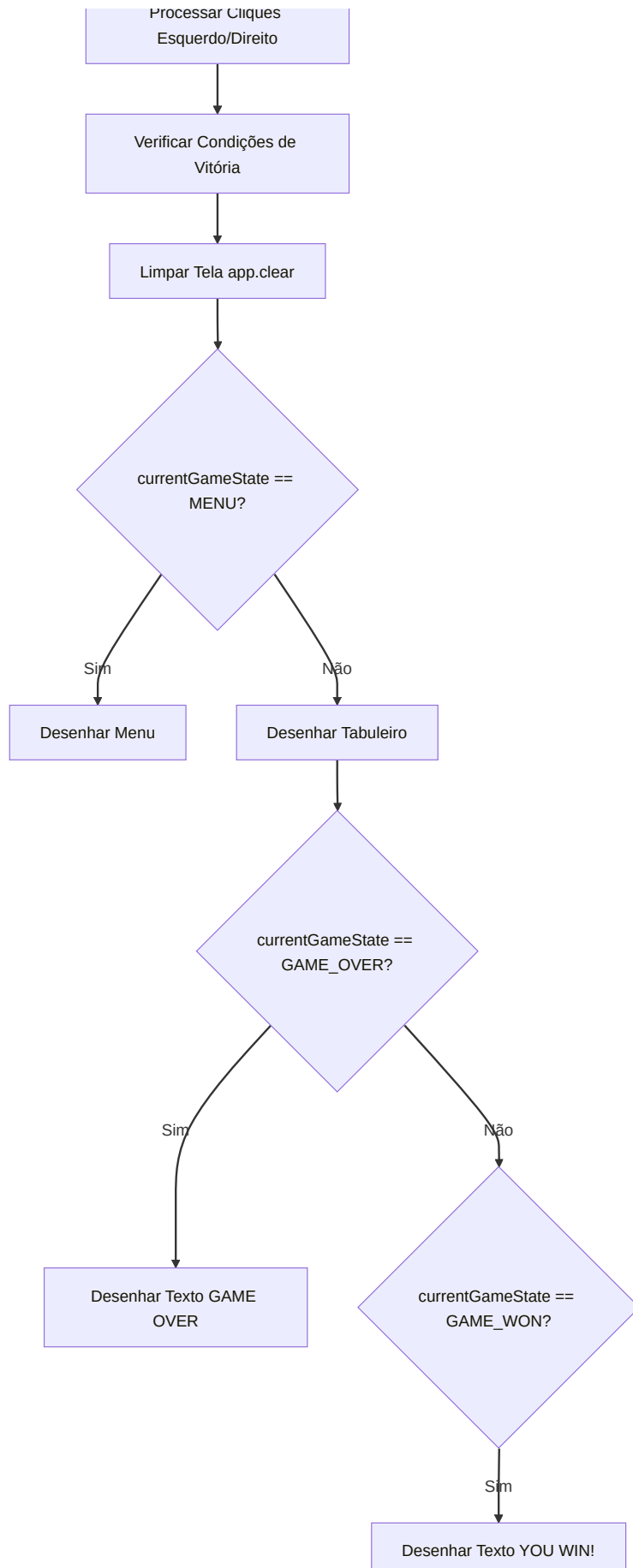
- `sf::Font::loadFromFile()`: Carrega o arquivo da fonte.
- `sf::Text`: Objeto usado para configurar o texto (string, fonte, tamanho, cor, contorno).
- `setTextureRect()`: Não aplicável a texto, mas `setPosition()` e `getGlobalBounds()` são usados para centralizar e posicionar o texto dinamicamente na janela.

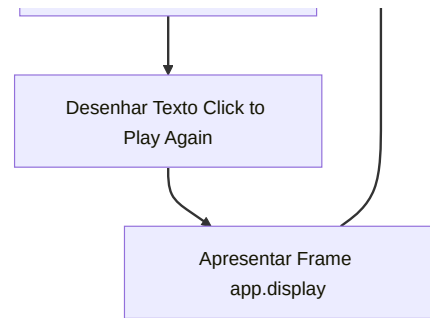
### Estrutura do Loop Principal (main function)

O loop principal do jogo é o coração da aplicação SFML, seguindo o padrão comum de jogos:









## Conceitos de Programação Avançados

### 1. Máquinas de Estado Finitos (FSM)

- A implementação do `GameState` é um exemplo claro de FSM, um padrão de design fundamental em desenvolvimento de jogos para gerenciar complexidade e fluxo de aplicação.

### 2. Recursão e Algoritmos de Busca (Flood Fill)

- A função `openCells()` é uma aplicação prática e eficiente da recursão para implementar o algoritmo de *flood fill*. É essencial para a jogabilidade do Minesweeper, revelando grandes áreas do tabuleiro com um único clique.

### 3. Manipulação de Matrizes 2D e Verificação de Limites

- O uso de `grid` e `sgrid` demonstra a manipulação de dados em estruturas 2D. A inclusão de uma borda invisível (12x12 para um tabuleiro 10x10) é uma técnica comum para simplificar a lógica de verificação de vizinhos, evitando a necessidade de múltiplas verificações de `if` para os cantos e bordas do tabuleiro real.

### 4. Programação Orientada a Eventos

- O loop principal do SFML é um exemplo clássico de programação orientada a eventos, onde o programa reage a interações do utilizador e eventos do sistema, em vez de seguir um fluxo linear.

### 5. Geração Procedural Simples

- A colocação aleatória de minas no `resetGame()` é uma forma básica de geração procedural de conteúdo, onde elementos do jogo são criados algoritmicamente em tempo de execução.

Este projeto de Minesweeper serve como um excelente estudo de caso para entender a aplicação de algoritmos fundamentais, padrões de design e técnicas de renderização em um contexto de desenvolvimento de jogos.

# Troubleshooting

Este guia resolve os problemas mais comuns encontrados durante a configuração e execução dos 16 jogos em C++. 🔧



## Problemas Mais Comuns

### 1. SFML não encontrado

#### Sintomas

```
CMake Error: Could not find SFML
pkg-config: sfml-all not found
```

#### Soluções

##### Linux:

```
# Ubuntu/Debian
sudo apt update
sudo apt install libsFML-dev

# Fedora
sudo dnf install SFML-devel

# Arch Linux
sudo pacman -S sfml

# Verificar instalação
pkg-config --exists sfml-all && echo "OK" || echo "ERRO"
```

##### macOS:

```
# Usando Homebrew
brew install sfml

# Se Homebrew não estiver instalado
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

##### Windows:

- Baixe SFML do site oficial (<https://www.sFML-dev.org/download.php>)

- Extraia para `C:\SFML`
- Configure variável de ambiente `SFML_ROOT=C:\SFML`

## 2. CMake versão muito antiga

### Sintomas

```
CMake Error: CMake 3.5 or higher is required. You are running version 2.8.12
```

### Soluções

#### Ubuntu/Debian:

```
# Remover versão antiga
sudo apt remove cmake

# Adicionar repositório oficial
wget -O - https://apt.kitware.com/keys/kitware-archive-latest.asc | sudo apt-
key add -
sudo apt-add-repository 'deb https://apt.kitware.com/ubuntu/ focal main'
sudo apt update
sudo apt install cmake
```

#### Compilar do código fonte:

```
wget https://github.com/Kitware/CMake/releases/download/v3.25.1/cmake-
3.25.1.tar.gz
tar -xzf cmake-3.25.1.tar.gz
cd cmake-3.25.1
./bootstrap --prefix=/usr/local
make -j$(nproc)
sudo make install
```

## 3. Compilador não suporta C++17

### Sintomas

```
error: 'auto' type specifier is a C++11 extension
error: range-based for loop is a C++11 extension
```

### Soluções

#### Ubuntu/Debian:

```
# Instalar GCC mais recente
sudo apt install gcc-9 g++-9
```

```
# Configurar como padrão
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-9 90
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-9 90

# Verificar versão
g++ --version
```

**Forçar compilador no CMake:**

```
cmake .. -DCMAKE_CXX_COMPILER=g++-9
```

## 4. Erro de linking com SFML

**Sintomas**

```
undefined reference to `sf::RenderWindow::RenderWindow()'
undefined reference to `sf::Texture::loadFromFile()'
```

**Soluções**

**Verificar bibliotecas SFML:**

```
# Linux
ldconfig -p | grep sfml
find /usr -name "*sfml*" 2>/dev/null

# Verificar pkg-config
pkg-config --cflags --libs sfml-all
```

**Reinstalar SFML:**

```
# Ubuntu/Debian
sudo apt remove libsfml-dev
sudo apt autoremove
sudo apt install libsfml-dev

# Verificar novamente
pkg-config --modversion sfml-all
```

## 5. Jogos não iniciam (sem janela)

**Sintomas**

- Executável compila mas não abre janela

- Erro "Failed to create OpenGL context"
- Tela preta

## Soluções

### Verificar drivers gráficos:

```
# Linux - informações da GPU
lspci | grep -i vga
glxinfo | grep "OpenGL version"

# Instalar drivers se necessário
# NVIDIA:
sudo apt install nvidia-driver-470

# AMD:
sudo apt install mesa-vulkan-drivers

# Intel:
sudo apt install intel-media-va-driver
```

### Testar OpenGL:

```
# Instalar mesa-utils
sudo apt install mesa-utils

# Testar OpenGL
glxgears
```

### Executar com debug:

```
# Executar com informações de debug
DISPLAY=:0 ./games/tetris/tetris
```

## 6. Assets não encontrados

### Sintomas

```
Failed to load image: images/tiles.png
Failed to load font: fonts/arial.ttf
```

## Soluções

### Verificar estrutura de arquivos:

```
# Ver se assets foram copiados
ls build/games/tetris/
ls build/games/tetris/images/

# Se não existirem, recompilar
make clean
make tetris
```

**Executar do diretório correto:**

```
# CORRETO - executar de dentro do diretório do jogo
cd build/games/tetris
./tetris

# INCORRETO - executar de outro lugar
cd build
./games/tetris/tetris # Pode não encontrar assets
```

## 7. Erro de permissão

**Sintomas**

```
Permission denied
make: *** [CMakeFiles/tetris.dir/all] Error 2
```

**Soluções**

**Corrigir permissões:**

```
# Dar permissão de execução aos scripts
chmod +x setup.sh
chmod +x *.sh

# Corrigir permissões do projeto
chmod -R 755 .
```

**Problemas de sudo:**

```
# Se instalou com sudo, corrigir ownership
sudo chown -R $USER:$USER ~/.cmake
sudo chown -R $USER:$USER ./build
```



## Problemas Específicos por Sistema



## Ubuntu/Debian Específicos

**Erro: "Package sfml-all was not found"**

```
# Atualizar lista de pacotes
sudo apt update

# Verificar se universe repository está habilitado
sudo add-apt-repository universe
sudo apt update

# Instalar SFML
sudo apt install libsFML-dev
```

**Erro: "Unable to locate package"**

```
# Verificar versão do Ubuntu
lsb_release -a

# Ubuntu muito antigo - usar PPA
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt update
```

## Fedora/CentOS Específicos

**Erro: "No package SFML-devel available"**

```
# Fedora - habilitar RPM Fusion
sudo dnf install https://mirrors.rpmfusion.org/free/fedora/rpmfusion-free-release-$(rpm -E %fedora).noarch.rpm

# CentOS - habilitar EPEL
sudo dnf install epel-release
```

## macOS Específicos

**Erro: "xcrun: error: invalid active developer path"**

```
# Instalar Command Line Tools
xcode-select --install

# Se já instalado, resetar
sudo xcode-select --reset
```

**Homebrew não funciona**

```
# Reinstalar Homebrew
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

# Adicionar ao PATH
echo 'export PATH="/opt/homebrew/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc
```

## Windows Específicos

### Visual Studio não encontra SFML

```
# No CMakeLists.txt, adicionar:
set(SFML_ROOT "C:/SFML")
find_package(SFML 2.5 COMPONENTS system window graphics audio REQUIRED)
```


### MinGW problemas de linking


```
# Usar bibliotecas estáticas
cmake .. -DSFML_STATIC_LIBRARIES=TRUE
```



## Ferramentas de Diagnóstico

### Script de Diagnóstico Completo

```
# Criar script de diagnóstico
cat > diagnose.sh << 'EOF'
#!/bin/bash

echo " Diagnóstico do Sistema - 16 Games in C++"
echo "===== "

# Sistema operacional
echo " Sistema:"
uname -a
echo ""

# Compilador
echo " Compilador:"
if command -v g++ &> /dev/null; then
    g++ --version | head -1
    echo " G++ disponível"
else
```

```

    echo "❌ G++ não encontrado"
fi

if command -v clang++ &> /dev/null; then
    clang++ --version | head -1
    echo "✅ Clang++ disponível"
else
    echo "❌ Clang++ não encontrado"
fi
echo ""

# CMake
echo "🔧 CMake:"
if command -v cmake &> /dev/null; then
    cmake --version | head -1
    echo "✅ CMake disponível"
else
    echo "❌ CMake não encontrado"
fi
echo ""

# SFML
echo "🎮 SFML:"
if pkg-config --exists sfml-all; then
    echo "✅ SFML $(pkg-config --modversion sfml-all) encontrado"
    echo "📌 Flags: $(pkg-config --cflags --libs sfml-all)"
else
    echo "❌ SFML não encontrado via pkg-config"

    # Procurar manualmente
    if find /usr -name "*sfml*" 2>/dev/null | head -5; then
        echo "💡 SFML pode estar instalado mas não configurado para pkg-
config"
    fi
fi
echo ""

# OpenGL
echo "🖥️ OpenGL:"
if command -v glxinfo &> /dev/null; then
    echo "OpenGL: $(glxinfo | grep "OpenGL version" | cut -d':' -f2)"
    echo "✅ OpenGL disponível"
else
    echo "⚠️ glxinfo não disponível (instale mesa-utils)"

```

```

fi
echo ""

# Espaço em disco
echo "💾 Espaço em disco:"
df -h . | tail -1
echo ""

# Resumo
echo "📋 Resumo:"
echo "======"

issues=0

if ! command -v g++ &> /dev/null && ! command -v clang++ &> /dev/null; then
    echo "❌ Nenhum compilador C++ encontrado"
    ((issues++))
fi

if ! command -v cmake &> /dev/null; then
    echo "❌ CMake não encontrado"
    ((issues++))
fi

if ! pkg-config --exists sfml-all; then
    echo "❌ SFML não encontrado"
    ((issues++))
fi

if [ $issues -eq 0 ]; then
    echo "🎉 Sistema parece estar configurado corretamente!"
    echo "💡 Se ainda há problemas, execute: ./setup.sh"
else
    echo "⚠️ $issues problema(s) encontrado(s)"
    echo "💡 Consulte a documentação para resolver os problemas acima"
fi
EOF

chmod +x diagnose.sh
./diagnose.sh

```

## Verificação de Build

```

# Script para verificar build específico
cat > check_build.sh << 'EOF'
#!/bin/bash

if [ ! -d "build" ]; then
    echo "❌ Diretório build não existe"
    echo "Execute: mkdir build && cd build && cmake .."
    exit 1
fi

cd build

if [ ! -f "CMakeCache.txt" ]; then
    echo "❌ CMake não foi configurado"
    echo "Execute: cmake .."
    exit 1
fi

echo "✅ Build configurado"
echo "📁 Jogos compilados:"

count=0
for game_dir in games/*/; do
    if [ -d "$game_dir" ]; then
        game_name=$(basename "$game_dir")
        if [ -f "$game_dir/$game_name" ]; then
            echo "✅ $game_name"
            ((count++))
        else
            echo "❌ $game_name (não compilado)"
        fi
    fi
done

echo ""
echo "📊 Total: $count jogos compilados"

if [ $count -eq 0 ]; then
    echo "💡 Execute: make all_games"
fi
EOF

```

```
chmod +x check_build.sh
./check_build.sh
```

## Últimos Recursos

### Resetar Ambiente Completamente

```
# Script de reset total
cat > reset_environment.sh << 'EOF'
#!/bin/bash

echo "🔥 RESETANDO AMBIENTE COMPLETAMENTE"
echo "===== "

# Fazer backup se necessário
if [ -d "build" ]; then
    echo "📦 Fazendo backup do build atual..."
    mv build build_backup_$(date +%Y%m%d_%H%M%S)
fi

# Limpar completamente
echo "🧹 Limpando arquivos temporários..."
rm -rf build
rm -rf .cache
find . -name "*.o" -delete
find . -name "*.cmake" -delete 2>/dev/null

# Recriar build
echo "📁 Recriando estrutura..."
mkdir build
cd build

# Configurar do zero
echo "⚙️ Configurando CMake do zero..."
cmake .. -DCMAKE_BUILD_TYPE=Debug

# Compilar teste
echo "🔨 Testando compilação..."
make tetris

if [ $? -eq 0 ]; then
    echo "✅ Reset concluído com sucesso!"
    echo "🎮 Teste: make run_tetris"
```

```
else
    echo "❌ Ainda há problemas após reset"
    echo "💡 Execute o diagnóstico: ../diagnose.sh"
fi
EOF

chmod +x reset_environment.sh
```

## Suporte da Comunidade

Se nenhuma solução funcionou:

1. Execute o diagnóstico completo: `./diagnose.sh`

2. Tente o reset total: `./reset_environment.sh`

3. Procure ajuda online:

- Stack Overflow: tag `sfml` + `cmake`
- Reddit: `r/cpp`, `r/gamedev`
- Discord: servidores de C++ e game dev

4. Documente seu problema:

- Sistema operacional e versão
- Saída do script de diagnóstico
- Mensagens de erro completas
- Passos que já tentou

## Prevenção de Problemas

### Manutenção Regular

```
# Atualizar dependências mensalmente
sudo apt update && sudo apt upgrade # Linux
brew update && brew upgrade          # macOS

# Limpar builds antigos
find . -name "build*" -type d -mtime +30 -exec rm -rf {} \;
```

### Backup de Configuração

```
# Backup da configuração funcionando
tar -czf working_config_$(date +%Y%m%d).tar.gz \
    CMakeLists.txt setup.sh build/CMakeCache.txt
```

**Lembre-se:** A maioria dos problemas pode ser resolvida com `./setup.sh`. Em caso de dúvida, sempre comece pelo diagnóstico automático!