Universidade da Beira Interior
Departamento de Informática

Licenciatura em Engenharia Informática
Unidade Curricular: Computação Gráfica

Breakout 3D

Um Jogo de Arcade 3D com Motor Gráfico Próprio

Relatório Técnico

Autores:

Francisco Pereira (52129)
Daniel Cardoso (52218)

Covilhã, Dezembro de 2025

Resumo

O *Breakout 3D* é um jogo de arcade tridimensional desenvolvido em equipa, em C++17, recorrendo à API gráfica OpenGL 3.3 (Compatibility Profile). O projeto parte do clássico *Breakout* e expande a sua fórmula através de múltiplos modos de jogo, sistemas de progressão, uma interface gráfica completa e um conjunto de funcionalidades técnicas orientadas a renderização em tempo real.

Do ponto de vista gráfico, o motor implementa um pipeline baseado em shaders GLSL (GLSL 330), suportando transformações geométricas através de matrizes de modelo, vista e projeção, iluminação dinâmica com o modelo de Phong calculado por fragmento, e texturização com carregamento de imagens via `stb_image`. A interface gráfica é renderizada num segundo passe em projeção ortográfica, permitindo menus interativos, overlays contextuais (pausa, vitória, derrota) e elementos de HUD. A renderização de texto é realizada a partir de fontes TrueType (TTF) com `stb_truetype`, recorrendo a um *atlas* de glifos pré-processado para desempenho consistente.

Ao nível de gameplay, o sistema inclui deteção e resposta de colisões 3D (AABB–AABB e esfera–AABB), power-ups com efeitos temporizados e quatro modos distintos: **Normal**, **Levels** (suportado por um conjunto de 20 níveis), **Endless** (sobrevivência com inserção progressiva de tijolos e pontuação acumulada) e **Rogue** (sistema de cartas com modificadores persistentes ao longo da sessão). A aplicação inclui ainda suporte para conteúdos animados (previews em GIF), cuja descodificação é feita fora da thread principal, garantindo que a criação de recursos OpenGL ocorre apenas na thread detentora do contexto. O áudio é gerido por um sistema dedicado (com base em *miniaudio*), incluindo música, efeitos sonoros e *stingers* para momentos específicos.

O resultado final é uma aplicação interativa de tempo real que demonstra, de forma integrada, competências em Computação Gráfica, arquitetura modular (separação Engine/Game), gestão de estado, desempenho e desenvolvimento de sistemas interativos.

Conteúdo

Capítulo 1

Introdução

1.1 Descrição da Proposta

O *Breakout 3D* consiste numa reinterpretação moderna do clássico jogo de arcade *Breakout*, transportando a sua mecânica central para um ambiente tridimensional e suportando-a por um motor gráfico desenvolvido de raiz. Este projeto foi realizado no âmbito da unidade curricular de Computação Gráfica e tem como finalidade aplicar, em contexto prático, conceitos fundamentais de renderização em tempo real, incluindo pipeline gráfico, shaders, iluminação, texturização e interação com o utilizador.

A escolha de um jogo como problema-base tem duas vantagens principais. Em primeiro lugar, força a integração de múltiplos subsistemas (gráficos, input, estado de jogo e UI), aproximando o trabalho de um cenário real de desenvolvimento. Em segundo lugar, permite validar diretamente decisões técnicas: um motor gráfico pode ser funcional num teste isolado, mas um jogo em tempo real evidencia imediatamente questões de desempenho, consistência visual, robustez de colisões e clareza de interface.

Em termos de interação, o jogador controla uma pá e deve impedir a queda da bola, destruindo tijolos organizados em grelhas. Para além da mecânica tradicional, o projeto introduz modos alternativos e sistemas adicionais que aumentam a longevidade e diversidade do jogo, nomeadamente progressão por níveis, pontuação em sobrevivência e um sistema de cartas que altera regras de forma persistente.

1.2 Enquadramento e Breve História do Breakout

O *Breakout* é um dos jogos mais emblemáticos da era dos arcades, surgindo como evolução natural de jogos baseados em reflexão de bola, como o *Pong*. A sua premissa é simples e imediatamente compreensível: o jogador controla uma pá (*paddle*) que reflete uma bola para destruir blocos (*bricks*) dispostos na parte superior do ecrã. Apesar da simplicidade, a fórmula revelou-se extremamente eficaz devido ao ciclo de feedback rápido, ao aumento progressivo de dificuldade e à componente de precisão motora exigida.

Do ponto de vista histórico, o *Breakout* consolidou um padrão de design que influenciou vários jogos posteriores: progressão por níveis com padrões diferentes de obstáculos, mecânica de risco/recompensa (jogar agressivo para limpar mais rápido versus jogar seguro para evitar perder a bola) e a utilização de variações de velocidade e ângulo para criar profundidade sem aumentar significativamente a complexidade do modelo de interação.

A adaptação do *Breakout* para 3D permite explorar conceitos de Computação Gráfica com maior riqueza: modelação e transformação de objetos tridimensionais, iluminação e materiais, texturização, e ainda a necessidade de articular uma interface 2D (menus/HUD) sobre uma cena 3D. Neste projeto, a inspiração do jogo original é preservada ao nível mecânico, mas expandida através de modos de jogo, progressão estruturada e sistemas adicionais (power-ups, cartas e conteúdo multimédia).

1.3 Contexto Técnico e Ferramentas

A implementação foi realizada em C++17, tirando partido de uma abordagem modular. As principais tecnologias e bibliotecas utilizadas incluem:

- **OpenGL 3.3 Compatibility Profile** para renderização, com um pipeline baseado em shaders GLSL;
- **GLFW** para criação de janela, gestão do contexto OpenGL e captura de input (teclado e rato);
- **GLEW** para carregamento de extensões e funções OpenGL;
- **GLM** para matemática (vetores, matrizes e transformações);
- `stb_image` para carregamento de texturas (PNG/JPG);
- `stb_truetype` para renderização de texto com fontes TTF via *atlas* de glifos;
- **miniaudio** para reprodução e gestão de áudio (música, SFX e *stingers*);
- **Git/GitHub** para controlo de versões e integração de trabalho em equipa.

O jogo decorre predominantemente num plano XZ, utilizando o eixo Y para composição visual (elevação ligeira de modelos e separação da UI em modo ortográfico). Esta decisão simplifica a leitura do gameplay e reduz complexidade desnecessária, mantendo um aspeto tridimensional coerente.

1.4 Objetivos do Projeto

Os objetivos principais incluem:

- Implementar um motor gráfico funcional suportado por OpenGL 3.3 Compatibility Profile;
- Desenvolver um sistema de iluminação em tempo real baseado no modelo de Phong por fragmento;
- Integrar texturização e gestão de recursos (modelos, texturas e shaders);
- Implementar um sistema robusto de colisões 3D adequado a um jogo em tempo real;
- Construir uma camada de UI completa (menus, HUD, overlays) em projeção ortográfica;
- Suportar múltiplos modos de jogo, incluindo modos baseados em níveis e variantes de progressão;
- Integrar renderização de texto TTF em tempo real com foco em desempenho;
- Integrar um sistema de áudio modular com música, SFX e *stingers*;
- Garantir boas práticas de gestão de recursos, respeitando a regra do contexto OpenGL na thread principal.

1.5 Organização do Documento

Este relatório encontra-se organizado em **seis** capítulos principais:

- **Capítulo 1:** Apresenta o projeto, contexto e objetivos;
- **Capítulo 2:** Descreve engenharia de software, arquitetura, requisitos e organização do projeto;
- **Capítulo 3:** Detalha decisões de implementação e subsistemas técnicos;
- **Capítulo 4:** Apresenta os modos de jogo e o sistema Rogue;
- **Capítulo 5:** Reflete criticamente sobre dificuldades e soluções;
- **Capítulo 6:** Conclusões e trabalho futuro.

Capítulo 2

Engenharia de Software

Este capítulo descreve as decisões de engenharia de software adotadas no desenvolvimento do *Breakout 3D*. São apresentados os requisitos funcionais e não funcionais, a arquitetura geral do sistema, as regras de dependência entre módulos e a organização do projeto em termos de diretórios e componentes.

2.1 Requisitos do Sistema

2.1.1 Requisitos Funcionais

- **RF1** — Renderização de cenas tridimensionais em tempo real;
- **RF2** — Aplicação de iluminação dinâmica baseada no modelo de Phong;
- **RF3** — Carregamento e aplicação de texturas em objetos 3D;
- **RF4** — Gestão de câmara perspetiva com múltiplos modos;
- **RF5** — Sistema de colisões tridimensionais (AABB e esfera–AABB);
- **RF6** — Implementação de múltiplos modos de jogo (Normal, Endless, Rogue e Levels);
- **RF7** — Conjunto de 20 níveis jogáveis e acessíveis nos modos baseados em níveis (Normal e Levels);
- **RF8** — Sistema de cartas no modo Rogue com modificadores persistentes;
- **RF9** — Menus e interfaces gráficas interativas;
- **RF10** — Renderização de texto em tempo real a partir de fontes TTF;
- **RF11** — Suporte para conteúdos animados (previews em GIF) com descodificação fora da thread principal;
- **RF12** — Sistema de pontuação e feedback visual ao jogador;

- **RF13** — Sistema de áudio com música, SFX e *stingers*, integrado por eventos.

2.1.2 Requisitos Não Funcionais

- **RNF1** — Execução estável a 60 FPS em hardware moderno;
- **RNF2** — Arquitetura modular com separação clara entre motor gráfico e lógica de jogo;
- **RNF3** — Código legível, organizado e devidamente documentado;
- **RNF4** — Baixa latência na resposta a input do utilizador;
- **RNF5** — Gestão segura de recursos gráficos e de memória;
- **RNF6** — Compatibilidade com OpenGL 3.3 Compatibility Profile;
- **RNF7** — Respeito pela restrição do contexto OpenGL: criação de recursos OpenGL apenas na thread principal;
- **RNF8** — Facilidade de extensão para novos modos, níveis, cartas e conteúdos multimédia.

2.2 Arquitetura Geral do Sistema

O sistema encontra-se dividido em duas camadas principais:

- **Camada Engine** — infraestrutura gráfica e de baixo nível;
- **Camada Game** — lógica específica do Breakout 3D (modos, entidades, regras e UI do jogo).

2.3 Gestão do Projeto

O desenvolvimento do *Breakout 3D* foi realizado em equipa, com coordenação contínua e integração incremental de funcionalidades. A gestão do projeto teve como foco garantir uma evolução estável do sistema, evitando regressões e assegurando que os diferentes subsistemas (renderização, colisões, UI, modos de jogo e áudio) se integravam corretamente no ciclo principal de execução.

2.3.1 Metodologia e Abordagem de Desenvolvimento

Foi adotada uma abordagem incremental e orientada a marcos (*milestones*), onde cada fase introduziu um conjunto limitado de funcionalidades e validou o seu funcionamento antes de avançar. Esta estratégia é particularmente adequada para aplicações de tempo real, pois permite testar cedo aspetos críticos como desempenho, consistência visual e estabilidade do estado do jogo.

2.3.2 Cronograma de Alto Nível

O projeto foi desenvolvido por fases, seguindo um encadeamento lógico entre infraestrutura e funcionalidades:

- **Fase 1 — Base da Engine:** criação da janela e contexto OpenGL (GLFW), inicialização de extensões (GLEW), configuração do renderer e pipeline mínimo de shaders.
- **Fase 2 — Mundo 3D:** integração de câmara (matrizes View/Projection), carregamento de modelos e texturas, e iluminação dinâmica (Phong).
- **Fase 3 — Gameplay essencial:** implementação de entidades (bola, pá, tijolos), sistema de colisões e regras base do Breakout.
- **Fase 4 — UI e interação:** menus, HUD, overlays (pausa/vitória/-derrota) e renderização de texto com fontes TTF.
- **Fase 5 — Modos e extensões:** introdução dos modos Endless e Rogue, sistema de cartas, progressão e pontuação.
- **Fase 6 — Multimédia e polimento:** áudio (miniaudio), previews animados (GIF) e ajustes finais de usabilidade e desempenho.

2.3.3 Controlo de Versões e Integração

O controlo de versões com Git/GitHub foi utilizado para manter histórico, facilitar integração e reduzir risco de regressões. Esta prática permitiu testar alterações de forma incremental e manter o projeto coerente ao longo do desenvolvimento. A utilização de commits frequentes e integração regular ajudou a detetar problemas cedo, especialmente em áreas sensíveis como input/UI e sincronização associada a recursos OpenGL.

2.4 Camada Engine

A camada *Engine* concentra componentes genéricos relacionados com computação gráfica e infraestrutura base. Os principais módulos incluem:

- **Window** — Criação da janela, contexto OpenGL e gestão de eventos;
- **Renderer** — Configuração do pipeline gráfico e desenho;
- **Shader** — Compilação e gestão de shaders GLSL;
- **Mesh** — Representação de geometrias e buffers na GPU;
- **Texture** — Carregamento e binding de texturas;
- **Input** — Leitura de teclado e rato;
- **Time** — Gestão de tempo e *delta time*.

2.5 Camada Game

A camada *Game* contém a lógica específica do jogo. Os principais componentes incluem:

- **Game** — Classe principal que coordena atualização e renderização;
- **GameState** — Estrutura que representa o estado completo do jogo por frame;
- **GameConfig** — Parâmetros e constantes de gameplay;
- **Entidades** — Bola, pá, tijolos e power-ups;
- **Sistemas** — input, física, colisões, progressão, UI e regras específicas por modo;
- **AudioSystem** — gestão de música, SFX e *stingers* através de eventos.

2.6 Organização do Projeto

A estrutura de diretórios foi concebida para separar declarações, implementações e recursos:

```
Breakout3D/
        include/
              engine/
              game/
        src/
              engine/
              game/
        assets/
              shaders/
              models/
              textures/
              fonts/
              audio/
                    music/
                    sfx/
                    stingers_music/
              video/
        external/
            stb_*.h
            miniaudio.h
```

2.7 Conclusão do Capítulo

A separação entre motor gráfico e lógica do jogo facilita manutenção e extensibilidade, permitindo evoluir modos, UI e conteúdos sem comprometer o núcleo de rendering.

Capítulo 3

Implementação

Este capítulo descreve a implementação técnica do Breakout 3D, abordando pipeline de renderização, iluminação, câmara, UI, texto, áudio e considerações de threading.

3.1 Pipeline de Renderização

O sistema segue o modelo *Modelo–Vista–Projeção* (MVP), transformando vértices do espaço local para o *clip space*. Em cada frame:

1. Construção da matriz de modelo (`Model Matrix`);
2. Cálculo da matriz de vista (`View Matrix`) a partir da câmara;
3. Definição da matriz de projeção (`Projection Matrix`) em perspetiva;
4. Envio das matrizes para o shader via *uniforms*;
5. Execução dos shaders e escrita no framebuffer.

3.2 Descrição do Código e Fluxo de Execução

A implementação do Breakout 3D encontra-se organizada segundo uma separação clara entre **Engine** e **Game**. A camada *Engine* fornece infraestrutura reutilizável (janela, input, tempo e renderização), enquanto a camada *Game* define regras, entidades e modos de jogo.

3.2.1 Fluxo Principal por Frame

A aplicação segue o ciclo típico de um sistema de renderização em tempo real. Em cada frame:

1. **Atualização de tempo:** cálculo de *delta time* (diferença temporal desde o último frame), usado para movimento independente do framerate.

2. **Eventos e input:** leitura de teclado e rato (GLFW) e atualização do estado de input.
3. **Atualização do jogo:** execução de regras de gameplay, movimento, colisões, progressão e transições de estado.
4. **Renderização 3D:** desenho da arena, tijolos, pá, bola(s) e restantes entidades em perspetiva.
5. **Renderização 2D/UI:** desenho de HUD, menus e texto em projeção ortográfica.
6. **Apresentação:** *swap buffers* para mostrar o frame final.

Este fluxo permite manter a lógica do jogo desacoplada do código específico de OpenGL, promovendo modularidade e testabilidade.

3.2.2 Componentes Principais da Engine

Window A classe `Window` encapsula a criação e gestão do contexto OpenGL, bem como funções auxiliares associadas à janela (tamanho do framebuffer, *swap buffers*, pedido de fecho). Centraliza a inicialização GLFW e reduz dependência do jogo em tipos específicos da biblioteca.

Renderer O `Renderer` é responsável por configurar o estado global do OpenGL (por exemplo, *depth test*), iniciar e terminar frames, definir câmara (matrizes View/Projection) e desenhar meshes. As funções típicas incluem:

- `beginFrame(fbW, fbH)` — limpa buffers e define viewport;
- `setCamera(V, P, camPos)` — atualiza matrizes e posição do observador;
- `drawMesh(mesh, M)` / `drawMesh(mesh, pos, size)` — desenha geometria aplicando transformações;
- `beginUI(...)` / `endUI()` — configura passe ortográfico para UI;
- rotinas auxiliares para quads 2D e elementos de UI.

Shader e Recursos O módulo `Shader` compila e liga programas GLSL, oferecendo interface para envio de *uniforms*. Os módulos `Mesh` e `Texture` encapsulam buffers (VAO/VBO/EBO) e texturas OpenGL, garantindo que a camada *Game* não precisa manipular diretamente chamadas OpenGL de baixo nível.

Input e Time O `Input` mantém o estado de teclas/botões e fornece utilitários para coordenadas do rato no framebuffer. O `Time` calcula *delta time*, garantindo movimento consistente em diferentes framerates.

3.2.3 Camada Game: Estado, Regras e Entidades

Game e GameState A classe `Game` coordena o ciclo `update()` e `render()`. Toda a informação dinâmica do jogo (modo atual, vidas, pontuação, entidades e estado da UI) encontra-se agregada em `GameState`. Esta decisão facilita serialização mental do estado e reduz dependências entre funções, pois todos os sistemas leem/escrevem no mesmo conjunto de dados.

GameConfig A estrutura `GameConfig` armazena parâmetros constantes e ajustáveis (velocidades, limites da arena, tamanhos de entidades, probabilidades de power-ups, etc.). Isto separa valores de *tuning* da lógica, facilitando balanceamento sem reescrever regras.

Sistemas A lógica é implementada de forma modular (por exemplo, input, física, colisões, progressão), permitindo isolar responsabilidades e reduzir complexidade. Em particular:

- **Input/Controlo:** converte input em ações (mover pá, lançar bola, pausar, selecionar UI);
- **Movimento/Física:** integra posições com base em velocidades e *delta time*;
- **Colisões:** testa e resolve colisões esfera–AABB e AABB–AABB;
- **Regras por modo:** altera condições de progressão, inserção de tijolos e pontuação.

3.2.4 Renderização em Dois Passes

A renderização encontra-se dividida em dois passes para manter clareza visual:

- **Passe 3D (perspetiva):** arena, objetos e iluminação Phong;
- **Passe UI (ortográfico):** menus, overlays, HUD e texto TTF.

Esta separação simplifica o desenho de UI e evita interferência do *depth buffer* da cena 3D com elementos 2D.

3.3 Gestão da Câmara

A câmara garante visibilidade adequada da arena, ajustando-se à razão de aspeto do framebuffer. A matriz de vista é obtida com `lookAt`, usando posição, alvo (centro da arena) e vetor *up* global.

3.4 Iluminação Dinâmica — Modelo de Phong

A iluminação é calculada no shader de fragmento, combinando componentes ambiente, difusa e especular:

$$I = I_a \cdot k_a + I_d \cdot k_d \cdot \max(0, \vec{N} \cdot \vec{L}) + I_s \cdot k_s \cdot \max(0, \vec{R} \cdot \vec{V})^n$$

3.5 Shaders

O motor utiliza um par de shaders (vértice e fragmento) carregado a partir de ficheiros externos. O mesmo programa de shader é reutilizado tanto para o mundo 3D como para UI e texto, recorrendo a *uniforms* para alterar o modo de amostragem/iluminação (por exemplo, parâmetros distintos para elementos do HUD versus geometria do mundo).

3.6 Texturização

As texturas são carregadas com `stb_image` e enviadas para a GPU, configurando filtros, mipmaps e modos de repetição. O shader decide, via *uniforms*, se a textura deve ser aplicada ou se a cor/material é usada diretamente.

3.7 Renderização de Interface Gráfica (UI)

A UI é renderizada num segundo passe ortográfico após o desenho da cena 3D. Durante este passe:

- o *depth test* é desativado;
- a projeção é ortográfica;
- os elementos são desenhados como quads 2D em coordenadas de ecrã.

3.8 Renderização de Texto

O texto é renderizado com `stb_truetype` usando um *atlas* de glifos gerado na inicialização. Cada carácter é um quad texturizado em projeção ortográfica, usando métricas do glyph para posicionamento consistente.

3.9 Sistema de Áudio

O Breakout 3D integra um sistema de áudio dedicado, baseado em *miniaudio*, responsável por música de fundo, efeitos sonoros e *stingers* em momentos específicos (vitória, derrota, transições).

3.9.1 Tipos de Áudio

- **Música de fundo** — faixas em loop durante menus e gameplay;
- **Efeitos sonoros (SFX)** — sons curtos associados a eventos (colisões, tijolos, power-ups, perda de vida);
- **Stingers** — excertos curtos para momentos-chave (vitória/derrota, etc.).

3.9.2 Organização e Gestão

Os ficheiros encontram-se organizados em `assets/audio/music/`, `assets/audio/sfx/` e `assets/audio/stingers_music/`. A reprodução é desencadeada por eventos do jogo (por exemplo, colisão, recolha de power-up, *game over*), mantendo o sistema desacoplado do renderer.

3.9.3 Considerações de Desempenho

Efeitos frequentemente usados são mantidos prontos a reproduzir, reduzindo latência. O controlo de volumes por categoria (música/SFX/*stingers*) permite equilibrar a mistura sonora sem impacto no desempenho do render.

3.10 Multithreading e Recursos OpenGL

Determinados conteúdos multimédia (nomeadamente previews animados em GIF) requerem descodificação que pode ser dispendiosa. Para manter a fluidez da aplicação, a descodificação é feita fora da thread principal. No entanto, a criação de recursos OpenGL (texturas, buffers, etc.) ocorre sempre na thread detentora do contexto, evitando condições de corrida e comportamento indefinido.

3.11 Conclusão do Capítulo

A implementação demonstra a integração coerente de pipeline, shaders, iluminação, UI, texto e áudio, com atenção às restrições de tempo real e às regras do contexto OpenGL.

Capítulo 4

Modos de Jogo e Sistema Rogue

O Breakout 3D disponibiliza quatro modos distintos, partilhando a mesma infraestrutura técnica (renderização, colisões, input), mas com regras e objetivos próprios.

4.1 Visão Geral dos Modos de Jogo

- **Normal Mode**;
- **Levels Mode**;
- **Endless Mode**;
- **Rogue Mode**.

4.2 Normal Mode

O *Normal Mode* corresponde ao modo clássico do Breakout com condições de vitória claras por nível: destruir todos os tijolos e evitar perder todas as vidas. Este modo utiliza o conjunto de **20 níveis**, permitindo jogar de forma direta e previsível.

4.2.1 Regras Fundamentais

- Número fixo de vidas no início;
- Lançamento da bola após interação do utilizador;
- Tijolos com durabilidade, reduzida a cada colisão;
- O nível termina ao destruir todos os tijolos;
- A perda de todas as vidas resulta em *Game Over*.

4.3 Levels Mode

O *Levels Mode* introduz progressão estruturada através do mesmo conjunto de **20 níveis**. Ao contrário do Normal, o foco é na progressão e continuidade da experiência ao longo dos níveis, mantendo uma curva de dificuldade crescente.

4.3.1 Variação entre Níveis

Cada nível varia em:

- Disposição espacial e densidade de tijolos;
- Durabilidade dos tijolos;
- Parâmetros iniciais (por exemplo, velocidade inicial da bola).

4.3.2 Estrutura de Dificuldade

Os níveis podem ser agrupados por complexidade:

- Níveis introdutórios (1–10);
- Níveis intermédios (11–15);
- Níveis avançados (16–20).

4.4 Endless Mode

No *Endless Mode*, não existe uma condição de vitória fixa. O jogo introduz novas linhas de tijolos progressivamente, aumentando a pressão sobre o jogador.

4.4.1 Mecânica de Progressão

- Inserção de linhas após um número definido de destruições;
- Inserção automática de linhas após determinado intervalo de tempo.

4.4.2 Pontuação

A pontuação é acumulativa e incentiva consistência (por exemplo, *streaks* e sequências de destruições sem perdas).

4.5 Rogue Mode

O *Rogue Mode* introduz uma camada de variabilidade e progressão por sessão através de um sistema de cartas. Ao completar ondas/objetivos intermédios, o jogador escolhe uma carta entre várias opções, aplicando modificadores persistentes até ao final da sessão.

4.5.1 Exemplos de Modificadores

- Velocidade da bola;
- Dimensões e controlo da pá;
- Frequência e tipo de power-ups;
- Penalizações e benefícios permanentes durante a sessão.

4.6 Integração Técnica dos Modos

Todos os modos utilizam a mesma base de dados (`GameState`) e sistemas partilhados, sendo as diferenças implementadas por regras específicas por modo, evitando duplicação e reforçando a modularidade.

4.7 Conclusão do Capítulo

A variedade de modos acrescenta profundidade e longevidade ao projeto, permitindo explorar progressão, sobrevivência e variabilidade estratégica com base numa infraestrutura técnica comum.

Capítulo 5

Reflexão Crítica e Problemas Encontrados

5.1 Objetivos Propostos vs. Objetivos Alcançados

Tabela 5.1: Objetivos propostos e respetivo estado de implementação

Objetivo	**Estado**
Motor gráfico próprio com OpenGL 3.3 (Compatibility)	Concluído
Iluminação dinâmica (modelo de Phong)	Concluído
Sistema de colisões tridimensionais	Concluído
Renderização de UI e texto TTF	Concluído
Múltiplos modos de jogo	Concluído
Sistema de cartas (Rogue Mode)	Concluído
Arquitetura modular Engine/Game	Concluído
Previews GIF com descodificação fora da thread principal	Concluído
Sistema de áudio (música, SFX, stingers)	Concluído

5.2 Problemas Encontrados e Soluções

5.2.1 Diferenças entre Sistemas de Coordenadas

Problema: O OpenGL considera a origem do ecrã no canto inferior esquerdo, enquanto o input do rato tipicamente reporta a origem no canto superior esquerdo, causando inconsistências em cliques na UI.

Solução: Foi implementada uma conversão consistente de coordenadas do rato para o sistema usado na renderização ortográfica da UI.

5.2.2 Restrições do Contexto OpenGL em Multithreading

Problema: A criação de recursos OpenGL fora da thread principal pode causar comportamento indefinido, uma vez que o contexto OpenGL deve ser usado pela thread que o detém.

Solução: Separou-se o processo em duas fases:

- Descoding/processamento de dados (por exemplo, frames de GIF) fora da thread principal;
- Criação de texturas/buffers OpenGL exclusivamente na thread principal.

5.2.3 Desempenho na Renderização de Texto

Problema: Gerar glyphs a cada frame é dispendioso.

Solução: Adoção de um *atlas* de glifos pré-processado, reduzindo a renderização a quads com amostragem de textura.

5.3 Reflexão Crítica

O desenvolvimento consolidou conhecimentos em pipeline gráfico, shaders, arquitetura modular e programação em tempo real, com especial atenção a desempenho, organização e robustez.

Capítulo 6

Conclusões e Trabalho Futuro

6.1 Conclusões

O Breakout 3D constitui um projeto académico sólido, demonstrando a aplicação prática dos conceitos lecionados na unidade curricular de Computação Gráfica. A implementação de um motor gráfico próprio, aliada a uma arquitetura modular, permitiu desenvolver um jogo funcional, visualmente coerente e tecnicamente robusto.

6.2 Pontos Fortes do Projeto

- Separação clara entre motor gráfico e lógica de jogo;
- Implementação correta do modelo de iluminação de Phong;
- Diversidade de modos (Normal, Levels, Endless e Rogue);
- Sistema de cartas com modificadores persistentes no Rogue;
- UI e texto TTF eficientes em projeção ortográfica;
- Integração de áudio com música, SFX e *stingers*;
- Previews GIF com abordagem segura face às restrições do contexto OpenGL.

6.3 Trabalho Futuro

6.3.1 Melhorias Técnicas

- Efeitos visuais avançados (partículas, sombras dinâmicas);
- Otimizações adicionais ao renderer e à UI;
- Melhorias de ferramentas internas para facilitar criação de níveis e conteúdos.

6.3.2 Expansão de Gameplay

- Novos power-ups e cartas;
- Editor de níveis com interface gráfica;
- Estatísticas persistentes e conquistas;
- Possível suporte a modos multijogador.

Bibliografia

- Akenine-Möller, T., Haines, E., Hoffman, N. *Real-Time Rendering*. CRC Press.
- Gregory, J. *Game Engine Architecture*. CRC Press.
- Khronos Group. OpenGL Documentation. https://www.khronos.org/opengl/
- LearnOpenGL. https://learnopengl.com/
- GLFW Documentation. https://www.glfw.org/documentation.html
- GLM Manual. https://glm.g-truc.net/