**Universidade da Beira Interior**
Departamento de Informática


# Bohr: *Very Small PDB Molecular Visualizer*

Elaborado por:

**41266 — Diogo Castanheira Simões**
**41381 — Igor Cordeiro Bordalo Nunes**

Orientador:

**Professor Doutor Abel João Padrão Gomes**

14 de Janeiro de 2020

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Acrónimos

**API** *Application Programming Interface*
**CG** Computação Gráfica
**GLM** *OpenGL Mathematics*
**GLSL** *OpenGL Shader Language*
**PDB** *Protein Database*
**SWOT** *Strength, Weakness, Opportunity, and Threat Analysis*
**TI** Tecnologias de Informação
**UC** Unidade Curricular

# Capítulo 1

# Introdução

## 1.1 Descrição da proposta

Várias áreas científicas de elevada importância na sociedade moderna dependem do estudo ao nível molecular e atómico dos componentes que sustentam a vida na Terra. Entre elas destacam-se em particular a indústria farmacêutica, a biotecnologia e as áreas afetas como a química orgânica.

Contudo, apesar da existência de vários *softwares* no mercado para a visualização e estudo destas, é de grande interesse na área da Computação Gráfica (CG) perceber como se pode renderizar estas moléculas de forma a poderem ser observadas virtualmente e estudadas.

Neste sentido, o grupo desafiou-se a desenvolver um visualizador molecular simplificado de ficheiros *Protein Database* (PDB) no âmbito da Unidade Curricular (UC) de CG ao invés dos projetos clássicos propostos inicialmente.

A proposta inicial consiste num visualizador molecular que renderize a superfície implícita *pi* [1]. Contudo, devido a contratempos descritos *a posteriori* no presente documento, a implementação final centra-se na renderização da superfície de *van der Walls*.

## 1.2 Constituição do grupo

O presente projeto foi realizado pela equipa constituída pelos elementos listados na Tabela 1.1. O trabalho realizado por cada membro é descrito na Secção 4.3.

| Nº | Nome |
|---|---|
| 41266 | Diogo Castanheira Simões |
| 41381 | Igor Cordeiro Bordalo Nunes |

Tabela 1.1: Constituição do grupo.

## 1.3 Organização do Documento

De modo a refletir o projeto realizado, este relatório encontra-se estruturado em cinco capítulos:

1. No primeiro capítulo — **Introdução** — é apresentado o projeto, em particular os seus objetivos, a equipa desenvolvedora, a respetiva organização do relatório.

2. No segundo capítulo — **Tecnologias utilizadas** — são delineadas as tecnologias utilizadas durante o seu desenvolvimento.

3. No terceiro capítulo — **Desenvolvimento e Implementação** — são descritas as escolhas e os detalhes de implementação da aplicação.

4. No quarto capítulo — **Reflexão Crítica e Problemas Encontrados** — são indicados os objetivos alcançados, quais as tarefas realizadas por cada membro do grupo, assim como são expostos os problemas enfrentados e é feita uma reflexão crítica sobre o trabalho.

5. No quinto capítulo — **Conclusões e Trabalho Futuro** — são analisados os conhecimentos adquiridos ao longo do desenvolvimento do projeto e, em contrapartida, o que não se conseguiu alcançar e que poderá ser explorado futuramente.

# Capítulo 2

# Tecnologias utilizadas

## 2.1 Introdução

Antes da implementação propriamente dita da aplicação, é necessário estabelecer quais as ferramentas e as tecnologias utilizadas a fim de alcançar os objetivos propostos para o projeto. Este Capítulo aborda, portanto, este tópico.

## 2.2 Ferramentas e tecnologias utilizadas

As ferramentas utilizadas no âmbito da realização do projeto, sumariadas na Tabela 2.1, visam três componentes essenciais na sua gestão: 1) aplicação *OpenGL*®, 2) relatório, e 3) controlo de versões.

Segue-se uma descrição do uso de cada uma das tecnologias:

- ***OpenGL*® [2]**: *Application Programming Interface* (API) multi-plataforma e com suporte a múltiplas linguagens de programação para a renderização de gráficos vetoriais 2D e 3D com recurso à placa gráfica;
- **GLFW [3]**: API simplificada para o *OpenGL*®, igualmente multi-plataforma, permitindo a gestão de janelas, contextos, superfícies e comandos (rato, teclado e *joystick*);
- **GLM [4]**: biblioteca matemática baseada na linguagem dos *shaders* do *OpenGL*®, *OpenGL Shader Language* (GLSL).
- **GLAD! (GLAD!) [5, 6]**: gerador automático de *loaders* para *OpenGL*®;
- ***FreeType* [7]**: biblioteca de desenvolvimento dedicada à renderização de fontes em *bitmaps* utilizáveis, por exemplo, pelo *OpenGL*®.

| *Software* / Tecnologia | Versão |
|---|---|
| **Aplicação *OpenGL*®** | |
| *OpenGL*® | 4.6 |
| GLFW | 3.3.2 |
| GLAD | 0.1.34 |
| GLM | 0.9.9.8 |
| *FreeType* | 2.10.4 |
| **Relatório** | |
| XeTEX | 3.14159265-2.6-0.999991 |
| *TeXstudio*© | 3.0.1 |
| **Controlo de versões** | |
| *git* | 2.17.1 |
| *GitKraken* | 7.4.1 |

Tabela 2.1: Ferramentas e tecnologias utilizadas, organizadas por categoria.

## 2.3 Código *open source*

Foi utilizado código *open source*, adaptado para o projeto sob as respetivas licenças, para as seguintes funcionalidades:

- ***Sphere*** [8, 9]: obtido dos códigos de exemplo do *OpenGL Shading Language Cookbook, 2nd Edition* para renderizar esferas sem recurso a ficheiros `*.obj` criados externamente;
- ***osdialog*** [10]: biblioteca multi-plataforma para acesso facilitado às caixas de diálogo do sistema operativo, com o fim de permitir abrir qualquer ficheiro PDB dinamicamente a qualquer momento e mostrar mensagens de erro ao utilizador.

O projeto *Bohr* será por sua vez disponibilizado sob a licença *GNU General Public License, version 3* [11], após apresentação e caso seja autorizado pelo Professor regente.

## 2.4 Conclusões

Após delineadas as ferramentas e tecnologias a utilizar, segue-se a fase de desenvolvimento do projeto, descrito no Capítulo seguinte. A utilização de código *open source* e de ferramentas bem conhecidas e testadas pela comunidade mundial permitirá, em princípio, uma implementação eficiente.

# Capítulo 3

# Desenvolvimento e Implementação

## 3.1 Introdução

A fase de implementação, que se estendeu por cerca de 2 semanas, envolveu a execução paralela de diferentes tarefas pelos dois elementos do grupo. Este Capítulo aborda em particular os seguintes aspetos desta fase do projeto:

- Escolhas de implementação (Secção 3.2): explica as decisões feitas durante a implementação do código-fonte;
- Detalhes de implementação (Secção 3.3): explora os detalhes mais importantes e/ou interessantes no código-fonte.

Adicionalmente, são descritos o manual de instalação (Secção 3.4) e o manual de utilização (Secção 3.5).

## 3.2 Escolhas de Implementação

A primeira e grande escolha de implementação que se faz notar ao longo de todo o código é a utilização de classes e das bibliotecas providenciadas pelo C++ a fim de tirar o máximo partido desta linguagem de programação.

Desta forma, o código-fonte está organizado nas seguintes pastas:

- `bin`: inclui o binário compilado e as seguintes pastas:
  - `fonts`: localização das fontes utilizadas para renderizar texto;
  - `shaders`: alberga os *fragment shaders* e os *vertex shaders* para a renderização das moléculas e do texto.

- `deps`: pasta gerada pelo *Makefile* para gerir as dependências durante a compilação;
- `doc`: aloja a documentação do projeto (nomeadamente o presente relatório);
- `include`: pasta com os *header files* de todos os métodos e classes utilizados no projeto;
- `obj`: pasta gerada pelo *Makefile* para guardar os ficheiros objeto a serem compilados no executável final;
- `src`: alberga todos os códigos-fonte (ficheiros `*.cpp`) com a implementação dos métodos e das classes declaradas nos *header files* da pasta `include`.

## 3.3 Detalhes de Implementação

O ficheiro principal, `bohr.cpp`, aloja a função `main()` de onde o programa arranca. Aqui são invocadas as seguintes funções de inicialização:

```
1 GLFWwindow* initialize_glfw(int width, int height,
  ↪  const char* title);
2 int initialize_glad(void);
```

Estas funções inicializam, respetivamente, as bibliotecas GLFW e GLAD, necessárias para a comunicação com a API do *OpenGL*®. Caso a primeira função retorne `NULL` ou a segunda retorne `0`, é gerada uma mensagem de erro na linha de comandos e o programa é abortado. A função de inicialização do GLFW em particular associa as funções de *callback* para o teclado e para o rato.

De seguida são compilados os *shaders* de renderização das moléculas com recurso à classe *Shader* (implementada em `shader_m.cpp`):

```
1 debug("Loading shaders...\n");
2 Shader lightingShader =
  ↪  Shader("shaders/lighting_vs.glsl",
  ↪  "shaders/lighting_fs.glsl");
```

Esta classe fornece métodos rápidos e muito acessíveis para manipular os dados passados aos *shaders* e a respetiva utilização.

É ainda inicializado o renderizador de texto:

```
TextRenderer textrenderer = TextRenderer(SCR_WIDTH,
 ↪  SCR_HEIGHT);
try {
    textrenderer.Load("fonts/UbuntuMono-R.ttf", 24);
} catch (const std::exception &e) {
    std::cerr << e.what() << '\n';
}
```

A classe *TextRenderer* está disponível *online* [12] para livre utilização, assim como os *shaders* para a renderização deste mesmo texto.

Por fim, o programa entra no seu ciclo de renderização. Este começa por limpar o *buffer* atual e renderiza os textos, nomeadamente as instruções no canto superior esquerdo e o nome do ficheiro (se aberto) no canto inferior esquerdo.

De imediato é processado o *input* de teclado do utilizador com a seguinte função:

```
action processInput(GLFWwindow *window, char **fname);
```

Esta função ajusta os parâmetros da câmara conforme as teclas selecionadas, mas em particular trata de abrir a janela de diálogo para abrir um ficheiro novo com o seguinte excerto de código:

```
if (glfwGetKey(window, GLFW_KEY_O) == GLFW_PRESS) {
    switchModeView(window, false);
    if (*fname != NULL) free(*fname);
    *fname = openPDBFileDialog();
    switchModeView(window, true);
    return (*fname != NULL) ? action::OPEN_FILE :
     ↪  action::NO_ACTION;
}
```

O modo de visualização é temporariamente alterado a fim de permitir que o cursor do rato seja visível durante a seleção do ficheiro uma vez que este é oculto durante a renderização da molécula. Após obter o caminho para o ficheiro pretendido, a função verifica se porventura não será `NULL`: neste caso, a molécula renderizada atualmente mantém-se e não é apagada da memória. Caso contrário, é devolvido um enumerador do tipo `action` que indica que o utilizador selecionou um novo ficheiro:

```
typedef enum {
    NO_ACTION,
    CAMERA_RESET,
    OPEN_FILE
} action;
```

No ciclo de renderização é feita uma análise deste retorno a fim de determinar a próxima ação. O *reset* da câmara envolve a invocação da seguinte chamada:

```
camera = molecule.resetCamera();
```

Esta função será vista mais à frente.

Caso seja aberto um novo ficheiro, a sua extensão é verificada como forma de validação primária:

```
bool isPBD(char *fname) {
    return
    string(std::experimental::filesystem::
        path(fname).extension())
        .compare(".pdb") == 0;
}
```

Sendo um ficheiro PDB, a seguinte linha de código irá carregar uma nova molécula a partir do ficheiro selecionado pelo utilizador:

```
    molecule = Molecule().fromPDB(fname);
```

Por fim, a molécula é renderizada de forma a apresentar a superfície de *van der Walls*:

```
molecule.render_vanderWalls(lightingShader, camera,
 ↪  screen_width, screen_height, molrotx, molroty);
```

À classe *Molecule*, implementada no ficheiro `pdbreader.cpp`, são delegadas todas as seguintes tarefas:

- Ler o ficheiro PDB e fazer o respetivo *parsing* a fim de obter os átomos e as respetivas coordenadas;
- Obter os dados relativos a cada átomo dada a tabela periódica implementada em `ptable.h`, nomeadamente o raio de *van der Walls* [13] e a cor CPK [14];
- Computar as esferas associadas a cada átomo dado o raio determinado anteriormente;
- Renderizar as esferas nas posições correspondentes às coordenadas do respetivo átomo e com a respetiva cor;

- Determinar a posição inicial da câmara dada a dimensão da molécula processada através do método `resetCamera()`.

Após estudo do formato de ficheiros PDB, determinou-se que o *parsing* seria simplificado uma vez que as posições de cada dado é esperado exatamente sempre na mesma posição de cada linha, em particular:

- Três coordenadas nas posições 30, 38 e 46 da linha;
- Símbolo atómico na posição 76.

Só as linhas começadas por `ATOM` ou `HETATM` são interpretadas uma vez que são estas aquelas que contêm os dados relativos aos átomos. Uma vez que só é pretendida a superfície, não é necessário calcular as ligações entre átomos (o formato PDB não fornece esta informação explicitamente).

Dada a política determinada anteriormente na Secção 3.2, a classe *Molecule* **não** é responsável por calcular os pontos das esferas. Uma classe própria existe para o efeito, *VBOSphere*, sendo então criadas instâncias desta classe num vetor:

```
1 bool Molecule::generateSpheres(void) {
2     this->spheres = vector<VBOSphere>();
3     for (auto atom : this->atoms) {
4         this->spheres.push_back(VBOSphere(atom.radius,
          ↪  50, 50,
          ↪  PeriodicTable::getColorFromSymbol(atom.name)
5             .toVec3()));
6     }
7     return true;
8 }
```

Classes e estruturas (*structs*) foram criadas para lidar com pontos e cores a fim de fornecer métodos imediatos para os converter no tipo de dados `glm::vec3` através da função `toVec3()`, comum a ambas as classes.

A renderização da molécula passa por um ciclo que percorre todas as esferas anteriormente computadas, carregando no respetivo *shader* a cor de cada esfera e a posição da câmara como fonte de luz. É de igual forma feita a transformação no mundo através de rotação, translação e escala (exatamente por esta ordem) a fim de posicionar a esfera no local exato. Bastará nesta fase invocar o seguinte método:

```
1 this->spheres[i].render();
```

O resultado final é semelhante ao do exemplo da Figura 3.4.

## 3.4 Manual de Instalação

O projeto foi implementado primariamente para o sistema operativo *Linux*. Em especial, foi utilizado o ***Linux Mint* 20.0** com as mais recentes atualizações em dia.

A fim de compilar o projeto, é necessário ter as seguintes bibliotecas instaladas no sistema:

- `g++`;
- `make`;
- `libglfw3-dev`;
- `libglm-dev`;
- `freetype`.

O *Makefile* incluído tratará da compilação de todos os ficheiros de forma automática, sendo fornecidos três modos:

1. `make debug`: compila no modo *debug*, o qual providencia informações adicionais durante a execução para fins de desenvolvimento;
2. `make release`: compila no modo *release*, ou seja, cria uma versão final para o utilizador final;
3. `make clean`: elimina todas as dependências, ficheiros objeto e executáveis a fim de se poder realizar uma compilação fresca de seguida.

## 3.5 Manual de Utilização

Com o projeto compilado, é **peremptória** a execução do programa através da linha de comandos **dentro da pasta** onde se encontra o executável. A sua execução é alcançada da seguinte forma:

```
./bohr
```

Ao abrir, a aplicação apresenta apenas as opções disponíveis e indica no fundo a informação "*No file opened*" (Figura 3.1). As opções são as seguintes:

- Tecla `O`: abrir ficheiro PDB;
- Teclas `WASD`: mover a câmara;
- Rato: rodar a câmara;
- Teclas `2468` ou setas: rodar a molécula;
- Teclas `ESC` ou `Q`: fechar a aplicação.


Figura 3.1: Aplicação após abertura ou quando não tem uma molécula aberta para renderização.

Com a opção O (abrir ficheiro) é aberto um diálogo do sistema operativo para escolher um ficheiro à discrição do utilizador (Figura 3.2). Caso o ficheiro selecionado não seja um ficheiro PDB, é apresentada uma mensagem de erro num diálogo gráfico (Figura 3.3). Contudo, caso o ficheiro seja válido, a aplicação irá interpretar os dados e renderizar a nova molécula (exemplo na Figura 3.4).

## 3.6 Conclusões

A exposição dos pontos mais importantes relacionados com a fase de implementação da aplicação *Bohr* permitiu ao grupo fazer uma retrospetiva do seu trabalho e perceber quais foram os pontos fortes e os pontos fracos do resultado final. Tal abre a porta para a última fase do projeto, uma fase sem código nem questões técnicas: a reflexão crítica.

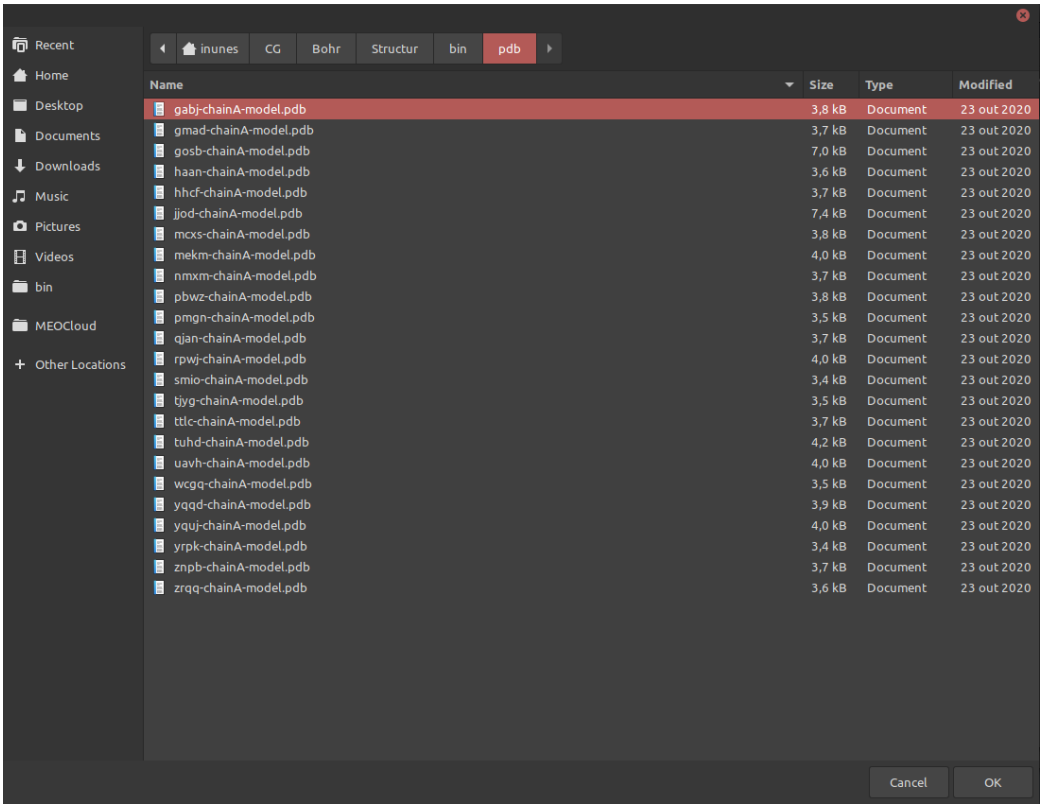


Figura 3.2: Janela de diálogo do sistema operativo para abertura de um novo ficheiro PDB.

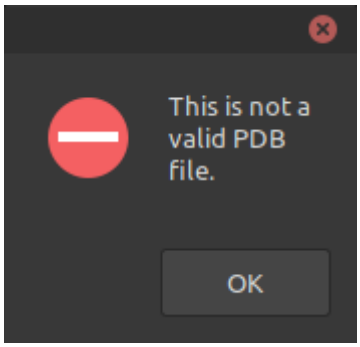


Figura 3.3: Mensagem de erro caso o ficheiro não seja válido.

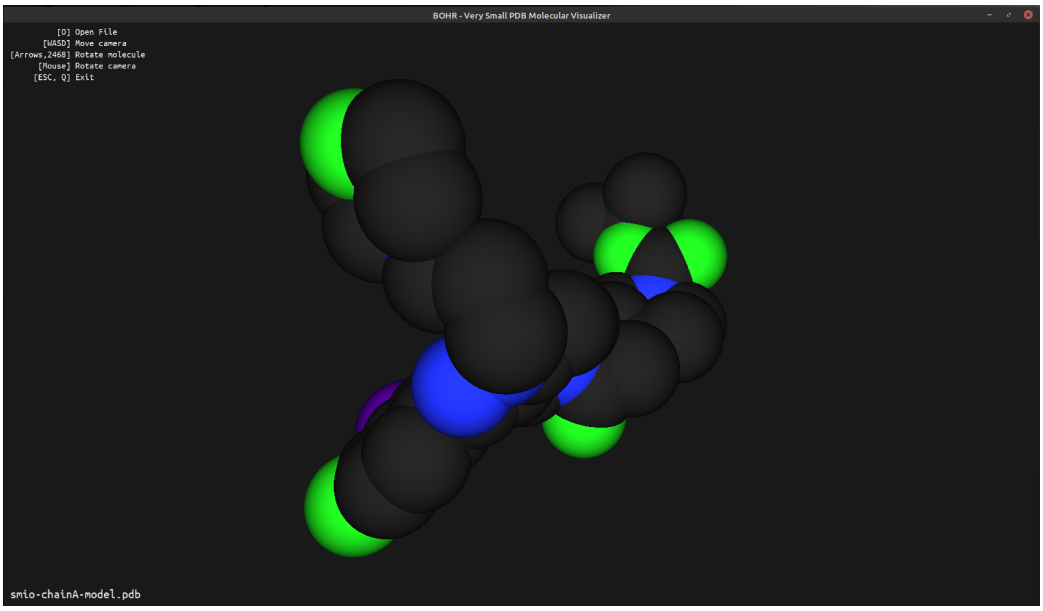


Figura 3.4: Exemplo de uma molécula renderizada na aplicação após interpretação do ficheiro PDB.

# Capítulo 4

# Reflexão Crítica e Problemas Encontrados

## 4.1 Introdução

Não obstante o planeamento feito *a priori*, o projeto *Bohr*, tal como qualquer outro na área das Tecnologias de Informação (TI), enfrentou alguns contratempos e, devido a problemas de saúde e pessoais severos por parte de um dos membros do grupo, não se revelou possível almejar todas as ambições inicialmente imaginadas. É preciso, pois, refletir sobre o desenvolvimento deste projeto.

Neste Capítulo são, portanto, explorados os seguintes tópicos:

- Objetivos propostos vs. alcançados (Secção 4.2): compara os objetivos inicialmente propostos com aqueles que foram concluídos no projeto final;
- Divisão de trabalho pelos elementos do grupo (Secção 4.3): lista as tarefas realizadas por cada elemento da equipa;
- Problemas encontrados (Secção 4.4): na sequência da Secção 4.2, explora os problemas encontrados durante a implementação da aplicação;
- Reflexão crítica (Secção 4.5): é feita uma *Strength, Weakness, Opportunity, and Threat Analysis* (SWOT) em retrospetiva pela equipa acerca do projeto.

## 4.2 Objetivos Propostos vs. Alcançados

A Tabela 4.1 expõe os objetivos propostos inicialmente para o projeto e identifica quais foram alcançados na sua plenitude, quais foram alcançados apenas parcialmente, e quais não tiveram sucesso.

| Objetivo proposto | Alcançado? |
|---|---|
| *Parsing* de ficheiros PDB | • |
| Computação das esferas | • |
| Obtenção do raio de cada átomo | • |
| Obtenção da cor CPK de cada átomo | • |
| Renderização de uma esfera | • |
| Renderização de uma molécula | • |
| Rotação de uma molécula | • |
| Movimentação da câmara | • |
| Abertura dinâmica de ficheiros por interface gráfica | • |
| Renderização da superfície de *van der Walls* | • |
| Suporte a proteínas | ○ |
| Renderização da superfície implícita *pi* | – |
| Utilização regular do *Trello* | – |

Tabela 4.1: Objetivos propostos e respetiva indicação de sucesso.
*Legenda.* • Alcançado em pleno; ○ Alcançado parcialmente. – Não alcançado.

| Tarefa | DS | IN |
|---|---|---|
| *Parsing* de ficheiros PDB | | • |
| Computação das esferas | | • |
| Obtenção do raio de cada átomo | • | |
| Obtenção da cor CPK de cada átomo | • | |
| Renderização de uma esfera | | • |
| Renderização de uma molécula | | • |
| Rotação de uma molécula | • | |
| Movimentação da câmara | • | |
| Abertura dinâmica de ficheiros por interface gráfica | | • |
| Renderização da superfície de *van der Walls* | | • |

Tabela 4.2: Distribuição de tarefas pelos elementos do grupo.
*Legenda.* • principal responsável; ○ auxiliou. DS: Diogo Simões; IN: Igor Nunes.

## 4.3 Divisão de Trabalho pelos Elementos do Grupo

A fim de agilizar o desenvolvimento do projeto, as tarefas foram distribuídas de forma balanceada pelos dois elementos da equipa (Tabela 4.2). De notar que, apesar da divisão de tarefas, a participação do grupo no seu todo foi essencial nos momentos de entreajuda, esclarecimento de dúvidas e consolidação da aplicação final.

## 4.4 Problemas Encontrados

A implementação da aplicação levou a que o grupo encontrasse alguns problemas e contratempos, os quais teve de ultrapassar a fim de terminar o projeto. Os problemas mais notáveis são resumidos na Tabela 4.3, incluindo as soluções encontradas para os ultrapassar.

| **Problema** | **Solução** |
|---|---|
| *Parsing* dos ficheiros PDB. | Estudo do formato segundo descrito em diversos fóruns *online* a fim de determinar as posições exatas dos dados relativo a cada átomo. |
| Procura do método mais eficiente para renderizar uma esfera. | Utilização da classe *Sphere* disponibilizada no *OpenGL Shading Language Cookbook, 2nd Edition* [9] a fim de evitar a importação de objetos externos. |
| Determinação da superfície implícita *pi*. | Sem solução encontrada. É necessário mais estudo. |
| Renderização da superfície implícita *pi* e dificuldade do algoritmo *marching cubes*. | Adoção do algoritmo *marching triangles* com recurso ao código *open source* disponível em [15]. Solução não presente na aplicação final devido ao problema anterior não solucionado. |

Tabela 4.3: Problemas encontrados durante o desenvolvimento da aplicação e respetivas soluções adotadas.

## 4.5 Reflexão Crítica

Tendo por objetivo expor a reflexão da *Lunáticos* face ao trabalho enveredado no desenvolvimento da aplicação *Bohr*, propõe-se efetivá-la com uma análise SWOT.

### 4.5.1 Pontos Fortes

1. A aplicação permite abrir uma nova molécula a qualquer momento.
2. É utilizada interface gráfica do sistema operativo para interagir com o utilizador.
3. As esferas são calculadas dinamicamente pela aplicação sem recurso a objetos externos.
4. O renderizador tem em conta a cor de cada átomo e o respetivo raio de *van der Walls*, mostrando assim a superfície de *van der Walls*.

### 4.5.2 Pontos Fracos

1. O *parser* PDB não contempla ficheiros relativos a proteínas.
2. O mesmo código-fonte compilado em diferentes sistemas operativos não garante exatamente o mesmo comportamento em relação à câmara e ao teclado.
3. Não é calculada a superfície implícita *pi*, não sendo então possível renderizar esta.
4. Só é renderizado um tipo de superfície molecular.

### 4.5.3 Ameaças

1. O *software* não contempla a possibilidade de fechar a molécula atualmente aberta nem se mostrar várias moléculas lado-a-lado.
2. O uso de interface gráfica não tem um comportamento garantido em *cross-platform*.
3. Para moléculas de grandes dimensões, a aplicação poderá facilmente tornar-se exigente em termos de recursos de memória.
4. Já existem variados *softwares* de visualização molecular, apesar da pertinência deste projeto no âmbito da UC de CG.

### 4.5.4 Oportunidades

1. Adicionar suporte para aminoácidos.
2. Adaptar com *flags* do pré-processador do C++ o código-fonte a fim de ultrapassar as limitações em cada sistema operativo.
3. Determinar a superfície implícita *pi* a fim de dar uso ao algoritmo *marching triangles* estudado para este projeto.
4. Renderizar mais superfícies, tendo em conta os vários raios armazenados na tabela periódica interna da aplicação.

## 4.6 Notas finais

Antes de terminar a reflexão crítica, é pertinente levar em conta os problemas pessoais e de saúde enfrentados por cada um dos membros do grupo que acabou por levar a um significativo atraso da conclusão do projeto face à data prevista.

O grupo reconhece, portanto, este mesmo atraso, o qual culminou no não cumprimento total dos objetivos inicialmente propostos para este projeto.

Não obstante, o facto de esta aplicação ter por base op cálculo e a renderização dinâmica de todos os seus elementos, ao invés de uma abordagem *hard-coded*, levantou novos desafios até então não encontrados durante as aulas práticas da UC de CG.

Consideramos, portanto, que, apesar do lamentável atraso, o resultado final obtido é satisfatório.

## 4.7 Conclusões

Esta fase de reflexão permitiu analisar o trabalho enveredado ao longo das semanas de planeamento, execução e teste. Com esta análise, a equipa pôde tirar conclusões não só sobre o seu desempenho, mas também acerca das tecnologias utilizadas, as quais serão expostas no Capítulo seguinte.

# Capítulo 5

# Conclusões e Trabalho Futuro

## 5.1 Conclusões Principais

Após conclusão do projeto "Bohr: *Very Small PDB Molecular Visualizer*", o grupo alcançou como produto final uma aplicação minimalista de visualização molecular a partir de ficheiros PDB para moléculas mais simples.

Apesar de não se ter obtido o cálculo da superfície implícita *pi*, foi alcançado com sucesso a renderização da superfície de *van der Walls*, sem dúvida uma das mais comuns e familiares na área da química.

Tendo sido o primeiro projeto completo na área da Computação Gráfica, concluímos que o projeto teve um resultado satisfatório, apesar de não ideal.

## 5.2 Trabalho Futuro

A implementação do cálculo da superfície implícita *pi* é, sem dúvida, o principal objetivo de trabalho futuro.

Dois objetivo secundário serão a melhoria do aspeto gráfico da aplicação, assim como a melhor adaptação a diferentes sistemas operativos.

Por fim, a adição de suporte a proteínas poderá revelar interesses práticos na área de estudos químicos e bioquímicos *in silico*.

# Bibliografia


[1] A. N. Raposo and A. J. P. Gomes, "Pi-surfaces: products of implicit surfaces towards constructive composition of 3d objects," *CoRR*, vol. abs/1906.06751, 2019. [Online]. Available: http://arxiv.org/abs/1906.06751

[2] T. K. G. Inc, "OpenGL," 2020, [Online] https://www.opengl.org/. Último acesso a 21 de Setembro de 2020.

[3] G. Project, "An OpenGL library | GLFW," 2020, [Online] https://www.glfw.org/. Último acesso a 21 de Setembro de 2020.

[4] T. K. G. Inc, "GLM SDK contribution," 2020, [Online] https://www.opengl.org/sdk/libs/GLM/. Último acesso a 21 de Setembro de 2020.

[5] D. Herberth, "Dav1dde/glad: Multi-Language Vulkan/GL/GLES/EGL/-GLX/WGL Loader-Generator based on the official specs," 2020, [Online] https://github.com/Dav1dde/glad. Último acesso a 24 de Novembro de 2020.

[6] ——, "Glad," 2020, [Online] https://glad.dav1d.de/. Último acesso a 24 de Novembro de 2020.

[7] W. Lemberg, "The FreeType Project," 2020, [Online] https://www.freetype.org/. Último acesso a 3 de Janeiro de 2020.

[8] "daw42/glslcookbook: Example code for the OpenGL Shading Language Cookbook - 2nd Edition," 2020, [Online] https://github.com/daw42/glslcookbook. Último acesso a 19 de Dezembro de 2020.

[9] "glslcookbook/sphere.cpp at master ů daw42/glslcookbook," 2020, [Online] https://github.com/daw42/glslcookbook/blob/master/ingredients/sphere.cpp. Último acesso a 25 de Dezembro de 2020.

[10] A. Belt, "AndrewBelt/osdialog: A cross platform wrapper for OS dialogs like file save, open, message boxes, inputs, color picking, etc." 2020, [Online] https://github.com/AndrewBelt/osdialog. Último acesso a 19 de Dezembro de 2020.

[11] G. Project, "The GNU General Public License v3.0 - GNU Project - Free Software Foundation," 2021, [Online] https://www.gnu.org/licenses/gpl-3.0.en.html. Último acesso a 8 de Janeiro de 2021.

[12] J. de Vries, "LearnOpenGL - Render text," 2021, [Online] https://learnopengl.com/In-Practice/2D-Game/Render-text. Último acesso a 12 de Janeiro de 2021.

[13] "Atomic radii of the elements (data page) - Wikipedia," 2021, [Online] https://en.wikipedia.org/wiki/Atomic_radii_of_the_elements_(data_page). Último acesso a 27 de Dezembro de 2020.

[14] "CPK Coloring - Wikipedia," 2021, [Online] https://en.wikipedia.org/wiki/CPK_coloring. Último acesso a 27 de Dezembro de 2020.

[15] Evan, "etcwilde/MarchingTriangles: Implementing the Marching Triangles Algorithm," 2021, [Online] https://github.com/etcwilde/MarchingTriangles. Último acesso a 13 de Janeiro de 2021.