Universidade da Beira Interior
Departamento de Informática

Nº 2 - 2022: *Isosurfacing Scalar Fields Throught Compute Shaders*

Elaborado por:

(E11005) - Nuno Miguel Freire Monteiro

(M12285) - Diogo Castanheira Simões

(M12688) - Manuel Salvador de Sousa Santos

Orientador:

Professor/a Doutor/a Abel J.P. Gomes

29 de janeiro de 2023

Conteúdo

Lista de Figuras

Acrónimos

CIV Computação Interativa e Visualização

CPU Central Processing Unit

GPU Graphics processing unit

GUI Graphical User Interface

UBI Universidade da Beira Interior

RGB Red, Green and Blue

GLAD Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator

GLM OpenGL Mathematics

GLSL OpenGL Shader Language

GLFW Graphics Library Framework

Capítulo

1

Introdução

1.1 Enquadramento

Este relatório foi feito no contexto da Unidade Curricular de Mestrado Computação Interativa e Visualização (CIV) da Universidade da Beira Interior (UBI).

1.2 Objetivos

O objetivo principal deste projeto é extrair triangulações das superfícies implícitas. O processo de triangular uma superfície implícita é realizado através do *Marching Cubes Algorithm*. Efetivamente, neste projeto usaremos funções implícitamentes definidas para gerar superfícies implícitas. Para além disso, vamos assumir que as superfícies não tenham singularidades como cúspides e vincos.

1.3 Organização do Documento

De modo a refletir o trabalho que foi feito, este documento encontra-se estruturado da seguinte forma:

1. O primeiro capítulo – **Introdução** – apresenta o projeto, a motivação para a sua escolha, o enquadramento para o mesmo, os seus objetivos e a respetiva organização do documento;
2. O segundo capítulo – **Estado de Arte** – descreve todo o conhecimento adquirido sobre o tema do projeto;

3. O segundo capítulo – **Tecnologias Utilizadas** – descreve todas as tecnologias utilizadas durante o desenvolvimento deste projeto;
4. O quarto capítulo – **Implementação** – descreve o protocolo prático realizado relacionado ao tema;
5. O quinto capítulo – **Conclusões e Trabalho Futuro** – as conclusóes que tiramos ao longo deste projeto e o trabalho futuro com as bases deste projeto estão escritas nesta capítulo.

Capítulo

2

Estado da Arte

2.1 Introdução

Para o planeamento de um trabalho, primeiro é necessário adquirir as bases teóricas necessários para ultrapassar os nossos desafios passo a passo, e esses conhecimentos teóricos são então explicados no Estado da Arte.

2.2 Funções Implícitas

Às funções definidas em função de uma variável dá-se o nome de funções explícitas. Exemplos clássicos em $\mathbb{R}^2$ incluem equações de retas ($y = mx + b$) e parábolas ($y = ax^2 + bx + c$). No entanto, nem todos os subconjuntos de pontos no espaço cartesiano podem ser definidos por funções explícitas. Um exemplo comum em $\mathbb{R}^2$ é a circunferência:

$$(x - x_0)^2 + (y - y_0)^2 = r^2 \tag{2.1}$$

onde (x_0, y_0) é o centro e r o raio. Exemplifica-se na Figura 2.1 a circunferência definida por $(x-1)^2 + (y-2)^2 = 4$.

Sendo uma equação de segundo grau, é possível representá-la através de duas funções explícitas:

$$y = y_0 + \sqrt{r^2 - (x - x_0)^2} \tag{2.2}$$

$$y = y_0 - \sqrt{r^2 - (x - x_0)^2} \tag{2.3}$$

Esta transformação não é possível para todos os polinómios, em particular para graus superiores a 4. Contudo, estas expressões polinomiais continuam a

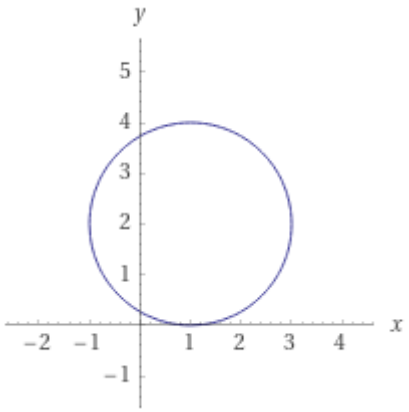

Figura 2.1: Exemplo de uma circunferência de centro (1,2) e raio 2.

ser subconjuntos válidos de $\mathbb{R}^n$, necessitando então de formas alternativas de representação. Dois métodos e respetivas representações da circunferência são:

1. **Funções paramétricas**: cada eixo é definido em ordem a uma variável adicional t:

$$\begin{cases} x = r\cos(t) \\ y = r\sin(t) \end{cases} \tag{2.4}$$

2. **Funções implícitas**: a equação não é definida a ordem a uma variável em particular (equação (2.1)).

Uma **função implícita** é então definida por $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, ou seja, para qualquer ponto em $\mathbb{R}^n$ é determinado um resultado em $\mathbb{R}$. Dependendo da função, o valor obtido pode ter significado, tal como uma grandeza física (*e.g.* densidade de um líquido ou sua temperatura a cada ponto do espaço). Esta função diz-se **algébrica** caso seja polinomial em cada variável.

Por seu turno, em $\mathbb{R}^3$, a **iso-superfície** de uma função implícita é a superfície que satisfaz a condição $f(\mathbf{x}) = 0$ (onde, doravante, $\mathbf{x} \equiv (x, y, z)$). Esta pode ser suavizada através de um parâmetro $s \in \mathbb{R}$ tal que:

$$f(\mathbf{x}) - s = 0 \tag{2.5}$$

2.3 *Marching Cubes*

2.3.1 Introdução

Marching Cubes (Cubos Marchantes), é um algoritmo de computação gráfica, publicado em 1987 nos artigos do **SIGGRAPH** por Lorensen e Cline, para a extração de malhas geométricas de isosuperfícies a partir de um campo escalar tridimensional(por vezes nomeados como *voxels*) [1].

2.3.2 Origem do *Marching Cubes*

O algoritmo foi criado para resolver um problema específico da computação gráfica que é: *Definir uma superfície fechado ao redor de pontos num espaço 3D*. Efetivamente, supondo que temos um *dataset* de pontos distribuídos num espaço 3D e cada um deles tem um valor intrínseco, que é denominado como ***Isovalue***, que classificará cada ponto como estando "dentro"ou "fora"de uma superfície fechada. Por exemplo, valores acima do *Isovalue* são considerados como pontos "fora"e valores abaixo dele são considerados como pontos "dentro".

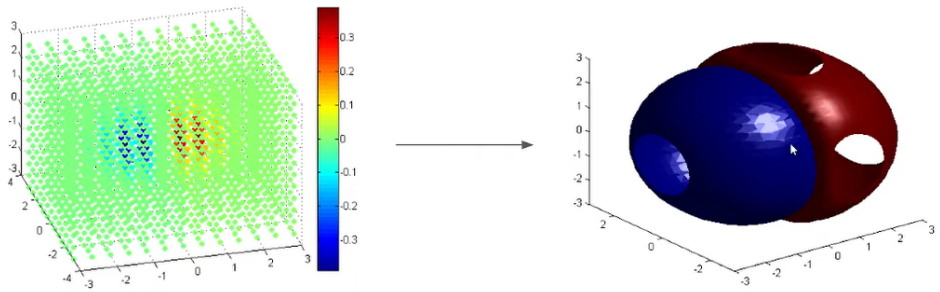

Figura 2.2: Problema que originou o *Cube Marching*

Como resposta a esse problema, o *Marching Cubes* baseia-se em:

- Subdividir o *dataset* do espaço 3D em cubos pequeno;
- Analisar cubo a cubo, conferindo os vértices do cubo que estão "fora"ou "dentro"da superfície definida pelo *isovalue*;
- Para cada cubo, é possível atribuir, independentemente do outros cubos, uma pequena parcela da superfície total, utilizando triângulos;
- Após ter passado por todos os cubos do, obtém-se a superfície que engloba todos os pontos de "dentro"do *dataset*.

2.3.3 Algoritmo de *Marching Cubes*

Sendo aquela solução proposta pelo *Marching Cubes*, o algoritmo poderia ser descrito da seguinte maneira:

1. Construir um cubo a partir de 8 pontos do *dataset*;
2. Classificar os vértices do cubo a partir do *Isovalue* ("dentro"ou "fora");
3. Construir um index de 8 *bits* dos vértices;
4. Calcular as densidades;
5. Consultar a tabela de triangulação usando o index;
6. Interpolar a borda da superfícies;
7. Resolver casos ambíguos;
8. Ir para os próximos 8 pontos (cubo) do *dataset*.

No 1º passo, vai acontecer uma coleta dos oito pontos do *dataset* que nós iremos usar. Outra coisa importante a ser mencionada é o tamanho do cubo vai impactar na qualidade da superfície gerada.

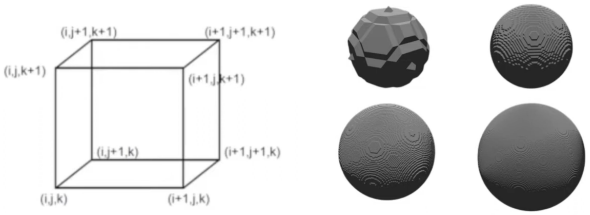

Figura 2.3: Exemplo de um cubo e a diferença que o tamanho faz

Após isso, no 2º passo, vai ocorrer a classificação dos vértices a partir do *isovalue*, ou seja, vamos ditar quais pontos são de "dentro"da superfície e "fora"dela. Quando um valor é maior que o *isovalue*, ele será classificado como um ponto de "fora"e, caso contrário, se o valor foi menor que o *isovalue*, será considerado um ponto de "dentro"da superfície.

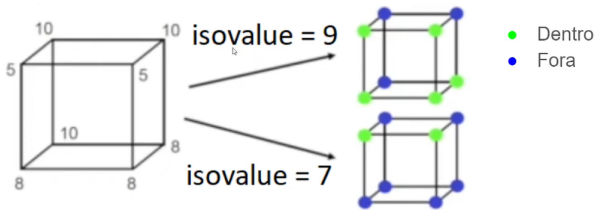

Figura 2.4: Exemplo de aplicação da classificação de *Isovalues*

Tendo em conta as classificados adquiridas no 2º passo, no 3º passo, para criar um vetor onde os vértices serão organizados e ser-lhes-á atribuir o valor de "0"para os vértices de "fora"e "1"para os vértices de dentro.

Como tal, no 4º passo, vamos qualquer os vetores normais para cada vértice, a partir dos componentes do vetor que subtrai o valor dos vértices adjacentes àquele em que o valor está a ser calculado.

$$G_x = V_{x+1,\,y,\,z} - V_{x-1,\,y,\,z}$$
$$G_y = V_{x,\,y+1,\,z} - V_{x,\,y-1,\,z}$$
$$G_z = V_{x,\,y,\,z+1} - V_{x,\,y,\,z-1}$$

Figura 2.5: Formulas das componentes do vetor

A partir do vetor, no 5º passo, vamos obter a definição da superfície que cruza o cubo que está a ser examinado. Aprofundando um pouco mais este passo, a partir dos oito vértices que podem estar "dentro"ou "fora", existem 256 possibilidades de formas de superfície que atravessam o cubo descritas na tabela que criamos. Porém só 15 definições de superfície são necessárias, como podemos ver na figura 4.4 , para atender a todas as 256 combinações diferentes devido à simetria.

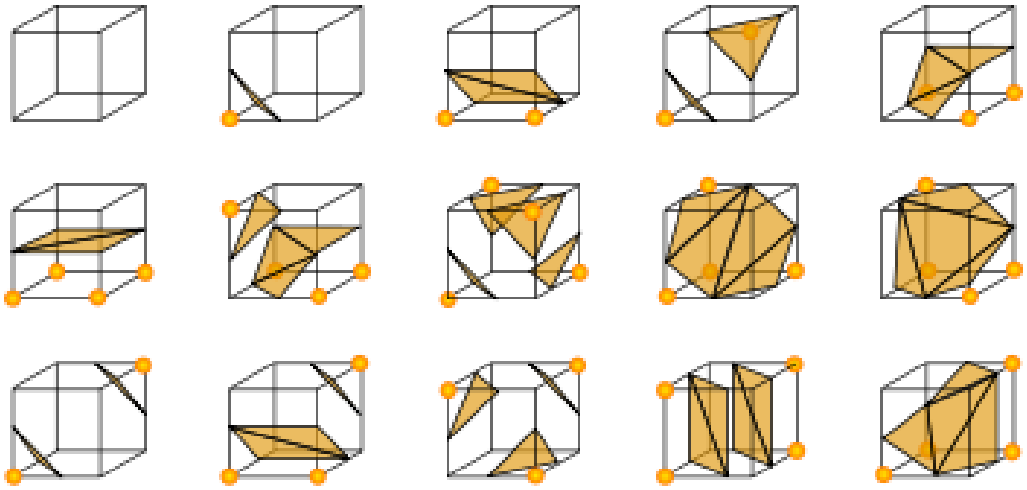

Figura 2.6: As 15 definições de superfície necessárias

No 6º passo, iremos definir o tamanho dos triângulos a partir da coordenada exata das arestas do cubo que o triângulo vai cruzar. Este processo é efeito através da interpolação do valor dos vértices com o *isovalue* para encontrar a coordenada desejado do triângulo.

Por último, no 7º passo, podem existir casos de ambiguidade no algoritmo e uma combinação de vértices podem gerar duas superfícies que dividem os vértices de formas diferentes. Para resolver tal problema, podemos fazer o seguinte:

1. Subdividir o cubo em 8 cubos menores;
2. Obter vetores normais para os vértices internos;
3. Comprar sinais dos vetores normais;
4. Os sinais dos vetores do cubo original devem alinhar-se com os sinais dos vetores dos cubos menores.

Na seguinte figura podemos ver um exemplo de um erro e a sua respetiva correção.

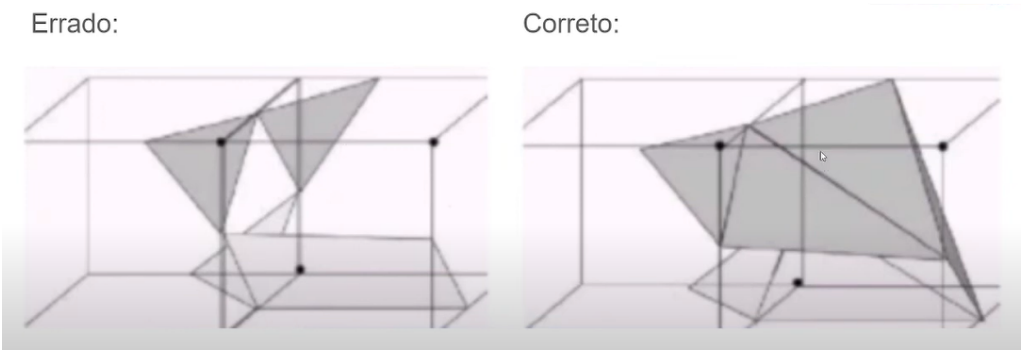

Figura 2.7: Erro de ambiguidade

Para facilitar a compreensão do algoritmo, recomendamos a assistir a esta simulação [2].

2.4 *Compute Shaders*

2.4.1 Para que são usados?

Os *Compute Shaders* não possuem um propósito especifico, sendo assim, eles são independentes da *Graphics Pipeline*. Como eles são independentes e não fazem parte da *Pipeline*, também não existiram nenhuns *users inputs* nem *outputs*, ao invés disso eles fazem as mudanças diretamente na Memória do Central Processing Unit (CPU).

2.4.2 Como eles funcionam?

Uma pequena analogia que podemos fazer relativo ao *Compute Shaders* é que eles são um conjunto de pequenos computadores chamados *"Work Groups"*. É bom manter em mente que estes *"Work Groups"* são independentes uns dos outros, e como tal, não devem de depender uns dos outros. Dentro destes *"Work Groups"* temos diversos caixotes chamados de *"Invocations"* que conseguem comunicar umas com as outras.

2.4.3 Warps/Wavefronts

Aprofundando um pouco mais sobre o tamanho dos *Work Groups*, cada fabricante dos CPUs tem uma otimização feita para um certo tamanho para *Work*

Groups, chamado um *Warp* ou *Wavefront*. Por exemplo, quem tem um CPU da intel, o seu *Wrap* será 32, enquanto para a AMD será 64. Efetivamente, possuir vários destes *Warps* melhoram a performance.

2.4.4 Inputs

Como não temos os *inputs* necessário não conseguimos descobrir para qual *pixel* cada *invocation* corresponde. Para começar, podemos buscar o ID da nossa atual *invocation*.Como dito anteriores, os *Compute Shaders* não tem *inputs* definidos pelos utilizadores, mas eles têm *inputs*, mais precisamente os 5 seguintes:

- `gl_NumWorkGroups` – que contém os tamanhos dos *"Work Groups"*;
- `gl_WorkGroupID` – contém o ID do *"Work Group"* que estamos atualmente;
- `gl_LocalInvocationID` – contém o ID da *"Invocation"* que estamos atualmente em respeito ao seu *"Work Group"*;
- `gl_GlobalInvocationID` – contém o ID da *"Invocation"* que estamos atualmente em respeito ao seu *"Chunk"*;
- `gl_LocalInvocationIndex` – contém o index da atual *"Invocation"* que estamos atualmente em respeito ao seu *"Work Group"*.

Nota: A diferença entre o ID e o Index é que o ID contém a localizações x e y, enquanto o Index só tem um inteiro, como se toda a estrutura 3D fosse *unraveled* como uma `String` num vetor de uma só dimensão.

2.4.5 Sincronização de *Invocations*

Após a variavél ter sido inicializada, para trabalhar com ela seguramente, em outras *invocations* então temos que garantir que elas estão todas sincronizadas, para isso podemos usar:

- `memoryBarrier();`
- `memoryBarrierAtomicCounter;`
- `memoryBarrierImage;`
- `memoryBarrierBuffer;`
- `memoryBarrierShared;`
- `groupMemorybarrier();`
- `barrier()`

2.4.6 *Atomic Operations*

Porém. se ao invés de um vetor compartilhado, nós temos um inteiro compartilhado. Devido às *invocations* serem paralelas, poderião ocorrer diversas leituras erradas de certos valores compartilhados que estariamos a operar. É nesta situação em que as *Atomic Operations* são necessárias, elas são as seguintes:

- `atomicAdd(mem, data)`
- `atomicMin(mem, data)`
- `atomicMax(mem, data)`
- `atomicAnd(mem, data)`
- `atomicOr(mem, data)`
- `atomicXor(mem, data)`
- `atomicExchange(mem, data)`
- `atomicCompSwap(mem, compare, data)`

2.4.7 *Group Voting*

Supondo um exemplo, existe uma divergência de caminhos no código onde uma permite fazer x ação, enquanto outra permite fazer y ação podemos usar:

- `anyInvocationARB(condition);`
- `allInvocationsARB(condition);`
- `allInvocationsEqualARB(condition).`

2.5 Conclusões

Com o todo o conjunto de conhecimento acumulado, conseguimos adquirir uma base teórica sólida para a futura implementação do projeto.

Capítulo

3

Tecnologias e Ferramentas Utilizadas

3.1 Introdução

Neste capítulo serão descritas todas as tecnologias e ferramentas que foram utilizadas ao longo da realização do projeto.

3.2 Tecnologias e Ferramentas Utilizadas

Para a implementação do projeto *MarchGL* com *OpenGL*©foi escolhida a linguagem de programação C++, em particular o *standard* C++20 (indicado ao compilador g++ com a flag `-std=c++20`). A fim de melhorar a experiência de utilização do *OpenGL*©, foram utilizadas as seguintes bibliotecas e *frameworks*:

- **OpenGL Shader Language (GLSL)** – é uma linguagem de *shading* de alto nível baseada na linguagem de programação C/C++. Foi criada pela *OpenGL ARB* para dar aos desenvolvedores controle mais direto do pipeline de gráficos sem ter de usar a linguagem de *assembly* ou linguagens específicas de hardware; [3]
- **Graphics Library Framework (GLFW)** – é uma biblioteca multiplataforma de código aberto para desenvolvimento *OpenGL*, *OpenGL ES* e *Vulkan* no desktop. Ela fornece uma API simples para criar janelas, contextos e superfícies, recebendo entradas e eventos; [4]
- **Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator (GLAD)** – gerados automático de *loaders* para *OpenGL*©; [5]

- **OpenGL Mathematics (GLM)** – é uma biblioteca matemática C++ somente de cabeçalho para software gráfico baseada nas especificações OpenGL Shading Language (GLSL); [6]

Tabela 3.1: Ferramentas e tecnologias utilizadas, organizadas por categoria.

***Software* / Tecnologia**	**Versão**
Aplicação OpenGL	
OpenGL	4.6
GLFW	3.3.5
GLAD	0.1.34
GLM	0.9.9.8
FreeType	2.10.4
Controlo de versões	
git	2.36.1
GitKraken	8.6.1

3.3 Código *Open Source*

Para além das tecnologias usadas referidas na Secção 3.2, o Código *Open Source* adicional foi utilizado para facilitar a implementação de componentes que não fazem parte do objetivo do projeto. Essas são:

- ***CParse*** – biblioteca para *parsing* de uma sequência de caracteres como uma expressão usando o algoritmo *Shunting-yard* de Dijkstra;
- ***Dear ImGui*** – uma biblioteca *bloat-free* para criação de Graphical User Interface (GUI) em C++;
- ***Learn OpenGL*** – coleção de códigos de exemplo para o propósito de ensino de *OpenGL* ©

3.4 Distribuição do Trabalho

Todos os constituintes do grupo empenharam-se para realizar o trabalho. Efetivamente, todos os membros contribuíram o máximo que puderam para cada parte do trabalho, porém houveram partes em que cada membro participou mais ativamente e eficientemente, mais precisamente:

- **Código-Fonte:** Nuno Monteiro e Diogo Simões;

- **Relatório:** Manuel Santos.

3.5 Conclusões

A partir das diversas tecnologias, ferramentas utilizadas e do trabalho dos diversos membros do grupo, conseguimos fazer um plano para a implementação do nosso projeto.

Capítulo

4

Implementação

4.1 Introdução

Após as bases teóricas adquiridas e o planeamento de como realizar o projeto, é necessário colocar tudo isso em prática, demonstrar como foi posto em prático, demonstrar os testes que foram aplicados e os respetivos resultados associados.

4.2 Funcionalidades e Requisitos

As funcionalidades a implementar do projeto *MarchGL* devem ser as seguintes:

- Renderização de funções implícitas com uso do algoritmo *Marching Cubes*;
- Suporte a dois motores de cálculo:
    1. Por *software* (cálculo do algoritmo exclusivo à CPU)
    2. Aceleração por *hardware* (com o uso da Graphics processing unit (GPU) através de *Compute Shaders*);
- Apresentar uma GUI capaz de personalizar e demonstrar os vários aspetos do programa;

4.3 Lógica e Estruturação

A aplicação é construída usando várias classes que trabalham juntas para manter a representação de funções implícitas armazenadas em memória.

O programa é controlado por uma classe principal encarregue do ciclo de renderização e gestão da GUI.

4.3.1 Parâmetros de arranque

O programa disponibiliza um conjunto de argumentos que podem ser passados pela linha de comandos (ou terminal) para alterar o seu comportamento:

- `--width` ou `-W`:

  Modifica a largura da janela de renderização, em pixeis. *Default*: 1920. Exemplo: `-W 1000` inicia o programa com uma largura de 1000 pixeis.

- `--width` ou `-W`:

  Modifica a largura da janela de renderização, em pixeis. *Default*: 1080. Exemplo: `-W 1000` inicia o programa com uma largura de 1000 pixeis.

- `--render` ou `-r`: Define qual o motor de renderização a que o programa recorre. Existem dois modos implementados:

  - `CPU`: motor de renderização por software com uso do algoritmo *marching cubes*;
  - `GPU`: motor de renderização por aceleração de *hardware* com uso do algoritmo *marching cubes* implementado em *compute shaders*;

  Default: `CPU`

- `--threads` ou `-t`: Indica o número de *threads* que o motor de renderização por *software* deve usar. *Default*: metade do número de núcleos lógicos disponibilizados pela CPU.

  Esta opção só é considerada caso o modo de renderização selecionado seja `CPU`. Se o valor especificado for superior ao *default*, o utilizador é alertado para a possibilidade de problemas de *performance* no seu sistema.

  Exemplo: `-t 6` inicia o programa com seis *threads* prontas a serem usadas.

É ainda disponibilizado o comando de ajuda descrito como o argumento `--help` ou `-h`, nesta situação o programa emite a informação descrevendo os argumentos disponíveis e logo em seguida o mesmo sai do programa.

4.3.2 *Shader Manager* e *Compute Shader Manager*

Este passo de arranque do programa só é feito caso a inicialização das bibliotecas/*frameworks* GLFW e GLAD seja corretamente efetuada.

Os *shaders* são carregados pelo *Shader Manager*, este é instanciado num único programa com dois *shaders* a compilar (contendo o respetivo *vertex* e *fragment shaders*). De parte é compilado também o *compute shader* contendo a implementação do algoritmo de *cube marching*.

4.4 Interface Gráfica

A partir da interface gráfica, o utilizador consegue interagir com o programa da maneira que ele desejar e obter os resultados que ele procura. Na figura a seguir está o exemplo da interface criada para o projeto. Aprofundando um

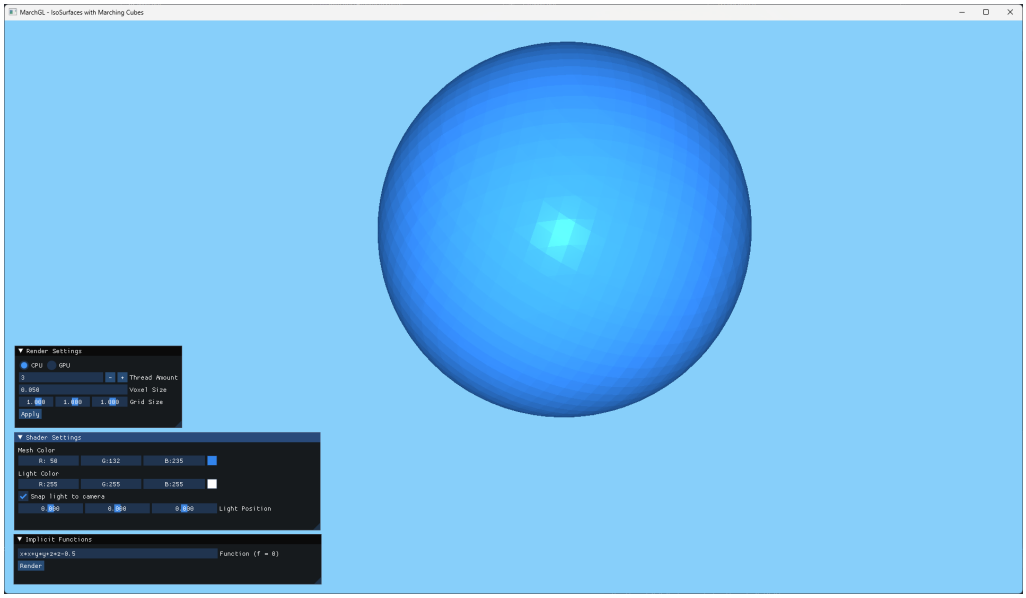

Figura 4.1: Interface Gráfica

pouco mais a interface, nós vamos depararmo-nos com três janelas principais: *Render Settings*, *Shader Settings* e a *Implicit Functions*.

4.4.1 *Render Settings*

Nas *Render Settings*, primeiramente temos uma opção que permite-nos utilizar o método de renderização pela CPU ou pela GPU (*"Compute Shaders"*). Após isso nós podemos aumentar ou diminuir o tamanho das *threads*, o tamanho dos *voxels* e a *grid size*. Para aplicar as mudanças desejadas, é necessário clicar no botão *"Apply"*.

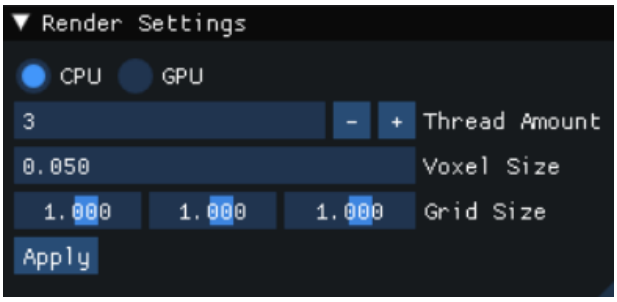

Figura 4.2: *Render Settings*

4.4.2 *Shader Settings*

Para além disso, nas *Shader Settings*, primeiramente nós podemos alterar a *Mesh Color* e depois a *Light Color* através da escala Red, Green and Blue (RGB). Por fim, podemos alterar a posição da luz e é nos permitido selecionar *"Snap Light"* para as coordenadas da fonte de luz e da câmara passarem a ser uma só.

Todas as alterações feitas nas *Shader Settings* são realizadas em tempo real.

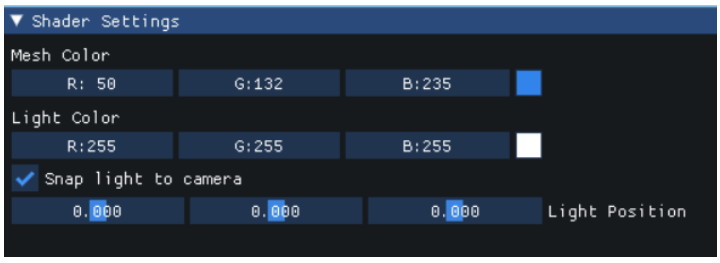

Figura 4.3: *Shader Settings*

4.4.3 *Implicit Functions*

Por último, na janela *Implicit Functions*, nós podemos colocar uma função implícita e o *Cube Marching* construirá a forma associada função. É obrigatório também que as funções estejam escritas segundo a sintaxe da linguagem GLSL.

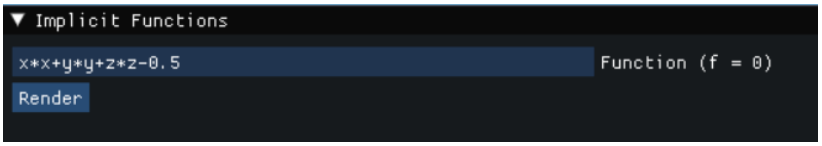

Figura 4.4: *Implicit Functions* com a função implícita de uma esfera

4.5 Motores de renderização

Em ambos os modos de renderização fornecidos, a função implícita para o calculo é obtida pelo campo de texto pela GUI.

4.5.1 Cálculo por *Software*

Este motor de renderização pode executar em várias *threads*, onde cada uma processa um conjunto de linhas contíguas da grade do algoritmo *marching cubes*. As funções são processadas pela biblioteca CParse.

4.5.2 Aceleração por *hardware*

O motor de renderização acelerado por *hardware* faz a **injeção de funções** no *compute shader* utilizado por este modo. Uma vez que o *Shader Manager* pode compilar programas a qualquer momento, as funções lidas são injetadas num local predeterminado do *compute shader* e um novo programa é compilado com este novo *shader*, sendo o anterior descartado.

```
1 float getDensity(vec3 p) {
2         float x, y, z;
3         x = p.x;
4         y = p.y;
5         z = p.z;
6
```

```
7           // <IFunction>
8
9           return 1.f;
10  }
```

A injeção da função lida é feita no local do comentário indicado por `<IFuntion>`. Por exemplo, caso da função `x*x + y*y + z*z - 1.0` o *compute shader* será recompilado com a seguinte linha de código:

```
1       return x*x + y*y + z*z - 1.0;
```

Desta forma é possível o calculo de uma iso-superficie com o recurso a apenas uma função implícita na GPU.

4.6 Conclusões

Em suma, através das funcionalidades e requisitos especificados *a priori*, duma lógica e estruturação minuciosamente planeanada com respectivos parâmetros de arranque, ambos *shaders* e motores de renderização, conseguimos obter o resultado desejado, que fora demonstrado através da nossa interface gráfica compostas pelas diversas *settings* que permite ao utilizador usufruir do nosso trabalho da maneira que desejar.

Capítulo

5

Conclusões e Trabalho Futuro

5.1 Conclusões

Com o desenvolvimento deste projeto foi então possível:

1. Estudar as funções implícitas e o método de cálculo das respetivas iso-superfícies;
2. Estudar o algoritmo da área de **CG!** (**CG!**) *Marching Cubes*;
3. Aprofundar o conhecimento na linguagem GLSL;
4. Estudar o funcionamento dos *Compute Shaders*;

Desta forma foi alcançado o objetivo principal: desenvolver e implementar um visualizador de iso-superfícies com recurso ao algoritmo *Marching Cubes* através de *Compute Shaders*.

O uso do motor de renderização por *software* mostrou ser minimamente adequado para o propósito. No entanto, não podendo ser comparado com a rapidez alcançada no uso de aceleração por *hardware*.

5.2 Trabalho Futuro

Ainda que, no contexto do projeto proposto, tenham sido cumpridos os objetivos definidos, a conclusão deste revelou a existência de oportunidades de expansão do trabalho realizado. Nomeadamente, o uso de ruído para a geração de "mundos"através deste mesmo algoritmo.

Bibliografia

[1] Wikipedia. Cubos Marchantes, 2018. [Online] https://pt.wikipedia.org/wiki/Marching_cubes. Último acesso a 26 de janeiro de 2023.

[2] Algorithms Visualized. Marching Cubes Animation | Algorithms Visualized, 2018. [Online] https://www.youtube.com/watch?v=B_xk71YopsA. Último acesso a 26 de janeiro de 2023.

[3] John M. Kessenich; Jacobo Rodriguez; Randi J. Rost; David Wolff. GLSL, 2023. [Online] https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language. Último acesso a 20 de janeiro de 2023.

[4] Marcus Geelnard. GLFW, 2023. [Online] https://code.visualstudio.com/. Último acesso a 20 de janeiro de 2023.

[5] GLAD. GLAD, 2023. [Online] https://glad.dav1d.de/. Último acesso a 20 de janeiro de 2023.

[6] Christophe Riccio. OpenGL Mathematics, 2023. [Online] https://glm.g-truc.net/0.9.9/. Último acesso a 20 de janeiro de 2023.