Sumário

Projeto Escultor 3D

Agostinho Brito

> *Leia com atenção*
> As descrições dos projetos devem ser lidas com atenção. Se não estiver claro, procure o professor para que o texto seja atualizado.

1. Apresentação

O objetivo desse projeto é construir uma ferramenta em C++ para realizar esculturas em blocos representados por matrizes digitais, algo como a ideia usada pelo jogo minecraft.

A ideia é permitir que o utilizador da ferramenta seja capaz de criar um arquivo em texto simples que contém uma sequência de passos a serem seguidos para definir propriedades de uma matriz tridimensional. As propriedades contêm especificações de cor para os elementos, de sorte que com o auxílio de softwares de visualização o usuário possa apreciar a escultura criada.

O projeto consiste em dois módulos: uma classe em C++ destinada à interpretação de um arquivo em formato de texto simples e geração das matrizes tridimensionais; e uma ferramenta visual para desenho, algo equivalente a um "paint" 3D.

Os sistemas de escultura tridimensional prevêem alguns tipos de formas geométricas simples que poderão ser usadas pelo usuário para pintar e deverão ser implementadas no projeto.

1.1. Parte 1 - Biblioteca de leitura e traçado de esculturas

A primeira etapa do projeto consiste em conceber uma classe em C++ que permita realizar operações em uma matriz tridimensional alocada dinamicamente. Os elementos dessa matriz guardam propriedades da escultura e são denominados `Voxels` (*volume elements*), algo equivalente aos `Pixels` que comumente são usados em imagens digitais. Nos Voxels seria possível armazenar informações como cor e transparência, necessárias para idealizar os elementos de uma escultura.

As matrizes tridimensionais possuem dimensões de largura, altura e largura especificadas pelo usuário, onde os índices dos elementos nessa matriz coresponderiam às posições dos voxels no espaço discreto.

Por exemplo, para uma matriz de dimensões $2 \times 3 \times 4$ elementos, sua representação em linguagem C++ poderia ser da forma `Voxel m[2][3][4];`. Essa matriz teria, portanto, um total de 24 voxels. Entretanto, é importante ressaltar que se trata apenas de um exemplo e que na implementação essa matriz seria alocada de forma dinâmica, com a quantidade de elementos em cada dimensão sendo fornecida pelo usuário.

O tipo de dado `Voxel` é definido como um `struct` e comporta as propriedades necessárias para permitir armazenar três tipos de informações: a cor do voxel, sua transparência e se ele deverá ser incluído ou não no modelo digital que representa a escultura. A estrutura `Voxel` é definida na listagem Código fonte da estrutura `Voxel`.

Listagem 1. Código fonte da estrutura `Voxel`

```cpp
struct Voxel {
  float r,g,b; // Colors
  float a; // Transparency
  bool isOn; // Included or not
};
```

A cor do voxel é armazenada nas propriedades `r`, `g` e `b` da estrutura. Cada uma das variáveis representam as dosagens de vermelho (red), verde (green) e azul (blue) usadas para compor uma cor específica. Essas dosagens necessariamente devem assumir valores na faixa $[0, 1]$, onde `0` denota ausência total da componente e `1` denota a presença total da componente de cor.

A propriedade de transparência, por sua vez, é definida pela variável `a`. Em computação gráfica, essa propriedade é normalmente denominada de canal alfa e deve assumir valores na faixa $[0, 1]$, onde `0` denota transparência total e `1` denota total opacidade.

Finalmente, a propriedade `isOn` define se o voxel correspondente deve ser incluído no arquivo digital que irá representar a escultura. Para ilustrar a ideia, considere o exemplo de uma matriz 3D de dimensões $[2 \times 2 \times 2]$ voxels com as seguintes propriedades:

coordenadas			**propriedades**				
x	y	z	r	g	b	alpha	isOn
0	0	0	0	1	0	1.0	true
0	0	1	1	0	0	0.8	true
0	1	0	0	0	1	1.0	true
1	0	0	1	1	0	0.5	true
1	0	0	*	*	*	*	false
1	0	1	*	*	*	*	false
1	1	0	*	*	*	*	false
1	1	1	*	*	*	*	false

A escultura tridimensional correspondente ao conjunto de voxels com tais propriedades é mostrado na Figura Exemplo de Escultura.

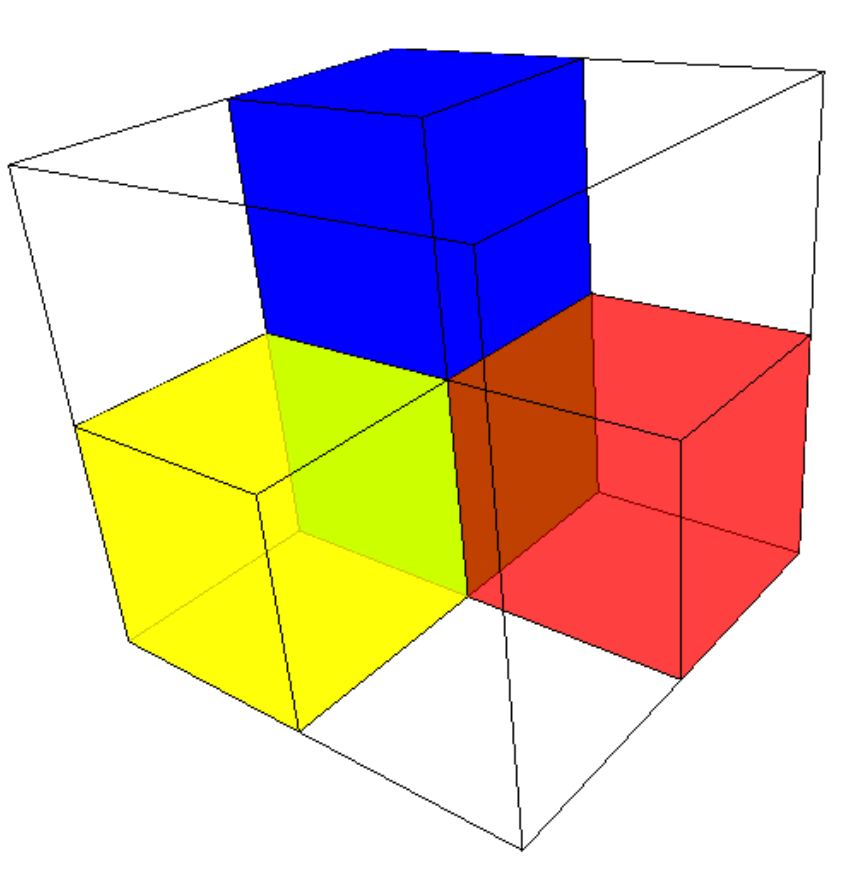

Figura 1. Exemplo de escultura 3d para criada com uma matrix de $2 \times 2 \times 2$ voxels

2. Objetivos

O que o desenvolvedor deverá implementar nesse projeto? A proposta é que a classe disponibilize uma série de métodos apropriados para realizar a escultura no bloco digital. Espera-se que um utilizador da classe seja capaz de desenhar ou apagar pelo menos três tipos de estruturas em uma escultura: blocos, voxels individuais e elipsóides, cujos métodos serão detalhados adiante.

Em se tratando de um elemento abstrato, precisa-se de uma ferramenta de visualização para ser exibir a escultura no computador. Entre os programas que podem ser usados para visualização, destaca-se o geomview (www.geomview.org), livremente disponível em sistemas Unix. No Ubuntu, por exemplo, a instalação é quase trivial, pois ele se encontra presente na loja virtual de aplicativos. Para usuários de outros sistemas operacionais existe o meshlab (meshlab.sourceforge.net), que também é livre e está disponível para várias plataformas (inclusive android e ios).

3. Formatos de armazenamento de modelos digitais

Para ser visualizada, a representação da escultura digital deve ser armazenada um arquivo utilizando um formato suportado pela ferramenta de visualização. Diversos formatos de armazenamento de modelos tridimensionais podem ser usados, mas optamos pelo de um formato bastante simples, de fácil criação e entendimento, que é o formato OFF. Ambos são suportados pelo geomview ou meshlab, e pode ser usado para representar malhas tridimensionais na forma de polígonos planares. A Exemplo de Escultura, por exemplo, foi obtida a partir da representação dos voxels na forma de cubos coloridos armazenados no formato OFF.

3.1. O formato de armazenamento OFF

O formato de armazenamento OFF é geralmente associados a arquivos com a extensão *.off*. Ele prevê a representação de superfícies usando conjuntos de vértices e polígonos que os interconectam. Para fins do projeto proposto, um arquivo OFF pode ser descrito da seguinte maneira:

```
OFF
NVertices  NFaces  NArestas
x[0]  y[0]  z[0]
...
x[NVertices-1]  y[NVertices-1]  z[NVertices-1]
Nv v[0] v[1] ... v[Nv-1]  r g b a
Nv v[0] v[1] ... v[Nv-1]  r g b a
...
Nv v[0] v[1] ... v[Nv-1]  r g b a
```

A primeira linha contém apenas a palavra `OFF`, servindo para identificar o tipo do arquivo.

A segunda linha especifica em números inteiros a quantidade de vértices (`NVertices`), faces (`NFaces`) e arestas (`NArestas`) que são representados ao todo na figura geométrica. O número de arestas normalmente não é utilizado pelas ferramentas de visualização podendo sempre assumir valor `0`.

A partir da terceira linha são apresentadas as coordenadas espaciais tridimensionais dos `NVertices` presentes na figura.

Seguindo as coordenadas vêm as especificações das faces. Cada face é definida por uma sequência de índices em uma linha do arquivo. O primeiro elemento da linha define o número de vértices na referida face. Em seguida, são apresentadas os índices dos `NVertices` vértices da face, bem como a cor da face no formato RGBA (Red, Green, Blue e Alpha). Cada cor é composta por um conjunto de quatro números em ponto flutuante na faixa $[0, 1]$, onde os três primeiros definem a cor da face (Red, Green, Blue) e o último define a sua transparência (0 para uma face totalmente transparente e 1 para uma face totalmente opaca).

Um fato importante sobre a especificação da face diz respeito à sequência em que os vértices são apresentados. O sentido do vetor normal a esta superfície é indicado pela regra da mão direita, onde os dedos acompanham a sequência fornecida (ver Figura 2)). Cada sequência de pontos produz um polígono planar com duas faces opostas e, nos programas de visualização, a direção e sentido do vetor normal são usados para escolher e iluminar corretamente a face que se deseja exibir.

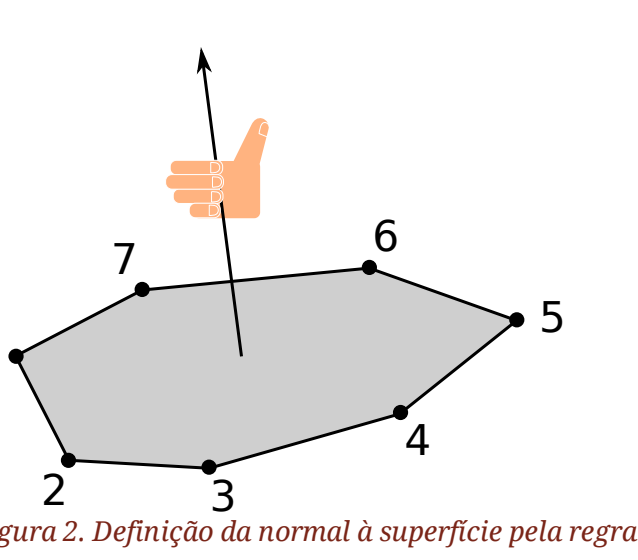

Figura 2. Definição da normal à superfície pela regra da mão direita

Para exemplificar a criação de um arquivo OFF, considere o cubo apresentado na Figura 3.

Sua representação deverá ser dada pelas posições dos seus oito vértices $(P_0 ... P_7)$ e pelas sequências de vértices que definem suas faces, que deverão ser organizadas de modo às suas normais apontarem para fora do cubo. Só assim as superfícies das faces são corretamente iluminadas, proporcionando o efeito visual desejado na ferramenta de exibição.

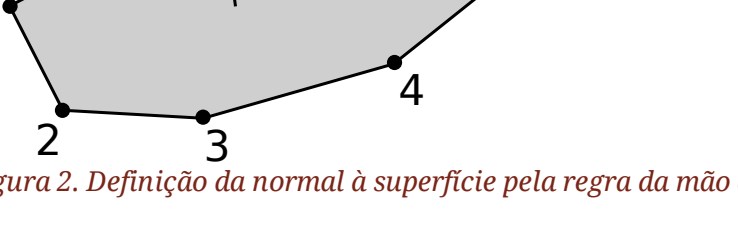

Figura 3. Representação de um cubo com centro no ponto .

As faces deste cubo seriam definidas então pelas seguintes sequências de vértices:

```
Face 1: P0 P3 P2 P1
Face 2: P4 P5 P6 P7
Face 3: P0 P1 P5 P4
Face 4: P0 P4 P7 P3
Face 5: P7 P6 P2 P3
Face 6: P1 P2 P6 P5
```

Para um cubo de lado de tamanho igual a 1, com centro no ponto e faces opacas desenhadas na cor branca, sua representação no formato OFF seria dada pelo arquivo com o seguinte conteúdo:

```
OFF
8 6 0
-0.5 0.5 -0.5
-0.5 -0.5 -0.5
0.5 -0.5 -0.5
0.5 0.5 -0.5
-0.5 0.5 0.5
-0.5 -0.5 0.5
0.5 -0.5 0.5
0.5 0.5 0.5
4 0 3 2 1 1 1 1 1
4 4 5 6 7 1 1 1 1
4 0 1 5 4 1 1 1 1
4 0 4 7 3 1 1 1 1
4 3 7 6 2 1 1 1 1
4 1 2 6 5 1 1 1 1
```

Neste arquivo, o modelo 3d possui 8 vértices e 6 faces declarados. O número de arestas igual a zero não é utilizado.

Perceba que, para construir uma estrutura com outros cubos como esse, o programador deverá observar que a sequência dos índices dos próximos cubos será semelhante ao do primeiro, acrescentando-se, porém, 8 unidades ao cubo anterior, oriundas dos novos grupos de vértices para os cubos que serão representados.

Ainda, as quantidades de vértices e faces previstas no cabeçalho do arquivo serão múltiplos de 8 e 6, respectivamente, conforme a quantidade de cubos existentes.

4. A classe Escultor - Parte 1 - Estruturas básicas

Para manipular os pixels da matriz tridimensional, o desenvolvedor deverá usar um conjunto de funcionalidades previstas para uma classe denominada `Sculptor` que deverá possuir a seguinte declaração:

```cpp
class Sculptor {
private:
  Voxel ***v; // 3D matrix
  int nx,ny,nz; // Dimensions
  float r,g,b,a; // Current drawing color
public:
  Sculptor(int _nx, int _ny, int _nz);
  ~Sculptor();
  void setColor(float r, float g, float b, float a);
  void putVoxel(int x, int y, int z);
  void cutVoxel(int x, int y, int z);
  void putBox(int x0, int x1, int y0, int y1, int z0, int z1);
  void cutBox(int x0, int x1, int y0, int y1, int z0, int z1);
  void putSphere(int xcenter, int ycenter, int zcenter, int radius);
  void cutSphere(int xcenter, int ycenter, int zcenter, int radius);
  void putEllipsoid(int xcenter, int ycenter, int zcenter, int rx, int ry, int rz);
  void cutEllipsoid(int xcenter, int ycenter, int zcenter, int rx, int ry, int rz);
  void writeOFF(const char* filename);
};
```

Cada um dos métodos da classe `Sculptor` realiza uma modificação específica na matriz de voxels definida pela variável `v`. A descrição de cada um desses métodos é apresentada na Tabela 1

Tabela 1. Métodos da classe `Sculptor`

Método	Descrição
`Sculptor(int _nx, int _ny, int _nz)`	Construtor da classe
`~Sculptor(int _nx, int _ny, int _nz)`	Destrutor da classe
`void setColor(float r, float g, float b, float a)`	Define a cor atual de desenho.
`void putVoxel(int x, int y, int z)`	Ativa o voxel na posição (x,y,z) (fazendo `isOn = true`) e atribui ao mesmo a cor atual de desenho
`void cutVoxel(int x, int y, int z)`	Desativa o voxel na posição (x,y,z) (fazendo `isOn = false`)
`void putBox(int x0, int x1, int y0, int y1, int z0, int z1)`	Ativa todos os voxels no intervalo $x \in [x0, x1]$, $y \in [y0, y1]$, $z \in [z0, z1]$ e atribui aos mesmos a cor atual de desenho
`void cutBox(int x0, int x1, int y0, int y1, int z0, int z1)`	Desativa todos os voxels no intervalo $x \in [x0, x1]$, $y \in [y0, y1]$, $z \in [z0, z1]$ e atribui aos mesmos a cor atual de desenho
`void putSphere(int xcenter, int ycenter, int zcenter, int radius)`	Ativa todos os voxels que satisfazem à equação da esfera e atribui aos mesmos a cor atual de desenho (r, g, b, a)
`void cutSphere(int xcenter, int ycenter, int zcenter, int radius)`	Desativa todos os voxels que satisfazem à equação da esfera
`void putEllipsoid(int xcenter, int ycenter, int zcenter, int rx, int ry, int rz)`	Ativa todos os voxels que satisfazem à equação do elipsóide e atribui aos mesmos a cor atual de desenho
`void cutEllipsoid(int xcenter, int ycenter, int zcenter, int rx, int ry, int rz)`	Desativa todos os voxels que satisfazem à equação do elipsóide
`void writeOFF(const char* filename)`	grava a escultura no formato OFF no arquivo `filename`

Desses métodos apresentados, o método `writeOFF()` merece uma explicação mais detalhada acerca da forma como ele representará a escultura digital.

É fácil perceber que cada voxel da matriz possui apenas quatro propriedades e apresentá-lo num software de visualização requer um cuidado adicional: decidir que estrutura geométrica será usada para representar esse voxel e como ela será construída.

Poder-se-ia pensar em várias representações possíveis, tais como esferas ou pirâmides, mas a representação cúbica deverá ser a escolhida para este projeto. Dessa forma, cada vez que um voxel de coordenadas (x, y, z) tiver sua propriedade `isOn = true`, o método de gravação do formato OFF deverá prever o desenho de um cubo com lado de comprimento igual a 1 com as mesmas propriedades de cor e transparência do voxel cujo centro coincide com o centro desse voxel.

Por exemplo, se o voxel de posição $(x, y, z) = (3, 2, 5)$ tiver sua propriedade `isOn = true`, deveria ser previsto um cubo cujas faces possuem a mesma cor do cubo e abrangendo a região $x \in [2.5, 3.5]$, $y \in [1.5, 2.5]$ e $z \in [4.5, 5.5]$. Para criar o cubo, deve-se portanto, prever-se no arquivo OFF um conjunto de 8 vértices e as 6 faces que os interconectam.

Portanto, é tarefa do método `writeOFF()` criar um arquivo de computador escrito no formato `OFF` onde para um matriz com `Nvoxels` voxels com propriedade `isOn=true`, exista um total de `8*Nvoxels` vértices e `6*Nvoxels` faces para representar escultura digital equivalente às propriedades da matriz.

O uso da classe `Sculptor` é mostrado no exemplo seguinte:

```
  // cria um escultor cuja matriz tem 10x10x10 voxels
  Sculptor trono(10,10,10);
  // para mudar a cor do voxel
  trono.setColor(0,0,1.0,1.0); // azul
  // ativa os voxels na faixa de [x,y,z] pertencendo a [0-9]
  trono.putBox(0,9,0,9,0,9);
  // desativa os voxels na faixa de [x,y,z] pertencendo a [0-9]
  trono.cutBox(1,8,1,9,1,9);
  // grava a escultura digital no arquivo "trono.off"
  trono.writeOFF("trono.off");
```

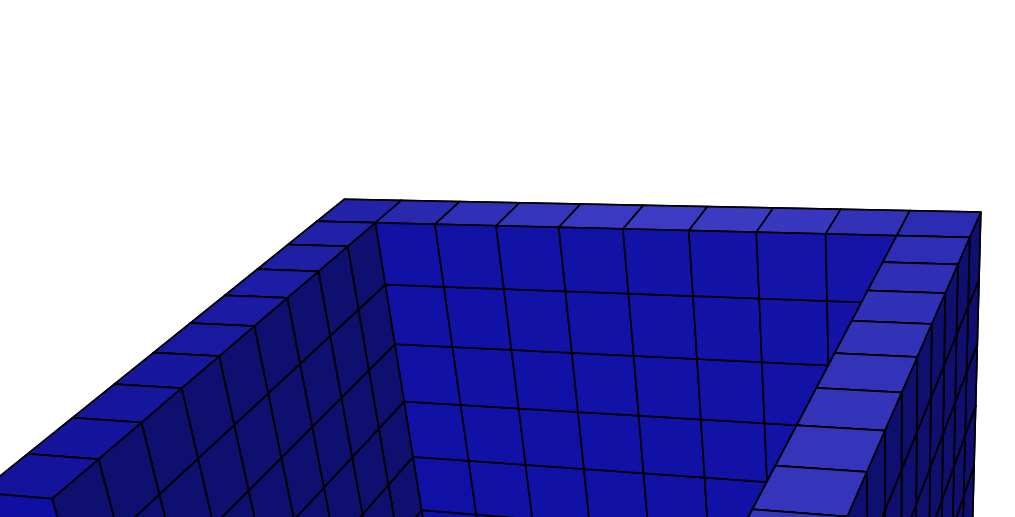

Figura 4. Representação a figura trono.off.

Veja que os processo de alocação e liberação da memória devem ser previstos no construtor e destrutor da classe Sculptor.

4.1. Criação de um programa de testes

Elabore um pequeno programa de testes que explore o uso de TODAS os métodos especificados para desenvolver uma escultura digital e salvá-la no formatos `OFF`

5. A classe Escultor - Parte 2 - Estruturas avançadas

Seu projeto deverá ser capaz de tratar classes abstratas para manipulação de figuras geométricas simples, realizando operações de desenho em uma matriz de pontos.

Para isso, a estrutura criada para a classe `Sculptor` deverá ser repensada e uma nova classe chamada `FiguraGeometrica` entrará como participante do processo de desenho.

Não serão criados objetos dessa nova classe, mas ela agora será a base para várias outras classes concretas, que proverão funcionalidades semelhantes àquelas providas pela classe `Sculptor` do projeto anterior.

5.1. Criação da classe abstrata `FiguraGeometrica`

Crie uma classe abstrata `FiguraGeometrica` para representar objetos primitivos genéricos, e preveja para esta classe pelo menos uma característica:

- Uma função virtual pura `draw(Sculptor &t)` que instrua o objeto a se desenhar em um objeto do tipo `Sculptor`. Essa função deverá ser implementada nas subclasses da classe `FiguraGeometrica`.

5.2. Criação da classe concreta `PutVoxel`

- Crie uma classe `PutVoxel` derivada de `FiguraGeometrica`. Um voxel deverá ser especificado para desenho conforme sua posição no espaco, sua cor e transparência.
- O método `draw(Sculptor &t)` deverá ser implementado.

5.3. Criação da classe concreta `CutVoxel`

- Crie uma classe `CutVoxel` derivada de `FiguraGeometrica`. Um voxel deverá ser especificado para remoção conforme sua posição no espaco.
- O método `draw(Sculptor &t)` deverá ser implementado.

5.4. Criação da classe concreta `PutBox`

- Crie uma classe `PutBox` derivada de `FiguraGeometrica`. Uma caixa deverá ser especificada para desenho conforme os limites espaciais fornecidos para as três dimensões, sua cor e transparência.
- O método `draw(Sculptor &t)` deverá ser implementado.

5.5. Criação da classe concreta `CutBox`

- Crie uma classe `CutBox` derivada de `FiguraGeometrica`. Uma caixa deverá ser especificada para remoção conforme os limites espaciais fornecidos.
- O método `draw(Sculptor &t)` deverá ser implementado.

5.6. Criação da classe concreta `PutSphere`

- Crie uma classe `PutSphere` derivada de `FiguraGeometrica`. Uma esfera deverá ser especificada para desenho conforme a posição do centro, seu raio em pixels, bem como sua cor e transparência.
- O método `draw(Sculptor &t)` deverá ser implementado.

5.7. Criação da classe concreta `CutSphere`

- Crie uma classe `CutSphere` derivada de `FiguraGeometrica`. Uma esfera deverá ser especificada para remoção conforme a posição do centro e seu raio em pixels.
- O método `draw(Sculptor &t)` deverá ser implementado.

5.8. Criação da classe concreta `PutEllipsoid`

- Crie uma classe `PutEllipsoid` derivada de `FiguraGeometrica`. Uma esfera deverá ser especificada para desenho conforme a posição do centro, seus raios em pixels, bem como sua cor e transparência.
- O método `draw(Sculptor &t)` deverá ser implementado.

5.9. Criação da classe concreta `CutEllipsoid`

- Crie uma classe `CutEllipsoid` derivada de `FiguraGeometrica`. Uma esfera deverá ser especificada para remoção conforme a posição do centro e seus raios em pixels.
- O método `draw(Sculptor &t)` deverá ser implementado.

5.10. Armazenamento de figuras

- Prepare um recurso para ler figuras gravadas em um arquivo e desenhá-las conforme as instruções presentes. O arquivo poderá conter os seguintes códigos, que deverão ser interpretados pela sua aplicação e utilizar a classe abstrata `FiguraGeometrica` para tratar e gerar o desenho conforme os códigos utilizados na tabela Tabela 2.
- O arquivo conterá várias dessas linhas, cada uma contendo uma instrução orientando o tipo de desenho que se deseja realizar na sequência. Logo, novo programa lerá esse arquivo e criará um arquivo de saída no formato `OFF` contendo o desenho conceptualizado no arquivo fornecido.

Tabela 2. Códigos das figuras geométricas

Código	Função
`dim largura altura profundidade`	define as dimensões do desenho. Ex: `dim 60 40 55`
`putvoxel x0 y0 z0 r g b a`	desenha voxel na posição $(x0, y0, z0)$ com a cor (r, g, b, a) Ex: `putvoxel 5 5 10 1.0 0.8 0.4 0.3`
`cutvoxel x0 y0 z0`	apaga voxel na posição $(x0, y0, z0)$ com a cor (r, g, b, a) Ex: `cutvoxel 5 5 10`
`putbox x0 x1 y0 y1 z0 z1 r g b a`	desenha um paralelepípedo delimitado por $x \in (x0, x1)$, $y \in (y0, y1)$, $z \in (z0, z1)$ com a cor (r, g, b, a). Ex: `putbox 10 15 12 33 1 8 0.4 0.5 0.8 1`
`cutbox x0 x1 y0 y1 z0 z1`	apaga um paralelepípedo delimitado por $x \in (x0, x1)$, $y \in (y0, y1)$, $z \in (z0, z1)$. Ex: `cutbox 10 15 12 33 1 8`
`putsphere x0 y0 z0 raio r g b a`	desenha uma esfera com centro no ponto $(x0, y0, z0)$, raio especificado e com a cor (r, g, b, a). Ex: `putsphere 10 10 5 3 1.0 1.0 0.5 0.3`
`cutsphere x0 y0 z0 raio`	apaga uma esfera com centro no ponto $(x0, y0, z0)$ e raio especificado. Ex: `cutsphere 10 10 5 3`
`putellipsoid x0 y0 z0 raiox raioy raioz r g b a`	desenha um elipsóide com centro no ponto $(x0, y0, z0)$, raios especificados e com a cor (r, g, b, a). Ex: `putellipsoid 10 10 5 3 4 5 1.0 0.5 0.9 0.4`
`cutellipsoid x0 y0 z0 raio`	apaga um elipsóide com centro no ponto $(x0, y0, z0)$ e raios especificados. Ex: `cutellipsoid 10 10 5 3 4 5`

5.11. Teste das funcionalidades implementadas

Crie uma figura e um programa-exemplo para testar as implementações que você realizou. Sugere-se que os seguintes recursos possam ser explorados para facilitar a tarefa de criação do código-fonte.

- Toda a figura poderá ser armazenada em um container de ponteiros para `FiguraGeometrica`.
- Faça utilização exaustiva das classes da biblioteca padrão de gabaritos. As classes `vector`, `string` e `stringstream` poderão ser amplamente utilizados para realizar a implementação das funcionalidades.
- Crie uma figura e represente-a em um arquivo utilizando os códigos fornecidos. Use uma combinação adequada de classes para converter sua figura em uma escultura digital no formato `OFF`. A matriz deverá ser salva em outro arquivo usando um dos métodos da classe `Sculptor`.

6. Programa interativo para criação de esculturas

Prepare um programa interativo baseado nas biblioteca de programação visual Qt para utilizar a classe `Sculptor` para permitir que um usuário realize desenhos em uma espécie de *Paint 3D*.

Considere-se à vontade para criar o programa conforme suas próprias idealizações, mas que a construção permite que esse seja utilizável por alguém que deseje trabalhar com as funções de sua classe `Sculptor`.

Será exigido que seu programa realize as seguintes tarefas:

- Criar uma abstração de um objeto da classe `Sculptor` com dimensões fornecidas pelo usuário.
- Permitir que o usuário visualize o objeto criado utilizando algum artifício.
- Permitir que o usuário modifique os parâmetros dos vários tipos de métodos previstos na classe escultor, tais como raios de uma esfera a ser desenhada, dimensões de uma caixa (*box*) ou cor de desenho.
- Agregar os métodos especificados para a classe `Sculptor` e permitir que o usuário os selecione algum deles usando botões de uma toolbox.
- Ao movimentar o mouse sobre a abstração do objeto da classe com o botão esquerdo pressionado, o método associado [illegible]