Trabalho de CG1

Carolina, Daniel, Heitor, Leonardo e Mariana

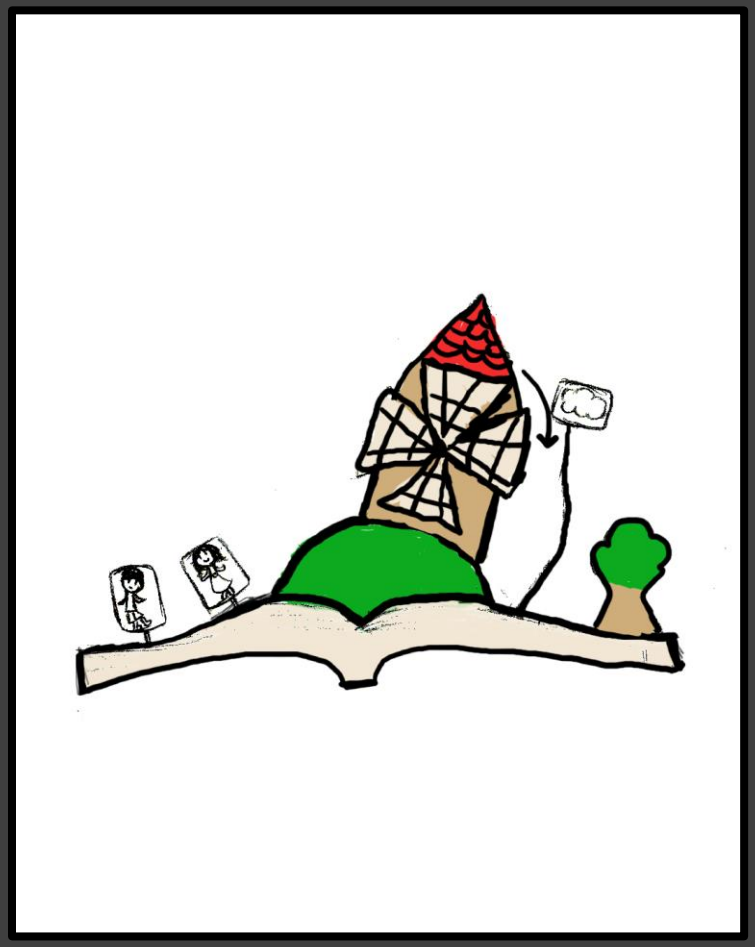
Como as coisas funcionam

- Criação do modelo
- Leitura
  - Processamentos antes da renderização
- Matrizes de Transformação
- Funções do OpenGL

Criação do Modelo

Design

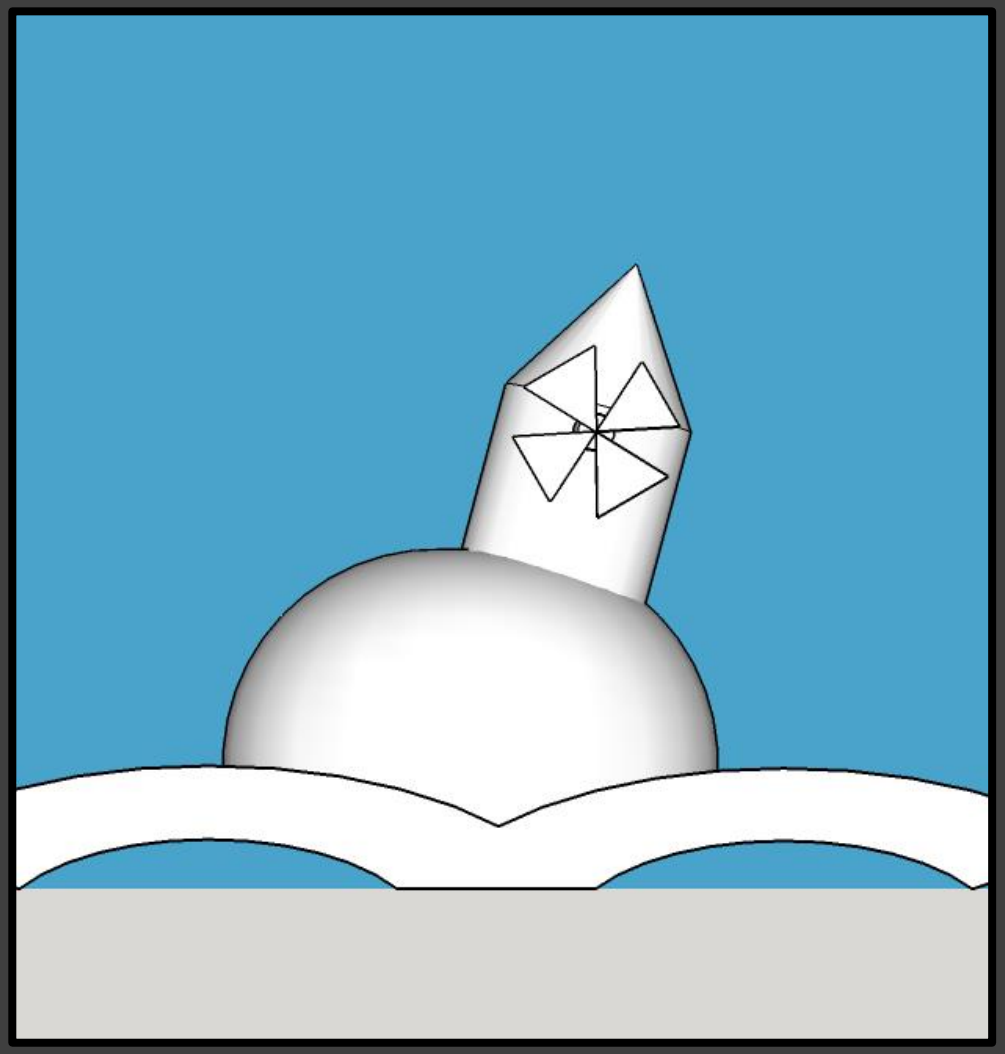
- Tema de fantasia
- Inspiração em livros de pop-up
- Mais artificial do que orgânico.



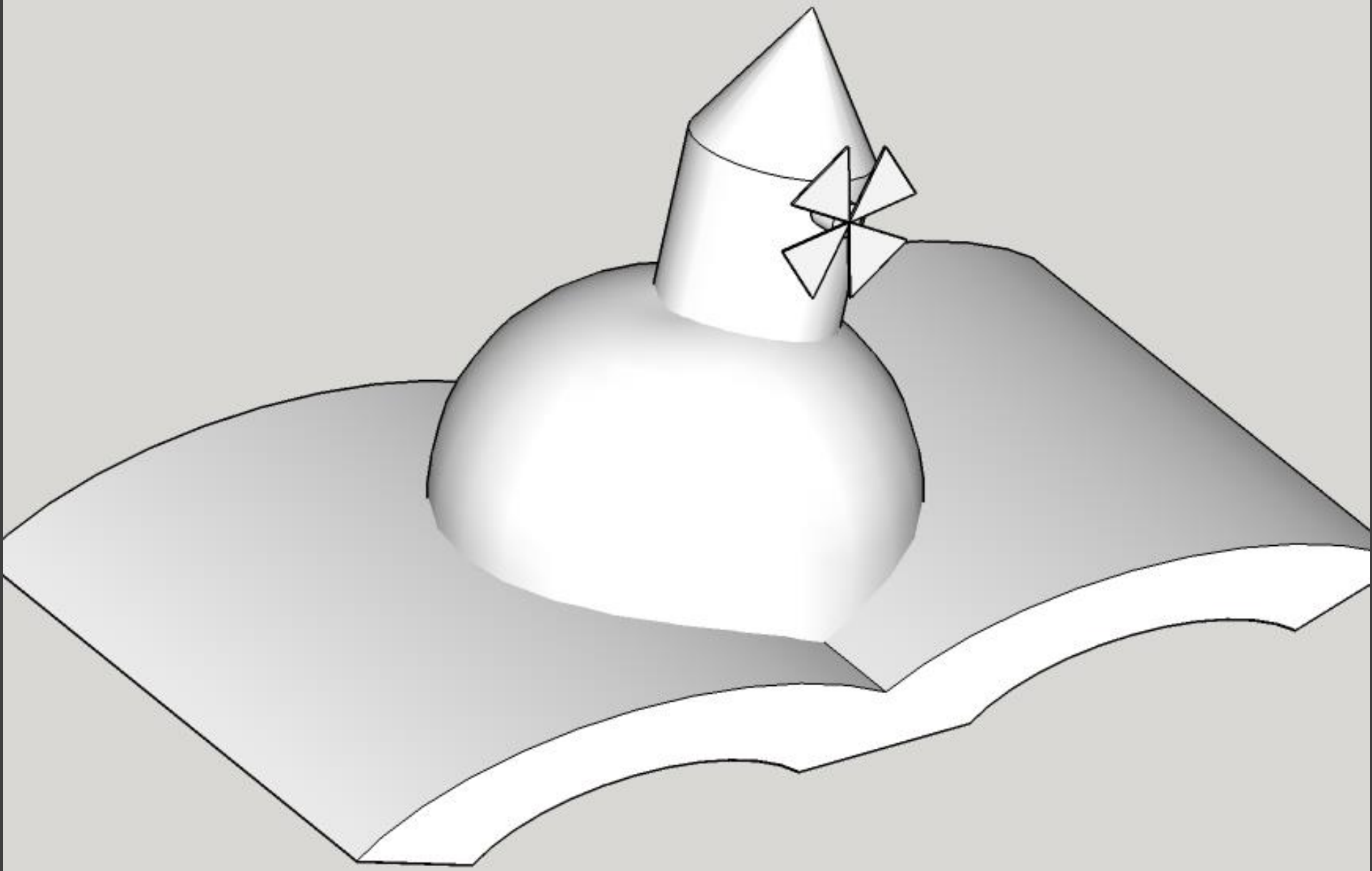
SketchUp Make

Sketchup era uma ferramenta do @Last Software (Depois comprada pela Google e atualmente pela Trimble).

Usado por arquitetos, engenharia mecânica e civil.



Modelo Atual



Leitura e Processamento

Arquivo .obj

```
# Comentário
g Nome Do Grupo

v 1193.55 106.86 1.59046e-014
# Representa um vértice (x, y, z) o w é opcional.
vt -2.53675e-013 -44.6388
# Representa coordenadas da textura (u, v) - Não usado
vn 0.249008 0.968501 -9.9527e-016
# Representa as coordenadas de uma normal - Não usado
v 1129 120.18 -792.172
vt 31.1879 -42.044
vn 0.160157 0.987092 -1.22081e-015
v 1129 120.18 1.59046e-014
vt -2.39435e-013 -42.044
f 1/1/1 2/2/2 3/3/2
# Representa um face com índices para v/vt/vn
```

Lendo um .obj

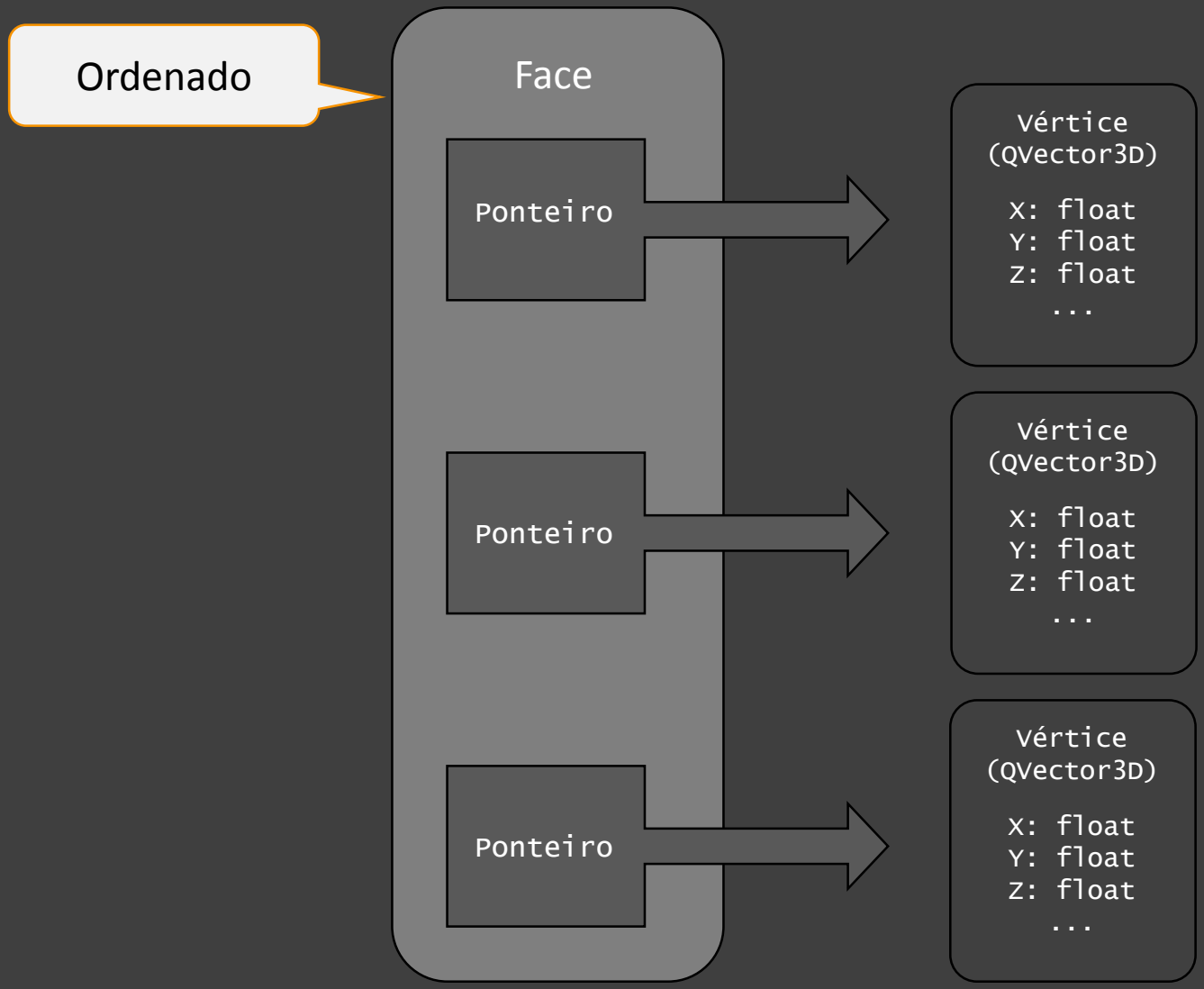
- Uma máquina de estados para cada “identificador”
- Enquanto não estamos no final do arquivo vamos ver o que fazer baseado na primeira letra do arquivo
- Mas primeiro temos que descrever como é um modelo no nossos sistema

Modelo

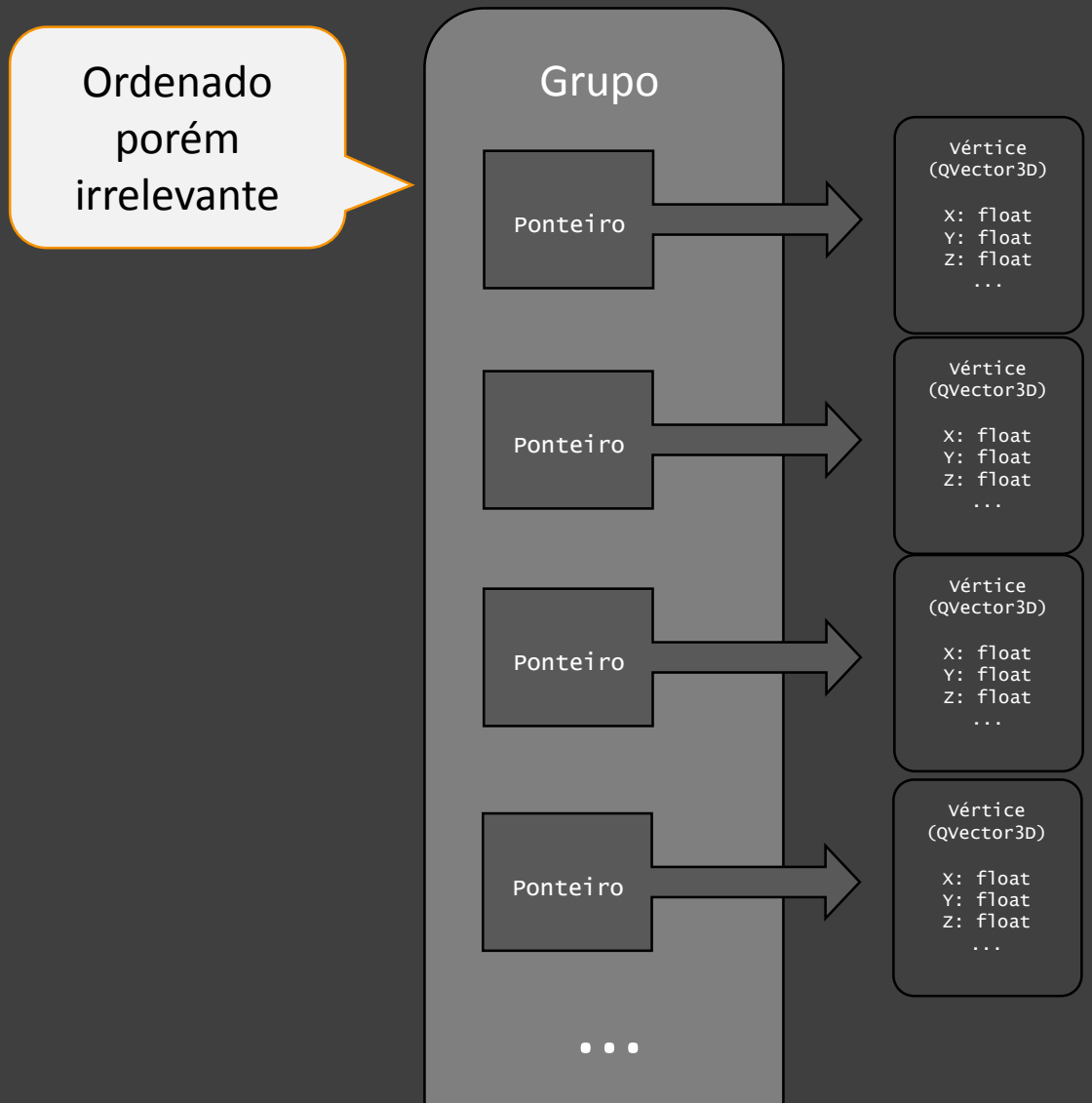
```
/** Uma face é simplesmente uma tripla de Vertices. */
typedef std::tuple<QVector3D*,QVector3D*,QVector3D*> Face;

/** Um vertice tem a mesma representação que um QVector3D. */
typedef QVector3D Vertice;

/* Um grupo de vértices é uma lista de vértices */
typedef QList<Vertice*> Grupo;

/**
 *  Esta classe representa um modelo.
 */
class Model {
private:
    QList<Vertice> vertices; // A lista de vertices normalzados do modelo.
    QList<Face> faces; // A lista de faces do modelo.
    QList<Grupo> grupos; // A lista de grupos
    Vertice pontoMedio; // Ponto médio do objeto, pode estar desatualizado
public:
    //Métodos
    void desenhar();
    void aplicarTransformacao(TransformMatrix m); // Aplica a transformação no modelo inteiro
    void aplicarTransformacao(TransformMatrix m,int i); // Aplica a transformação no i-ésimo grupo do modelo.
    // Get and Setters
    QList<Vertice> getVertices();
    QList<Face> getFaces();
    Vertice getPontoMedio(); // Retorna e atualiza o ponto médio
    Vertice getPontoMedio(int i); // Retorna o ponto médio do i-ésimo grupo
    // Construtores
    Model(QString pathname); // Construtor que recebe um arquivo OBJ.
    Model(QList<Vertice> vertices,QList<Face> faces); // Construtor que recebe uma lista de vertices e faces.
};
```

Vértices, Faces e Grupos

Vértices, Faces e Grupos



Vértices, Faces e Grupos



E agora?

- No .obj, as linhas com ‘v’ e ‘f’ seguem uma *expressão regular*.
- "([\S]*)([\s]*)([\S]*[\s]*)([\S]*[\s]*)([\S]*[\s]*)“
  - Identificador (‘f’ ou ‘v’)
  - Primeiro vértice/coordenada
  - Segundo vértice/coordenada
  - Terceiro vértice/coordenada
- No caso de faces temos que analisar os ‘/’ para só ler o índice do vértice do modelo, não de coordenadas de textura ou de normais.
- No fim, esses valores serão normalizados e também iremos calcular o ponto médio do modelo.

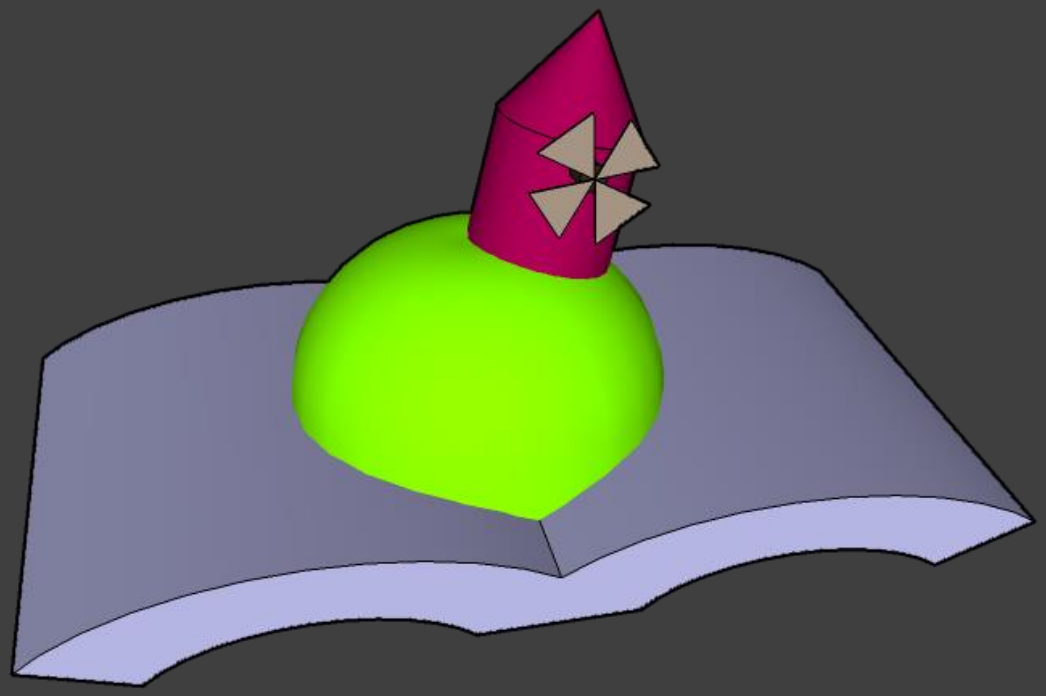
Arquivo .obj

```
# Comentário
g Nome Do Grupo

v 1193.55 106.86 1.59046e-014
vt -2.53675e-013 -44.6388
vn 0.249008 0.968501 -9.9527e-016
v 1129 120.18 -792.172
vt 31.1879 -42.044
vn 0.160157 0.987092 -1.22081e-015
v 1129 120.18 1.59046e-014
vt -2.39435e-013 -42.044
f 1/1/1 2/2/2 3/3/2
```

E agora?

- Quando um ‘g’ é lido, empilhamos um novo grupo e todos os próximos vértices irão fazer parte do topo da pilha.
- Os grupos são úteis quando queremos transformar só alguns objetos.
  - Motivo principal para os grupos não saberem as faces que seus vértices pertencem.
  - É isso que faz o moinho girar.

Modelo com Grupos Explicitados



Processamentos

- Quando recebemos um modelo .obj procuramos a maior de suas coordenadas para normalizar o modelo.
- Cálculo do ponto médio também é feito nessa hora, contudo não temos acesso direto a esse valor já que com as transformações o modelo vai mudando seu ponto médio.

Matrizes de Transformação

Matrix de Transformação

```
class TransformMatrix
{
private:
    double** matrix;
public:
    // Métodos
    void mostrar();

    // Operadores
    TransformMatrix operator*(const TransformMatrix& direita);  // Multiplicação de Matriz
    TransformMatrix& operator= (const TransformMatrix& direita);// Atribuição
    double& operator() (unsigned i, unsigned j); // Acesso para modificação
    double operator() (unsigned i, unsigned j) const; // Acesso para visualização

    // Construtores
    TransformMatrix(); // Identidade
    TransformMatrix(double a[4][4]); // Por um vetor
    TransformMatrix(double** a); // Por um vetor dinâmico
    TransformMatrix(const TransformMatrix &obj); //Copiador

    // Desconstrutor
    ~TransformMatrix();

    //Get and Setter
    void setMatrix(double a[4][4]);
};

//Operador Matrix * QVector4D
QVector4D operator*(const TransformMatrix& esquerda, const QVector4D& direita);

//Funções que retornam as matrizes de transformação
TransformMatrix translacao(double x, double y, double z);
TransformMatrix translacao(QVector3D v);
TransformMatrix rotacaoX(double angGraus);
TransformMatrix rotacaoY(double angGraus);
TransformMatrix rotacaoZ(double angGraus);
TransformMatrix rotacaoVetor(double angGraus, QVector3D v);
TransformMatrix escala(double x, double y, double z);
```

Sobrecarga de Operador

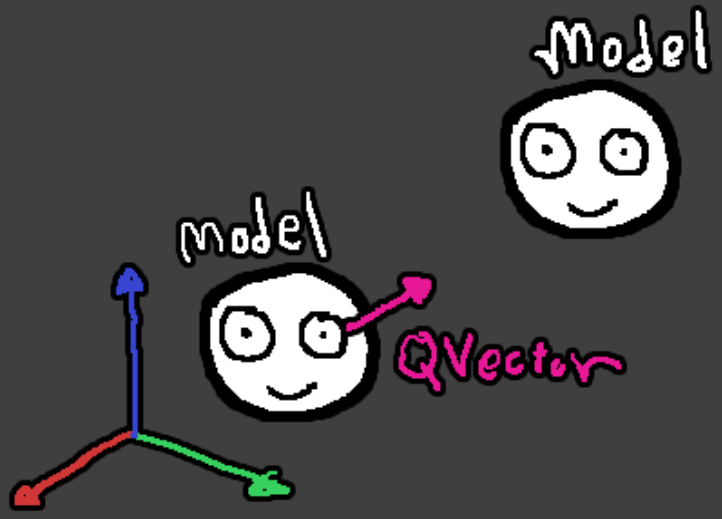
- `TransformMatrix operator*(const TransformMatrix& direita) const`
  - Multiplicação de Matriz em $O(n^3)$ sem modificar a matriz
- `QVector4D operator*(const TransformMatrix& esquerda, const QVector4D& direita)`
  - Multiplicação Matrix e Vetor, praticamente a mesma função de cima
- `double operator()(unsigned i, unsigned j)`
  - Operadores de acesso da matriz. Vem em pares: de acesso e de modificação

Funções de Transformação

```
//Funções que retornam as matrizes de transformação
TransformMatrix translacao(double x, double y, double z);
TransformMatrix translacao(QVector3D v);
TransformMatrix rotacaoX(double angGraus);
TransformMatrix rotacaoY(double angGraus);
TransformMatrix rotacaoZ(double angGraus);
TransformMatrix rotacaoVetor(double angGraus, QVector3D v);
TransformMatrix escala(double x, double y, double z);
```

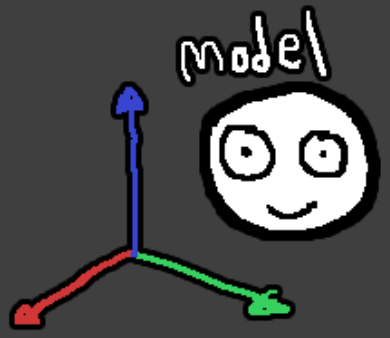
- Definidos no mesmo arquivo da matriz de transformação.
  - Porém, não pertencem a classe.
- Recebem parâmetros específicos a cada transformação, constroem e retornam um objeto da classe TransformMatrix que descreve a transformação desejada.

Translação



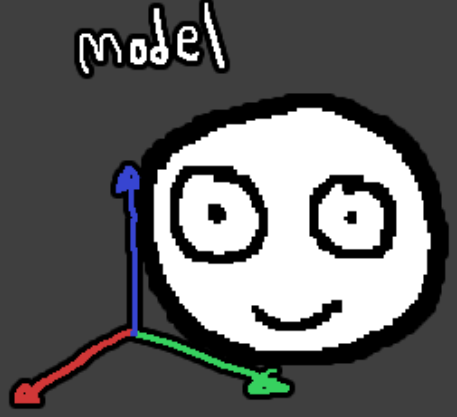
- Faz adição Ponto-Vetor
- Parâmetros:
  - Três valores double, representando o deslocamento em X,Y e Z
  - Ou um QVector3D com o mesmo significado.

Escala



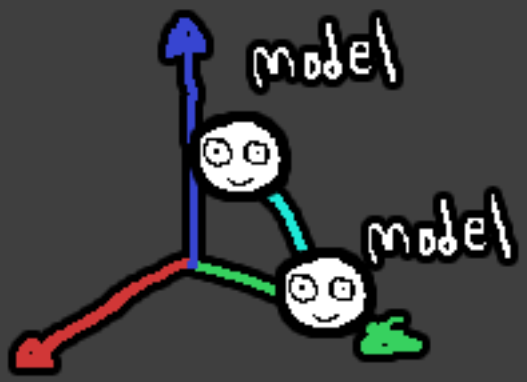
- Uma escala potencialmente não uniforme
- Parâmetros
  - Três valores double representando a escala em X,Y,Z

Escala



- Uma escala potencialmente não uniforme
- Parâmetros
  - Três valores double representando a escala em X,Y,Z

Rotação



- Em algum dos planos ou em um vetor específico
- Parâmetros
  - Ângulo em graus
  - Ângulo e um QVector3D representando o eixo de rotação

Rotação

- Em algum dos planos ou em um vetor específico
- Parâmetros
  - Ângulo em graus
  - Ângulo e um QVector3D representando o eixo de rotação

Como aplicar a transformação?

- Função do modelo que recebe uma matriz de transformação e itera por seus vértices, multiplicando a matriz por cada um deles
- Parâmetros
  - A matriz de transformação retornada pelas funções
  - Pode-se também especificar apenas um grupo no qual a transformação será aplicada, no caso a função só itera nos vértices daquele grupo

Aplicação e Renderização

- Cada face do modelo é enviado pro pipeline usando o glVertex3d.
- Os modelos tem que ser triangularizados previamente.
- Para aplicar uma transformação usamos um iterador mutável e multiplicamos cada vetor por uma matriz de transformação

```
void tnw::Model::desenhar()
{
    glBegin(GL_TRIANGLES);
    foreach (tnw::Face f, this->faces) {
        glVertex3d((std::get<0>(f))->x(),(std::get<0>(f))->y(),(std::get<0>(f))->z());
        glVertex3d((std::get<1>(f))->x(),(std::get<1>(f))->y(),(std::get<1>(f))->z());
        glVertex3d((std::get<2>(f))->x(),(std::get<2>(f))->y(),(std::get<2>(f))->z());
    }
    glEnd();
}

void tnw::Model::aplicarTransformacao(TransformMatrix m)
{
    QMutableListIterator<tnw::Vertice> iter(vertices);
    while (iter.hasNext()) {
        tnw::Vertice v = iter.next();
        iter.setValue((m*QVector4D(v,1)).toVector3D());
    }
}
```

Produto Final