Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu
Departamento de informática

Relatório do projeto de Programação Orientada a Objetos

Processamento de modelos criados no Blender

Licenciatura em Engenharia Informática

Grupo:
Bruno Lopes nº17740
Carlos Costa nº17741
Pedro Costa nº17754

Viseu, 2020

Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu
Departamento de informática

Relatório do projeto de Programação Orientada a Objetos

Licenciatura em Engenharia Informática
2º ano
1º semestre
Ano letivo 2019/2020

Grupo:
Bruno Lopes nº17740
Carlos Costa nº17741
Pedro Costa nº17754

Viseu, 2020

Agradecimentos

Queremos agradecer aos professores da disciplina Francisco Morgado, Carlos Simões e Luis Soares por todo o conhecimento transmitido.

Índice

Índice de figuras

1. Introdução

No âmbito da unidade curricular de Programação Orientada a Objetos (poo), foi-nos solicitado, como projeto, a criação de um programa que leia ficheiros .obj e calcule/determine algumas informações acerca dos modelos que se encontram gravados nos ficheiros. Uma das informações fundamentais é a área de cada modelo.

2. Trabalho Prático

2.1. Load

A função load é uma função do tipo booleano, em que retorna true caso não ocorra nenhum erro, e retorna falso se ocorrer algum erro. Esta função vai verificar se um ficheiro passado por parâmetro já foi carregado ou não. Se já tiver sido carregado, aparece uma mensagem a dizer que o ficheiro já foi carregado previamente, e retorna falso. Se o ficheiro não tiver sido carregado, a função vai alocar espaço para assim o poder carregar para um novo modelo.

```
bool SGestao::Load(const string& fich)
{
    ifstream f(fich);
    if (f.is_open())
    {
        f.close();

        if (PesquisarModelo(fich))
        {
            cout << "Já foi carregado um ficheiro com o nome [" << fich << "] previamente!" << endl;
            return false;
        }

        Modelo* aux = new Modelo(fich);
        if (ERRO)
            delete aux;
        else
            Lista_Modelos.push_back(aux);  //criacao do modelo
    }
    else
        cout << "ERRO: O ficheiro [" << fich << "] não existe!" << endl;

    return not ERRO;  //retorna true se nao ocorrer erro e false se ocorrer
}
```

Figura 2-1 - Função Load

2.2. Função Contar

A função contar tem por objetivo contar o número de vértices, arestas ou faces. Para tal, o programa executa um ciclo for desde o início até ao final da lista de modelos, contando assim o número de vértices, faces ou arestas, de acordo com o pretendido.

```
int SGestao::Contar(Tipo T)
{
    int contador = 0;
    switch (T)
    {
    case VERTICE:
        for (list<Modelo*>::iterator it = Lista_Modelos.begin(); it != Lista_Modelos.end(); ++it)
            contador += (*it)->get_numero_vertices();
        break;

    case ARESTA:
        for (list<Modelo*>::iterator it = Lista_Modelos.begin(); it != Lista_Modelos.end(); ++it)
            contador += (*it)->get_numero_arestas();
        break;

    case FACE:
        for (list<Modelo*>::iterator it = Lista_Modelos.begin(); it != Lista_Modelos.end(); ++it)
            contador += (*it)->get_numero_faces();
        break;
    }

    return contador;
}
```

Figura 2-2 - Função Contar

2.3. Envolvente

A função envolvente é uma função booleana, ou seja, retorna apenas true ou false. É passado um ficheiro que contém um modelo por parâmetro. Se esse modelo não existir na lista de modelos, significa que o modelo não está carregado na lista de modelos e consequentemente a função retorna false.

No caso de existir o modelo, o programa vai calcular o ponto mínimo e o ponto máximo, e retorna true.

```
bool SGestao::Envolvente(const string& fich, Ponto& Pmin, Ponto& Pmax) {
    Modelo* m = PesquisarModelo(fich);
    if (m == NULL) // nao foi encontrado o modelo/a lista esta vazia
        return false;
    else
    {
        m->Ponto_minimo(Pmin);
        m->Ponto_maximo(Pmax);
        return true;
    }
}
```

Figura 2-3 - Envolvente

O ponto mínimo e máximo é calculado na função Adicionar_vertice da classe Modelo, tal como podemos ver na figura seguinte.

```
bool Modelo::Adicionar_vertice(string linha)
{
    lista_vertices.push_back(new Vertice(linha));  //adicionar o vertice a lista

    if (ERRO)
        return false;

    if (lista_vertices.size() == 1)  //se for o primeiro vertice a ser adicionado, inicializar o ponto maximo e o ponto minimo
    {
        **lista_vertices.begin() = PontoMinimo;
        **lista_vertices.begin() = PontoMaximo;
    }
    else
    {
        list<Vertice*>::iterator it = lista_vertices.end();
        it--;
        **it < PontoMinimo;
        **it > PontoMaximo;
    }

    return true;
}
```

Figura 2-4 - Função adicionar vértice

Para calcular o ponto mínimo e máximo, calculamos através de operators presentes na classe vértice, tal como podemos ver na figura 2-5

```
void Vertice::operator < (Ponto& Pmin) {
    if (Pmin.x > x)
        Pmin.x = x;
    if (Pmin.y > y)
        Pmin.y = y;
    if (Pmin.z > z)
        Pmin.z = z;
}

void Vertice::operator > (Ponto& Pmax) {
    if (Pmax.x < x)
        Pmax.x = x;
    if (Pmax.y < y)
        Pmax.y = y;
    if (Pmax.z < z)
        Pmax.z = z;
}

void Vertice::operator = (Ponto& P) {
    P.x = x;
    P.y = y;
    P.z = z;
}
```

Figura 2-5 - Operators

2.4. Memória

A função de memória vai calcular toda a memória usada no programa.

Primeiramente, guarda numa variável a memória ocupada pela lista de modelo. Seguidamente, adiciona o tamanho de cada elemento presente na lista de modelos.

```
int SGestao::Memoria()
{
    int memoria = 0;

    memoria += 1 * sizeof(list<Modelo*>);
    memoria += Lista_Modelos.size() * sizeof(Modelo*);
    for (list<Modelo*>::iterator it = Lista_Modelos.begin(); it != Lista_Modelos.end(); ++it)
    {
        memoria += (*it)->Memoria();
    }

    return memoria;
}
```

Figura 2-6 – Memória do modelo

Após isso, vai percorrer um ciclo for desde o inicio até ao final da lista de modelos, e vai adicionar o tamanho de memória de cada tipo de variável (double, int, string, list, classe, etc), o tamanho de cada vértice da lista (todos os vértices têm o mesmo tamanho de memória), o tamanho de memória de cada aresta, tamanho de memória de cada vetor de arestas e por fim, percorre um ciclo for desde o inicio da lista de faces até ao final da lista, adicionando o tamanho que cada face ocupa.

No final de tudo, temos uma variável que guarda o tamanho de memória que todo o programa ocupa.

```
int Modelo::Memoria()
{
    int memoria = 0;

    memoria += 5 * sizeof(double);
    memoria += 2 * sizeof(int);
    memoria += 3 * sizeof(Face*);
    memoria += 2 * sizeof(Ponto);
    memoria += 1 * sizeof(Aresta**);
    memoria += 1 * sizeof(list<Aresta*>);
    memoria += 1 * sizeof(Vertice**);
    memoria += 1 * sizeof(list<Vertice*>);
    memoria += 1 * sizeof(list<Face*>);
    memoria += 1 * sizeof(string);
    memoria += 2 * lista_vertices.size() * sizeof(Vertice*);
    memoria += lista_vertices.size() * (*lista_vertices.begin())->Memoria();  //os vertices tem todos a mesma memoria
    memoria += 2 * lista_arestas.size() * sizeof(Aresta*);
    for (int i = 0; i < lista_arestas.size(); ++i)
    {
        memoria += vetor_arestas[i]->Memoria();
    }
    memoria += 1 * lista_faces.size() * sizeof(Face*);
    for (list<Face*>::iterator it = lista_faces.begin(); it != lista_faces.end(); ++it)
    {
        memoria += (*it)->Memoria();
    }

    return memoria;
}
```

Figura 2-7 - Memória

2.5. Modelo com mais memória

Para determinar qual o modelo com mais memória, vai ser percorrida a lista de modelos, e guardar numa variável a memória que um modelo ocupa. Esta memória é calculada pela função de memória explicada anteriormente (função memória do modelo). Após isso, o programa verifica se aquele modelo ocupa mais memória que os modelos anteriormente calculados. Se ocupar, a variável "mais_mem" guarda o valor da memória ocupada por esse modelo. Se não ocupar mais memória que algum dos anteriores, o programa passa a verificar o modelo seguinte.

```
Modelo* SGestao::ModeloMaisMemoria()
{
    int mais_mem = -1;
    Modelo* mais_m = NULL;
    int aux;

    for (list<Modelo*>::iterator it = Lista_Modelos.begin(); it != Lista_Modelos.end(); ++it)
    {
        aux = (*it)->Memoria();
        if (mais_mem < aux)
        {
            mais_mem = aux;
            mais_m = *it;
        }
    }

    return mais_m;
}
```

Figura 2-8 - Modelo com mais memória

2.6. Número de interseções

A função NumIntersecoes calcula o número de interseções que ocorrem em cada modelo.

É criada uma variável que serve como contador, e criada uma reta, que contém o ponto A e um vetor, que define a direção da reta.

```
int SGestao::NumInterseccoes(Ponto A, Ponto B)
{
    int contador = 0;

    Reta R;
    R.A.x = A.x;
    R.A.y = A.y;
    R.A.z = A.z;

    R.v[0] = B.x - A.x;
    R.v[1] = B.y - A.y;
    R.v[2] = B.z - A.z;

    for (list<Modelo*>::iterator it = Lista_Modelos.begin(); it != Lista_Modelos.end(); ++it)
    {
        contador += (*it)->NumInterseccoes(R);
    }

    return contador;
}
```

Figura 2-9 - Número de interseções

O sistema vai percorrer toda a lista de modelos e vai calculando as interseções que existem em cada modelo, percorrendo todas as faces do modelo e verificando se a reta interseta a face. Se intersetar, adiciona 1 ao contador.

```
int Modelo::NumInterseccoes(Reta R)
{
    int contador = 0;

    for (list<Face*>::iterator it = lista_faces.begin(); it != lista_faces.end(); ++it)
    {
        contador += (*it)->intersecao(R);
    }

    return contador;
}
```

Figura 2-10 - Interseção da Face

2.7. Remover modelo

Esta função, tal como o nome indica, tem como objetivo remover um modelo.

O nome do modelo é passado como parâmetro da função, e o programa vai percorrer a lista de modelos e comparar o nome do ficheiro que pretendemos eliminar, com os nomes dos ficheiros presentes na lista de modelos. Se existir um modelo na lista de modelos com o nome igual ao que pretendemos eliminar, o programa vai eliminar o modelo e seguidamente vai eliminar o espaço alocado para este modelo na lista de modelos.

```
bool SGestao::RemoverModelo(const string& fich)
{
    for (list<Modelo*>::iterator it = Lista_Modelos.begin(); it != Lista_Modelos.end(); ++it)
    {
        if (fich.compare((*it)->getnome_ficheiro()) == 0)   // compara o nome do ficheiro com o nome dos fich. da Lista de modelos
        {
            delete* it;  //eliminar o modelo
            Lista_Modelos.erase(it);  //eliminar o espaco alocado para o modelo na lista
            return true;
        }
    }
    return false;
}
```

Figura 2-11 - Remover modelo

2.8. Modelos carregados

A função Numero_modelos_carregados retorna o tamanho da Lista_Modelos, o que nos dá o número total de modelos que foram carregados para o programa.

```
int Numero_modelos_carregados() { return Lista_Modelos.size(); }
```

Figura 2-12 - Número de modelos carregados

2.9. Área de um Modelo

A função AreaModelo recebe o ficheiro com o modelo. Primeiramente, a função pesquisa o modelo. Caso este não tenha encontrado o modelo ou a lista esteja vazia, retorna . Caso contrário, retorna o valor dado por get_area_total, tal como podemos ver na figura 2-13.

```
double SGestao::AreaModelo(const string& fich)
{
    Modelo* m = PesquisarModelo(fich);
    if (m == NULL) // nao foi encontrado o modelo/a lista esta vazia
        return 0;
    else
        return m->get_area_total();
}
```

Figura 2-13 - Área do modelo

A função get_area_total retorna AreaTotal.

```
double get_area_total() { return AreaTotal; }
```

Figura 2-14 - Função get_area_total

AreaTotal é a soma das áreas das faces (soma à medida que as funções são carregadas).

```
AreaTotal += f->get_area();  //calcular a area do objeto
```

Figura 2-15 - Área total

Visto que não conseguimos aceder à variaver area para ir buscar o seu valor, necessitamos de uma função que retorne o seu valor. Essa função é a get_area, tal como podemos ver na figura seguinte

```
double get_area() { return area; }
```

Figura 2-16 - Função get_area

A função calcular_area tem 2 casos. Se o número de vértices for igual a 3, executa a função calcular_area_triangulo, tal como podemos ver na figura 2-17. Senão, atribui o valor 1 à variável ERRO.

A função calcular_area_triangulo faz o cálculo da área de cada triângulo. Se normal_ao_plano for falso, então lê os vértices v1, v2 e v3 e atribui o valor do produto externo entre v1, v2 e v3 à variável normal_ao_plano.Atribui à variável area o resultado da área do triângulo.

```
void Face::calcular_area_triangulo()
{
    double a, b, c;
    double d, e, f;

    if (!normal_ao_plano)
    {
        Vertice* v1 = vetor_vertices[0];  //primeiro vertice
        Vertice* v2 = vetor_vertices[1];  //segundo vertice
        Vertice* v3 = vetor_vertices[2];  //terceiro vertice
        normal_ao_plano = produto_externo(v1, v2, v3);
    }

    area = (sqrt(pow(normal_ao_plano[0], 2) + pow(normal_ao_plano[1], 2) + pow(normal_ao_plano[2], 2))) / 2.0;
}

void Face::calcular_area()
{
    switch (numero_vertices)
    {
    case 3:
        calcular_area_triangulo();
        break;

    default:
        ERRO = 1;
        break;
    }
}
```

Figura 2-17 - Calcular área do triângulo

3. Conclusão

Com a realização deste projeto, aplicamos todos os conhecimentos adquiridos em sala de aula, bem como os métodos lecionadas ao longo do semestre. Como resultado, obtivemos um programa que é capaz de ler os arquivos .obj e de calcular todas as funcionalidades impostas, cumprindo assim todos os objetivos.
Este trabalho foi muito importante para a compreensão e aprofundamento dos métodos de desenvolvimento estudados, permitindo um aprofundamento dos nossos conhecimentos.