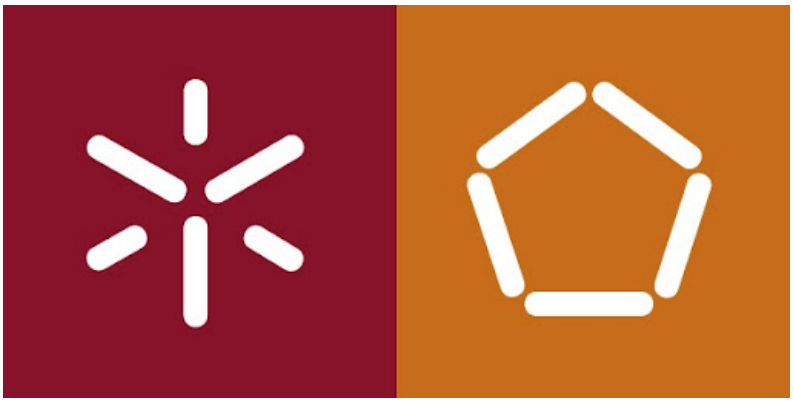
Universidade do Minho
Escola de Engenharia



Geometrical Transforms

Computação Gráfica

Relatório do Trabalho Prático

Grupo nº44
Alexandre Eduardo Vieira Martins A93242
José Eduardo Silva Monteiro Santos Oliveira A100547
Pedro Afonso Moreira Lopes A100759
Pedro Silva Ferreira A97646

5 de abril de 2024

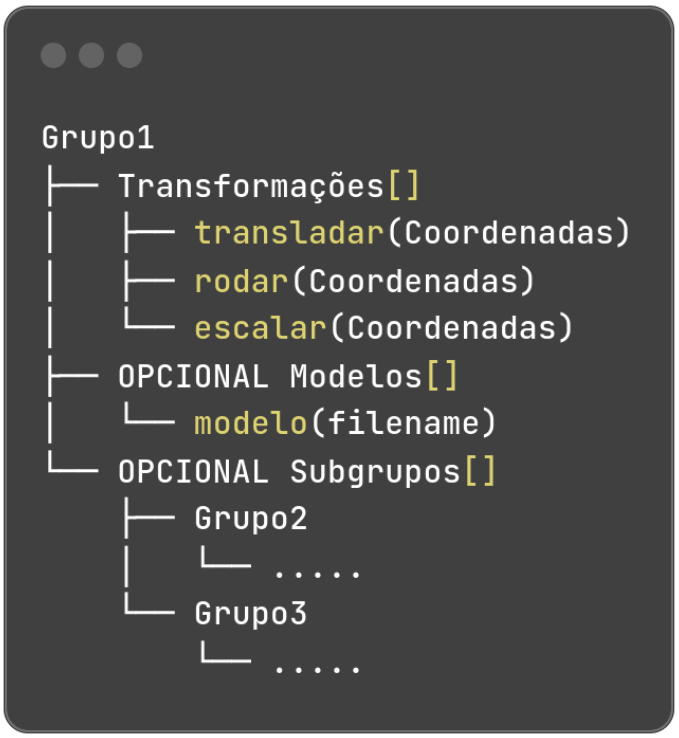
Índice

Introdução

Nesta segunda fase do trabalho prático, a abordagem principal incidiu sobre transformações geométricas, tendo como objetivo a implementação de cenas hierárquicas com elas.

Enquanto que, na fase anterior, o ficheiro XML de input para a Engine continha unicamente referências aos modelos .3d agregados num grupo, agora este passa a uma disposição que armazena transformações geométricas (translação e/ou rotação e/ou escalagem) e, opcionalmente, os tais modelos de antes e/ou "subgrupos" que contenham as suas próprias transformações. Ficam, assim, os grupos organizados em árvore.

Dum ponto de vista abstrato:



É necessário que qualquer transformação do Grupo1 seja aplicada não só aos seus modelos, mas também aos modelos dos Grupo2 e Grupo3, para manter o caráter hierárquico.

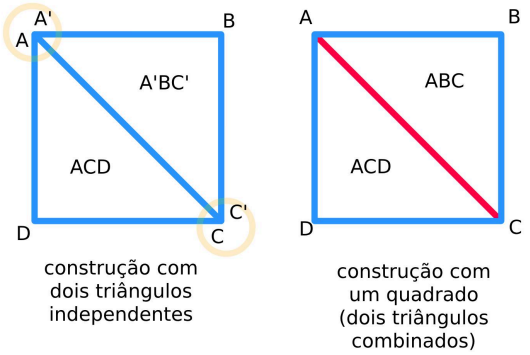
Ao contrário da fase anterior, esta não teve qualquer impacto no Generator, pois não se alterou o modo como os modelos são formados.

Generator

Enquanto que na implementação anterior se definiam faces unicamente com o triângulo, constatou-se que formas geométricas como o cubo e o plano beneficiariam duma renderização à base de quadriláteros. Tal provém do facto de que, sendo cada divisão composta por um par de triângulos adjacentes, cada um deles terá dois pontos comuns.

Assim, o número de pontos distintos são quatro, e são armazenados segundo uma estrutura *Square* e metade deles partilhados para o desenho de ambos os triângulos segundo a aresta comum, assimilando a utilização de VBOs, que vai ainda ser implementada no futuro.

```
struct Coordenadas{
  double p1, p2, p3;
};

struct Triangle{
  std::vector<Coordenadas> pontos;
};

struct Square{
  std::vector<Coordenadas> pontos;
};
```



Aquando da implementação da nova funcionalidade, notou-se um bug vestígio da 1ª fase que, apesar de não ter qualquer interferência com o objetivo desta fase diretamente, afetava a renderização correta do cubo, e como tal, merece a sua atenção neste relatório.

Ao movimentar a câmara em torno do cubo, certas faces iam desaparecendo, o que tornou a atenção para o estado de culling a ser usado, mas sem sucesso. Eventualmente, verificou-se que a ordem de armazenamento das coordenadas dos pontos no Generator resultava em triângulos orientados segundo um sistema de coordenadas de mão esquerda para certas faces, o que fazia com que as faces fossem geradas pela Engine viradas para dentro - Assim tem-se a causa por de trás da inversão do culling não ter resolvido o problema.

Engine

Sendo que uma transformação armazenada num grupo pode ser de três tipos diferentes, foi criada a estrutura Transform para a abstrair. Enquanto isso, o grupo, podendo conter quantidades indefinidas de transformações, modelos e subgrupos, implementou-se como uma classe com vetores para cada componente.

```
struct Transform{
  std::string type;
  float angle, x, y, z;

  Transform(std::string type, float angle, float x, float y, float z):
    type(type), angle(angle), x(x), y(y), z(z){}
};
```

```
class Group{

  public:

  std::vector<Transform> transforms;
  std::vector<std::string> model_paths;
  std::vector<Group> groups;

  Group():transforms(), model_paths(), groups(){}

};
```

Engine (Cont.)

Tendo em consideração que a função *handle_form()* identifica a forma do modelo segundo o seu nome de ficheiro e seguidamente desenha-o, a *handle_groups()* conjuga a aplicação dessa primeira função com as transformações, isto duma forma recursiva, permitindo aos modelos dum grupo filho ser afetado pelas transformações do pai. Durante a execução desta função, de forma a sabermos o estado da Matriz original antes das transformações, nós armazenamos as matrizes atuais e retiramos da queue de Matrizes que fizemos através das funções *pushMatrix()* e *popMatrix()*.

```
void handle_groups(const Group& group) {

    pushMatrix();

    for (const auto& transform : group.transforms) {
        if (transform.type == "translate") {
            glTranslatef(transform.x, transform.y, transform.z);
        } else if (transform.type == "rotate") {
            glRotatef(transform.angle, transform.x, transform.y, transform.z);
        } else if (transform.type == "scale") {
            glScalef(transform.x, transform.y, transform.z);
        }
    }

    for (const auto& model_path : group.model_paths)
        handle_form(model_path);

    for (const auto& sub_group : group.groups)
        handle_groups(sub_group);

    popMatrix();
}
```

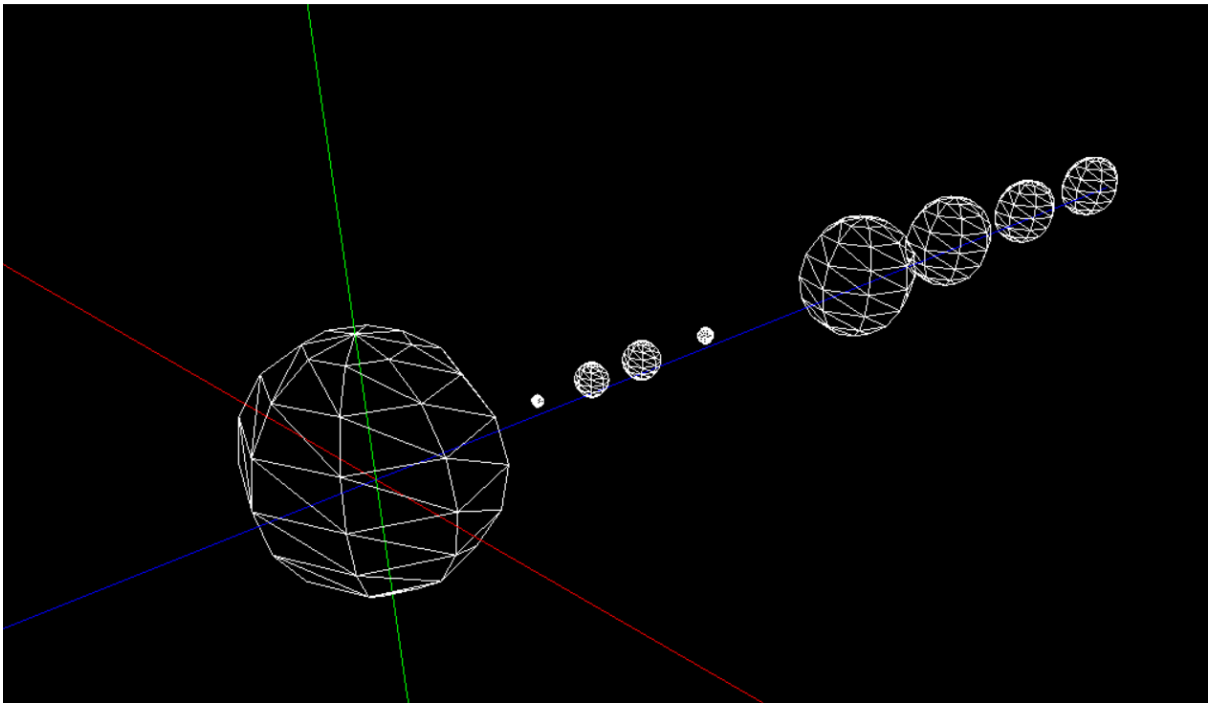
Uma das melhorias desta fase em relação à anterior foi o parsing de *xml*. Na fase 1, não existia qualquer tratamento de dados de *transforms*, nem quando um grupo se encontrava dentro de outro. Com vista às próximas fases, criamos também o tratamento de dados das *lights*.

```
void processLightElement(tinyxml2::XMLElement* lightElement) {
    float px, py, pz, dx, dy, dz, sx, sy, sz, sdx, sdy, sdz, c;

    for (tinyxml2::XMLElement* child = lightElement->FirstChildElement();
         child;
         child = child->NextSiblingElement()){

        const char* childName = child->Name();

        if (strcmp(childName, "point") == 0) {
            child->QueryFloatAttribute("posX", &px);
            child->QueryFloatAttribute("posY", &py);
            child->QueryFloatAttribute("posZ", &pz);
        }
```

```
        else if (strcmp(childName, "directional") == 0) {
            child->QueryFloatAttribute("dirX", &dx);
            child->QueryFloatAttribute("dirY", &dy);
            child->QueryFloatAttribute("dirZ", &dz);
        }
        else if (strcmp(childName, "spotlight") == 0) {
            child->QueryFloatAttribute("posX", &sx);
            child->QueryFloatAttribute("posY", &sy);
            child->QueryFloatAttribute("posZ", &sz);
            child->QueryFloatAttribute("dirX", &sdx);
            child->QueryFloatAttribute("dirY", &sdy);
            child->QueryFloatAttribute("dirZ", &sdz);
            child->QueryFloatAttribute("cutoff", &c);
        }
    }
}
```

Demo Scene

Para criar a *demo scene,* definimos um *group* que contém todos os planetas e o Sol. Tentou-se preservar alguma sensação de escala, tanto do tamanho dos planetas como das distâncias entre os mesmos. Dadas as disparidades das medidas reais, reduzimos significativamente o tamanho e distâncias dos gigantes gasosos e o tamanho do Sol por motivos meramente de apresentação.



Conclusão

Dada a fase como terminada, ficamos mais cientes de como a conjugação das transformações e dos modelos hierarquicamente torna o desenvolvimento de *scenes* complexas mais modular e flexível.

Consideramos que todos os objetivos para esta fase foram cumpridos, e inclusive melhoramos a fundação para as fases futuras, tornando-as assim mais simples de serem implementadas.