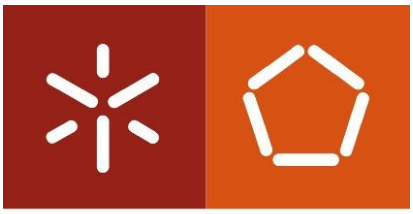


Universidade do Minho

Licenciatura em Engenharia Informática

Computação Gráfica - 2ª Fase

Grupo 14

Luís Faria (A93209) Joaquim Roque (A93310)
Gabriela Prata (A93288)

Ano Lectivo 2021/2022



0.1 Introdução

Para a segunda fase do trabalho prático, é proposto que se adicione a capacidade de aplicar transformações geométricas, de forma hierárquica, especificando-as no ficheiro XML de *input* do *engine*. No que se refere ao *generator*, não é necessário efetuar qualquer alteração.

0.2 Decisões e abordagens escolhidas

0.2.1 Estruturas de dados

De forma a poder suportar tais requisitos, a criação de estruturas de dados na fase de *parsing* revelou-se como sendo um passo fundamental. Dessa forma, surgiram as estruturas *group* e *transform*. Como os nomes sugerem, estas agregam a informação das *tags* correspondentes do ficheiro de *input*.

group

```
typedef struct group {

    std::vector<Transform> transforms;
    std::vector<std::string> models;
    uint nest_level;

    group(uint nest_level);

} Group;
```

Figura 1: Estrutura *group*

O *group* é composto por dois vetores, um para as transformações e outro para os modelos necessários, correspondendo este último a uma lista de nomes de ficheiros. Apesar da ordem de desenho dos modelos ser essencialmente irrelevante para esta fase, tal não é verdade para as transformações, daí que o *std::vector* garanta essa restrição, uma vez que a ordem relativa de inserção não é alterada.

O inteiro *nest_level* indica o nível de aninhamento, em termos de hierarquia, em que o grupo em questão se encontra. Por outras palavras, este valor corresponde a adicionar 1 ao valor herdado do grupo-pai. Se se tratar do grupo mais externo, ou seja, se não tiver grupo-pai, o seu *nest_level* é 1. Este permite, na fase de de desenho da imagem, ajustar a matriz *modelview* adequadamente (através de operações de *pop* e/ou *push*), de modo a respeitar a hierarquia estipulada. De salientar também que esta estratégia é notável na medida em que evita que os grupos contenham referências uns para os outros, o que adicionaria complexidade e, consequentemente, maior propensão a erros.

transform

```
typedef enum transform_type {
    rotate,
    translate,
    scale,
} TransformType;

typedef struct transform {

    CartPoint3d point;
    std::optional<angle_t> angle;
    TransformType type;

    transform(angle_t angle, const CartPoint3d &point);
    transform(TransformType type, const CartPoint3d &point);

} Transform;
```

Figura 2: Estrutura *transform*

As transformações, apesar de distintas, apresentam semelhanças suficientes para que a mesma estrutura seja suficiente para as representar, nomeadamente o facto de todas necessitarem de um triplo de coordenadas espaciais. O seu tipo é distinguido através do *enum transform_type*. Mais ainda, o ângulo é marcado como opcional, uma vez que apenas é necessário nas rotações.

0.2.2 Correções e otimizações

Nesta fase, o grupo decidiu efetuar algumas correções ao código desenvolvido na fase anterior. Desde logo, a geração da primitiva caixa não estava a ser feita de forma adequada, pois não estava centrada na origem, mas sim com a base contida no plano *xOz*. A primitiva plano passou a ter as diagonais dos quadrados que a formam no sentido oposto, garantindo, assim, que os nossos testes se aproximavam do esperado, de acordo com as imagens fornecidas pela equipa docente. O cone continha redundância, que foi eliminada, no número de pontos gerados, uma vez que a última *stack* era desenhada duas vezes.

No que diz respeito ao *engine*, previamente à inicialização das funções do *OpenGL*, os pontos dos modelos referidos em todos os grupos a desenhar são carregados num *map*, garantindo que cada ficheiro *.3d* só é lido uma única vez.

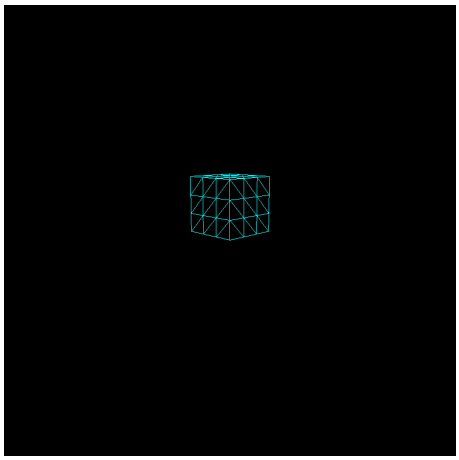
0.2.3 Extras

Adicionalmente, desenvolveu-se a primitiva *torus*. Necessita de um raio externo, de um raio interno, número de *stacks* e número de *slices*. A estratégia para desenhar a mesma passa por visualizar uma circunferência contida no plano *zOy*, de raio igual a metade da diferença dos raios de *input*, centrada na origem e com tantos lados quanto o dobro do número de *stacks*. Esta é transladada sobre o eixo *Oz* e daí é orientada, em termos de ângulo com plano *zOy*, para corresponder à *slice* a desenhar no momento.

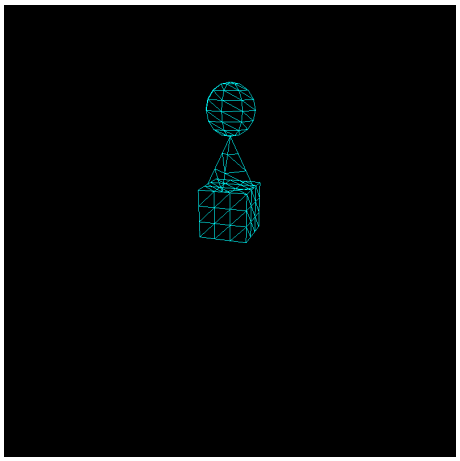
Também desenvolvemos outro modo para a câmara, a que podemos chamar *third person camera motion*, que permite ao utilizador "viajar"pelo mapa, utilizando as setas para controlar o movimento e a posição do rato para controlar a direção. Para além disso, o modo *default/explorer* da câmara foi modificado. Agora, com as setas, o utilizador pode andar na superfície de uma "esfera"centrada no ponto (0,0,0) e com as teclas *page_up* e *page_down*, o utilizador pode aumentar ou diminuir o raio da mesma, respetivamente. A tecla *P* é usada para mudar o modo da câmara.

0.3 Testes

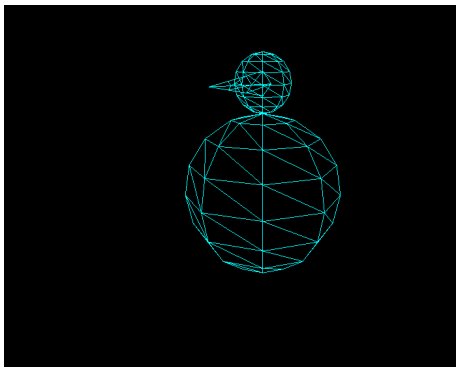
0.3.1 teste_2_1



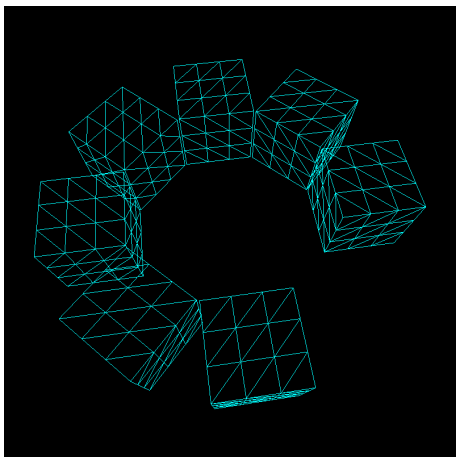
0.3.2 teste_2_2



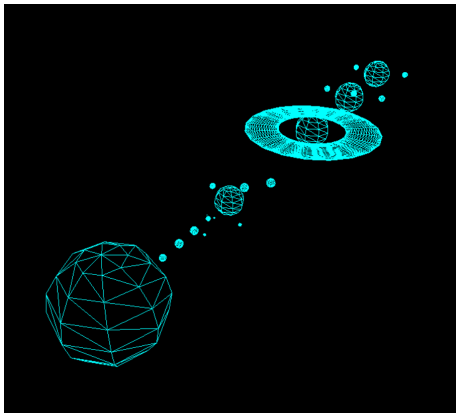
0.3.3 teste_2_3



0.3.4 teste_2_4



0.3.5 solar_system



0.4 Conclusão

Concluída esta segunda fase do trabalho prático, consideramos que atingimos os objetivos exigidos para mesma de forma satisfatória e elegante. Demos, assim, mais um passo em direção à próxima fase, com uma base para a mesma consolidada.