Universidade do Minho
Departamento de Informática

Computação Gráfica

Fase 2 - Grupo 59

5 de abril, 2024

Carlos Ferreira
a89509

Gerson Junior
a88000

Pedro Sousa
a100823

Pedro Viana
a100701

Sumário

1 Introdução

Serve o seguinte relatório para documentar o processo de desenvolvimento da segunda fase do trabalho prático da unidade curricular de Computação Gráfica, do ano letivo 2023/2024.

O objetivo desta fase é criar cenas hierárquicas usando transformações geométricas. Uma cena é definida como uma árvore onde cada nodo contém um conjunto de transformações geométricas (translação, rotação e escala) e, opcionalmente, um conjunto de modelos. Cada nodo também pode ter nodos filhos.

2 Engine

De forma a melhor representar os dados fornecidos no ficheiro de configuração XML, nesta fase foi melhorada a abstração do estado do *engine* com o objetivo de obter um tipo de dados que corresponda mais de perto com os elementos do ficheiro.

Para tal foi usado o seguinte exemplo fornecido pela equipa docente:

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="10" y="3" z="10" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="1" far="1000" />
    </camera>
    <group>
        <transform>
            <translate x="0" y="1" z="0" />
        </transform>
        <models>
            <model file="box_2_3.3d" />
        </models>
    </group>
</world>
```

Com este foi decidido que carregar um ficheiro de configuração deve gerar um tipo de dados `World` com uma `Window`, uma `Camera` e um `Group` que por sua vez tem `Transform`'s, `Model`'s e outros `Group`'s.

O tipo de dados `Model` já foi abordado no relatório da primeira fase pelo que tal não será repetido aqui.

Foi ainda decidido criar um módulo responsável pelo carregamento dos ficheiros de configuração XML denominado `XMLWorldLoader`.

2.1 World

Pelo seguinte excerto do ficheiro XML com os elementos relevantes ao elemento `world`:

```
<world>
    <window ... />
    <camera>
        ...
    </camera>
    <group>
        ...
    </group>
</world>
```

Foi decidido que este deve ser representado por uma classe `World` com 3 variáveis de instância que correspondam aos elementos `window`, `camera` e `group`.

Definição simplificada do estado da classe `World`:

```
class World {
    Window window{};
    Camera camera;
    Group group;
}
```

2.1.1 Window

Pelo seguinte exemplo do elemento `window`:

```
<window width="512" height="512" />
```

Foi decidido que este deve ser representado por uma classe "Window" com dois inteiros que correspondem à altura e largura da janela e ainda o nome do ficheiro de configuração XML como título da janela.

Definição simplificada do estado da classe `Window`:

```
class Window {
    int width, height;
    char *title;
}
```

Na implementação atual da `Window` as variáveis `width` e `height` são atualizadas sempre que a janela é redimensionada, porém, estas variáveis são apenas usadas uma vez no início do programa para definir as dimensões da janela de acordo com o ficheiro de configuração. Ou seja, isto é apenas feito para manter estes valores corretos e mais tarde tal comportamento poderá ser removido se nunca se demonstrar útil.

2.1.2 Camera

Pelo seguinte exemplo do elemento `camera`:

```
<camera>
    <position x="10" y="3" z="10" />
    <lookAt x="0" y="0" z="0" />
    <up x="0" y="1" z="0" />
    <projection fov="60" near="1" far="1000" />
</camera>
```

Foi decidido que este deve ser representado por uma classe `Camera`, com as seguintes variáveis de instância:

- Posição da camera como um `Point3`.
- Posição alvo como um `Point3`.
- Vetor cima como um `Vector3`.
- Três floats para as variáveis relevantes à projeção, `fov`, `near` and `far`.

Foram ainda adicionadas mais três variáveis do tipo float - `pitch`, `yaw` e `radius` - usadas para controlar o movimento da câmera na terceira pessoa.

Definição simplificada do estado da classe `Camera`:

```
class Camera {
    Point3 position;
    Point3 lookAt;
    Vector3 up;
    float fov{}, near{}, far{};
    float pitch{}, yaw{};
    float radius{};
}
```

A classe `Camera` oferece ainda os seguintes métodos para colocar a câmera, alterar a prespetiva e reagir a controlos do teclado respetivamente:

```
void Camera::place();

void Camera::setPerspective(int w, int h) const;

void Camera::reactKey(unsigned char key, int x, int y);
```

2.1.3 Group

Pelo seguinte exemplo do elemento `group`:

```
<group>
    <transform>
        <translate x="0" y="1" z="0" />
    </transform>
    <models>
        <model file="box_2_3.3d" />
    </models>
    <group>
        <transform>
            <translate x="0" y="1" z="0" />
        </transform>
        <models>
            <model file="cone_1_2_4_3.3d" />
        </models>
    </group>
</group>
```

Foi decido que este deve ser representado por uma *tree like structure*. Assim, o elemento `group` é definido pela classe `Group` que, por usa vez, contém um vetor de `Group`'s com os subgrupos do grupo. Esta contém ainda mais dois vetores para as transformações e modelos do grupo.

Definição simplificada do estado da classe `Group`:

```
class Group {
    std::vector<std::unique_ptr<Transform>> transforms;
    std::vector<Model> models;
    std::vector<Group> subgroups;
}
```

A classe `Group` oferece ainda o seguinte método para desenhar o grupo:

```
void Group::draw() const
{
    glPushMatrix();

    for (const auto &transform: this->transforms)
        transform->apply();

    for (const auto &model: this->models)
        model.draw();

    for (const auto &group: this->subgroups)
        group.draw();

    glPopMatrix();
}
```

Este é um método recursivo que desenha os modelos do grupo e os modelos nos seus subgrupos. O comportamento do método pode ser descrito pela seguinte sucessão de eventos:

1. faz *push* da matriz atual (`GL_MODELVIEW`).
2. aplica as transformações do grupo pela ordem dáda no ficheiro de configuração
3. desenha os modelos do grupos
4. desenha os seus subgrupos
5. faz *pop* da matriz atual (`GL_MODELVIEW`).

2.1.3.1 Transform

Pelo seguinte exemplo do elemento `transform`:

```
<transform>
    <translate x="0" y="1" z="0" />
    <rotate angle="90" x="0" y="1" z="0" />
    <scale x="2" y="2" z="2" />
</transform>
```

Foi decidido que este deve ser representado por uma classe abstrata `Transform` que é implementada pelas classes concretas `Translate`, `Rotate` e `Scale` em concordância com os três elementos filhos possíveis, `translate`, `rotate` e `scale` respetivamente.

A classe abstrata `Tranform` deve ainda especificar que as suas subclasses devem implementar um método `apply()` que aplica a correspondente tranformação à matriz atual.

Definição simplificada da classe `Transform`:

```
class Transform {
    virtual void apply() const noexcept = 0;
};
```

A classe `Translate` tem como estado um Vector3 e implementa o método `apply()` com a função `glTranslatef(GLfloat x, GLfloat y, GLfloat z)`.

A classe `Rotate` tem como estado um ângulo do tipo float e um Vector3 e implementa o método `apply()` com a função `glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)`.

A classe `Scale` tem como estado os três fatores de escala do tipo float e implementa o método `apply()` com a função `glScalef(GLfloat x, GLfloat y, GLfloat z)`.

Definição simplificada das classes `Translate`, `Rotate` e `Scale` respetivamente:

```
class Translate : public Transform {
    Vector3 vector;

    void apply() const noexcept override {
        glTranslatef(this->vector.x, this->vector.y, this->vector.z);
    }
};

class Rotate : public Transform {
    float angle;
    Vector3 vector;

    void apply() const noexcept override {
        glRotatef(this->angle, this->vector.x, this->vector.y, this->vector.z);
    }
};

class Scale : public Transform {
    float factorX, factorY, factorZ;

    void apply() const noexcept override {
        glScalef(this->factorX, this->factorY, this->factorZ);
    }
};
```

2.2 XMLWorldLoader

O módulo `XMLWorldLoader` é responsável por ler e interpretar os ficheiros de configuração XML e gerar o objeto do tipo `World` que será usado como estado do programa.

Este oferece a função `World load(char *filename)` que, dádo o nome do ficheiro de configuração XML, devolve um objeto do tipo `World`. Esta função apenas abstrai para um módulo opáco o trabalho que já era feito na primeira fase para ler o ficheiro de configuração.

Para ser possível ler e interpretar o novo elemento `group`, neste módulo foi definida ainda a função Group `parseGroup(XMLElement *groupElement)` que, dádo um elemento `group` do tipo `XMLElement`, devolve um objeto do tipo `Group`.

Esta função está definida de forma recursiva de forma a esta seja aplicada aos subgrupos de um grupo e, assim, aplicar esta função ao grupo do topo da árvore devolve um `Group` com todos os nodos no ficheiro de configuração XML. A função tem ainda o cuidado de ler as transformações pela ordem que estão escritas no ficheiro.

3 Demo

Como pedido no enunciado foi criado um demo do sistema solar (Figura 1) com o novo motor.

Este contém o Sol, os oito planetas do sistema solar, a lua da Terra, 3 luas de Júpiter e os "anéis" de Saturno. Nenhum destes foi desenhado com rigor, mas apenas de forma a que sejam reconheciveis como os corpos que tentam representar.

Todo a cena foi desenhado com o mesmo modelo, uma esfera de raio 1 com 32 *stacks* e 32 *slices*.

O Sol foi desenhado no centro do cena sem qualquer transformação.

A posição dos planetas é definida através de uma rotação seguida a uma translação a partir da posição do Sol e são depois escalados apropriadamente. O facto de que translação é feita a partir do Sol não tem impacto neste demo, mas caso fose necessário mover o Sol, através desta estratégia, os planetas também moverão conformemente.

As luas da Terra e de Júpiter tem a mesma relação que os seus planetas tem com o Sol.

Os "anéis" de Saturno são obtidos desenhando uma esfera na mesma posição de Saturno com maior dimensão nos eixos x e z, mas com uma dimensão muito menor no eixo do y obtendo assim um disco com raio superior a Saturno. Este é ainda rodado de forma a ser mais visível para a câmera.

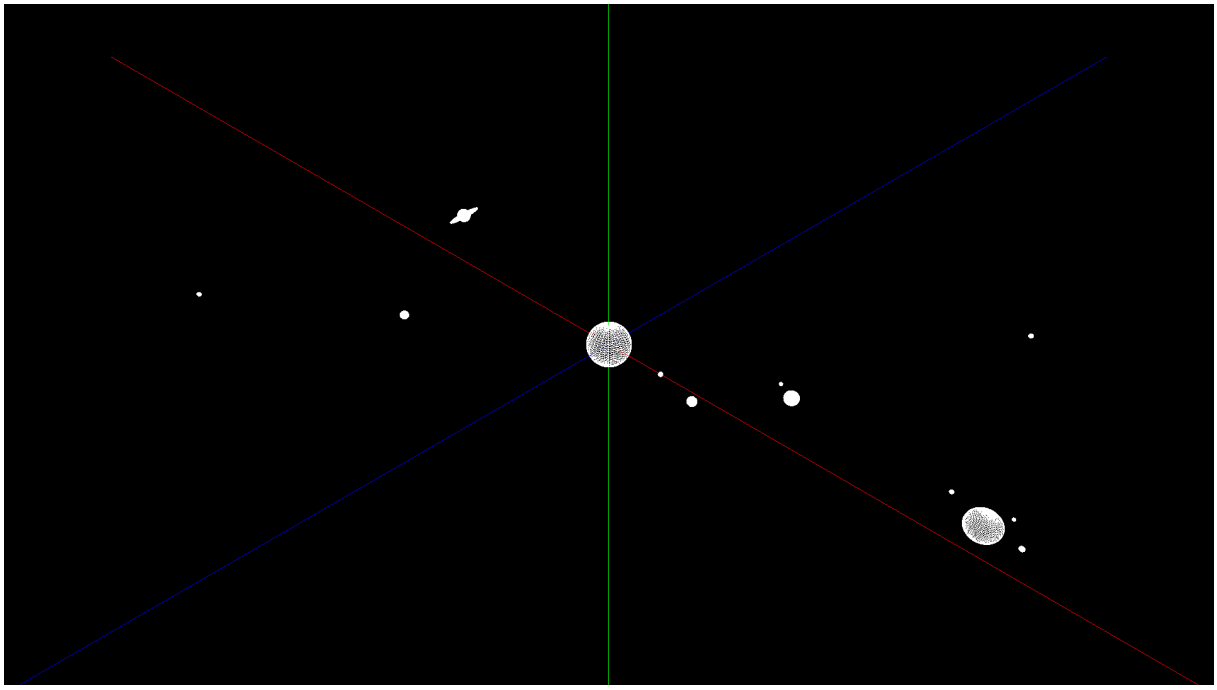

Figura 1: Sistema solar

4 Conclusão

Nesta fase do projeto foi possível identificar e desenvolver as estratégia necessárias para carregar a nova iteração dos ficheiros de configuração XML, assim como desenhar vários modelos descritos numa estrutura em forma de árvore pela qual lhes são aplicadas transformações.

Considera-se que nesta fase foi possível consolidar o uso de transformações fornecidas pelo OpenGL como o `glTranslatef`, `glRotatef` e `glScalef` assim como o uso do *push* e *pop* de matrizes de modo a compor estas transformações da forma desejada.

Assim, o grupo considera ser agora capaz de criar cenas mais complexas.

Foi ainda possível melhorar a abstração do código de forma a obter soluções mais familiares ao contexto do projeto.