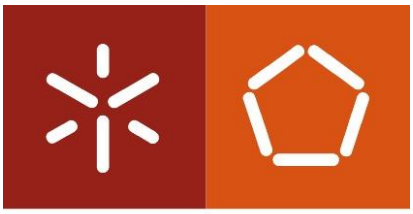


Universidade do Minho

Licenciatura em Engenharia Informática

Computação Gráfica - 4ª Fase

Grupo 14

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

Ano Lectivo 2021/2022



0.1 Introdução

A quarta e última fase tem em vista a implementação de funcionalidades de iluminação e de aplicação de texturas aos modelos. Para tal, o *generator* deve gerar, para cada vértice, um ponto 3D - correspondente ao vetor normal ao mesmo - e um ponto 2D, que será usado de forma a mapear os pontos de uma imagem sobre as faces da primitiva respetiva. No que toca ao *engine*, o mesmo deve ser capaz de ler a nova informação gerada e aplicá-la adequadamente. Os ficheiros XML usados no mesmo também são estendidos, podendo ser especificados três tipos de luzes, bem como cor e um imagem a servir como textura para os modelos.

0.2 Decisões e abordagens escolhidas

0.2.1 Estruturas de Dados

Cor

Para suportar a opção de especificar cores para os modelos, foi implementada uma nova estrutura que agrega a informação relativa às várias componentes que compõem a cor de um dado material.

```
struct Color {

    RGB diffuse;
    RGB ambient;
    RGB specular;
    RGB emissive;
    unsigned shininess;

    Color();
    Color(const RGB& diffuse, const RGB& ambient,
          const RGB& specular, const RGB& emissive,
          unsigned shininess);
};
```

Figura 1: Estrutura *Color*

```
class RGB {

private:
    std::array<uint8_t, 3> arr;

public:
    constexpr uint8_t operator[](size_t i) const {
        return this->arr[i];
    }

    RGB(uint8_t r, uint8_t g, uint8_t b);
    std::array<float, 4> as_float_array() const;
};
```

Figura 2: Estrutura *RGB*

A estrutura *Color* contém, então, 4 instâncias da estrutura *RGB*. Esta última não é mais que um *array* de 3 inteiros de um *byte* (nota para o uso do tipo *uint8_t*). Desta forma garante-se que não há valores que estão fora do intervalo 0-255, que são os valores entre os quais as 3 componentes da luz (isto é, vermelho, verde e azul) devem variar. De salientar, ainda, a função *RGB::as_float_array()*. Esta retorna um *array* de *floats*, uma vez que o *OpenGL* necessita de valores reais entre 0 e 1 para a especificação das cores. O último destes valores é o canal alfa, que é assumido como sendo 1.

Posto isto, cada modelo inclui agora a sua cor e, opcionalmente, o caminho da imagem a usar como textura, como podemos ver na figura 0.3.6.

```
struct Model {

    std::string model_filename;
    std::optional<std::string> texture_filename;
    Color color;

    Model(std::string&& model_fn, const Color& color);
    Model(std::string&& model_fn, std::string&& texture_fn, const Color& color);
};
```

Figura 3: Estrutura *Model*

Nota para o facto de apenas o ficheiro de textura ser opcional. Para a cor assumem-se os valores especificados no enunciado, se esta não for especificada no XML.

Luz

No que toca à iluminação, a aplicação deve suportar três tipos de luzes. Como tal, justifica-se o uso de uma classe abstrata, ilustrada na figura 4. Cada implementação da mesma deverá consequentemente implementar o método que indica o seu tipo, bem como as variáveis de instância necessárias para a mesma.

```
enum LightType {
    point,
    directional,
    spotlight,
};

class Light {

public:
    Light();
    virtual LightType get_type() const = 0;
};
```

Figura 4: Estrutura *Light*

Texturas

Para as texturas, surge uma classe em tudo semelhante à classe *VBO*.

```
class TexturesHandler {

private:
    static std::shared_ptr<TexturesHandler> singleton;

    std::vector<unsigned> images;
    std::map<std::string, unsigned> image_info;


    TexturesHandler(const std::set<std::string> &textures_fns);

public:
    static void init(const std::set<std::string> &textures_fns);
    static std::shared_ptr<TexturesHandler> get_instance();

    bool bind(const std::string& texture_fn) const;
    void clear() const;
};
```

Figura 5: Estrutura *TexturesHandler*

A cada nome de ficheiro associa-se o índice correspondente à imagem, carregada em memória do GPU. A classe em questão disponibiliza ainda os métodos necessárias para se ativar a textura necessária aquando do momento de desenho do modelo.

Pontos

Uma vez que as texturas são especificadas usando duas coordenadas, desenvolveram-se estruturas adequadas para suportar este requisito, podendo as mesmas ser especificadas através de coordenadas cartesianas ou polares, de forma análoga ao que se havia implementada para os pontos 3D.

0.2.2 Cálculo das normais e coordenadas de textura

O *generator* deve agora gerar normais para cada vértice, para além de coordenadas de textura. Para tal, optamos por usar mais duas extensões de ficheiros (*.norm* e *.text*, respetivamente).

Para o plano e para caixa, as normais são iguais para cada vértice da mesma face, de modo que é trivial o cálculo das mesmas. Na esfera, basta normalizar cada vértice e esse assim obtemos o vetor correspondente à normal. No cone, devemos ter em consideração o declive para obter a normal, declive esse que pode ser obtido como o arco tangente da divisão da altura pelo raio. Já para os *patches* de Bézier, calculam-se dois vetores tangentes ao vértice (equações 1 e 2) e através do produto externo dos mesmos obtém-se uma aproximação à normal para aquele ponto. Isto é possível porque cada ponto P é calculado em função de u e de v:

$$t_1 = \frac{\partial P(u,v)}{\partial u} \tag{1}$$

$$t_2 = \frac{\partial P(u,v)}{\partial v} \tag{2}$$

Por último, para o *torus*, o processo é semelhante ao da esfera, ou seja, devemos tomar em consideração os ângulos do vértice para obter a normal.

Para as texturas, os casos do plano e da caixa são, mais uma vez, triviais por se tratarem de faces planas. No entanto, convém referir que no caso do plano, pretende-se que se repita a textura, em vez de a esticar sobre o modelo. Para a esfera, cone e *torus*, os valores de s e de t (x e y em coordenadas de textura) correspondem à *slice* e à *stack*, respetivamente. Já nos *patches* de Bézier, uma vez que cada ponto é função de u e v, ambos valores entre 0 e 1, os mesmos são usados diretamente como valores de s e de t, respetivamente.

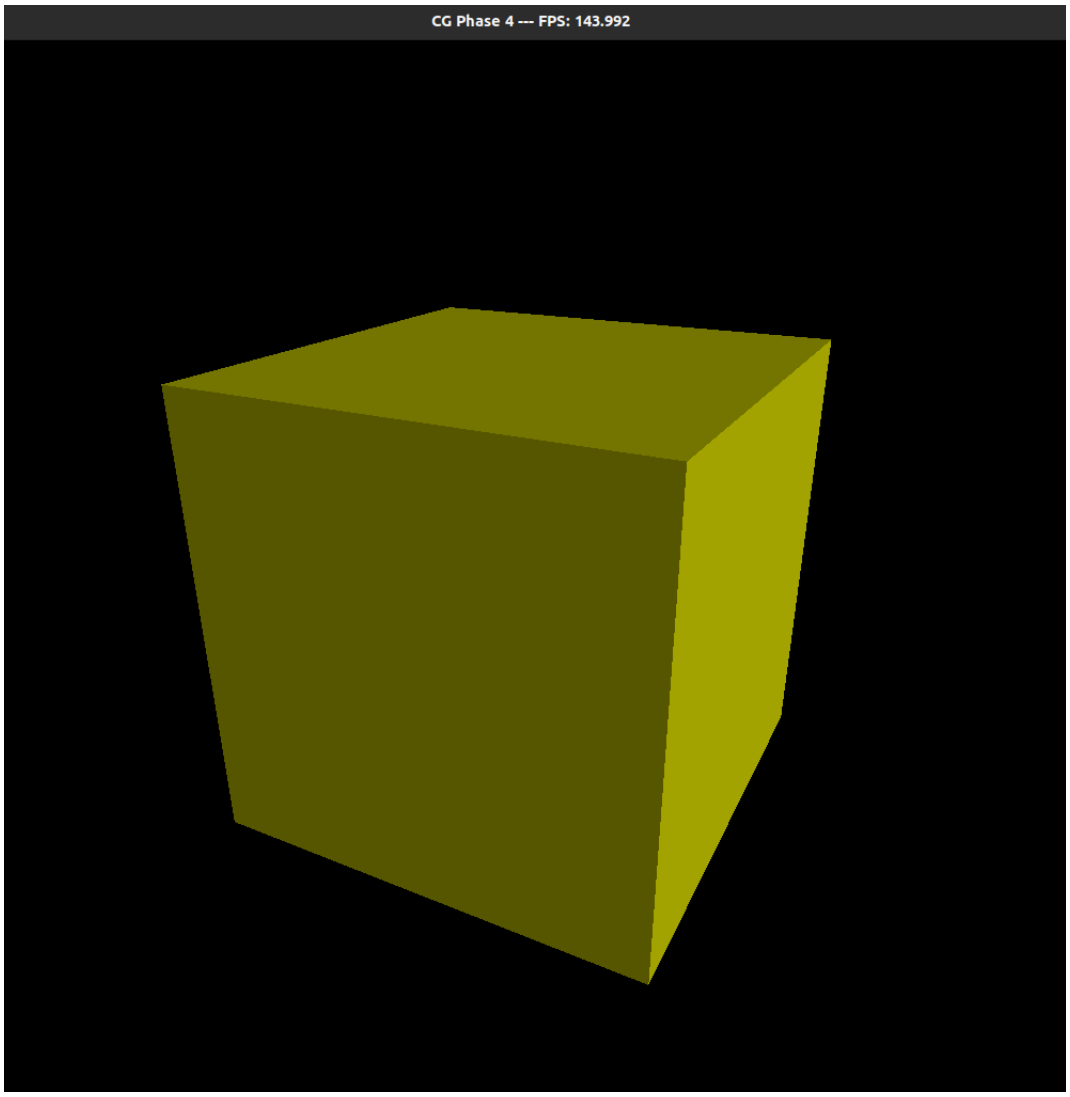
0.2.3 Correções e otimizações

A única alteração substancial foi na câmara. Esta apresentava algumas inconsistências, que foram corrigidas. Tanto o modo fixo como o modo primeira pessoa encontram-se completamente funcionais, sendo usado as teclas W, A, S, D (ou as setas) para mover e o rato para direcionar.

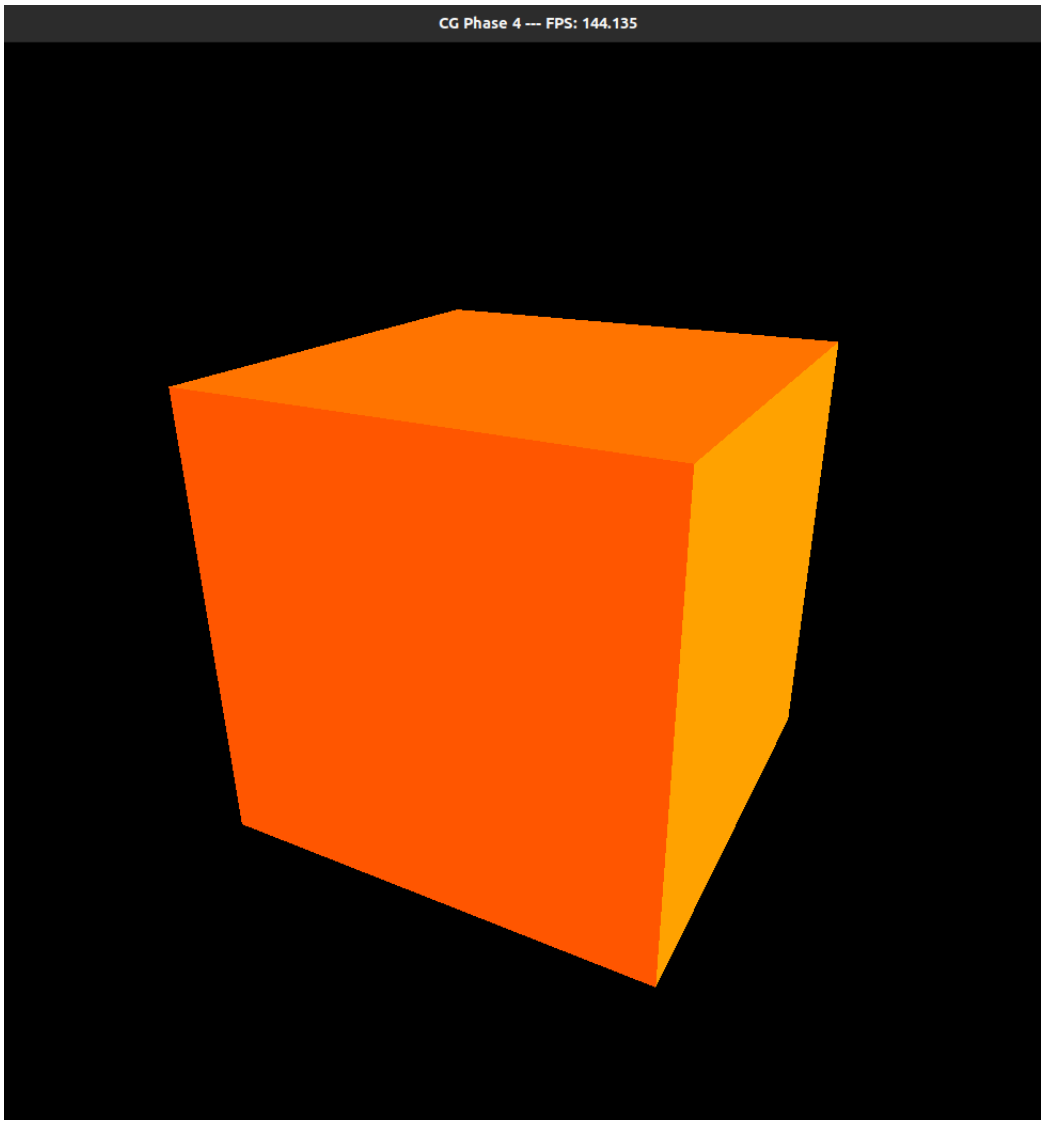
0.3 Testes

Nesta secção mostramos o *rendering* dos testes fornecidos pelos professores e a demonstração do sistema solar com texturas feito pelo grupo.

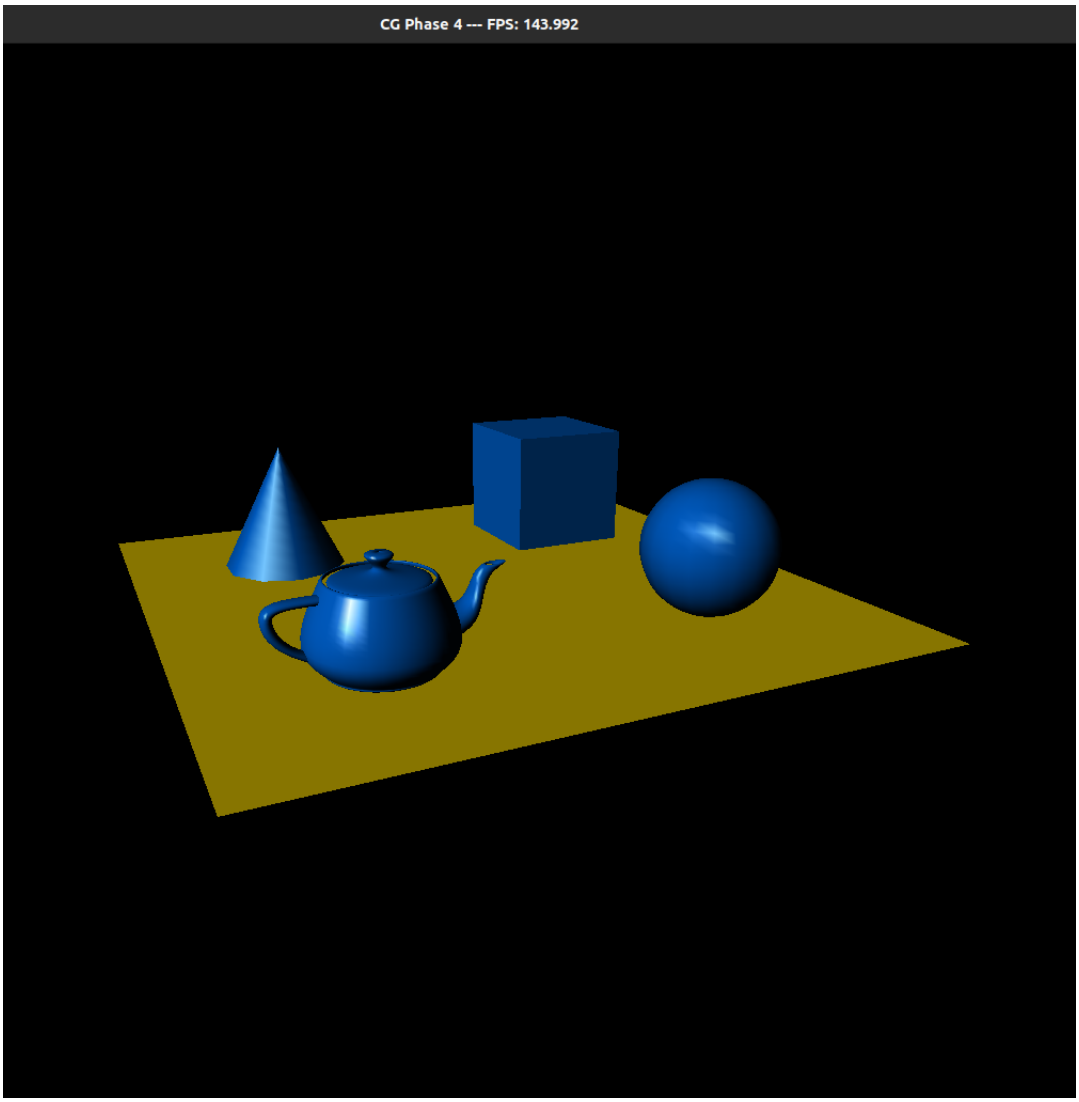
0.3.1 test_4_1



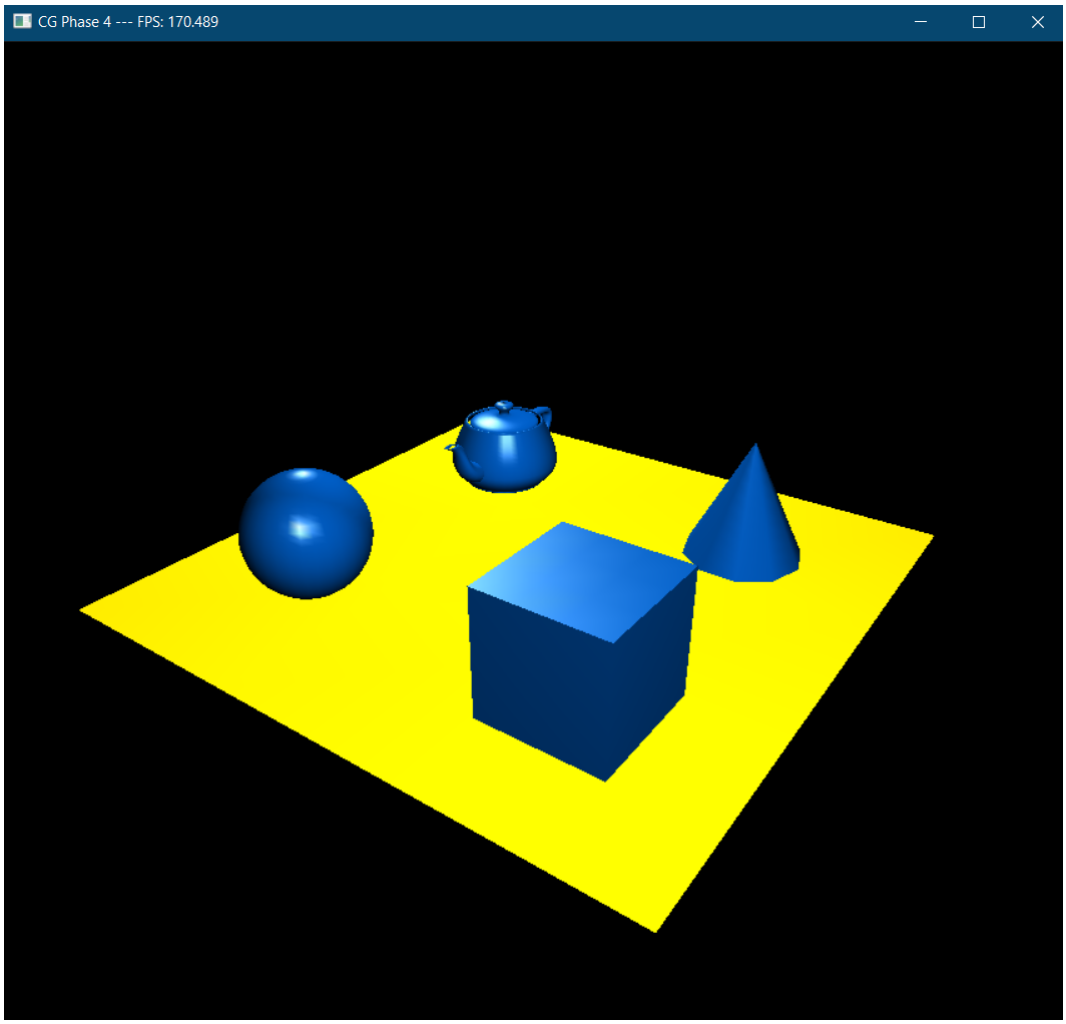
0.3.2 test_4_2



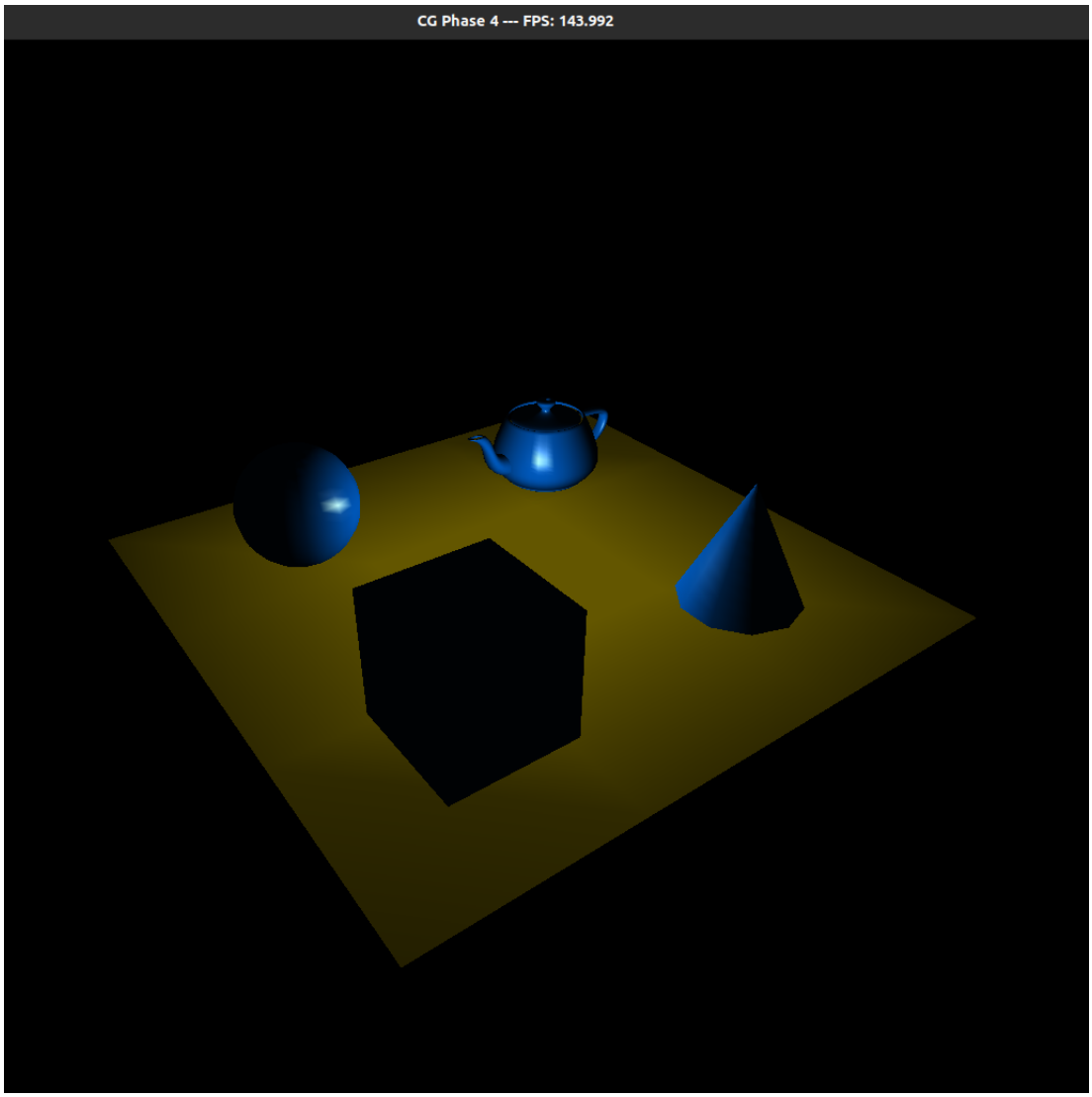
0.3.3 test_4_3



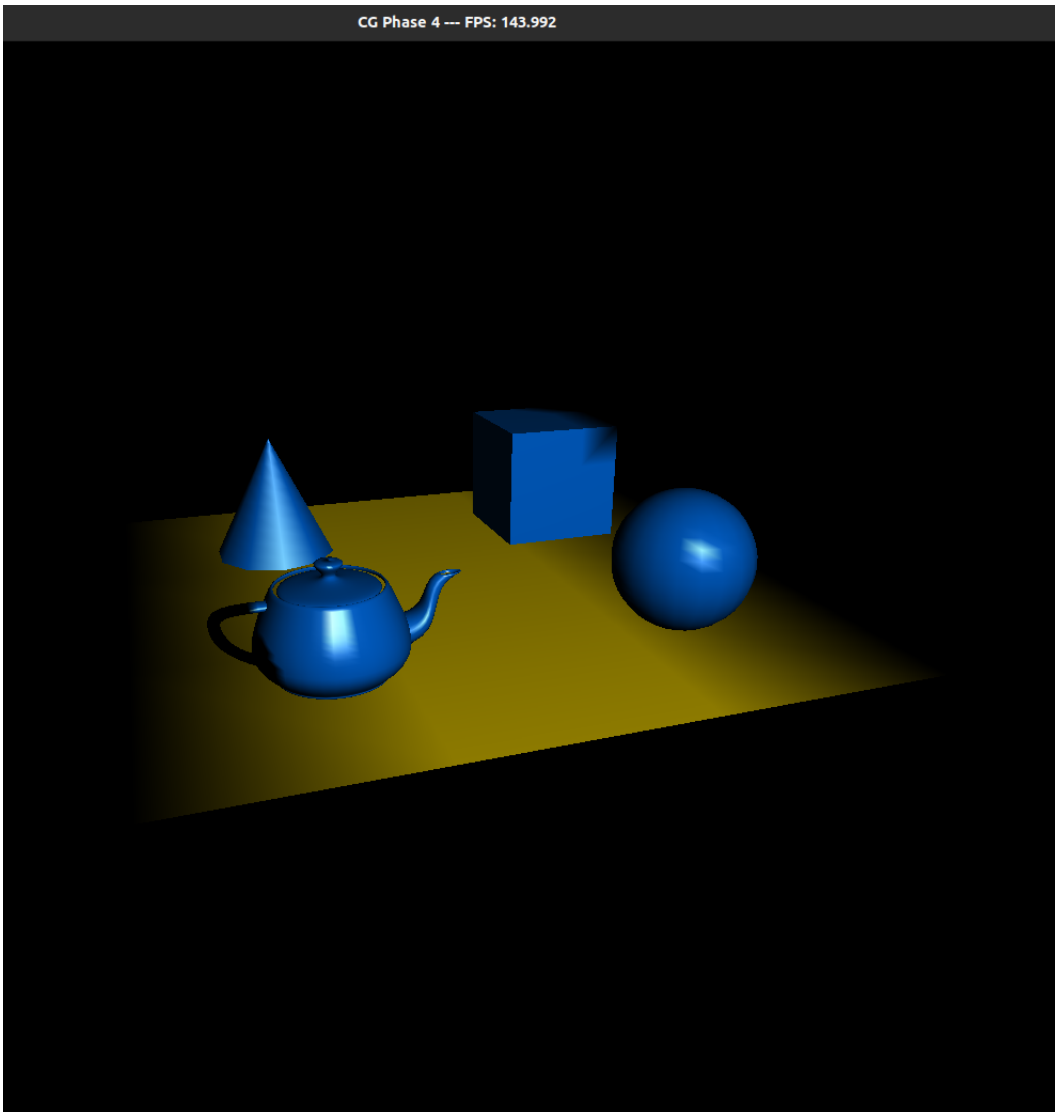
0.3.4 test_4_4



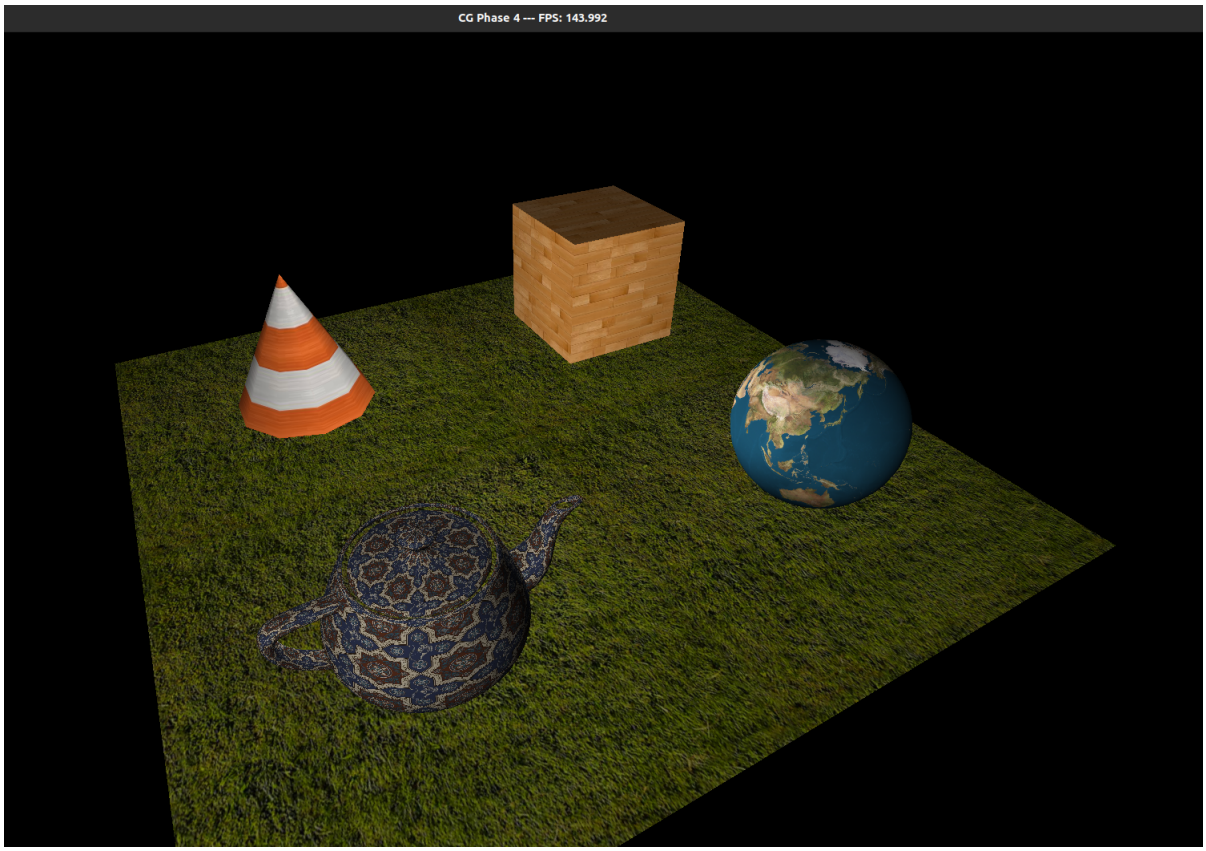
0.3.5 test_4_5



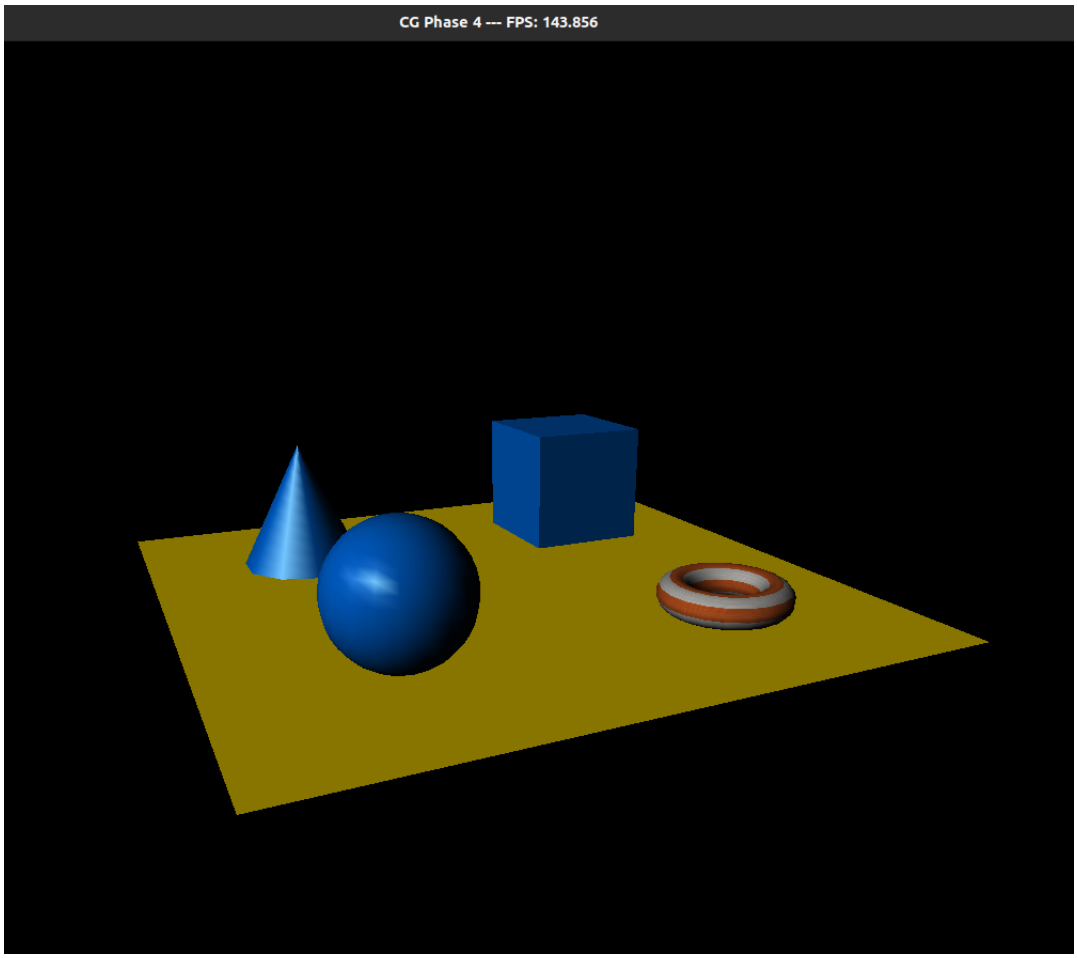
0.3.6 test_4_6



0.3.7 test_4_7



0.3.8 test_4_8



0.3.9 solar_system_textures

Como existe movimento, captamos dois momentos para o demonstrar.

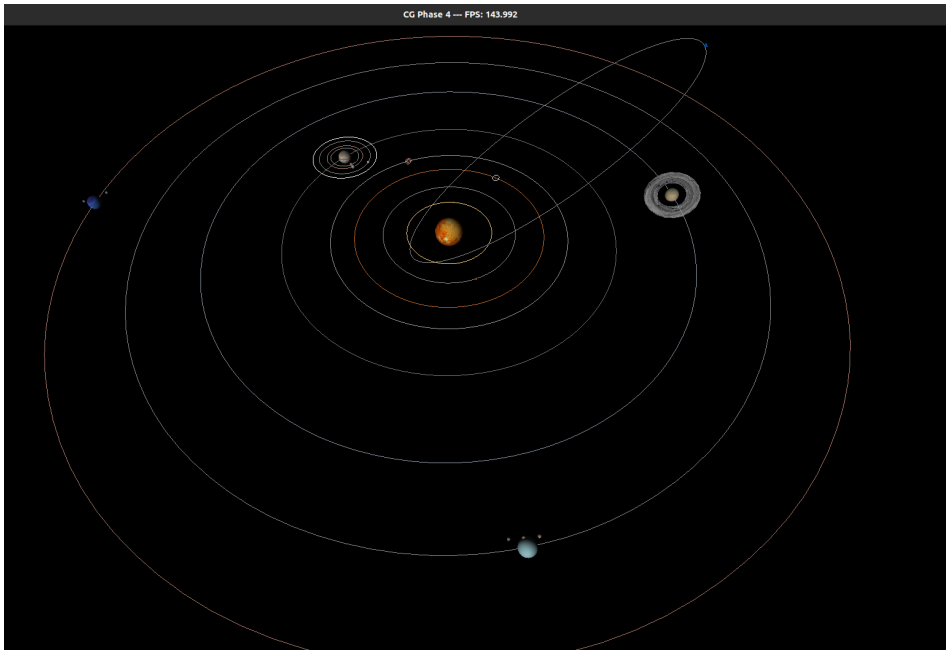


Figura 6: 1º Momento

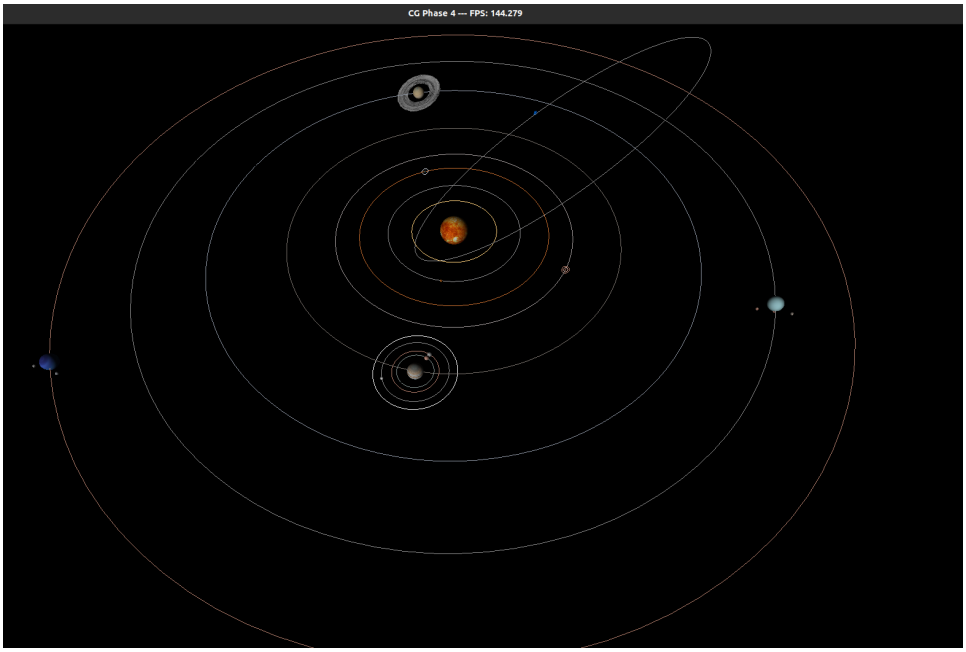


Figura 7: 2º Momento

0.4 Conclusão

Terminada esta fase, dá-se assim por concluído o projeto de Computação Gráfica. Em retrospetiva, consideramos que em todas as fases cumprimos na íntegra aquilo que era proposto para cada uma delas. Como trabalho futuro, seria interessante desenvolver outros extras, como os mencionados no enunciado, nomeadamente *view frustum culling*, e eventualmente apresentar mais *demos*, de forma a provar a capacidade das aplicações desenvolvidas.