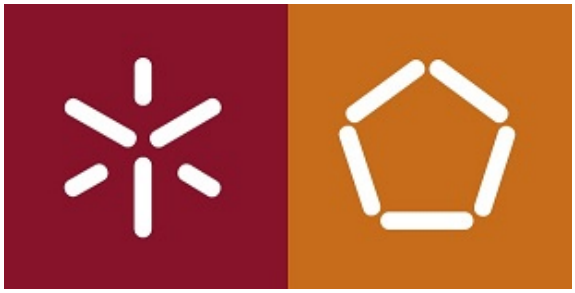
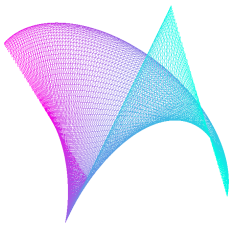


Universidade do Minho
Departamento de Informática

Solaris



Computação Gráfica
Grupo 24 - Fase 2

4 de abril, 2022


Beatriz Rodrigues
(*a93230*)


Francisco Neves
(*a93202*)


Guilherme Fernandes
(*a93216*)


João Carvalho
(*a93166*)

Índice

1. Introdução

O programa *Solaris* foi atualizado de forma a permitir novas funcionalidades. Desta forma, o *engine* foi modificado de forma a conseguir interpretar *scenes* que incluam transformações geométricas definidas em hierarquia.

Para além disso, foi desenvolvida uma *scene* relativa a uma representação do Sistema Solar. Esta pode ser obtida com o auxílio de um *script* escrito em *Python*. A representação mencionada inclui o Sol, os planetas e os satélites naturais correspondentes.

2. Engine

2.1 Transformações

Uma vez que a ordem das transformações é relevante, estas serão colecionadas num vetor pela ordem em que aparecem no `XML`. Como tal, `Transform` é uma classe abstrata que disponibliza o método `apply` que aplica a transformação.

Esta tem 3 implementações, uma para cada tipo de transformação referido nesta fase.

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

class Translation: public Transform {
  private:
    Point _coords;

  public:
    Translation() : _coords(Point::cartesian(0, 0, 0)) {}
    Translation(Point coords) : _coords(coords) {}
    void apply() const noexcept;
};

class Rotation: public Transform {
  private:
    float _angle;
    Point _coords;

  public:
    Rotation() : _angle(0), _coords(Point::cartesian(0, 0, 0)) {}
    Rotation(float angle, Point coords) : _angle(angle), _coords(coords) {}
    void apply() const noexcept;
};

class Scale: public Transform {
  private:
    Point _coords;

  public:
    Scale() : _coords(Point::cartesian(1, 1, 1)) {}
```

```
    Scale(Point coords) : _coords(coords) {}
    void apply() const noexcept;
};
```

As funções responsáveis por aplicar as transformações apenas chamam as funções relativas do OpenGL e são as seguintes:

```
void Translation::apply() const noexcept {
    glTranslatef(_coords.x(), _coords.y(), _coords.z());
}

void Rotation::apply() const noexcept {
    glRotatef(_angle, _coords.x(), _coords.y(), _coords.z());
}

void Scale::apply() const noexcept {
    glScalef(_coords.x(), _coords.y(), _coords.z());
}
```

Para garantir que as transformações são aplicadas aos subgrupos mas não afetam grupos isolados, a função de desenho de um grupo envolve ações de *push* e *pop* da matriz.

```
auto Group::draw() const noexcept -> void {
    glPushMatrix();
    for (auto&& transform : _transforms) {
        transform->apply();
    }
    for (auto&& model : _models) {
        model.draw();
    }
    for (auto&& group : _subgroups) {
        group.draw();
    }
    glPopMatrix();
}
```

2.2 XML parsing

Visto que o *parser* já estava desenvolvido tendo em vista a hierarquia do XML, para implementar a possibilidade da interpretação de transformações geométricas bastou acrescentar à função responsável pelo *parsing* de um grupo, (`parse_group`), a seguinte porção de código:

```
auto parse_group(
    XMLElement const* const node,
    std::unordered_map<
        std::string, std::shared_ptr<std::vector<Point>>>* points_map
) noexcept -> cpp::result<Group, ParseError> {
    ...
```

```
    auto transforms = std::vector<std::unique_ptr<Transform>>();
    auto transforms_elem = node->FirstChildElement("transform");
    if (transforms_elem != nullptr) {
        for (auto transform_elem = transforms_elem->FirstChildElement();
             transform_elem;
             transform_elem = transform_elem->NextSiblingElement()
        ) {
            auto transform = parse_transform(transform_elem);
            CHECK_RESULT(transform);
            transforms.push_back(std::move(*transform));
        }
    }
    ...
}
```

Esta é dependente de uma nova função designada por `parse_transform` que instancia a transformação correspondente.

```
auto parse_transform(XMLElement const* const node) noexcept
    -> cpp::result<std::shared_ptr<Transform>, ParseError>
{
    auto transform_type = std::string_view(node->Value());

    if (transform_type == "translate") {
        auto coords = parse_point(node);
        CHECK_RESULT(coords);

        return std::make_unique<Translation>(coords.value());

    } else if (transform_type == "rotate") {
        float angle;
        if (node->QueryFloatAttribute("angle", &angle) != XML_SUCCESS) {
            return cpp::fail(ParseError::MALFORMED_ROTATION);
        }

        auto coords = parse_point(node);
        CHECK_RESULT(coords);

        return std::make_unique<Rotation>(angle, *coords);

    } else if (transform_type == "scale") {
        auto coords = parse_point(node);
        CHECK_RESULT(coords);

        return  std::make_unique<Scale>(*coords);
    }

    return cpp::fail(ParseError::UNKNOWN_TRANFORMATION);
}
```

3. Sistema Solar

3.1 *Script*

Com o auxílio de algumas fontes (incluídas nas referências) acerca das dimensões do Sol, dos planetas e dos seus satélites naturais correspondentes, para além da distâncias entre eles, foi produzida uma *script* que permite gerar um ficheiro *XML* cuja interpretação feita pelo *engine* resulta no Sistema Solar.

Desta forma, para conferir um toque da realidade do Sistema Solar, foi decidido que as dimensões dos astros e as distâncias entre eles deveriam sofrer uma escala, ou não seria possível que este fosse visualizado.

Assim, com recurso a algumas *libraries* de *Python*, foi possível a identação típica de um ficheiro *XML*, a leitura dos dados dos astros presentes em ficheiros *csv* e a aleatoriedade na colocação dos asteróides presentes na cintura de asteróides.

Tendo tudo isto em conta, foi gerado um ficheiro *XML* que contém todos os planetas e satélites naturais de cada um deles, bem como 500 asteróides, de forma a representar a cintura de asteróides, e o anel de Saturno.

Por fim, de forma a otimizar a geração do sistema solar, para astros de menores dimensões como os asteróides da cintura de asteróides ou os satélites naturais dos planetas, é utilizado um ficheiro *.3d* com uma esfera que consideramos ser de baixa resolução, visto que possui um número de *slices* e *stacks* diminuído. Por outro lado, de forma a trazer fiabilidade à *scene* gerada, para o desenho dos diversos planetas e do Sol é utilizado um ficheiro *.3d* de uma esfera com um maior número de *stacks* e *slices* para que, esta se pareça o máximo possível com uma esfera.

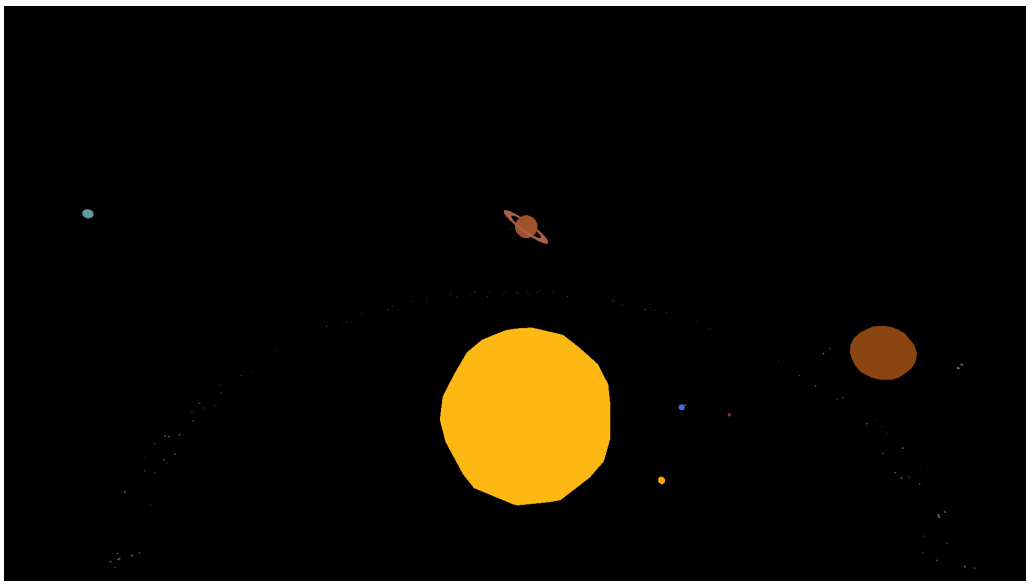


Figura 3.1: Sistema Solar - Baixa Resolução

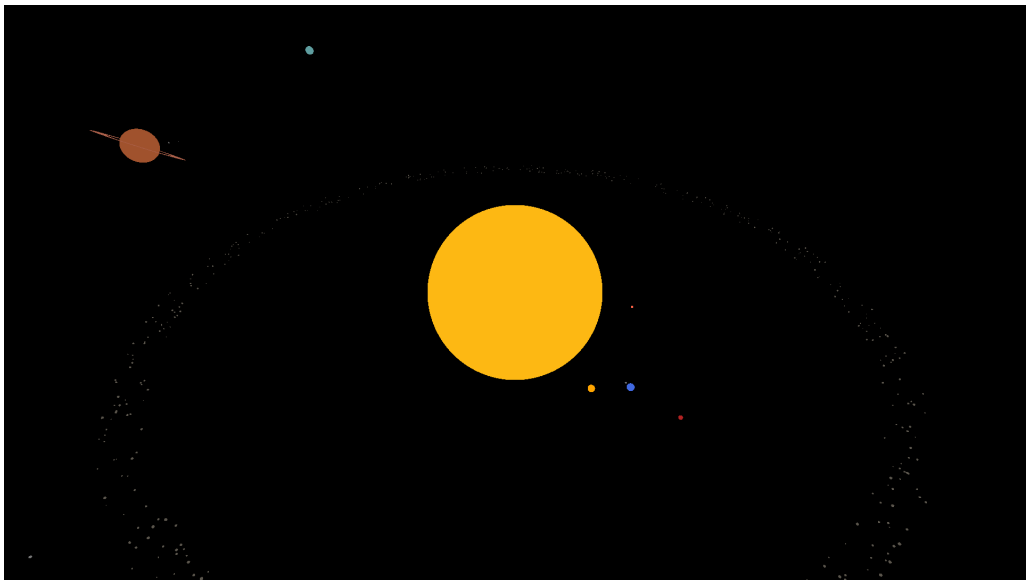


Figura 3.2: Sistema Solar - Elevada Resolução

3.2 Aplicação de Cores

Para ser obtida uma representação fiável e mais percetível do Sistema Solar foram utilizadas cores para cada astro, sendo que, as cores dos satélites naturais e dos asteróides são genéricas, enquanto que as cores dos planetas e do anel de Saturno são as suas cores reais.

Para isto, foram utilizadas cores com expressões hexadecimais nos ficheiros *.csv* dos planetas e as genéricas declaradas diretamente na *script* geradora do Sistema Solar. Posteriormente, recorrendo à *library* `Pillow` do *Python* as expressões hexadecimais são convertidas em expressões *RGB*.

Este ficheiro *.csv* foi editado de forma a conter apenas a informação relevante para o *script*.

Por fim, de forma a ser permitida a existência de campos *model* sem qualquer cor atribuída, é utilizada a cor branca para representar esses sólidos.

```
<group name="Sun">
    <models>
        <model file="models/sphere.3d">
            <color r="253" g="184" b="19" />
        </model>
    </models>
    ...
</group>
```

Relativamente ao *parsing* das cores num ficheiro semelhante ao inserido acima, foi acrescentado ao `parse_model` a seguinte porção de código que permite obter o código RGB de cada modelo. Este deve ser dividido por 255 de forma a obter o seu valor em `float`, normalizado entre 0 e 1, conforme é requerido pela função do *OpenGL* utilizada.

```
auto parse_model(
    XMLElement const* const node,
    std::unordered_map<
        std::string, std::shared_ptr<std::vector<Point>>>* points_map
) noexcept -> cpp::result<Model, ParseError> {
    ...
    auto color_elem = node->FirstChildElement("color");
    auto color = Color{1, 1, 1};
    if (color_elem != nullptr) {
        float r, g, b;
        if (color_elem->QueryFloatAttribute("r", &r) == XML_SUCCESS
            && color_elem->QueryFloatAttribute("g", &g) == XML_SUCCESS
            && color_elem->QueryFloatAttribute("b", &b) == XML_SUCCESS
        ) {
            color = Color{r / 255, g / 255, b / 255};
        } else {
            return cpp::fail(ParseError::MALFORMED_COLOR);
        }
    }
    ...
}
```

Por fim, as cores são aplicadas ao desenhar o modelo na seguinte função:

```
auto Model::draw() const noexcept -> void {
    glBegin(GL_TRIANGLES);
    glColor3f(_color.r, _color.g, _color.b);
    for (auto&& p : (*_points)) {
        glVertex3f(p.x(), p.y(), p.z());
    }
    glEnd();
}
```

4. Otimizações

Uma vez que o sistema solar contém centenas de astros, surgiu a necessidade de fazer algumas optimizações no *engine*.

O maior *bottleneck* encontrado estava presente na leitura dos ficheiros modelo, pois o mesmo ficheiro estava a ser lido centenas de vezes. Isto foi facilmente corrigido com a introdução de um `unordered_map` onde são colocados vetores de pontos relativos a cada modelo após a leitura do ficheiro relativo. Assim, se um modelo já tiver sido lido é utilizada apenas uma referência ao vetor de pontos lido anteriormente.

```
auto parse(std::string_view file_path) noexcept
    -> cpp::result<World, ParseError>
{
    ...
    auto points_map = new
        std::unordered_map<std::string, std::shared_ptr<std::vector<Point>>>();
    auto group = parse_group(group_element, points_map);
    delete points_map;
    ...
}

auto parse_group(
    XMLElement const* const node,
    std::unordered_map<
        std::string, std::shared_ptr<std::vector<Point>>>* points_map
) noexcept -> cpp::result<Group, ParseError> {
    auto models = std::vector<Model>();
    auto const models_elem = node->FirstChildElement("models");
    if (models_elem != nullptr) {
        for (auto model_elem = models_elem->FirstChildElement("model");
            model_elem;
            model_elem = model_elem->NextSiblingElement("model")
        ) {
            auto model = parse_model(model_elem, points_map);
            CHECK_RESULT(model);
            models.push_back(*model);
        }
    }
    ...
}
```

```
auto parse_model(
    XMLElement const* const node,
    std::unordered_map<
        std::string, std::shared_ptr<std::vector<Point>>>* points_map
) noexcept -> cpp::result<Model, ParseError> {
    ...
    auto stored_points = (*points_map)[file_path];
    if (!stored_points) {
        std::ifstream file(file_path);
        if (!file.is_open()) {
            return cpp::fail(ParseError::PRIMITIVE_FILE_NOT_FOUND);
        }
        float x, y, z;
        auto points = std::vector<Point>();
        while (file >> x >> y >> z) {
            points.push_back(Point::cartesian(x, y, z));
        }
        stored_points = std::make_shared<std::vector<Point>>(std::move(points));
    }
    ...
}
```

Além de poupar imenso tempo no arranque do programa, resulta na utilização de uma quantidade muito mais reduzida de memória, uma vez que o vetor de pontos existe apenas uma vez para o qual os modelos possuem referências.

5. Testes Efetuados

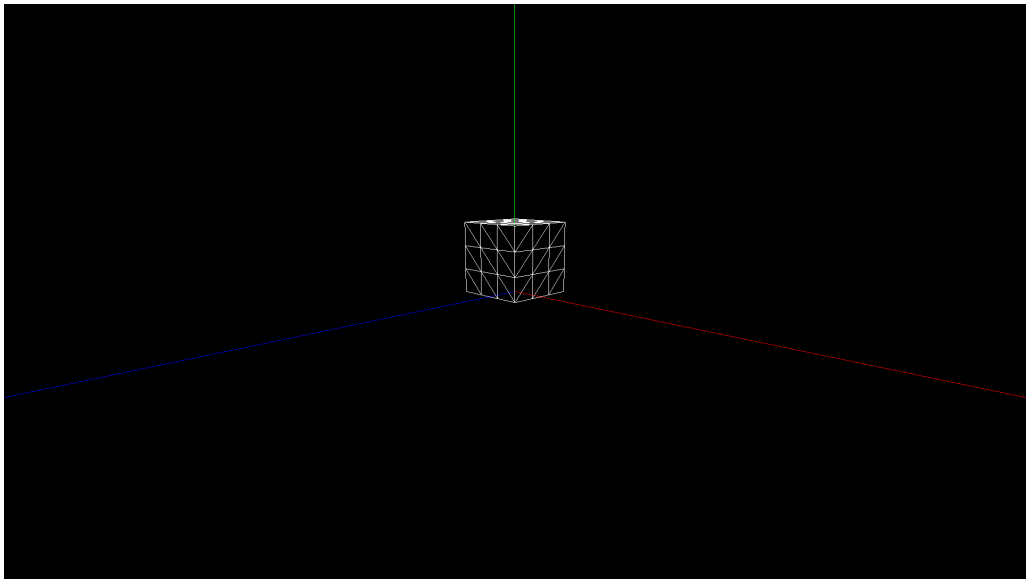


Figura 5.1: Teste 1

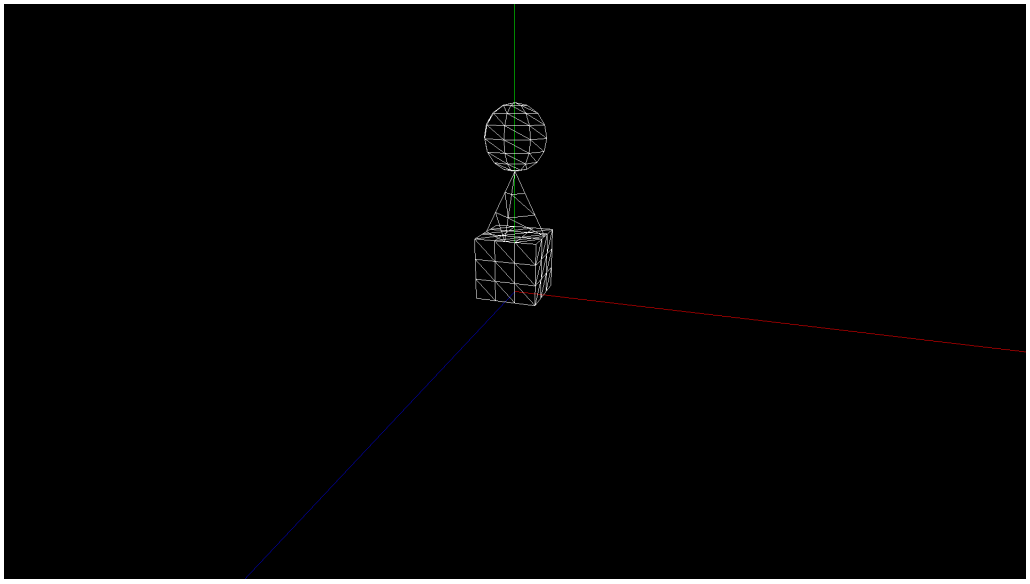


Figura 5.2: Teste 2

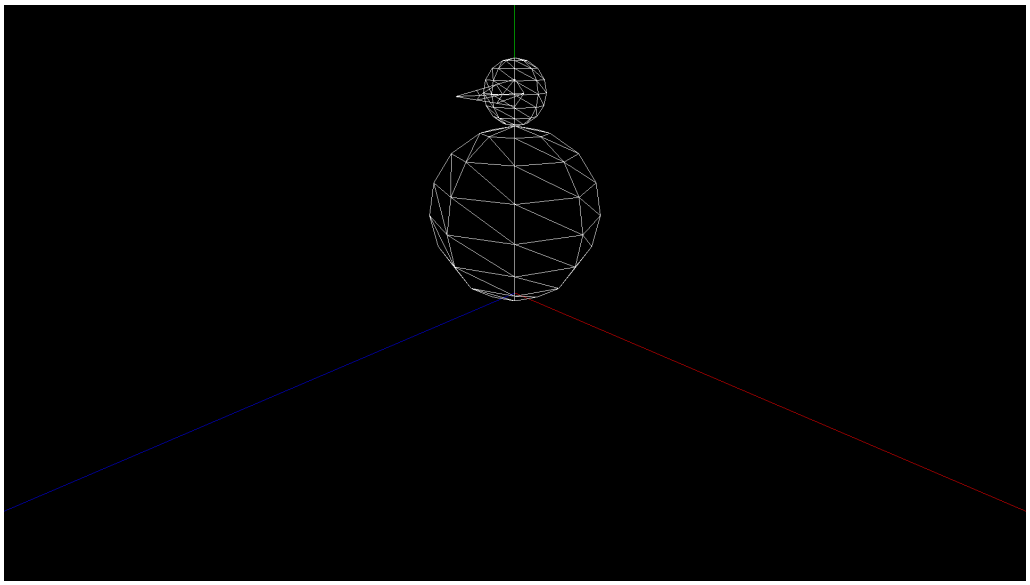


Figura 5.3: Teste 3

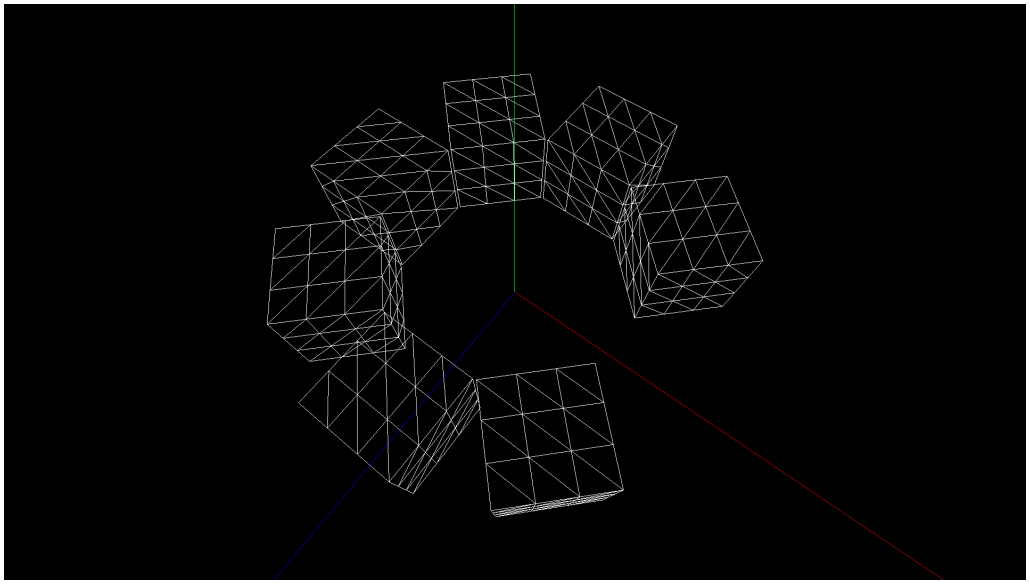


Figura 5.4: Teste 4

6. Conclusões e Trabalho Futuro

Considera-se que foi efetuado um bom trabalho e que o programa implementa uma boa variedade de funcionalidades. No entanto, no futuro, será melhorado relativamente ao seu aspeto estético, utilizando texturas e iluminação.

Para além disso pretendemos trocar o vetor de pontos dos modelos por *buffers* armazenados na placa gráfica (*VBOs*).

7. Referências

1. **Dados sobre planetas e luas (NASA)** . Consultado pela última vez a 4 de abril de 2022.

2. **Distância relativa entre planetas (National Geographic)** . Consultado pela última vez a 4 de abril de 2022.