Universidade do Minho
Escola de Engenharia

Licenciatura em Engenharia Informática 2024/25

Computação Gráfica:
Relatório de Projeto (2ª Fase)

GRUPO 8

Gustavo Barros	Pedro Ferreira	Enzo Vieira	Luís Figueiredo
A100656	A97646	A98352	A100549

1. Generator

1.1. Primitiva “Tubo” (Cilindro Oco)

Esta primitiva é a representação de um tubo (cilindro oco), caracterizado por 2 raios (interno e externo), uma altura e o número de slices. Esta primitiva vai ser usada para complementar Saturno no sistema solar com o seu anel.

A geração desta geometria é realizada através de uma função que constroi um grupo de vértices, organizados em triângulos, que definem as superfícies do tubo: as faces superior e inferior e as superfícies lateral externa e interna.

As faces superior e inferior são feitas como aneis planos, que se localizam nas extremidades da altura do cilindro (y = height/2 e y=-height/2). Cada face é composta por um número de slices (triângulos), que estabelecem a segmentação angular em intervalos de 2pi/slices. As coordenadas dos vértices são calculadas com o uso das funções trigonométricas (cosf e sinf).

As superfícies laterais são construídas verticalmente, e estas conectam as bordas das faces superior e inferior. A superficie lateral externa é definida pelos vértices do raio externo (oradius) e a interna pelo raio interno (iradius). Cada faixa é dividida em segmentos de slices, que formam uma série de triângulos. A orientação dos triângulos é definida respeitando a convenção anti-horária.

A função de implementação utiliza um vetor dinâmico que armazena as coordenadas dos vértices, a altura é centrada na origem com o valor de halfHeight(height * 0.5f).

2. Engine

2.1. Leitura Estruturada de Configurações XML

A leitura do ficheiro XML de configuração leva despoleta a inicialização dos seus dados nas estruturas definidas no ficheiro *src/include/engine/Config.h*:

```
using Transform = std::variant<
    Translation,
    Rotation,
    Scaling
>;

struct Window {
    int width = 512;
    int height = 512;
};

struct Camera {
    std::array<double, 3> position = { 0, 0, 0 };
    std::array<double, 3> lookAt = { 0, 0, 0 };
    std::array<double, 3> up = { 0, 1, 0 };

    double fov = 60.0;
    double nearClip = 1.0;
    double farClip = 1000.0;
};


struct Group {
    glm::vec3 colour = { 1.0f, 1.0f, 1.0f };
    std::vector<Model> models;
    std::vector<Transform> transforms;
    std::vector<Group> subgroups;
};

struct World {
    Window window;
    Camera camera;
    std::vector<Group> groups;
};
```

Group: note-se que é uma estrutura de dados que inclui um campo de coleção de variáveis do seu próprio tipo. Nisto assenta a noção hierárquica inerente à renderização das cenas mais tarde.

Transform: Uma estratégia para uniformizar a aplicação de transformações em modelos, independentemente de qual cariz seja, através de mapeamento polimórfico.

Camera: Um registo constante dos detalhes a utilizar para inicialização da função LookAt. O vetor up é dado como o global, algo especialmente útil no mantimento da ortogonalidade dos eixos locais à câmara em primeira pessoa.

2.2. Renderização de Grupos Hierárquicos

```
void renderGroup(const Group& group) {
    glPushMatrix();
    glPushAttrib(GL_CURRENT_BIT);

    // Apply colours
    glColor3f(group.colour.r, group.colour.g, group.colour.b);

    // Apply transforms
    for (const auto& transform : group.transforms) {
        applyTransform(transform);
    }

    // Draw this group's models
    glBegin(GL_TRIANGLES);
    for (const auto& model : group.models) {
        for (size_t i = 0; i < model.vertices.size(); i += 3) {
            glVertex3f(
                model.vertices[i],
                model.vertices[i + 1],
                model.vertices[i + 2]
            );
        }
    }
    glEnd();

    // Recursively render subgroups
    for (const auto& subgroup : group.subgroups) {
        renderGroup(subgroup);
    }


    glPopAttrib();
    glPopMatrix();
}
```

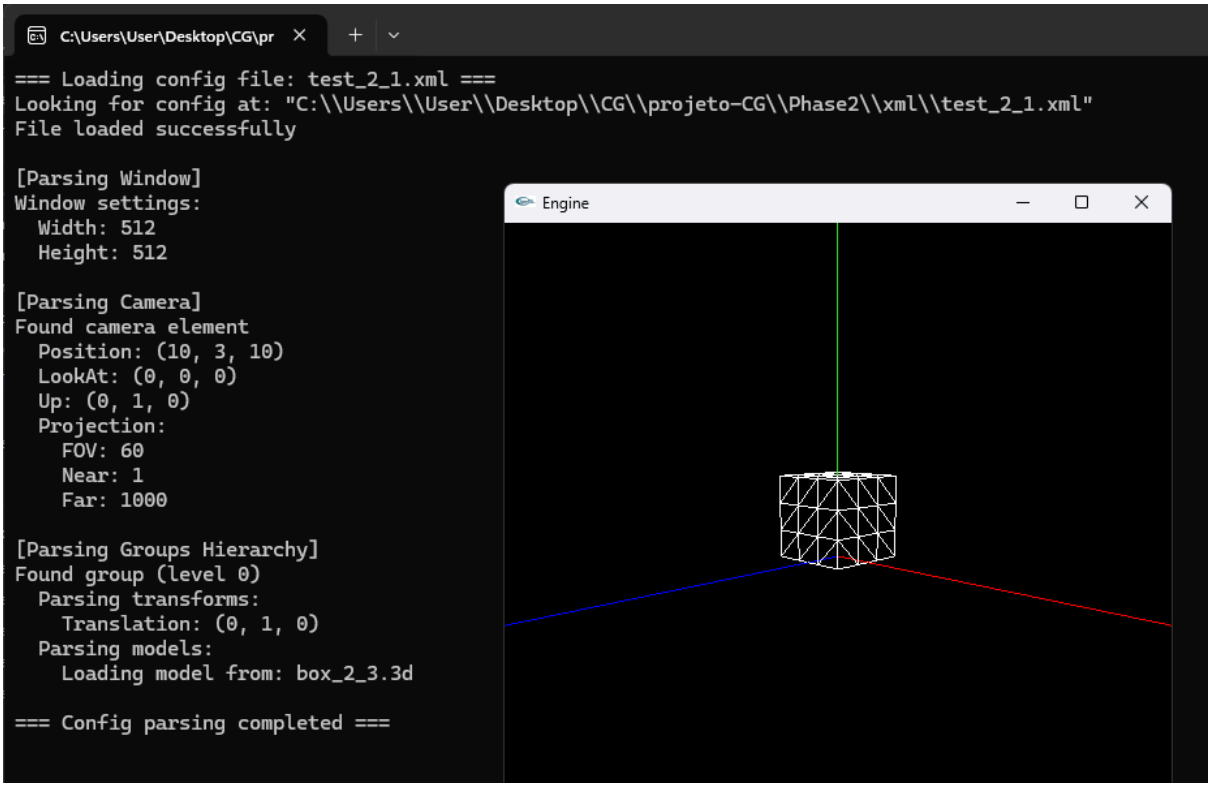
A disposição hierárquica dos grupos a renderizar incentiva o uso da recursividade. Em cada grupo, tem-se a ordem transformações, modelos, subgrupos, para garantir que:

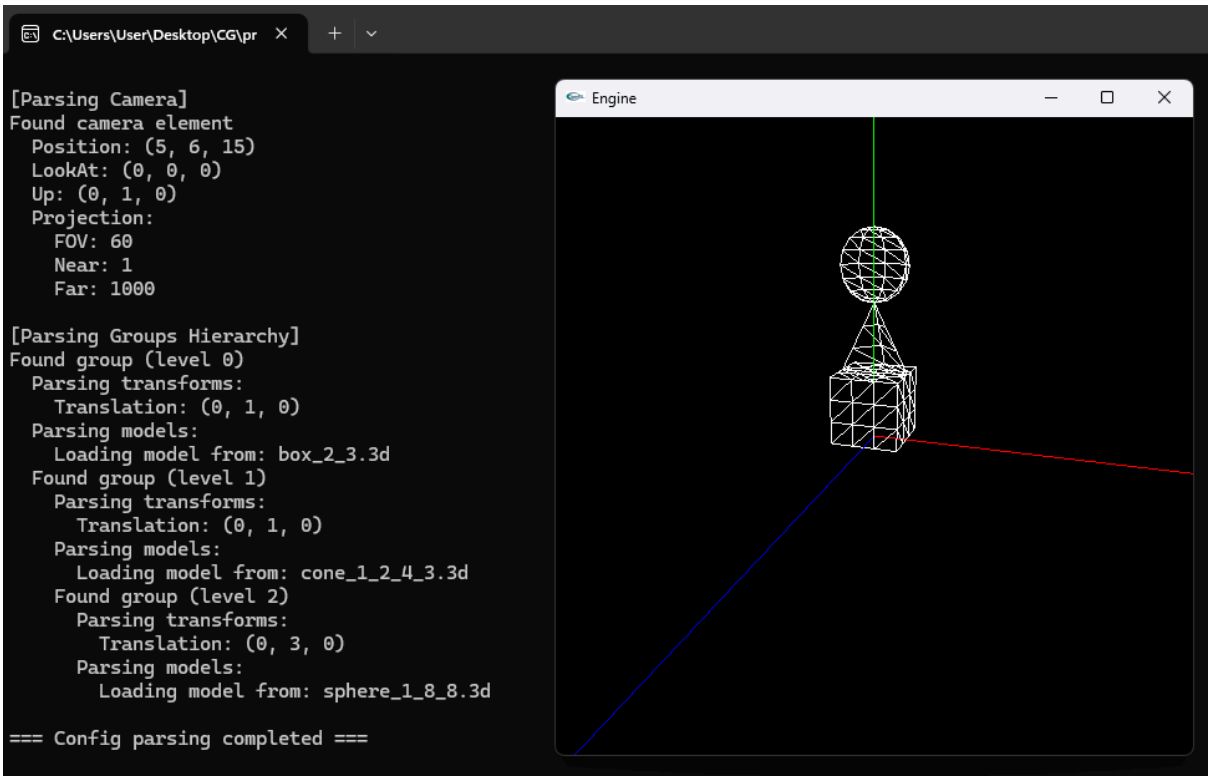
a) As transformações de um grupo, revertidas no final, têm efeito só nele e nos seus subgrupos.

b) As transformações do subgrupo são aplicadas *em cima* das do seu grupo parente.

Cada grupo também tem uma cor associada, mas, visto que esta só afeta os modelos, é comutativa em ordem às transformações. É anulável com push/popAttrib.

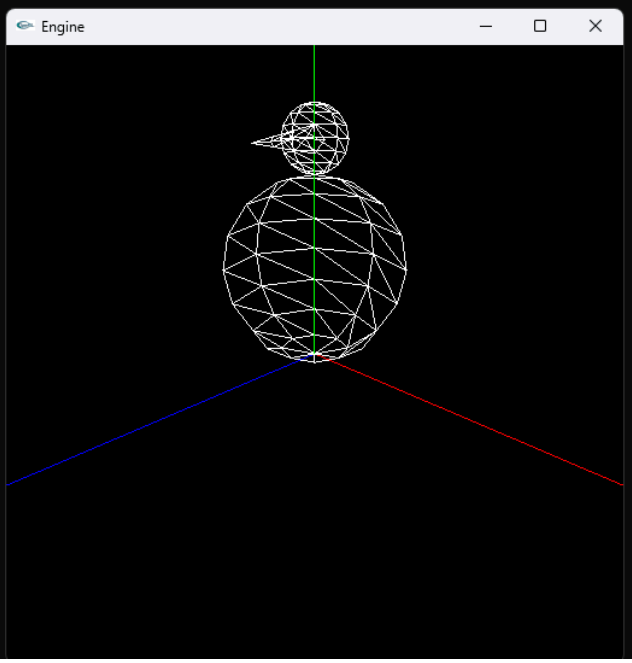
2.3. Cenas Teste Fornecidas





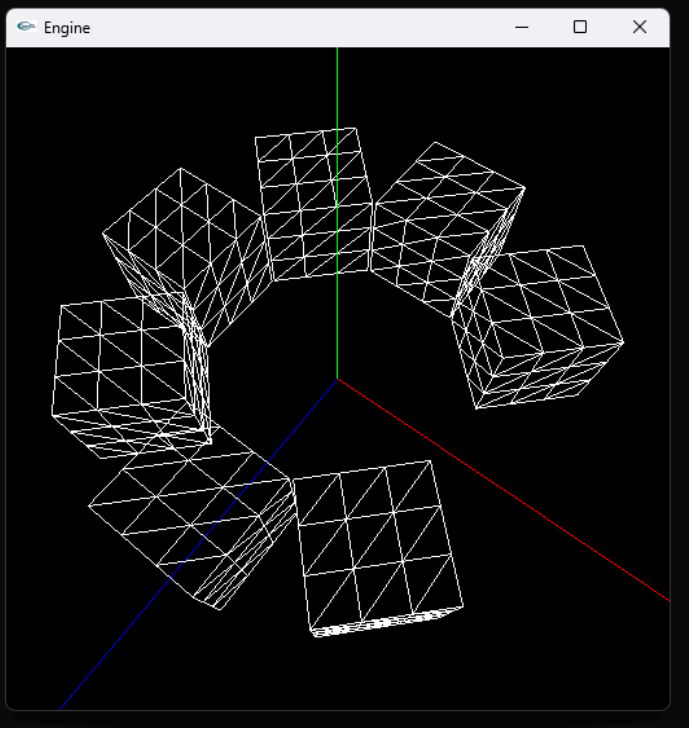
C:\Users\User\Desktop\CG\pr

```
    FOV: 60
    Near: 1
    Far: 1000

[Parsing Groups Hierarchy]
Found group (level 0)
  Parsing transforms:
    Translation: (0, 0.75, 0)
  Found group (level 1)
    Parsing transforms:
      Scaling: (0.75, 0.75, 0.75)
    Parsing models:
      Loading model from: sphere_1_8_8.3d
  Found group (level 1)
    Parsing transforms:
      Translation: (0, 1, 0)
    Found group (level 2)
      Parsing transforms:
        Scaling: (0.25, 0.25, 0.25)
      Parsing models:
        Loading model from: sphere_1_8_8.3d
    Found group (level 2)
      Parsing transforms:
        Rotation: angle=90 axis=(1, 0, 0)
        Scaling: (0.1, 0.3, 0.1)
      Parsing models:
        Loading model from: cone_1_2_4_3.3d

=== Config parsing completed ===
```



C:\Users\User\Desktop\CG\pr

```
[Parsing Window]
Window settings:
  Width: 512
  Height: 512

[Parsing Camera]
Found camera element
  Position: (3, 10, 4)
  LookAt: (0, 0, 0)
  Up: (0, 1, 0)
  Projection:
    FOV: 60
    Near: 1
    Far: 1000

[Parsing Groups Hierarchy]
Found group (level 0)
  Parsing transforms:
    Translation: (0, 1, 0)
  Found group (level 1)
    Parsing transforms:
      Rotation: angle=45 axis=(0, 1, 0)
      Translation: (3.5, 0, 0)
    Parsing models:
      Loading model from: box_2_3.3d
  Found group (level 1)
    Parsing transforms:
      Rotation: angle=90 axis=(0, 1, 0)
      Translation: (3.5, 0, 0)
```



2.4. Posicionamento de Câmaras

Foram implementadas duas câmaras:

a) **Orbital:** gira em torno do eixo global Y olhando para o ponto lookAt fornecido na configuração, sendo a sua posição restrita a coordenadas pertencentes a uma esfera que tenha lookAt como centro e um raio arbitrário ajustável (de início justamente calculado pela distância da posição da câmara ao lookAt).

b) Free Roam (1ª Pessoa): torna-se e movimenta-se segundo os seus eixos locais, definidos pelas suas direções front, right e up.

2.4.1. Câmara 1ª Pessoa

```
if (inFreeroam) {
    gluLookAt(
        freeroam::pos.x,
        freeroam::pos.y,
        freeroam::pos.z,

        freeroam::pos.x + freeroam::front.x,
        freeroam::pos.y + freeroam::front.y,
        freeroam::pos.z + freeroam::front.z,

        freeroam::up.x,
        freeroam::up.y,
        freeroam::up.z
    );
}
```

Conceptualizou-se a câmara em primeira pessoa como um objeto centrado numa esfera de raio 1 apontando segundo um vetor normalizado *front* com direção ao longo do -Z local. A arbitrariedade da posição da câmara e da relação do referencial local para com o global faz com que a única forma de obter um ponto lookAt seja somando a posição a esse vetor front.

```
void moveForward() {
    pos += front * movementSpeed;
}

void moveBackward() {
    pos -= front * movementSpeed;
}

void moveRight() {
    glm::vec3 right = glm::normalize(glm::cross(-front, worldUp));
    pos -= right * movementSpeed;
}

void moveLeft() {
    glm::vec3 right = glm::normalize(glm::cross(-front, worldUp));
    pos += right * movementSpeed;
}
```

Para caminhar para a frente/atrás, só se modifica a posição segundo a direção do vetor front. Análogo para cima/baixo segundo o vetor local up.

Para os lados, há primeiro que obter um vetor representante da direita local. Este nada mais é do que o perpendicular tanto ao up global como ao eixo Z local, pelo que é obtível por produto vetorial.

```
void updateVectors() {
    // Calculate new front vector
    front.x = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
    front.y = sin(glm::radians(pitch));
    front.z = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    front = glm::normalize(front);

    // Re-calculate right and up vectors
    glm::vec3 right = glm::normalize(glm::cross(front, worldUp));
    up = glm::normalize(glm::cross(right, front));
}

void rotateYaw(float direction) {
    yaw += direction * sensitivity;
    updateVectors();
}

void rotatePitch(float direction) {
    pitch += direction * sensitivity;

    // Constrain pitch to prevent over-rotation
    if (pitch > 89.0f) pitch = 89.0f;
    if (pitch < -89.0f) pitch = -89.0f;

    updateVectors();
}
```

A rotação segundo certo ângulo é algo mais complexa. Rodar o vetor front segundo ângulos pitch/yaw implica aplicar-lhe escalas trigonométricas que lhe perturbam a normalidade, e também desatualizam os vetores unitários up e right em termos de ortogonalidade. Assim, é preciso recalculá-los por produto vetorial e normalizá-los para evitar o risco de desvios, causados por outrora se aplicar rotações em matrizes referenciais “corrompidas” com um acumular de transformações indevido. Adicionalmente, restringiu-se também a rotação pitch de modo a prevenir a perda do horizonte na perspetiva.

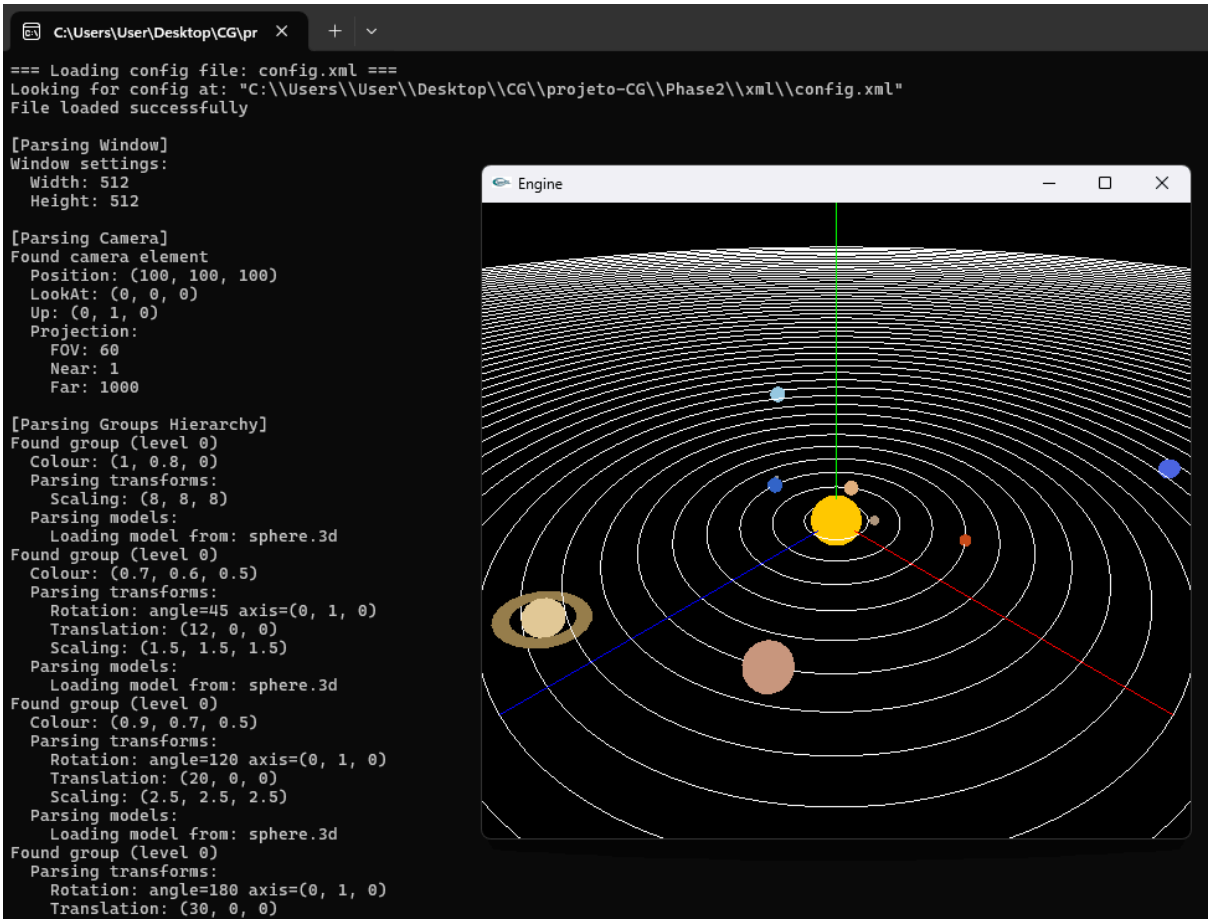
2.4.1. Câmara Orbital

```
else { // for orbital
    gluLookAt(
        orbital::position.x,
        orbital::position.y,
        orbital::position.z,

        world.camera.lookAt[0],
        world.camera.lookAt[1],
        world.camera.lookAt[2],

        world.camera.up[0],
        world.camera.up[1],
        world.camera.up[2]
    );
}
```

A câmara orbital é de conceção mais simples. À função lookAt basta passar diretamente o ponto lookAt e o vetor up configurados, pelo que os cálculos são feitos só para obter a posição. O único movimento possível é em torno do ponto, pelo que as coordenadas da posição não passam de uma conversão de coordenadas polares para cartesianas segundo os ângulos de rotação que são aplicados. A orientação da câmara é inerentemente ajustada ao ter o ponto lookAt como fixo. Note-se, também, de novo a restrição da elevação a 180 graus de modo a prevenir a perda de horizonte.

```
void updatePosition() {
    // Constrain elevation to prevent over-rotation
    elevation = glm::clamp(elevation, -89.0f, 89.0f);

    // Calculate spherical coordinates to cartesian
    float x = radius * cos(glm::radians(elevation)) * sin(glm::radians(azimuth));
    float y = radius * sin(glm::radians(elevation));
    float z = radius * cos(glm::radians(elevation)) * cos(glm::radians(azimuth));

    position = lookAt + glm::vec3(x, y, z);
}

void rotateAzimuth(float direction) {
    azimuth += direction * sensitivity;
    updatePosition();
}

void rotateElevation(float direction) {
    elevation += direction * sensitivity;
    updatePosition();
}
```

2.5. Cena Demonstrativa - Sistema Solar



Tentámos seguir as características dos planetas como a cor e tamanho com rigor em relação à realidade. Para esta fase decidimos por começar com um modelo mais simples que inclui o Sol, os planetas e apenas a lua do planeta Terra.

No XML os objetos tiveram um ordem para serem definidos, no caso o Sol foi definido primeiro tendo apenas uma transformação de escala. Decidimos definir os círculos orbitais para facilitar a visualização da rotação a definir em relação ao Sol. Seguidamente foram definidos os planetas nos quais foram usados transformações de rotações translações e scale, para definir o seu tamanho e posição consoante a órbita que têm ao redor do sol, quando definido um planeta que contenha uma lua, no caso a Terra, esta lua é definida após este planeta por causa de também necessitar de translações rotações e escala para obter o seu tamanho e a sua posição correta em órbita a esse planeta.

2.6. Controlos Implementados

- **Câmara Free Roam:** movimento plano WASD, movimento vertical QE, rotação setas
- **Câmara Orbital:** movimento WASD, zoom in/out setas
- **Trocar Câmara:** C
- **Modo de Polígono:** barra espaço