Computação Gráfica - Fase 2

3 de Abril de 2024

Grupo 2

Daniel Pereira	Duarte Ribeiro	Francisco Ferreira	Rui Lopes
A100545	A100764	A100660	A100643

Introdução

Este relatório descreve a segunda fase do projeto de Computação Gráfica. Nesta fase foi pedido uma extensão da fase anterior, onde foi pedido para ser implementado um sistema de hierarquias de transformações geométricas (translação, rotação e escala) onde são aplicadas aos modelos desse grupo e aos seus subgrupos. Para além disso, também foi pedido uma *demo* de um sistema solar com a utilização do sistema de hierarquias desenvolvido.

1. Hierarquia de Transformações

Para a implementação do sistema de hierarquias de transformações geométricas, foi criada uma estrutura de dados `GroupTransform` que cada `WorldGroup`[1] tem.

Este `GroupTransform` é essencialmente um *wrapper* para uma matriz de transformação (uma matriz 4x4) com uma lista de transformações. Uma transformação pode ser:

- um `Translate(Vec3f translation)`: que representa uma translação sobre o vetor `translation`;
- um `Rotate(float angle, Vec3f axis)`: que representa uma rotação de `angle` graus sobre o eixo `axis`;
- um `Scale(Vec3f scale)`: que representa uma escala sobre o vetor `scale`.

A partir da lista de transformações, a matriz de transformação é calculada e armazenada no `GroupTransform`. Esta operação é feita no carregamento do mundo ou a cada vez que a lista de transformações é alterada.

As formulas para calcular cada tipo de transformação são as seguintes:

- Translação em (x, y, z):

$$\mathcal{T}(x,y,z) = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Escala em (x, y, z):

$$\mathcal{S}(x,y,z) = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

[1] De relembrar que `WorldGroup` é a estrutura de dados que contém dados do grupo do mundo (lista de modelos, subgrupos e agora lista de transformações).

- Rotação em torno de um eixo arbitrário (x, y, z) por um ângulo α:

$$\mathcal{R}(\alpha, x, y, z) = \begin{pmatrix} x^2 + (1 - x^2)\cos(\alpha) & xy(1 - \cos(\alpha)) - z\sin(\alpha) & xz(1 - \cos(\alpha)) + y\sin(\alpha) & 0 \\ xy(1 - \cos(\alpha)) + z\sin(\alpha) & y^2 + (1 - y^2)\cos(\alpha) & yz(1 - \cos(\alpha)) - x\sin(\alpha) & 0 \\ xz(1 - \cos(\alpha)) - y\sin(\alpha) & yz(1 - \cos(\alpha)) + x\sin(\alpha) & z^2 + (1 - z^2)\cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Com isto, a matriz de transformação de um `GroupTransform` é calculada multiplicando todas as matrizes de transformação na ordem em que foram adicionadas à lista.

A rotação sendo em torno de um eixo arbitrário, em oposição em torno de um eixo principal, dá mais flexibilidade ao artista

Por exemplo, para este conjunto de transformações no xml:

```
<group>
    <translate X="1" Y="0" Z="0"/>
    <rotate angle="90" X="0" Y="1" Z="0"/>
    <scale X="2" Y="1" Z="1"/>
</group>
```

A matriz final de transformação seria (I sendo a matriz identidade):

$$R = I \times \mathcal{T}(1, 0, 0) \times \mathcal{R}(90\,\text{deg}, 0, 1, 0) \times \mathcal{S}(2, 1, 1) \tag{1}$$

O código de operações sobre matrizes já tinha sido desenvolvido na fase anterior, logo esta funcionalidade foi desenvolvida sem muitas alterações.

1.1. No *OpenGL*

Com a matriz de transformação pronta, foi apenas necessário chamar o método `glMultMatrixf` do *OpenGL* [1] com a matriz de transformação antes de renderizar os modelos do grupo. Esta função multiplica a matriz atual pela matriz selecionada atualmente (neste caso a `GL_MODELVIEW`), substituindo-a pela matriz resultante.

Devido a esta multiplicação, a função `glPushMatrix` e `glPopMatrix` foram usadas para guardar e restaurar a matriz anterior, respetivamente. Isto permite que a matriz atual seja restaurada para o estado anterior após a renderização do grupo.

```
void Engine::renderGroup(WorldGroup &group) {
    glPushMatrix()

    glMultMatrixf(*group.transform.mat);

    for (size_t model_index : group.models) {
        Model &model = m_models[model_index];
        renderModel(m_models[model_index]);
    }
    for (WorldGroup &child : group.children) {
        renderGroup(child);
    }

    glPopMatrix();
}
```

Devido à função `glMultMatrixf` do *OpenGL* interpretar as matrizes em *column-order* em vez de *row-order* (como é em *arrays* bidimensionais em *C++*), a matriz final de transformação (Equação 1) também tem que ser transposta antes de ser enviada para o *OpenGL*.

1.2. Visualização e manipulação das transformações

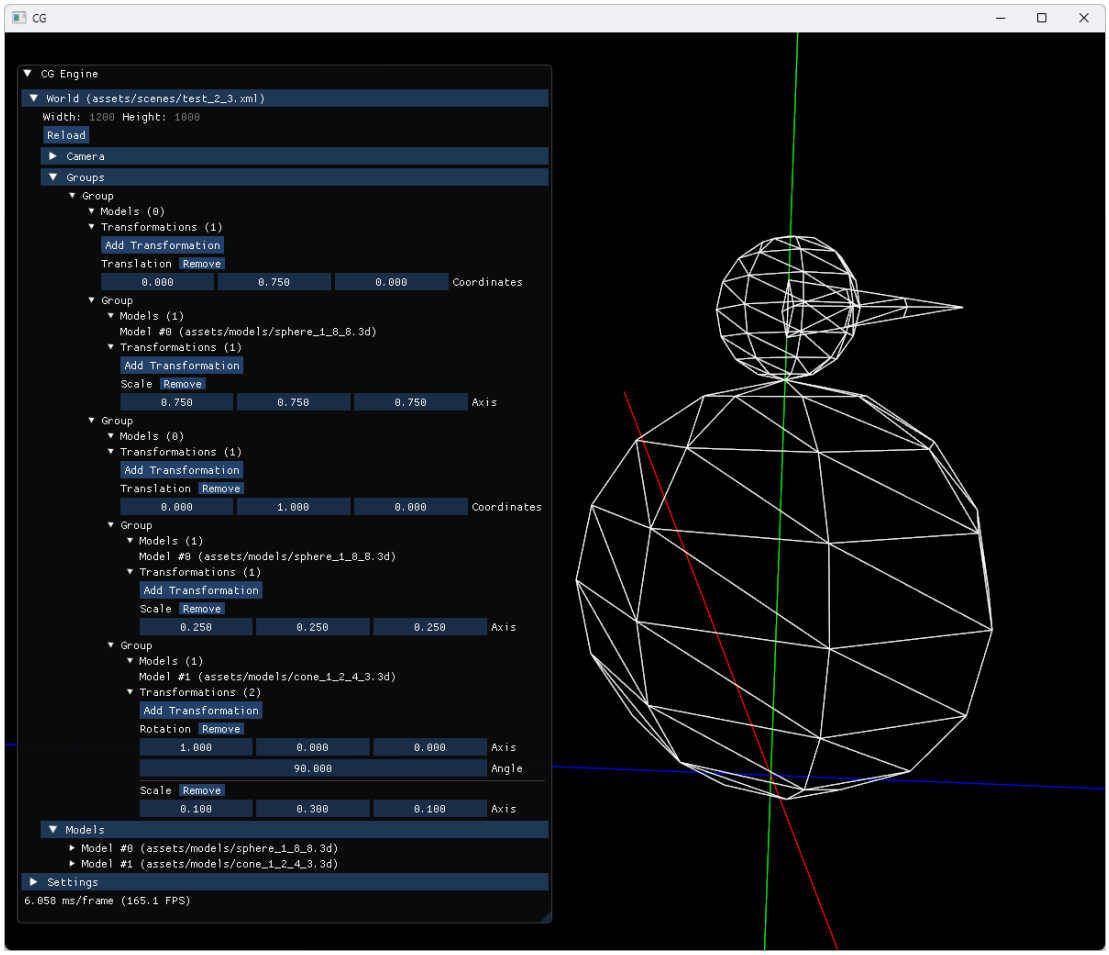

Figura 1: Visão geral das transformações no *ImGui*

O grupo tem a ambição de tornar esta *engine* **usável**. Isto implica que interações com o utilizador são essenciais e críticas. Com isto, recorrendo ao *ImGui* [2], tornamos possível a visualização e edição das transformações que são carregadas para o mundo.

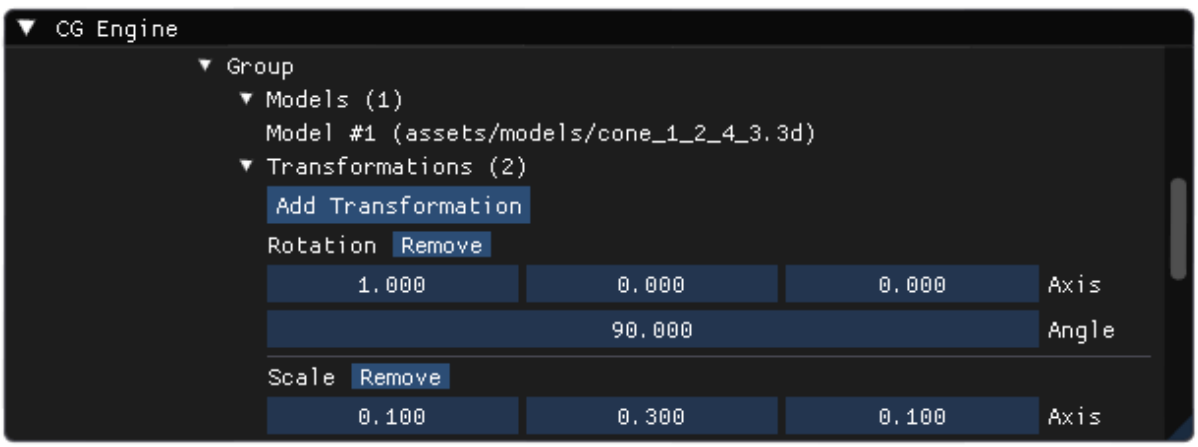

Figura 2: Lista de transformações num grupo no *ImGui*

Um grupo tem uma lista de transformações. Uma transformação pode ser **removida** clicando no botão Remove presente à frente do nome dessa. Qualquer parâmetro de qualquer transformação pode ser alterado em tempo real, com feedback instantâneo no mundo.

É possível adicionar uma nova transformação ao grupo. Para isso, basta clicar no botão Add Transformation e escolher o tipo de transformação que se quer adicionar. A transformação é adicionada ao fim da lista de transformações do grupo.

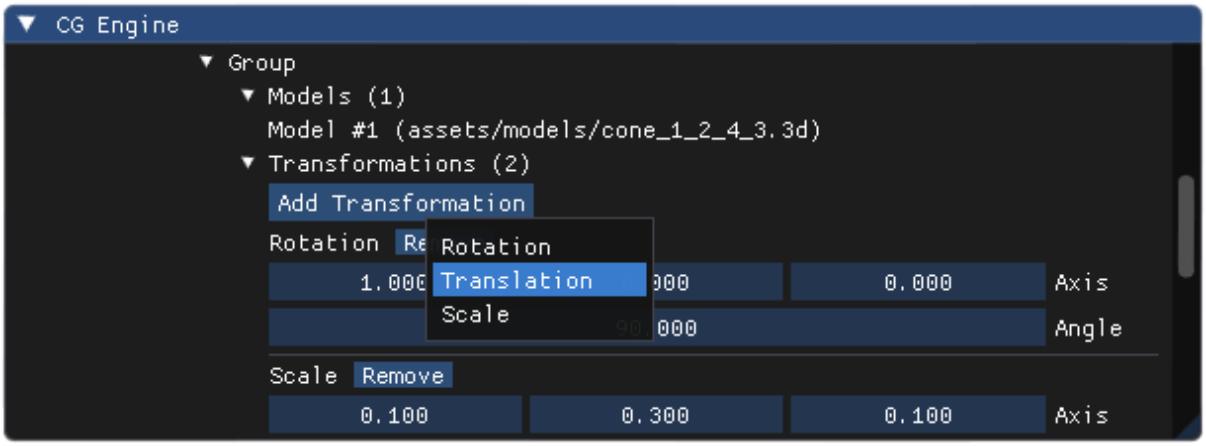

Figura 3: Adição de uma nova transformação no *ImGui*

Também é possível, arrastando uma transformação com o rato, trocar duas transformações de posições na lista entre elas.

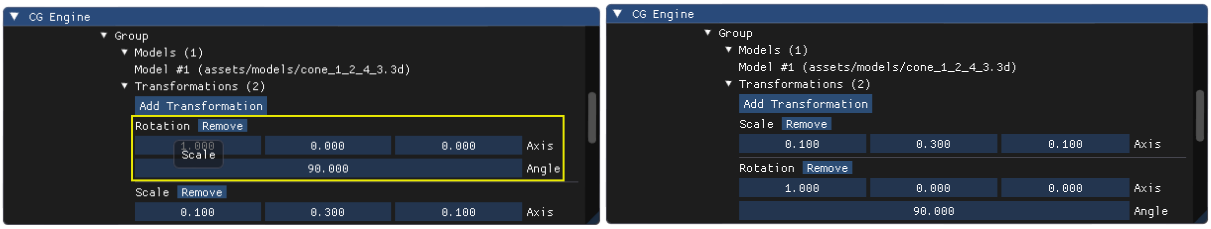

Figura 4: *Drag and Drop* de transformações

Com a mesma ação, dá também para mover a transformação de um grupo para outro.

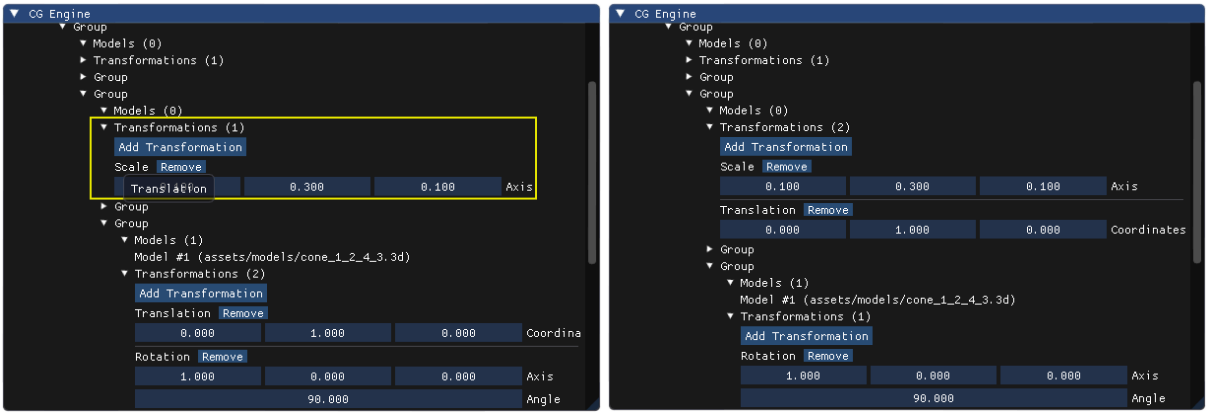

Figura 5: *Drag and Drop* de transformações entre grupos

Estas funcionalidades interativas serão essenciais para atingir o objetivo pretendido.

2. Câmera primeira pessoa

Como os requisitos desta fase não foram muito extensos, nem envolveram muitos *rewrites* de código na *engine*, implementamos também o modo de câmera em primeira pessoa.

Este modo baseia-se na mesma função `gluLookAt` que a câmera em terceira pessoa também usa. Com isto, a alteração entre os modos de primeira-pessoa e terceira-pessoa não causam alterações na posição câmera[2], o que torna a experiência de utilização muito mais intuitiva.

O cálculo de atualização da câmera em primeira-pessoa conforme o movimento do rato é muito semelhante à atualização dela em terceira-pessoa. Como foi descrito no relatório anterior, a partir de cálculos de coordenadas esféricas a partir de cartesianas é possível conseguir os valores de raio e ângulos α e β da câmera atual, somar a diferença de movimento do rato a esses ângulos, e converter de volta para coordenadas cartesianas. A diferença é que, em vez de se atualizar as coordenadas de posição da câmera, em primeira-pessoa, atualiza-se as coordenadas do *looking at*.

```
void UpdateCameraRotation(world::Camera &camera, float x_offset, float y_offset) {
    x_offset = degrees_to_radians(x_offset) * sensitivity;
    y_offset = degrees_to_radians(y_offset) * sensitivity;

    float radius, alpha, beta;
    Vec3f direction = camera.position - camera.looking_at;
    if (camera.first_person_mode) {
        direction = -direction;
        y_offset = -y_offset;
    }

    direction.ToSpherical(radius, alpha, beta);

    alpha -= x_offset;
    beta -= y_offset;

    // Clamp -180deg<beta<180deg

    const Vec3f new_direction = Vec3fSpherical(radius, alpha, beta);
    if (camera.first_person_mode)
        camera.looking_at = new_direction + camera.position;
    else
        camera.position = new_direction + camera.looking_at;
}
```

Em primeira-pessoa, o vetor direção é invertido visto que, em primeira pessoa, queremos mover a câmera à volta dela e não em torno do *looking at*. O *y offset* também é invertido para o movimento do rato ser traduzido num movimento mais natural da câmera.

No fim, a câmera é atualizada com a soma do vetor direção com a posição da câmera (em primeira-pessoa) ou com o *looking at* (em terceira-pessoa), para que o movimento da câmera seja feito em relação a esses pontos.

[2]Isto poderia acontecer devido a, por exemplo, usar-se variáveis de `yaw` e `pitch` diferentes para cada um dos modos.

2.1. Movimento

Para o movimento, a cada renderização de *frame*, a câmera é alimentada com os valores de *input* do teclado, se estiver em primeira pessoa, e *scroll* do rato.

Este *input* é um vetor que representa para onde o utilizador pretende andar.

- `W` pressionado $\rightarrow (0, 0, 1) \rightarrow$ Andar para frente
- `S` pressionado $\rightarrow (0, 0, -1) \rightarrow$ Andar para trás
- `D` pressionado $\rightarrow (1, 0, 0) \rightarrow$ Andar para a direita
- `A` pressionado $\rightarrow (-1, 0, 0) \rightarrow$ Andar para a esquerda
- `Espaço` pressionado $\rightarrow (0, 1, 0) \rightarrow$ Subir
- `Control Esquerdo` pressionado $\rightarrow (0, -1, 0) \rightarrow$ Descer

Para movimentações diagonais, por exemplo, várias destas teclas podem ser pressionadas ao mesmo tempo, e com isso, o vetor *input* será a soma dos vetores associados a todas as teclas pressionadas.

Com o vetor *input*, são calculados dois vetores, um que representa a direção para a frente da câmera, e outro que representa a direção para a direita da câmera.

$$\vec{f} = \left\| \overrightarrow{looking\_at} - \overrightarrow{position} \right\| \tag{2}$$

$$\vec{r} = \left\| \vec{f} \times \overrightarrow{up} \right\| \tag{3}$$

Sendo, $\vec{f}$ o vetor de direção para a frente da câmera, $\vec{r}$ o vetor de direção para a direita da câmera, $\overrightarrow{looking\_at}$, $\overrightarrow{position}$ e $\overrightarrow{up}$ vetores presentes nas informações da câmera, $\vec{a} \times \vec{b}$ o produto externo entre $\vec{a}$ e $\vec{b}$ e $\|\vec{a}\|$ o vetor normalizado de $\vec{a}$.

Com isto, o vetor direção final pode ser calculado da seguinte forma:

$$\vec{d} = \left\| \vec{f} \times \vec{i}_z + \vec{r} \times \vec{i}_x \right\| + \overrightarrow{up} \times \vec{i}_y \tag{4}$$

Sendo $\vec{i}$ o tal vetor de *input* e as variáveis denotadas anteriormente. Atenção que as multiplicações aqui (ex: $\vec{f} \times \vec{i}_z$) não denotam produtos externos, mas sim multiplicações de vetores por valores.

De notar que os movimentos no eixo xz são normalizados, pois, caso contrário, o movimento na diagonal seria mais rápido do que o movimento horizontal[3]. A normalização não inclui o eixo y para dar mais liberdade de movimentação ao utilizador na subida e descida enquanto se move horizontalmente.

2.1.1. Suavidade no movimento

Para dar um *feedback* de utilização agradável ao utilizador, a câmera move-se conforme equações físicas de aceleração e velocidade.

A cada *frame*, a câmera é atualizada com a função `TickCamera` que recebe o vetor *input* e o *timestep* (tempo que passou desde o último *frame*). Este *timestep* é calculado a partir da diferença do `glfwGetTime` [3] entre o *frame* atual e o *frame* anterior[4].

[3] O vetor $(1, 0, 1)$ tem tamanho $\sqrt{2}$, não é unitário.

[4] Isto trata-se apenas de uma aproximação visto que o *Framerate* pode não ser constante, mas como não estamos num cenário onde operações têm de ser precisas (por exemplo, em colisões de objetos), não há grande problema.

Desta forma, o nosso *loop* principal da *engine* foi alterado para incluir a atualização da câmera a cada *frame*.

```
void Engine::Run() {
    float currentTime = glfwGetTime();
    while (!glfwWindowShouldClose(m_window)) {
        ...
        const float newTime = glfwGetTime();
        ProcessInput(newTime - currentTime);
        currentTime = newTime;

        Render();
        ...
    }
}
```

Para se concretizar o movimento suave, foram adicionados parâmetros de velocidade, velocidade máxima, aceleração e fricção à câmera. Voltando à função `TickCamera`, esta função é responsável por atualizar esses parâmetros conforme o *input* do utilizador.

Esses parâmetros são atualizados da seguinte forma:

$$\vec{v} = \vec{d} \times camera_{acceleration/s} \times \Delta t \tag{5}$$

$$\vec{p} = \vec{v} \times \Delta t \tag{6}$$

Sendo $\vec{v}$ o vetor velocidade da câmera, $\vec{d}$ o vetor direção calculado anteriormente, $camera_{acceleration/s}$ uma constante da câmera, Δt o intervalo de tempo entre *frames*.

Caso o utilizador não esteja a pressionar nenhuma tecla de movimento de *input*, a velocidade da câmera é atualizada da seguinte forma:

$$\vec{v}_n = \vec{v}_a \times camera_{friction/s} \times \Delta t \tag{7}$$

Sendo $\vec{v}_n$ a nova velocidade, $\vec{v}_a$ a velocidade anterior e $camera_{friction/s}$ uma constante na câmera.

A lógica de implementação desta funcionalidade pode ser vista aqui:

```
void TickCamera(world::Camera &camera, const Vec3f input, const float timestep) {
    const Vec3f forward = (camera.looking_at - camera.position).Normalize();
    const Vec3f right = forward.Cross(camera.up).Normalize();

    const Vec3f move_dir = (forward * input.z + right * input.x).Normalize()
        + camera.up * input.y;
    const Vec3f acceleration = move_dir * camera.acceleration_per_sec * timestep;
    camera.speed += acceleration;

    if (camera.speed.Length() > camera.max_speed_per_second) {
        camera.speed = camera.speed.Normalize() * camera.max_speed_per_sec;
    }

    camera.position += camera.speed * timestep;
    camera.looking_at += camera.speed * timestep;

    if (move_dir.x == 0 && move_dir.y == 0 && move_dir.z == 0) {
        camera.speed -= camera.speed * timestep * camera.friction_per_second;
    }

    // Atualizar scroll
}
```

A mesma lógica também foi aplicada ao *scroll* do rato que agora dá *zoom in* e *zoom out* suavemente independente do modo da câmera (primeira ou terceira pessoa).

Estes parâmetros podem ser vistos e editados em tempo real na aba *"Camera"* do *ImGui*.

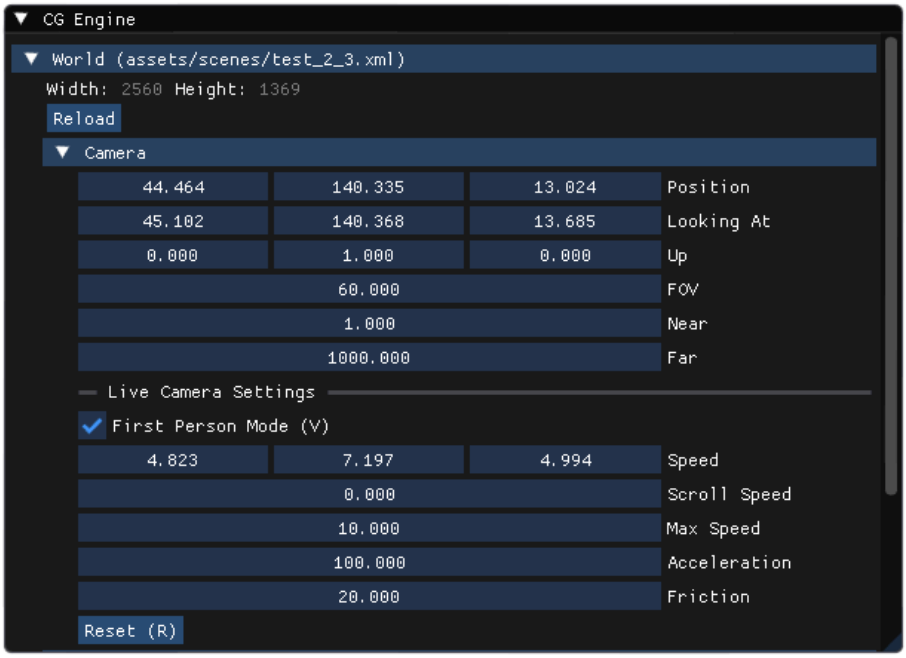

Figura 6: Visualização dos novos parâmetros da câmera no *ImGui*

3. Repetição de Modelos

Devido à grande quantidade de luas e planetas no sistema solar, que irão utilizar por baixo o mesmo modelo de esfera, foi feita algumas alterações nas estruturas de dados para não haver duplicação de memória em modelos iguais[5].

Anteriormente, a estrutura do modelo, estava guardada dentro da lista de modelos de um `WorldGroup`. Isto trará problemas de memória e lentidão de carregamento em cenas pesadas, que contenham modelos repetidos.

Agora, a estrutura dos modelos foi movida para dentro da estrutura do mundo completo, e dentro do `WorldGroup` guarda-se apenas um inteiro que representa o índice do modelo na lista de modelos do mundo.

[5]De relembrar que um modelo, atualmente, é simplesmente a lista de vértices que o representa graficamente.

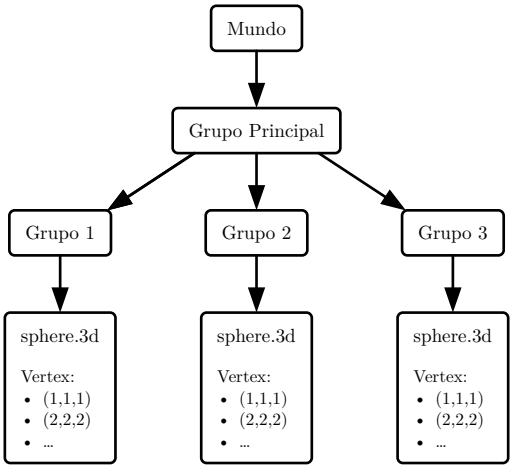

Figura 7: Sem identificadores de modelo

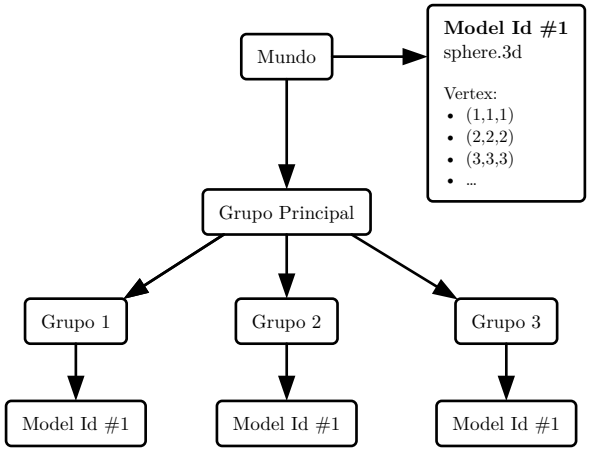

Figura 8: Com identificadores de modelo

3.1. Modelos no *ImGui*

A visualização dos modelos no *ImGui* também foi atualizada para constatar as mudanças nas estruturas de dados e mostrar novas informações sobre o modelo.

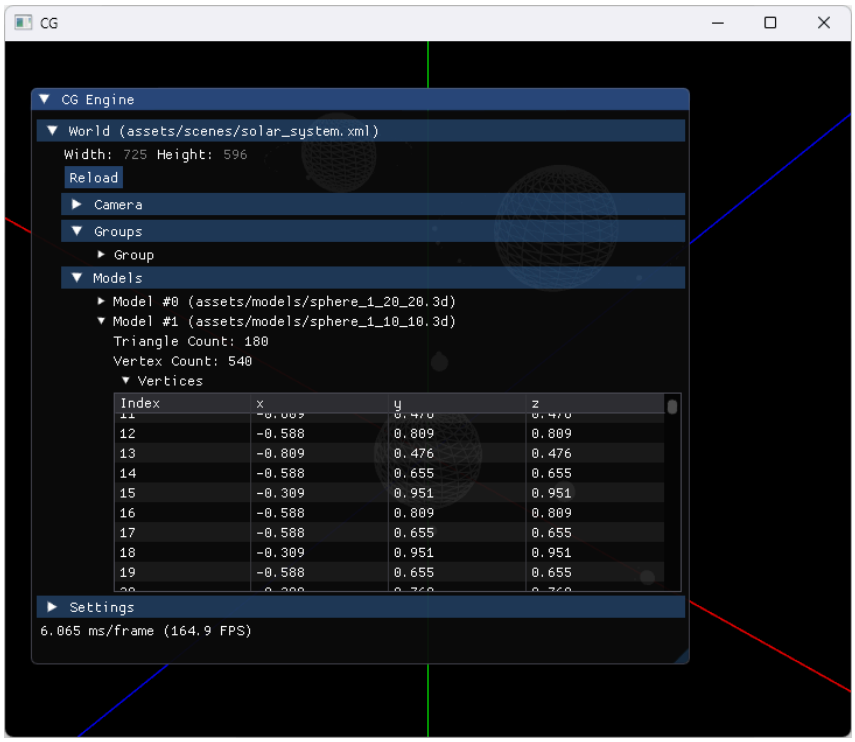

Figura 9: Visualização de modelos no ImGui

4. Serialização do Mundo

Para implementação da geração do mundo do sistema solar, será necessária uma forma programática de criação de mundos no formato *XML*. Para aproveitar as APIs já existentes de cálculos matemáticos, o carregamento do mundo da nossa *engine* foi estendido para também suportar escrita de mundos em *XML*.

Desta forma, agora, um mundo carregado em memória pode ser guardado num ficheiro *XML*, e um mundo guardado num ficheiro *XML* pode ser carregado em memória, tendo assim operações de *serialização* e *deserialização* implementadas.

Esta funcionalidade foi implementada também com a biblioteca *tinyxml2* [4], que também já foi usada na fase anterior para a leitura de mundos.

Isto veio com uma vantagem que, o mundo pode ser guardado em *runtime*, estendendo assim a criação de mundos para que o utilizador, dinamicamente, possa modificar o mundo a partir das funcionalidades mostradas no capítulo Secção 1.2 e guardá-lo em disco para uso posterior.

4.1. Funcionalidade de *Reload*

Com o *refactor* feito devido à implementação da serialização de mundos, também foi implementado um botão de Reload no *ImGui*, que relê o arquivo XML do mundo e altera o estado do mundo para o que foi lido. Este botão também funciona como um *reset* da *scene* caso não tenha havido mudanças.

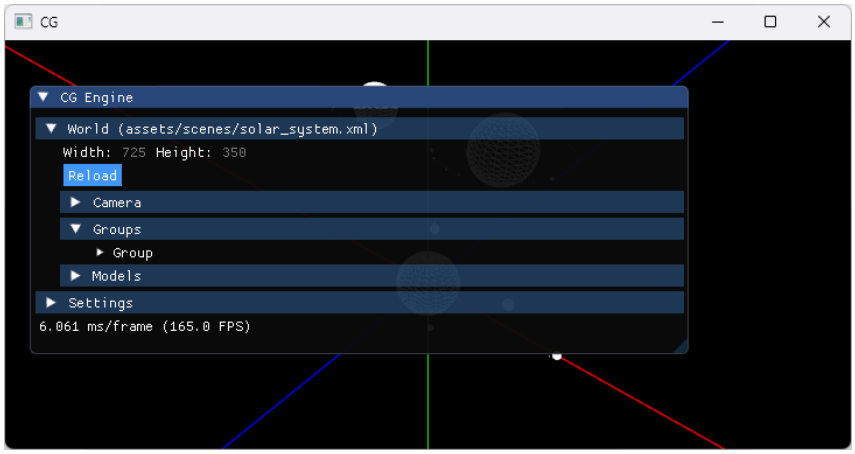

Figura 10: Botão de Reload no *ImGui*

5. Primeira versão do Sistema Solar

Com todas as funcionalidades implementadas, prosseguimos para a criação do gerador do sistema solar.

O sistema solar é gerado com o programa gerador, sendo um novo tipo de gerador possível (para além dos geradores de cubos e esferas).

```
$ generator solar-system <sun size scale factor> <planet distance scale factor>
<scene scale factor> <output file>
```

As informações dos planetas (com os seus satélites) do sistema solar como diâmetro, distância ao sol, período de rotação, período de translação, etc. foram retiradas do *dataset* https://devstronomy.martinovo.net/ e estão guardadas na pasta `assets/planets/`.

Como no CSV do *dataset* estão presentes distâncias demasiado grandes que não podem ser representadas em escala real no nosso mundo, foi necessário aplicar um fator de escala para todas as distâncias e tamanhos dos planetas do sistema solar. Este fator de escala é passado como argumento ao gerador.

Para alivar a distância entre os planetas, foi também aplicado um fator de escala para as distâncias entre os planetas e o sol para que a nossa representação do sistema solar tenha os planetas mais juntos. Também passado como argumento ao gerador.

Os planetas também são rodados ligeiramente de acordo com o ângulo de inclinação do seu plano orbital, retirado do mesmo *dataset*.

Como o modelo para já é estático, a posição do planeta foi escolhida aleatoriamente para cada planeta, de acordo com a sua distância ao sol, recorrendo a coordenadas esféricas.

Aplicando os fatores de escala vimos que alguns satélites de alguns planetas ficavam demasiado pequenos, então também tivemos que aplicar um valor mínimo para o tamanho do planeta.

Com isto ainda, para poupar recursos, em planetas que o seu tamanho sejam menores que 0.05 de unidades de tamanho no nosso mundo, o modelo do planeta é substituído por um modelo de esfera com menos detalhe (de 20 *stacks* e *slices* para 10 *stacks* e *slices*).

Com isto, a estrutura do *XML* do sistema solar gerado é a seguinte:

- Grupo principal
  - Grupos planetários
    - Transformações
      - Translação para a posição do planeta
      - Rotação para o ângulo de inclinação do plano orbital
    - Grupo planeta
      - Transformações
        - Escala para o tamanho do planeta
      - Modelo do planeta
    - Grupos satélites
      - Transformações
        - Translação para a distância da lua ao planeta
        - Escala para o tamanho do satélite

Os satélites dos planetas são gerados à volta do planeta, com o seu ângulo aleatório. A distância do satélite ao planeta é relacionada com o tamanho do planeta, visto que a distância real ficava demasiado grande para a nossa representação.

A primeira versão do sistema solar gerado pode ser visto na seguinte imagem:

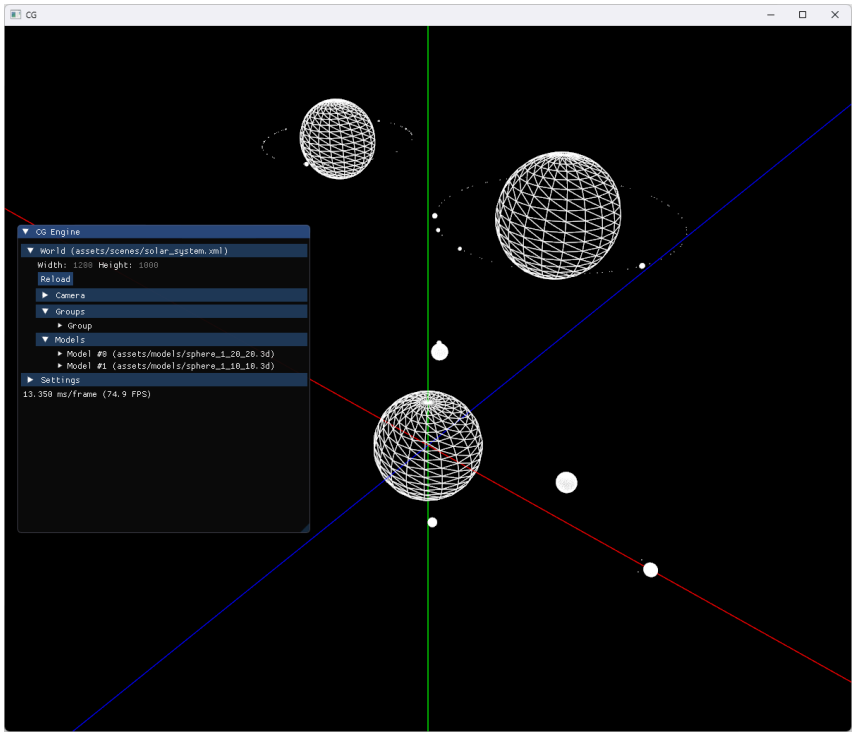

Figura 11: Primeira versão do sistema solar gerado

Para já o sistema solar está muito cru, sem texturas, sem movimento, sem efeitos de luzes, mas temos a ambição de fazer um sistema solar muito mais detalhado e realista com, por exemplo, anéis de saturno e Júpiter, efeitos de *bloom*, para além dos requisitos pedidos no enunciado.

Conclusão

O desenvolvimento do projeto está a decorrer positivamente, com o projeto a ganhar forma e a tornar-se cada vez mais complexo e interessante. A implementação do sistema de hierarquias de transformações geométricas com a visualização e manipulação no *ImGui* foi um grande passo para a usabilidade da *engine*. A câmera suave foi um grande *quality of life* para o utilizador, e a serialização do mundo e o gerador do sistema solar são funcionalidades que vão ser essenciais para a criação de cenas mais complexas e para a criação de cenas dinâmicas.

A CI implementada na fase anterior já se provou útil para detetar problemas de compatibilidades entre plataformas.

O grupo está satisfeito com o progresso feito até agora e está ansioso para continuar a desenvolver o projeto.

Bibliografia

[1] «OpenGL - The Industry Standard for High Performance Graphics». [Em linha]. Disponível em: https://www.opengl.org/

[2] ocornut, «ocornut/imgui». [Em linha]. Disponível em: https://github.com/ocornut/imgui

[3] «An OpenGL library». [Em linha]. Disponível em: https://www.glfw.org/

[4] L. Thomason, «TinyXML-2». [Em linha]. Disponível em: https://github.com/leethomason/tinyxml2