

Universidade da Beira Interior

Licenciatura em Engenharia Informática



**Departamento de
Informática**

Sistema Solar

Elaborado por:

**António Cruz, 47995
Francisco Santos, 47711
Leonardo Santos, 48708
Ruben Carvalho, 46180**

Orientador:

Professor Doutor Abel Gomes

Unidade Curricular:

Computação Gráfica

9 de janeiro de 2024, Covilhã

Agradecimentos

Após terminar o projeto, é impossível não refletir sobre a imensa contribuição que recebemos ao longo do tempo. Neste capítulo, gostaríamos de expressar os nossos mais profundos agradecimentos ao professor Abel João Padrão Gomes e a todos os membros do grupo que tornaram esta experiência educacional significativa. Ao professor, expressamos a nossa mais sincera gratidão. O seu comprometimento apaixonado com o ensino e a sua dedicação incansável ao compartilhamento de conhecimento foram fundamentais para o nosso aprendizado. Além disso, não podemos deixar de expressar nossa gratidão a cada membro do grupo. Cada um de nós trouxe uma perspectiva única, habilidades distintas e um compromisso inabalável com a realização dos objetivos do trabalho. As nossas reuniões foram marcadas por colaboração, criatividade e respeito mútuo, elementos essenciais para o nosso sucesso coletivo. A troca de ideias, o trabalho árduo e a camaradagem que cultivámos juntos foram fundamentais para enfrentar os desafios que encontrámos ao longo deste percurso académico. Cada membro do grupo desempenhou um papel vital, contribuindo não apenas para a conclusão do trabalho, mas também para o crescimento individual de cada um. É impossível deixar de agradecer também aos nossos familiares, que se fizeram presentes, nos dias bons e maus, e em momento algum nos deixaram cair, apoando-nos sempre para nos mantermos no bom sentido, nesta que é a nossa caminhada para o sucesso profissional.

Resumo

Pensar que cada ser humano é apenas uma peça pequena de um grande puzzle que constitui a nossa galáxia deixa muitas pessoas fascinadas, pela nossa inferioridade comparando com a complexidade e o tamanho do que nos rodeia. Para essas pessoas é ainda mais fascinante ver os elementos do sistema solar com os seus próprios olhos, com ajuda de telescópios e imagens partilhadas por satélites, apesar de nem sempre ser fácil de conseguir ver o que queremos. Mas e se conseguissemos representar o nosso sistema solar num computador? Seria muito mais fácil e barato ver tudo aquilo que queremos, sem sairmos das nossas próprias casas, para além de nos fornecer uma maior imersão. Ao longo deste documento vamos explicar detalhadamente como foi desenvolvida e implementada a nossa versão do sistema solar, com os vários planetas e informações sobre os mesmos. Vamos explicar também a magia que fez todas as físicas dos planetas funcionar, como utilizámos bibliotecas de *User Interface* (UI) para criar menus que possibilitam alterar partes do sistema em tempo real, e muito mais.

Acrónimos

CG	Computação Gráfica
GPU	<i>Graphics Processing Unit</i>
HDR	<i>High Dynamic Range</i>
IMGUI	<i>Immediate Mode Graphic User Interface</i>
LDR	<i>Low Dynamic Range</i>
RAM	<i>Random-Access Memory</i>
UC	Unidade Curricular
UI	<i>User Interface</i>

Lista de Excertos de Código

2.1	<i>Script Run</i>	7
2.2	Inicialização da Janela	7
2.3	Inicialização do <i>IMGUI</i>	7
2.4	Carregamento das <i>shaders</i> , modelos e texturas	8
2.5	Colisões parte 1	8
2.6	Colisões parte 2	8
2.7	<i>DeltaTime</i>	10
2.8	<i>Cubemap</i> para a <i>Skybox</i>	11
2.9	<i>Fragment Shader</i> do <i>framebuffer</i> para a <i>exposure</i>	14
2.10	<i>Fragment shader</i> do <i>bloom</i>	15
2.11	Escrita em 2 <i>colorbuffers</i> diferentes	15
2.12	<i>Fragment shader</i> do <i>framebuffer</i>	16
2.13	Verificação da interseção do <i>ray-cast</i>	17
2.14	<i>Thread</i> da Música	18

Lista de Figuras

2.1	Resultado Final	4
2.2	Terra de Noite	5
2.3	Terra com Núvens	5
2.4	Asteróides	9
2.5	Menu Lateral	10
2.6	Menu de Informações	10
2.7	<i>Labels</i>	11
2.8	<i>Skybox</i>	13
2.9	Passos do <i>Bloom</i>	14

Conteúdo

Lista de Figuras	viii
Conteúdo	ix
1 Introdução	2
1.1 Âmbito, Enquadramento e Motivação	2
1.2 Objetivos do Trabalho	2
1.3 Organização do Documento	2
2 Desenvolvimento e Implementação	4
2.1 Introdução	4
2.2 Demo	4
2.3 Dependências	6
2.3.1 IMGUI	6
2.3.2 assimp	6
2.3.3 <i>stb_image.h</i>	6
2.3.4 soloud	6
2.4 Compilação	6
2.5 Detalhes de Implementação	7
2.5.1 Inicialização	7
2.5.2 <i>Loop Principal</i>	10
2.5.3 <i>Cube Maps</i>	11
2.5.4 <i>Post-processing</i>	13
2.5.5 <i>Bloom</i>	14
2.5.6 <i>Lens Flare</i>	17
2.5.7 Música de Fundo	18
2.6 Conclusão	19
3 Conclusões e Trabalho Futuro	21
3.1 Conclusões Principais	21
3.2 Trabalho Futuro	21
Bibliografia	23

1 / Introdução

1.1 | Âmbito, Enquadramento e Motivação

Neste documento vamos descrever como foi feito, organizado e dividido o nosso projeto da Unidade Curricular (UC) de Computação Gráfica (CG), onde escolhemos desenvolver um Sistema Solar, o projeto que achámos ser mais interessante de fazer pela sua dificuldade e pelas possibilidades de conceitos que poderíamos implementar para o complementar.

O desenvolvimento deste projeto tornou-se importante para consolidar os básicos de CG e para aprender outros conceitos muito importantes e que nos deixaram fascinados pela sua complexidade e pelos resultados obtidos quando estes se aplicam.

1.2 | Objetivos do Trabalho

Com este projeto, o nosso maior objetivo foi melhorar e afinar os nossos conhecimentos sobre CG, pelo que a sua realização foi fundamental, assim como toda a investigação realizada para o mesmo.

Também tivemos como objetivos a aprendizagem de conceitos que desconhecíamos e que se mostraram ser muito interessantes e importantes, tanto nesta área como noutras, ganhando assim um conhecimento extra que certamente será útil no nosso futuro profissional.

1.3 | Organização do Documento

De modo a descrever fielmente o trabalho que foi feito, este documento encontra-se estruturado da seguinte forma:

1. O primeiro capítulo – **Introdução** – onde falámos sobre o projeto de um modo geral, referindo os seus objetivos e a sua organização.
2. O segundo capítulo – **Desenvolvimento e Implementação** – onde se falou sobre como se desenvolveu a aplicação e o que foi usado para desenvolver a mesma.
3. O terceiro capítulo – **Conclusões e Trabalho Futuro** – onde se falou sobre as aprendizagens que ficaram com o desenvolvimento deste projeto e o que pode ser feito no futuro para o melhorar.

2 / Desenvolvimento e Implementação

2.1 | Introdução

Neste capítulo será explicada toda a implementação da simulação do sistema solar. Iremos detalhar o máximo possível para que seja acessível a todos os tipos de conhecimento de CG, desde básicos a avançados.

Vamos falar sobre as dependências do projeto, explicar como tudo é inicializado e como o fluxo do sistema funciona.

2.2 | Demo

Na imagem abaixo conseguimos ter uma noção de como ficou o resultado final, com uma vista do sistema solar todo. É possível ver todos os planetas de forma distante e o sol a brilhar no centro.

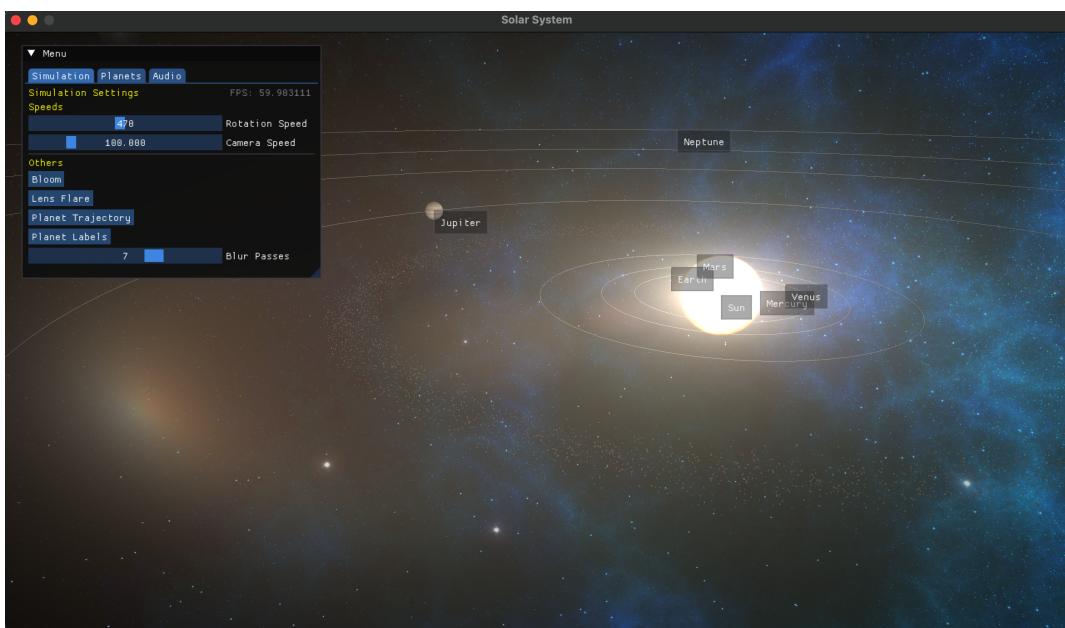


Figura 2.1: Resultado Final

Nas imagens seguintes, é possível observar que a terra tem várias texturas. Uma textura que dá à Terra nuvens que se movem "ao sabor do vento", uma textura para quando está de noite no planeta e se vêem as luzes e outra para quando está de dia.

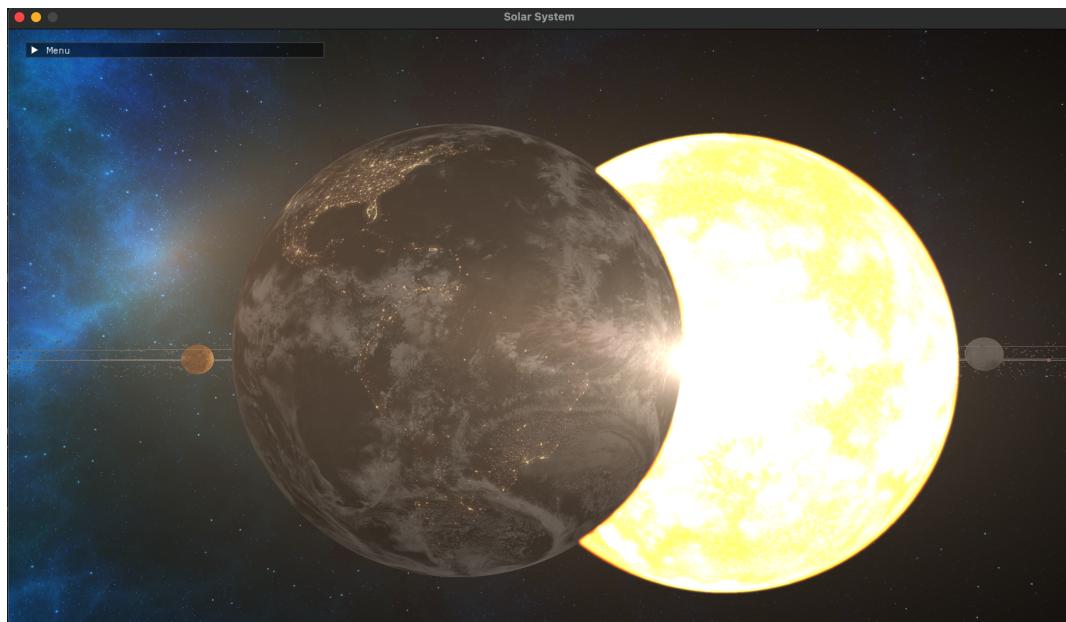


Figura 2.2: Terra de Noite

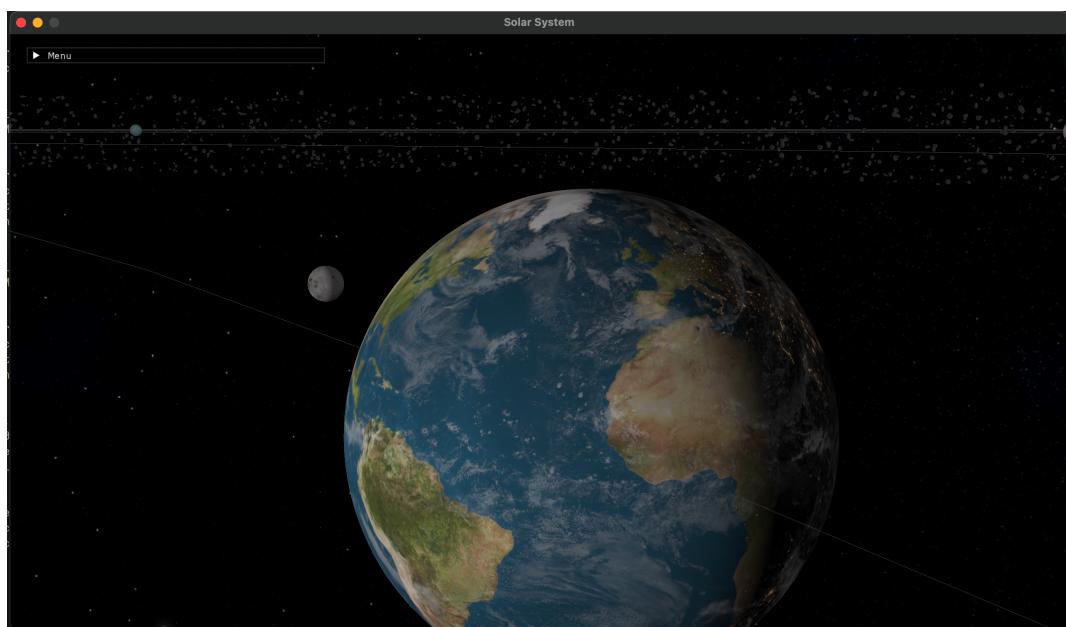


Figura 2.3: Terra com Núvens

2.3 | Dependências

Para a realização deste projeto, utilizamos várias bibliotecas externas. Algumas foram com aprendizagem por meio das aulas e com auxílio do professor e outras por estudo e aprendizado autónomo. Entre elas, encontramos bibliotecas para manipular o som, criar e controlar menus na *User Interface* (UI), e todo o sistema com *OpenGL*.

2.3.1 - IMGUI

Immediate Mode Graphic User Interface (IMGUI) [1] permite criar UI's para o nosso sistema *OpenGL* com muita facilidade. Possui uma comunidade ativa com muitos tutoriais e documentação *online*. Esta biblioteca é leve, eficiente e compatível com as mais diversas plataformas.

Ao utilizarmos esta biblioteca, conseguimos maior foco na resolução de problemas em vez de nos forçarmos em fazer os menus que não são o mais importante neste caso.

2.3.2 - assimp

Com o *Assimp* [2] conseguimos carregar ficheiros 3D com diferentes formatos de forma simples e rápida. A biblioteca cria uma estrutura de dados que consegue ser genérica e abstrata o suficiente para funcionar com muitos tipos de objetos 3D.

2.3.3 - stb_image.h

stb_image.h [3] é uma biblioteca que permite carregar texturas com facilidade, é *open source* como todas as outras bibliotecas que utilizamos e suporta várias plataformas.

2.3.4 - soloud

soloud [4] é uma biblioteca que permite o *playback* de áudios com facilidade, e é bastante utilizada em jogos e sistemas semelhantes por ser extremamente leve e simples de usar.

2.4 | Compilação

Para compilar todo o projeto, criamos um *script run* para compilar e executar o projeto. Mas este *script* nada mais é do que executar uma sequência de comandos com o *cmake*. O *CMakeLists.txt* é bastante complexo e por isso não o vamos mostrar.

Exerto de Código 2.1: Script Run

```
...
mkdir build && cd build
cmake .. && make
./solarsystem
```

Com isto, para executar o programa basta então executar o *script run*, fazendo `./run`.

2.5 | Detalhes de Implementação

2.5.1 - Inicialização

Vamos agora proceder à explicação do que é feito antes do *loop* principal da aplicação. Começamos por iniciar todos os *shaders*, guardar dados na *Graphics Processing Unit* (GPU), iniciar as configurações para algumas bibliotecas que usamos, fazer alguns cálculos, entre outras coisas.

Logo depois de criarmos a janela e definirmos algumas *callbacks* para tratarmos coisas como os *inputs* e o *scroll*. Definimos ainda o *Viewport* do sistema.

Exerto de Código 2.2: Inicialização da Janela

```
1 GLFWwindow *window = glfwCreateWindow(WIDTH, HEIGHT, "Solar System", nullptr, nullptr);
2 glfwViewport(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);
3 glfwSetKeyCallback(window, KeyCallback);
4 glfwSetCursorPosCallback(window, MouseCallback);
5 glfwSetScrollCallback(window, scroll_callback);
```

Depois iniciamos a nossa configuração para a biblioteca **IMGUI** que nos permite criar interfaces gráficas para o sistema com maior facilidade.

Exerto de Código 2.3: Inicialização do *IMGUI*

```
1 IMGUI_CHECKVERSION();
2 ImGui::CreateContext();
3 ImGuiIO &io = ImGui::GetIO();
4 io.ConfigFlags |=
5     ImGuiConfigFlags_NavEnableKeyboard;
```

De seguida, começamos a carregar todos os *shaders*, os modelos e as texturas necessárias para todo o sistema. Utilizamos classes para carregar e manipular as mesmas. Segue um pequeno exemplo:

Exerto de Código 2.4: Carregamento das *shaders*, modelos e texturas

```

1 Shader shader("resources/shaders/modelLoading.vs",
2         "resources/shaders/modelLoading.frag");
3 Model earthModel("resources/models/earth/earth.obj");
4 unsigned int earthNightTextureID =
5     TextureFromFile("resources/models/earth/earthnight.jpg", ".");

```

As classes ***Shader*** e ***Model*** são provenientes do código disponível no *GitHub* do *LearnOpenGL* [5] e que está explicado no site [6].

Criamos também uma *struct* para trabalharmos com as *hitboxes* dos planetas e desligarmos o *lens flare* caso, ao lançar um *ray-cast* desde a câmera, este colide com algum planeta no seu caminho até ao sol.

Exerto de Código 2.5: Colisões parte 1

```

1 // definir a estrutura
2 struct Sphere {
3     glm::vec3 center;
4     float radius;
5 };
6
7 // Funcao para criar a estrutura sphere
8 Sphere createSphere(float radius, glm::vec3 position) {
9     Sphere sphere;
10    sphere.center = position;
11    sphere.radius = radius;
12    return sphere;
13 }
14
15 Sphere sunSphere = createSphere(sunRadius, lightPos);
16 Sphere earthSphere =
17     createSphere(earthRadius, glm::vec3(0.0f, 0.0f, 1.0f * AU));

```

Para os asteróides, antes do *loop*, calculamos todas as coordenadas e desenhamos os mesmos.

Exerto de Código 2.6: Colisões parte 2

```

1 glm::mat4 model = glm::mat4(1.0f);
2 // 1. translation: displace along circle with 'radius' in range [-offset,
3 // offset]
4 float angle = (floati / (float)amount * 360.0f;
5 float displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
6 float x = sin(angle) * asteroidRadius + displacement;
7 displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;

```

```

8 float y = displacement * 0.4f; // keep height of asteroid field smaller
9                                     // compared to width of x and z
10 displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
11 float z = cos(angle) * asteroidRadius + displacement;
12 model = glm::translate(model, glm::vec3(x, y, z));
13
14 // 2. scale: Scale between 0.05 and 0.50f
15 float scale = static_cast<float>((rand() % 20) / 50.0 + 0.05);
16 model = glm::scale(model, glm::vec3(scale));
17
18 // 3. rotation: add random rotation around a (semi)randomly picked rotation
19 // axis vector
20 float rotAngle = static_cast<float>((rand() % 360));
21 model = glm::rotate(model, rotAngle, glm::vec3(0.4f, 0.6f, 0.8f));

```

Fazemos isso para cada asteróide que vamos colocar. O número de asteróides é pré-definido e fixo ao longo de todo o sistema. Logo depois de guardarmos os dados provenientes dos cálculos aplicados vamos colocar tudo num *buffer* da GPU e vamos também carregar as *meshes* dos asteróides.

O resultado dos asteróides é mostrado abaixo.

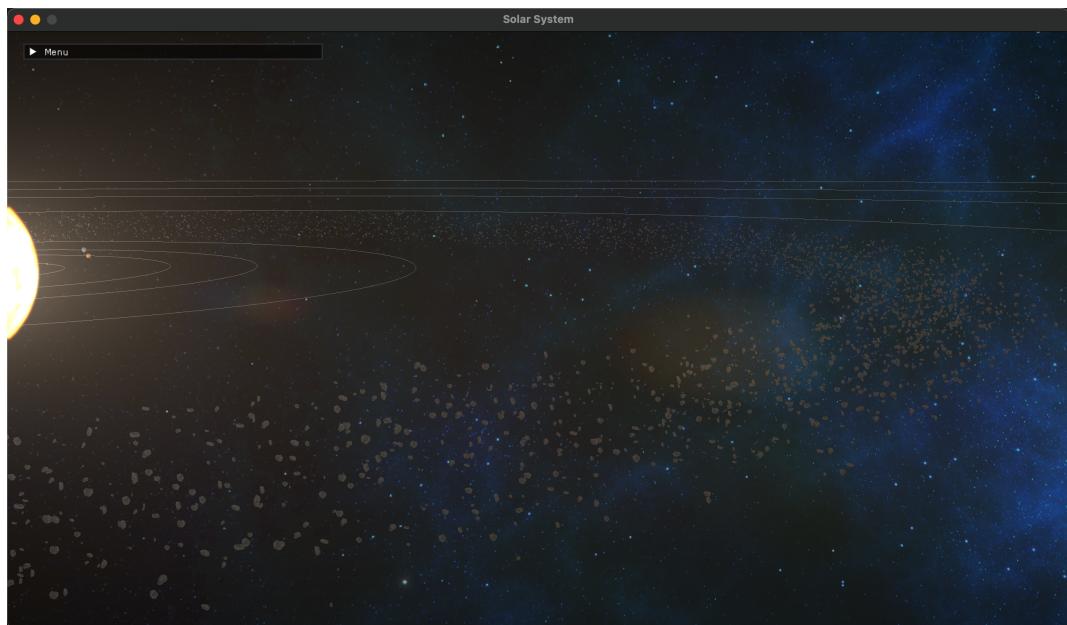


Figura 2.4: Asteróides

Iniciamos ainda os *cube maps* que serão explicados em detalhe no capítulo 2.5.3. Ainda antes de iniciarmos o *loop* do sistema, inicializamos o nosso áudio, referido no capítulo 2.5.7.

2.5.2 - Loop Principal

O *loop* principal vai executar infinitamente ou até receber o comando de parar que é dado ao pressionarmos a tecla *ESQ* ou o botão de fechar a janela.

A cada iteração desenhamos tudo o que vemos, tratamos o *input*, etc. Fazemos alguns cálculos e guardamos estados, como por exemplo o *deltaTime*.

Excerto de Código 2.7: *DeltaTime*

```
1 GLfloat currentFrame = glfwGetTime();  
2 deltaTime = currentFrame - lastFrame;
```

Anteriormente, definimos as *callbacks* para os *inputs* e guardamos todas as teclas pressionadas num *array*. Agora vamos utilizar isso para tratar as diferentes ações que podemos realizar. Essas ações incluem trocar entre mover a câmera ou utilizar os menus.

A cada *frame* é desenhado o menu lateral que permite controlar e mostrar dados relativos ao programa e até ao sistema solar que estamos a simular.



Figura 2.5: Menu Lateral

Na imagem a cima, é possível verificar que conseguimos controlar velocidades, ligar e desligar opções como o *lens flare* e controlar o nível de *blur*. Para trocarmos entre utilizar a câmera e o menu só precisamos de pressionar a tecla **P**.

Incluimos também menus com informações sobre cada planeta, para que não faltam informações sobre aquilo que estamos a ver.



Figura 2.6: Menu de Informações

Fazemos as *labels* dos planetas com o auxílio dos menus do IMGUI. Podemos ativar e desativar com o menu, ou pressionando a letra L.

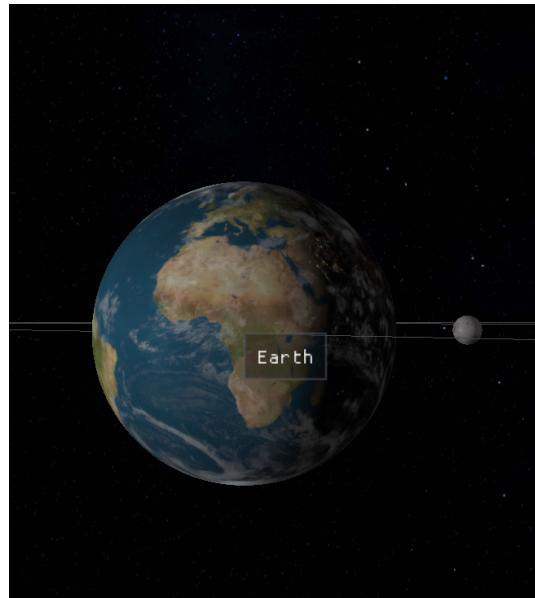


Figura 2.7: *Labels*

2.5.3 - *Cube Maps*

Uma *skybox* é um método de criação de imagens de fundo que faz parecer com que o mundo (neste caso, galáxia) pareça maior do que realmente é. Esta cria um cubóide à volta do mundo e usa seis texturas para cada lado/parede. Ao mexermos-nos em cena, essas paredes não aparecerão mudar de sítio como aconteceria se apenas criássemos um cubo com as texturas das paredes por dentro.

Excerto de Código 2.8: *Cubemap* para a *Skybox*

```
1 void system() {  
2     ...  
3     float skyboxVertices[] = {  
4         // positions  
5         -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,  
6         1.0f, -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f,  
7         -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f,  
8         -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, 1.0f,  
9         1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f,  
10        1.0f, 1.0f, 1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f,  
11        -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f,  
12        1.0f, 1.0f, 1.0f, 1.0f, -1.0f, 1.0f, -1.0f, -1.0f,  
13        -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f,  
14        1.0f, 1.0f, 1.0f, 1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f,  
15        1.0f, 1.0f, 1.0f, 1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f,  
16    }
```

```
17      -1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, 1.0f,
18      1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f,
19
20      -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 1.0f, -1.0f,
21      1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, -1.0f};
22
23  /* SKYBOX GENERATION */
24  unsigned int skyboxVAO, skyboxVBO;
25  glGenVertexArrays(1, &skyboxVAO);
26  glGenBuffers(1, &skyboxVBO);
27  glBindVertexArray(skyboxVAO);
28  glBindBuffer(GL_ARRAY_BUFFER, skyboxVBO);
29  glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices), &skyboxVertices,
30             GL_STATIC_DRAW);
31  glEnableVertexAttribArray(0);
32  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void *)0);
33  /* SKYBOX GENERATION */
34
35  while(!glfwWindowShouldClose()) {
36      ... // depois de renderizar tudo o resto
37
38  /* DRAW SKYBOX */
39  glDepthFunc(GL_LESS);
40  skyboxShader.use();
41  view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
42  skyboxShader.setMat4("view", view);
43  skyboxShader.setMat4("projection", projection);
44  // skybox cube
45  glBindVertexArray(skyboxVAO);
46  glActiveTexture(GL_TEXTURE0);
47  glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
48  glDrawArrays(GL_TRIANGLES, 0, 36);
49  glBindVertexArray(0);
50  glDepthFunc(GL_LESS);
51  /* DRAW SKYBOX */
52
53  ...
54 }
55 }
```

O resultado obtido é o seguinte.

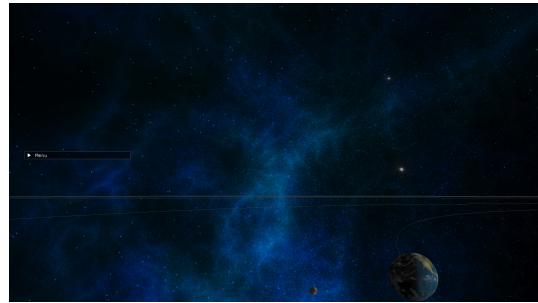


Figura 2.8: *Skybox*

2.5.4 - *Post-processing*

Para as técnicas de *Post-processing* que vamos explicar a seguir tivemos que adicionar algumas coisas à nossa simulação:

1. *Framebuffer*: O *framebuffer* é uma porção da *Random-Access Memory* (RAM) onde o *OpenGL* armazena os pixels que vão ser escritos no ecrã, ou seja, se tivermos acesso a ele podemos mudar os pixels para obter os efeitos que desejamos (*Post-processing*). Como o *framebuffer* padrão do *OpenGL* não é manipulável, criámos um onde renderizamos a nossa cena, aplicamos os efeitos, e escrevemos no mesmo um retângulo do tamanho do nosso *display*.
2. *High Dynamic Range* (HDR): Por defeito os valores de cor dos pixels no *framebuffer* estão limitados entre 0.0 e 1.0, isso faz com que os pixels com muito brilho percam definição por estarem limitados a 1.0. Ao passarmos o nosso *framebuffer* de GL_RGB para GL_RGBA16F, obtemos pixels com muita mais capacidade de representação dessa gama que antes estaria cortada. No entanto, normalmente os nossos *displays* são *Low Dynamic Range* (LDR), por esse motivo têm essas mesmas limitações de cor e o resultado vai ser igual, mesmo tendo mais informação para representar os pixels. Para mudar isto, precisamos de fazer o chamado *Tone-mapping*.
3. *Tone-mapping*: É o processo de transformar números de precisão *float* no limite [0.0; 1.0] esperado sem perder muito detalhe. Para usar este processo decidimos usar uma implementação com *exposure* e *gamma correction*.

Exerto de Código 2.9: *Fragment Shader* do *framebuffer* para a *exposure*

```

1 void main() {
2     ...
3     const float adjSpeed = 0.05;
4     float targetExposure = 0.5 / lum * 3.0f;
5     float sceneExposure = mix(exposure, targetExposure, adjSpeed);
6
7     sceneExposure = clamp(exposure, 0.3f, 3.0f);
8
9     vec3 mapped = vec3(1.0) - exp(-col * sceneExposure);
10    mapped = pow(mapped, vec3(1.0 / gamma));
11    ...
12 }
```

2.5.5 - Bloom

O *bloom* é um efeito que nos permite dar o efeito de uma luz muito brilhante numa cena, algo que é difícil de passar ao visualizador pelas limitações de cor anteriormente faladas. A ideia é extraír as partes mais brilhantes da cena, desfocá-las, e combinar esse resultado com a imagem original.

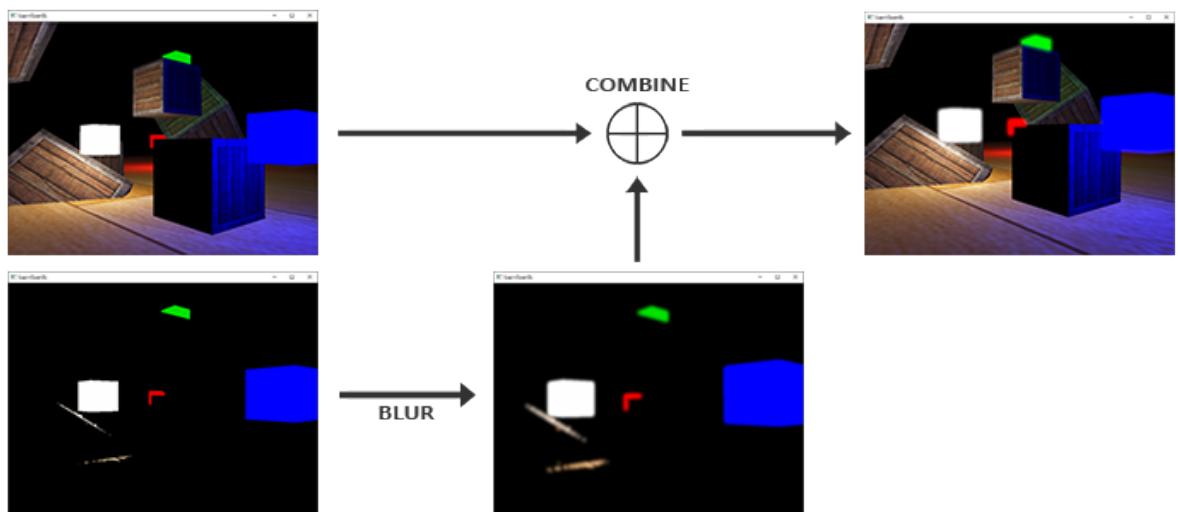


Figura 2.9: Passos do *Bloom*

Como temos a nossa imagem em HDR, podemos extraír as partes brilhantes da imagem facilmente:

Excerto de Código 2.10: *Fragment shader do bloom*

```

1 // lamp.frag
2 layout (location = 0) out vec4 FragColor;
3 layout (location = 1) out vec4 BrightColor;
4 ...
5 void main() {
6     color = texture(texture_diffuse, TexCoords) * sunIntensity;
7     FragColor = color;
8
9     float brightness = dot(color.rgb, vec3(0.2126, 0.7152, 0.0722));
10    if(brightness > 1.0)
11        BrightColor = vec4(color.rgb, 1.0);
12    else
13        BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
14 }
```

Para armazenarmos essa imagem nova, temos de informar o *OpenGL* que queremos escrever em dois *colorbuffers* no nosso *framebuffer*:

Excerto de Código 2.11: Escrita em 2 *colorbuffers* diferentes

```

1 void system()
2 ...
3 unsigned int textureColorbuffer;
4 glGenTextures(1, &textureColorbuffer);
5 glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
6 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCREEN_WIDTH, SCREEN_HEIGHT, 0,
7             GL_RGB, GL_FLOAT, NULL);
8 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
9 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
10 glBindFramebuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
11                   textureColorbuffer, 0);
12
13 unsigned int bloomTexture;
14 glGenTextures(1, &bloomTexture);
15 glBindTexture(GL_TEXTURE_2D, bloomTexture);
16 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCREEN_WIDTH, SCREEN_HEIGHT, 0,
17             GL_RGB, GL_FLOAT, NULL);
18 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
19 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
20 glBindFramebuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D,
```

```
21         bloomTexture, 0);  
22  
23     ...  
24  
25     unsigned int attachments[2] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1};  
26     glDrawBuffers(2, attachments);  
27     ...  
28 }
```

A nossa imagem com pontos brilhantes passa assim a estar no GL_COLOR_ATTACHMENT1 e a imagem original no GL_COLOR_ATTACHMENT0. O *FragColor* no *shader* anterior está a ser escrito no GL_COLOR_ATTACHMENT0 e, por isso, na textura *textureColorbuffer* enquanto o *BrightColor* está a ser escrito no GL_COLOR_ATTACHMENT1 e, por isso, na textura *bloomTexture*.

Aplicamos de seguida um filtro de **desfocagem Gaussiana**, que consiste em desfocar horizontalmente e depois verticalmente várias vezes seguidas, com pesos diferentes a cada iteração. Feito isto combinamos os *colorbuffers* **zero** e **um**, que são passados como texturas no *Fragment Shader* do nosso *framebuffer* e temos o efeito desejado.

Excerto de Código 2.12: *Fragment shader* do *framebuffer*

```
1 // framebuffer.frag  
2 uniform sampler2D screenTexture;  
3 uniform sampler2D bloomBlur;  
4  
5 ...  
6  
7 void main() {  
8     vec3 col = texture(screenTexture, TexCoords).rgb;  
9     vec4 bloomTex = texture(bloomBlur, TexCoords);  
10    vec3 bloomColor = bloomTex.rgb;  
11  
12    if (bloomActive)  
13        col += bloomColor;  
14  
15    ...  
16 }
```

2.5.6 - *Lens Flare*

Um *lens flare* é um efeito físico das câmeras reais que acontece quando uma luz intensa atinge o sistema de lentes de vidro que constitui a câmera. Este efeito estilístico faz com que a simulação pareça mais real, pois parece que estamos a ver algo gravado numa câmera verdadeira.

O *lens flare* é calculado em *real-time* e não usa nenhuma textura convencional, apenas uma combinação de círculos com aberração cromática, junto de uma textura de ruído para um efeito mais aleatório.

A *shader* usada foi adaptada deste *lens flare* [7].

Para conseguir o efeito em que o *lens flare* é obstruído por um planeta a passar entre a câmera e o sol, usámos o seguinte método:

1. Criar uma *hitbox* para cada planeta.
2. Fazer um *ray-cast* desde a câmera até ao centro do sol.
3. Se este raio intercetar algum planeta, o *lens flare* é desligado.
4. Quanto mais longe o planeta estiver da câmera, menor será o seu raio de interseção.

Excerto de Código 2.13: Verificação da interseção do *ray-cast*

```

1 bool isIntersecting(glm::vec3 rayOrigin, glm::vec3 rayDirection,
2                     const Sphere &sphere, float correction = 1.0f) {
3
4     float cameraToPlanetLength = glm::length(sphere.center - camera.Position);
5     float cutoff = cameraToPlanetLength / AU * correction;
6     glm::vec3 oc = rayOrigin - sphere.center;
7     float a = glm::dot(rayDirection, rayDirection);
8     float b = 2.0f * glm::dot(oc, rayDirection);
9     float c = glm::dot(oc, oc) -
10        (glm::clamp(sphere.radius - cutoff, 0.0f, sphere.radius)) *
11        (glm::clamp(sphere.radius - cutoff, 0.0f, sphere.radius));
12     float discriminant = b * b - 4 * a * c;
13
14     if (discriminant > 0) {
15         float t1 = (-b - sqrt(discriminant)) / (2.0f * a);
16         float t2 = (-b + sqrt(discriminant)) / (2.0f * a);
17
18         if ((t1 >= 0.0f) || (t2 >= 0.0f)) {
19             return true;
20         }
21     }
22 }
```

```

21     }
22     return false;
23 }
24
25 ...
26 void system() {
27 ...
28     while(!glfwWindowShouldClose()) {
29         ...
30         glm::vec3 rayDirection = glm::normalize(sunSphere.center – camera.Position);
31         float rayLength = glm::length(sunSphere.center – camera.Position);
32         glm::vec3 rayEndPoint = camera.Position + rayDirection * rayLength;
33         if (lensFlareActive) {
34             bool sunVisible =
35                 !isIntersecting(camera.Position, rayDirection, mercurySphere,
36                                 10.0f) &&
37                 !isIntersecting(camera.Position, rayDirection, venusSphere, 2.0f) &&
38                 !isIntersecting(camera.Position, rayDirection, earthSphere, 2.0f) &&
39                 !isIntersecting(camera.Position, rayDirection, marsSphere) &&
40                 !isIntersecting(camera.Position, rayDirection, jupiterSphere, 1.5f) &&
41                 !isIntersecting(camera.Position, rayDirection, saturnSphere) &&
42                 !isIntersecting(camera.Position, rayDirection, uranusSphere) &&
43                 !isIntersecting(camera.Position, rayDirection, neptuneSphere);
44             screenShader.setBool("sunVisibleAndEnabled", sunVisible);
45         } else {
46             screenShader.setBool("sunVisibleAndEnabled", false);
47         }
48 ...
49     }
50 ...
51 }
```

2.5.7 - Música de Fundo

Foi usada a biblioteca ***sould***, que nos permitiu facilmente adicionar música à nossa simulação. Há uma *thread* dedicada para esse efeito.

Excerto de Código 2.14: *Thread* da Música

```
1 std::thread audioThread(&initializeMiniaudio);
```

Três sons são escolhidos aleatoriamente ao iniciar o sistema. Podemos trocar os mesmos por via do menu principal, e baixar ou aumentar o seu volume.

2.6 | Conclusão

Concluimos assim a implementação da nossa simulação do sistema solar. Consideramos ter um sistema completo, tendo a terra com a sua lua, muitos asteróides espalhados, fundo no universo, luz solar que atinge a câmera e permite gerar um efeito mais realista e tendo também vários menus que permitem manipular o sistema em tempo de execução e ler informações sobre os diversos planetas.

3 / Conclusões e Trabalho Futuro

3.1 | Conclusões Principais

Dando por terminado o desenvolvimento deste projeto, concluímos que conseguimos alcançar todos os nossos objetivos, obtendo assim uma simulação realista do nosso sistema solar. Refletindo sobre tudo o que foi feito ao longo destes meses de trabalho, percebemos o quanto importante é a área da CG e o quanto importante foi este projeto para a nossa aprendizagem, já que com ele conseguimos aprender muito para além do que foi possível nas aulas. Concluímos também que fizemos um excelente trabalho, com uma boa organização e divisão do trabalho entre todos os elementos do grupo, estando muito orgulhosos do resultado final.

3.2 | Trabalho Futuro

Neste projeto, devido a problemas de falta de tempo livre para investirmos mais no projeto, acabamos por não ter tempo para realizar a renderização de texto com auxílio de *bitmaps* ou *billboards*, mas ficando assim como um objetivo futuro para aprimoramento do projeto. Para suprir a falta disto, criámos labels por via de menus com ajuda da biblioteca IMGUI.

Para além disso, seria bastante interessante adicionar à nossa simulação outros planetas e galáxias distantes, ou até mesmo simular buracos negros e "brincar" com a gravidade e todas as leis da física que conhecemos hoje.

Bibliografia

- [1] IMGUI. <https://github.com/ocornut/imgui>. [Online] Último acesso a 4 de Janeiro de 2023.
- [2] assimp. <https://github.com/assimp/assimp>. [Online] Último acesso a 4 de Janeiro de 2023.
- [3] nothings/stb. <https://github.com/nothings/stb>. [Online] Último acesso a 4 de Janeiro de 2023.
- [4] Jari Komppa. soloud. <https://github.com/jarikomppa/soloud>. [Online] Último acesso a 4 de Janeiro de 2023.
- [5] Joey De Vries. JoeyDeVries/LearnOpenGL. <https://github.com/JoeyDeVries/LearnOpenGL>. [Online] Último acesso a 4 de Janeiro de 2023.
- [6] Joey De Vries. LearnOpenGL, 2020. [Online] <https://learnopengl.com/>. Último acesso a 4 de Janeiro de 2023.
- [7] Shadertoy. <https://www.shadertoy.com/>. [Online] Último acesso a 4 de Janeiro de 2023.