

Maio 19, 2024

Relatório CG - Fase 4 - Grupo 24

Guilherme Barbosa (a100695)

Ivan Ribeiro (a100538)

Nuno Aguiar (a100480)

Pedro Ferreira (a100709)

Sumário

| | |
|---|----|
| 1) Introdução | 3 |
| 2) Mudanças em relação a fases anterior | 3 |
| 3) XMLParser | 5 |
| 4) Texturas e normais | 7 |
| 4.1) Plano | 7 |
| 4.2) Caixa | 8 |
| 4.3) Esfera | 9 |
| 4.4) Cone | 11 |
| 4.5) Cilindro | 13 |
| 4.6) Torus | 15 |
| 5) Shaders | 16 |
| 5.1) Iluminação | 17 |
| 5.1.1) Materiais e texturas | 17 |
| 5.1.2) Texturas | 17 |
| 5.1.3) Vertex shader | 17 |
| 5.1.4) Geometry shader | 18 |
| 5.1.5) Fragment shader | 19 |
| 5.2) Bloom, HDR e tone mapping | 22 |
| 6) Conclusão | 28 |

1) Introdução

Neste documento é caracterizado o processo de desenvolvimento da fase 4 do projeto da Unidade Curricular de Computação Gráfica. São descritas as mudanças realizadas no XML Parser, a aplicação das texturas e normais nas diversas primitivas, a iluminação da cena e as componentes de cor introduzidas e o shader desenvolvido.

2) Mudanças em relação a fases anterior

Nesta seção gostaríamos de referir as mudanças que ocorreram em relação às fases, falando em específico das superfícies de Bezier. Inicialmente estávamos a realizar a interpolação manualmente, para cada par de valores (u, v) e a desenhar os respetivos pontos. Até aqui estava tudo bem, porém, quando começamos a implementar as normais das superfícies, deparamo-nos com a dificuldade de as calcular. Inicialmente, o grupo fez uma pesquisa intensiva para resolver o impasse, chegando mesmo a implementar uma fórmula. O problema surgiu do facto de nenhum dos membros realmente compreender como os cálculos funcionavam e, em certo ponto, repararam que existiam casos em que o resultado estava mesmo mal.

Após uma conversa com a equipa docente, foi decidido que usariamos a fórmula lecionada em aula para resolver esta adversidade. Assim, para x, y e z , obtemos os valores do ponto através de multiplicações matriciais e, mudando apenas uma matriz em cada caso, obtemos a derivada parcial da superfície para u e v , sendo estes vetores usados para calcular a normal da superfície no ponto.

Após as mudanças referidas, a implementação destas superfícies tem o seguinte aspeto:

```
points = []

for patch in bezier_patches:
    points.append(dividePatch(patch, divisions))

def dividePatch(patch, divisions):
    ans = []
    prevPoints = []
    curPoints = []

    for i in range(0, divisions + 1):
        v = i / divisions

        curPoints = []

        for j in range(0, divisions + 1):
            u = j / divisions

            curPoints.append(calculatePatchPoint(patch, v, u))

    # Produce Triangles
    if i > 0:
        for j in range(0, divisions):
            curV = curPoints[j];
            rightV = curPoints[j+1];
            rightDownV = prevPoints[j+1];
            downV = prevPoints[j];

            ans.append(rightDownV);
            ans.append(rightV);
```

```

        ans.append(curV);

        ans.append(curV);
        ans.append(downV);
        ans.append(rightDownV);

    prevPoints = curPoints

    return ans

def calculatePatchPoint(patch, v, u):
    coords = []
    text = [1 - u, 1 - v];
    normal = [];

    uM = [powf(u, 3), powf(u, 2), u, 1]
    uMDeriv = [3 * powf(u, 2), 2 * u, 1, 0]

    vM = [powf(v, 3), powf(v, 2), v, 1]
    vMDeriv = [3 * powf(v, 2), 2 * v, 1, 0]

    # bezierCoefficients returns the bezier coefficients' matrix
    bezierM = bezierCoefficients();
    bezierMT = transpose(bezierM);

    uVector;
    vVector;

    # i = 0 -> x
    # i = 1 -> y
    # i = 2 -> z
    for i in range(0, 3):

        # Calc point coords
        coords.append(interpolateCoords(patch, bezierM, bezierMT, uM, vM, i))

        # Calc u partial derivate coords
        uVector.append(interpolateCoords(patch, bezierM, bezierMT, uMDeriv, vM, i))

        # Calc v partial derivate coords
        vVector.append(interpolateCoords(patch, bezierM, bezierMT, uM, vMDeriv, i))

    # Indicates that coords represents a point instead of a vector
    coords.append(1)

    normal = cross(uVector, vVector)

    return Vertex(coords, normal, text)

def interpolateCoords(patch, bezierCoef, bezierCoefT, uV, vV, coord)
    points = []
    ans = 0;
    tmp;

    for i in range(0, len(patch)):
        points[i/4][i%4] = patch[i].coords[coord];

```

```

tmp = uM * bezierM * points * bezierMT;

# Since glm can multiply the different elements, we need to manually do it
for i in range(0, 4):
    ans += tmp[i] * vM[i];

return ans

```

3) XMLParser

A nível deste componente, as mudanças refletiram-se na adição do reconhecimento das tags de luz, de cores e de textura. Começando pelas luzes, o grupo decidiu adicionar alguma informação extra além da fornecida nos testes base. Primeiro, para permitir que a luz tenha diferentes tons, o grupo adicionou, dentro da tag light, o parsing das seguintes tags (o formato apresentado será tag_name(attribute_list)):

- diffuse(R, G, B)
- ambient(R, G, B)
- specular(R, G, B)

Depois, para permitir o maior controlo alguns parâmetros de cada tipo de luz o grupo decidiu adicionar os seguintes atributos:

- na luz oriunda de um ponto é possível especificar o valor das variáveis constant, linear e quadratic
- na luz focal podemos especificar o valor das variáveis outerCutOff, constant, linear e quadratic

Além destes atributos, também foi realizado o parse dos atributos base indicados pela equipa docente.

A nível do processamento das cores e texturas, o grupo assume que existe uma cor padrão, neste caso a indicada pela equipa docente, e o processamento realizado refere-se somente aos valores indicados pela mesma. Assim, o aspeto final do XML Parser é:

```

xml_doc = TinyXML2.read(xml_file)

# Base xml tags in each document
window_config = xml_doc.child_tag("window")
camera_config = xml_doc.child_tag("camera")
group = xml_doc.child_tag("groups")
lights_tag = xml_doc.child_tag("lights")

# Read group tags (Transformation é uma abstração de uma matrix 4x4 'mat4' provida pelo
# glm)
engine_obj = parseEngineObject(group, Transformation());

# lights is an object that has 3 components: Spot lights, Point lights and Directional
# Lights
lights = parseLights(lights_tag)

def parseEngineObject(group : XML_Tag, transformation : Transformation) -> EngineObject
    obj_points = [] # each element is an array of vertices
    child_objects = []

```

```

for tag in group:
    if tag == "transformation":
        # read and construct matrix for rotation/translation/scale, appending it to group
        transformations.append(parseTransformation(tag))

    if tag == "models":
        obj_points.append(parseModels(tag))

    if tag == group:
        child_objects.append(parseEngineObject(tag, Transformation()))

return EngineObject(transformation, obj_points, child_objects)

def parseModels(models):
    vertices = []

    for model in models:
        # read file indicated by model and retrieve its vertices
        vertices.append(parseVertex(model))

    return vertices

def parseTransformation(transformTag):

    if transformTag == translate:
        t = Translate(translateTime, alignCurve, translateVector, isCurve, controlPoints,
                      upVector, alignVector)

    else if transformTag == rotate:
        t = Rotate(angle, time, rotationVector)

    else:
        t = Scale(scaleVector)

    return t

def parseLights(lights):
    spot_lights = []
    point_lights = []
    directional_lights = []

    if lights == None:
        # Each of this functions has a default value for when it receives None
        spot_lights.append(parseSpotLight(None))
        point_lights.append(parsePointLight(None))
        directional_lights.append(parseDirectionalLight(None))

    else:
        for l in lights:
            if l.name == "spot":
                spot_lights.append(parseSpotLight(l))
            elif l.name == "point":

```

```

        point_lights.append(parsePointLight(l))
    elif l.name == "directional":
        directional_lights.append(parseDirectionalLight(l))
    else:
        error("unrecognized tag")

    return (spot_lights, point_lights, directional_lights)

```

4) Texturas e normais

Para além da informação que já constava nos vértices das várias primitivas, nesta fase introduzimos coordenadas de texturas (as texturas devem estar localizadas na pasta /texture) e normais, sendo estas últimas essenciais na iluminação dos objetos. De seguida, descrevemos o processo seguido para cada uma das primitivas.

4.1) Plano

Como todos os pontos da primitiva pertencem a um mesmo plano, horizontal e voltado para cima, as normais dos pontos serão um vetor normalizado voltado para o semi-eixo positivo y (cima), ou seja, (0,1,0). Por sua vez, as coordenadas de textura dependem da coluna e linha respetiva em que cada ponto se encontra no plano. Dividimos as coordenadas de textura pelo tamanho do plano, para ficarem num intervalo de [0,1]. Invertemos os valores das coordenadas de textura v com (1 - valor), porque coordenadas de texturas começam em baixo à esquerda de textura.

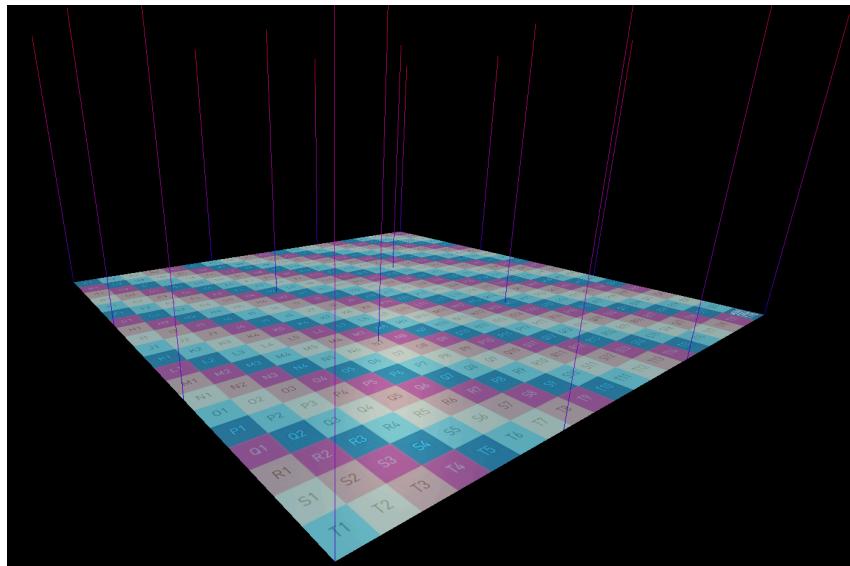


Figura 1: Textura e normais do plano

Tomando por base o pseudo-código apresentado no relatório da fase 1, temos então:

```

func plane (length, divisions) {
    points = [] // pontos intermédios que a estrutura utiliza

    step = length / divisions // salto vertical/horizontal entre pontos
    shift = length / 2 // translação para centrar pontos de plano
    for col in divisions:
        for line in divisions:
            // posição
            x = col * step - shift

```

```

y = 0
z = line * step - shift
// normal
nx = 0
ny = 1
nz = 0
// textura
tu = (col * step) / length
tv = 1 - ((line * step) / length)

points_add(x, y, z, nx, ny, nz, tu, tv)

// com os pontos da primitiva, construímos os triângulos de forma idêntica à 1ª fase
build_plane()
}

```

4.2) Caixa

Como descrito na fase 1, a caixa é construída com base na primitiva plano, aplicando transformações para obter cada face na posição correta. Para cada um dos planos, definimos as normais dos pontos dessa face com base na sua orientação. Para as texturas, mantivemos as coordenadas de textura que os pontos de cada plano já possuíam, tendo em alguns caso, de ajustar para a textura apresentar a orientação correta na respetiva face do cubo, após a transformação aplicada. Não fizemos um atlas de textura, todas as faces têm a mesma textura.

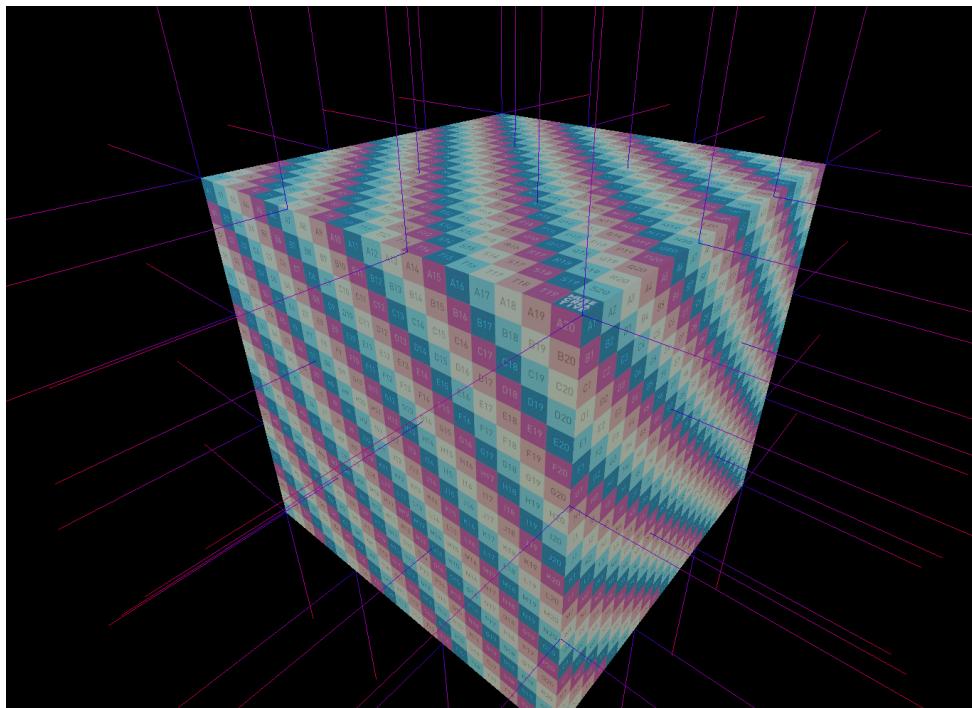


Figura 2: Textura e normais da caixa

Tomando por base o pseudo-código apresentado no relatório da fase 1, temos então:

```

func box (length, divisions) {
    triangles = [] // array that accumulates resulting vertices

    planeFacingUp =[vertex] // plane with triangles facing upwards, points are all x>=0,
    y=0, z>=0

```

```

planeFacingDown = reverse of planeFacingUp // plane with triangles facing downwards,
points are all x>=0, y=0, z>=0

shiftVector = (length/2, length/2, length/2) // vector to center cube

translateVector_x = (length, 0, 0)
translateVector_y = (0, length, 0)
translateVector_z = (0, 0, length)

// Down face
downVertices = planeFacingDown - shiftVector
downVertices.normal = (0,-1,0)
triangles_add(downVertices)

// Left face
leftVertices = rotate_z_matrix(90) * planeFacingUp - shiftVector
leftVertices.normal = (-1,0,0)
leftVertices.tex_coords = (1 - leftVertices.tv, 1 - leftVertices.tu)
triangles_add(leftVertices)

// Right face
rightVertices = rotate_z_matrix(90) * planeFacingDown -
    shiftVector + translateVector_x
rightVertices.normal = (1,0,0)
rightVertices.tex_coords = (1 - rightVertices.tv, rightVertices.tu)
triangles_add(rightVertices)

// Back face
backVertices = rotate_x_matrix(-90) * planeFacingUp - shiftVector
backVertices.normal = (0,0,-1)
backVertices.tex_coords = (backVertices.tu, 1 - backVertices.tv)
triangles_add(backVertices)

// Front face
translateVector = (0,0,length)
frontVertices = rotate_x_matrix(-90) * planeFacingDown -
    shiftVector + translateVector_z
frontVertices.normal = (0,0,1)
triangles_add(frontVertices)

// Up face
translateVector = (0,length,0)
upVertices = planeFacingUp - shiftVector + translateVector_y
upVertices.normal = (0,1,0)
upVertices.tex_coords = (1 - upVertices.tu, upVertices.tv)
triangles_add(upVertices)

return triangles
}

```

4.3) Esfera

Com referido na fase 1, a construção da esfera envolve construir triângulos entre pontos de camadas horizontais consecutivas da esfera. Na construção das coordenadas de textura, para evitar que no fim de cada linha, os pontos do fim da linha com coordenada de tu = 0.9 conectassem com os pontos do início da mesma linha com coordenada tu = 0.0, o que provocava a

renderização da textura inteira invertida nessa secção, no final da linha substituímos o valor de $tu = 0.0$ para 1.0 nos respetivos pontos, para a renderização ocorrer corretamente. Como a esfera é construída de cima para baixo, invertemos o valor de v com (1-valor).

Para a construção das normais, consideramos o vetor formado entre a origem da esfera $(0,0,0)$ e as coordenadas geométricas de cada ponto para obter a normal de cada ponto. Este vetor é multiplicado pelo fator $(1 / \text{raio})$, para ser normalizado.

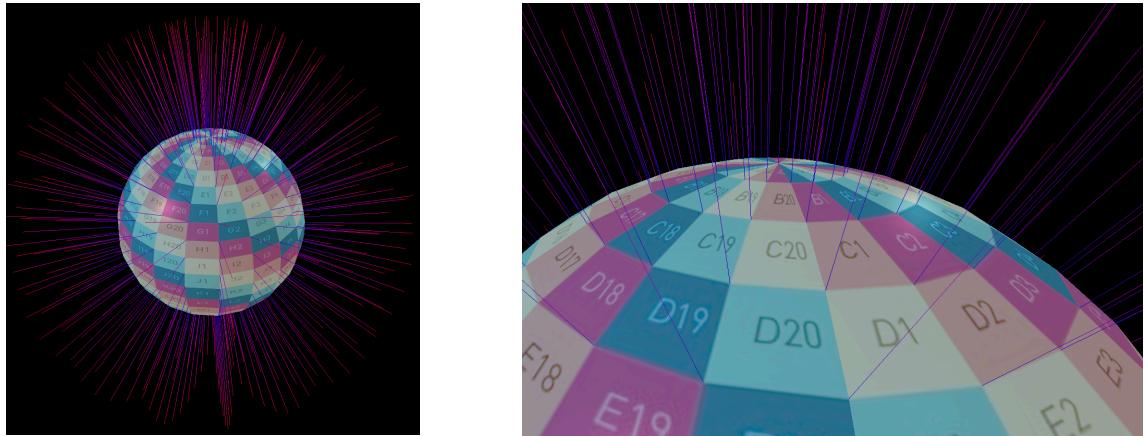


Figura 3: Textura e normais de esfera

Tomando por base o pseudo-código apresentado no relatório da fase 1, temos então:

```
func sphere (radius, slices, stacks) {
    points = [] // pontos intermédios que a estrutura utiliza
    triangles = [] // pontos finais que constituem a estrutura

    sector_step = 2PI / slices
    stack_step = PI / stacks
    norm_factor = 1 / radius

    // obter pontos que constituem a esfera
    for stack in stacks:
        curr_phi = PI / 2 - stack_step * stack
        y = radius * sin(curr_phi)

        tv = 1 - (stack / stacks)

        for slice in slices:
            tu = slice / slices

            curr_theta = sector_step * slice
            z = radius * cos(curr_phi) * cos(curr_theta)
            x = radius * cos(curr_phi) * sin(curr_theta)

            nx = x * norm_factor
            ny = y * norm_factor
            nz = z * norm_factor

            points_add(x,y,z,nx,ny,nz,tu,tv)

    // construir os triângulos com pontos criados
    for stack in stacks:
```

```

    for slice in slices:
        point = points[stack][slice]

        if point is (last_slice_in_stack):
            point_right.tu = 1
            point_right_down.tu = 1

            triangles_add(point, point_down, point_right_down)
            triangles_add(point, point_right_down, point_right)

    return triangles
}

```

4.4) Cone

Como todas as faces laterais do cone apresentam a mesma inclinação, todas as normais da lateral terão a mesma coordenada y. Para o cálculo desta coordenada y, imaginando um corte transversal do cone, obtemos dois triângulos retângulos cuja altura corresponde à altura do cone, e a base ao raio do cone. Pegando no ângulo associado ao ponto mais alto de um dos triângulos, e calculando a sua tangente, obtemos a coordenada y. Imaginando uma reta definida da origem até esse ponto (1,y), definimos uma reta perpendicular à hipotenusa do triângulo, ou por outras palavras, um vetor perpendicular à lateral do cone, correspondendo à normal das faces laterais do cone.

Relembrando a forma como o cone era construído, para cada camada horizontal do cone, partíamos de um ponto alinhado com o eixo x ($x > 0, y > 0, z = 0$), e aplicávamos uma rotação sobre o eixo y com ângulo positivo, para obter o próximo ponto da camada. Tomando partido deste ângulo de rotação, as coordenadas x e z da normal da lateral vão resultar, respetivamente, de calcular o cosseno e seno desse ângulo. Como a rotação se processa no sentido do eixo z negativo, temos de inverter o valor da coordenada z para obter a normal correta. Por sua vez, para o cálculo das normais na base do cone, visto que os pontos pertencem todos a um mesmo plano voltado para y negativo, a normal será (0,-1,0)

Para as faces laterais do cone, optamos por “enrolar” a textura em torno das várias faces, de modo a que a aresta inicial em que se começa a enrolar a textura, corresponde à aresta em que acabamos de enrolar a textura. Se a camada vertical em que os pontos se encontram for superior, terão uma coordenada v superior e, se o ângulo em torno de cada camada horizontal for superior, terão uma coordenada u superior. Encontramos o mesmo problema encontrado na esfera de conectar pontos no final de cada camada horizontal, pelo que adotamos a mesma solução para o cone.

Na base do cone, repetimos a textura inteira para cada triângulo construído (entre o centro da base e dois pontos consecutivos da borda da base). Para tal, o centro da base terá coordenadas $u=0.5$ e $v=1.0$, porque o ponto se encontra horizontalmente entre os dois pontos da borda, e verticalmente é a posição mais alta do triângulo. Por sua vez os pontos da base terão coordenadas $u=0$ e $v=0$, $u=1.0$ e $v=0$, porque são pontos nos extremos esquerdo e direito do triângulo, e verticalmente estão na posição mais baixa do triângulo.

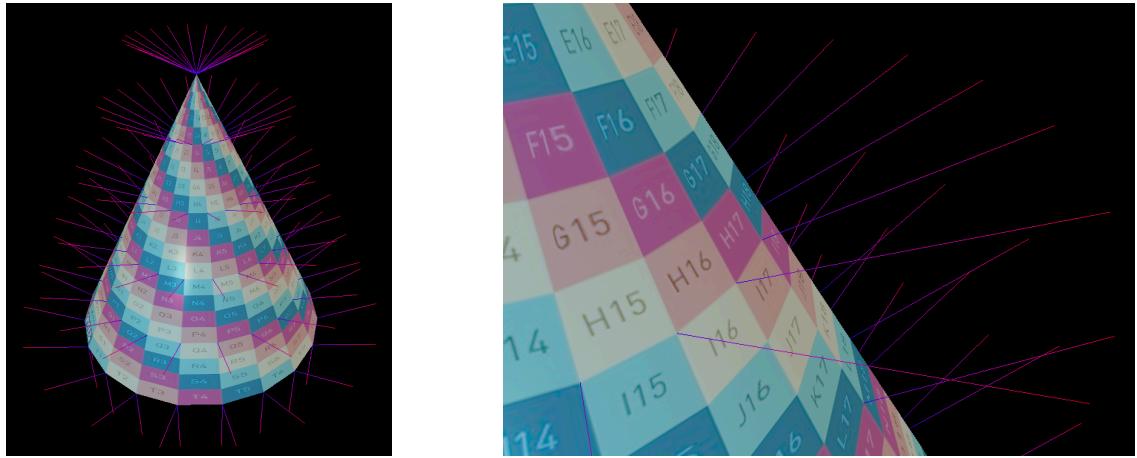


Figura 4: Textura e normais do cone

Tomando por base o pseudo-código apresentado no relatório da fase 1, temos então:

```

func cone (radius, height, slices, stacks) {
    // ângulo entre pontos da base adjacentes
    angle = 360 / slices

    // diferença de altura entre camadas horizontais
    stack_step = height / stacks

    // diminuição de raio entre camadas horizontais
    radius_step = radius / stacks

    // y de normal perpendicular à face lateral do cone
    normal_y = radius / height

    triangles = []

    for stack in stacks:
        prev_point = (radius - stack * radius_step, stack * stack_step, 0)

        for slice in slices:
            cur_point = rotate_y_matrix(angle) * prev_point

            // adicionar triângulos que vão constituir a base
            if stack == base_stack:
                base_point.normal = prev_point.normal = cur_point.normal = (0, -1, 0)
                base_point.tex_coords = (0.5, 1)
                prev_point.tex_coords = (1, 0)
                cur_point.tex_coords = (0, 0)
                triangles_add(base_point, prev_point, cur_point) // base_point = (0, 0, 0)

            // adicionar triângulos que constituem a lateral
            else:
                cur_point.normal.x = normalize(cos(angle * slice))
                cur_point.normal.y = normal_y
                cur_point.normal.z = normalize(sin(-angle * slice))

                cur_point.tex_coords.u = (angle * slice) / 360
                cur_point.tex_coords.v = (stack * stack_step) / height

```

```

    if point is (last_slice_in_stack):
        right_point.tu = 1
        right_down_point.tu = 1

    triangles_add(right_down_point, right_point, cur_point)
    triangles_add(cur_point, right_point, down_point)

    cur_point = prev_point

    // adicionar triângulos que vão constituir o chapéu do cone
    highest_point = (0, height, 0)

    for cur_point in last_stack_points:
        triangles_add(cur_point, highest_point, left_point)

    return triangles
}

```

4.5) Cilindro

Como as faces laterais do cilindro são perpendiculares ao plano XZ, as normais das laterais serão paralelas ao plano XZ. Assim, temos que todas as normais das faces laterais vão ter a coordenada y=0. Por sua vez, imaginando para uma camada horizontal com y=a o vetor definido entre (0,a,0) e um ponto da face lateral, compreendemos que as coordenadas x e z das normais laterais vão corresponder às coordenadas x e z geométricas do respetivo ponto. Por fim, todos os pontos da base superior e inferior do cilindro terão, respectivamente, as normais (0,1,0) e (0,-1,0).

Em concordância com a implementação de texturas no cone, para as bases do cilindro, seguimos a mesma estratégia de repetir a textura inteira para cada triângulo construído e, para as laterais, “enrolamos” a textura em torno das várias faces laterais do cilindro. O processo de cálculo das coordenadas de textura é semelhante ao do cone. No entanto, a coordenada u será invertida (1-valor) porque o cilindro é construído no sentido dos ponteiros do relógio (“direita” para a “esquerda” em relação à textura)

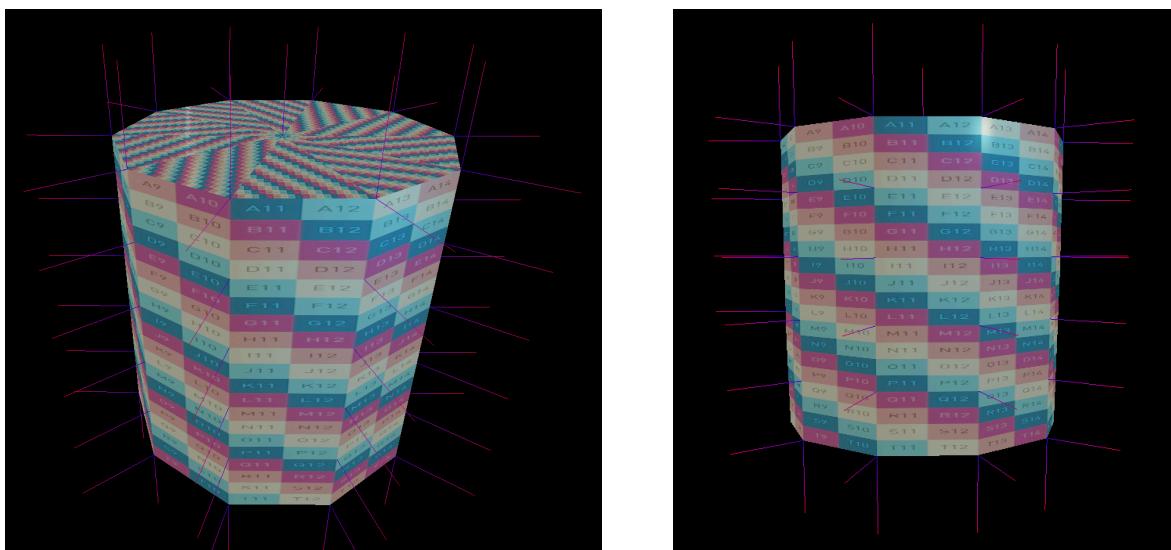


Figura 5: Textura e normais de cilindro

Tomando por base o pseudo-código apresentado no relatório da fase 1, temos então:

```

func cylinder (radius, height, slices, stacks) {

    // ângulo entre pontos da base adjacentes
    angle = 360 / slices

    // diferença de altura entre camadas horizontais
    stack_step = height / stacks

    triangles = []

    for stack in stacks:
        prev_point = (radius, stack * stack_step, 0)

        for slice in slices:
            cur_point = rotate_y_matrix(angle) * prev_point

            // adicionar triângulos que vão constituir a base inferior
            if stack == first_stack:
                base_point.normal = prev_point.normal = cur_point.normal = (0, -1, 0)

                base_point.tex_coords = (0.5, 0)
                prev_point.tex_coords = (1, 1)
                cur_point.tex_coords = (0, 1)

                triangles_add(base_point, prev_point, cur_point) // base_point = (0, 0, 0)

            // adicionar triângulos que vão constituir a base superior
            if stack == last_stack:
                base_point.normal = prev_point.normal = cur_point.normal = (0, 1, 0)

                cur_point.tex_coords = (1, 1)
                prev_point.tex_coords = (0, 1)
                base_point.tex_coords = (0.5, 0)

                triangles_add(cur_point, prev_point, base_point) // base_point = (0, height, 0)

            // adicionar triângulos que constituem a lateral
            if stack != first_stack:
                cur_point.normal = (cur_point.coords.x, 0, cur_point.coords.z)

                cur_point.tex_coords.u = 1 - (slice / slices)
                cur_point.tex_coords.v = stack / stacks

                if point is (last_slice_in_stack):
                    right_point.tu = 0
                    right_down_point.tu = 0

                triangles_add(right_down_point, right_point, cur_point)
                triangles_add(cur_point, down_point, right_down_point)

                cur_point = prev_point
            return triangles
}

```

4.6) Torus

Como referido na fase 1, a construção do torus implicava a criação de várias circunferências numa disposição em forma de torus, seguida da ligação das circunferências com a formação de triângulos entre dois pontos consecutivos de uma circunferência e da circunferência seguinte. No cálculo das coordenadas de textura, deparamo-nos com o mesmo problema de conectar pontos no final de cada linha encontrado na esfera: tanto na ligação do primeiro e último pontos de cada circunferência, como na ligação da primeira e última circunferências. A solução adotada foi idêntica à da circunferência.

A textura é “enrolada” em torno da primitiva de modo que cada circunferência que constitui o torus contenha um ponto em que o topo da textura conecta ao fundo da textura. Por sua vez, existe uma circunferência em que a esquerda da textura conecta à direita da textura. Para obter este mapeamento, uma circunferência mais avançada (com maior ângulo de rotação em relação à circunferência original) corresponde a uma coordenada u superior, e um ponto mais avançado numa circunferência (maior ângulo de rotação em relação ao ponto inicial na circunferência) corresponde a uma coordenada v superior.

Para obtenção das normais, a normal de um ponto corresponde ao vetor formado entre um ponto e o centro da circunferência a que ele pertence, multiplicado por um fator de normalização ($1 / ((\text{external_radius} - \text{internal_radius}) / 2)$)

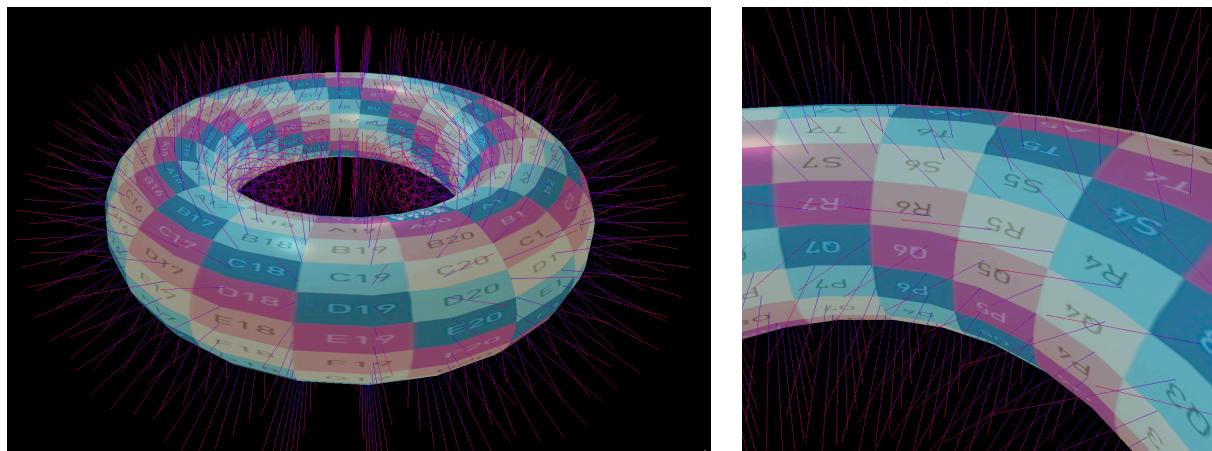


Figura 6: Textura e normais do torus

```
func Torus(raioInterior, raioExterior, slices, stacks) {
    // raio de cada circunferência que constitui o Torus
    raioTorus = (raioExterior - raioInterior) / 2

    // distância horizontal entre origem e centro de cada circunferência na posição final
    distPonto = raioInterior + raioTorus

    // ângulo entre as circunferências que constituem o Torus
    angleStacks = 360 / slices

    // ângulo entre os pontos de cada circunferência
    angleSlices = 360 / stacks

    // escalar para normalizar as normais
    normFactor = 1 / raioTorus
```

```

circunferenciaXZ = [] // pontos de circunferência no plano XZ centrada na origem
circunferenciaYZ = [] // pontos de circunferência no plano YZ centrada na origem
fatiasTorus = [] // pontos de todas as circunferências na posição final
triangles = []

// pontos da circunferência no plano XZ (idêntico à fase 1)
calc_points_xz()

// pontos da circunferência no plano YZ (idêntico a fase 1)
calc_points_yz()

// colocar as circunferências na posição correta
for stack in stacks:
    for slice in circunferenciaYZ: // cada stack é uma circunferênciaYZ

        // Matriz de translação que move o ponto para as coordenadas finais
        x = distPonto * sin(angleStacks * stack)
        y = 0
        z = distPonto * cos(angleStacks * stack)

        // Mover ponto para a posição correta e com a rotação pretendida
        posicaoFinal = translate_matrix(x,y,z) * rotate_y_matrix(angleStacks * stack)
        * curr_point

        ponto.coords = posicaoFinal
        ponto.normal = (posicaoFinal - circunferenciaYZ_center) * normFactor
        ponto.tex_coords = (stack / stacks, slice / slices)

        fatiasTorus_add(ponto)

// construir os triângulos
for stack in stacks:
    if stack is (last_stack_in_stacks):
        point_down.tex_coords.u = 0
        point_right_down.tex_coords.u = 0

    for slice in slices:
        point = fatiasTorus[stack][slice]

        if point is (last_slice_in_slices):
            point_right.tex_coords.u = 0
            point_right_down.tex_coords.u = 0

            triangles_add(point, point_down, point_right_down)
            triangles_add(point, point_right_down, point_right)
    return triangles
}

```

5) Shaders

Embora já tivessem sido utilizados em todas as fases, torna-se relevante para este relatório falar dos shaders

5.1) Iluminação

5.1.1) Materiais e texturas

No relatório da fase anterior foi referido que cada vértice tem um ID do seu objeto, através do qual o shader pode ir buscar a matriz de transformação a aplicar a um Texture Buffer. Reaproveitando este mesmo conceito para o material, o shader é também capaz de obter as informações correspondentes ao material do objeto acedendo ao mesmo buffer, passando esta a conter, adicionalmente, a informação pertinente sobre o material.

A informação necessária foi guardada numa struct com o seguinte formato:

```
struct Material {
    glm::vec3 diffuse;
    glm::vec3 ambient;
    glm::vec3 specular;
    glm::vec3 emissive;
    GLfloat shininess;
    GLfloat texture_id; // dificuldades em aceder como int/uint
    // para alinhar com vec4
    GLfloat padding_1;
    GLfloat padding_2;
}
```

5.1.2) Texturas

Como se pode observar através da struct anterior, existe uma textura para cada material, a qual pode ser acedida pelo respetivo ID. Ao invés de utilizar um *texture atlas*, decidimos que é mais conveniente um Texture Array, o qual foi abstraído na classe [TextureArray](#). Esta é responsável por inicializar e configurar o array de texturas e ler texturas a partir de imagens, enviando-a para o array, fazendo, se necessário, resize das mesmas para o tamanho definido, recorrendo para isso à biblioteca [stb_image](#). Assim, assumimos que a primeira textura tem o ID 0, a próxima 1, e assim sucessivamente.

Neste processo, o *Renderer* é o responsável por atribuir os IDs aos diversos nomes de textura. Este processo de identificação da textura por um ID permite a reutilização das imagens que já foram carregadas para a memória, em contraste com a hipótese de carregar a imagem 1 vez por objeto que a utilize. As implementações foram elaboradas de forma simples, não existindo a gestão dos slots de textura nem expansão do array quando necessário.

5.1.3) Vertex shader

A posição do vértice no ecrã só será calculada no *geometry shader*, mantendo-se igual à que é fornecida no VBO:

```
gl_Position = posicao_fornecida;
```

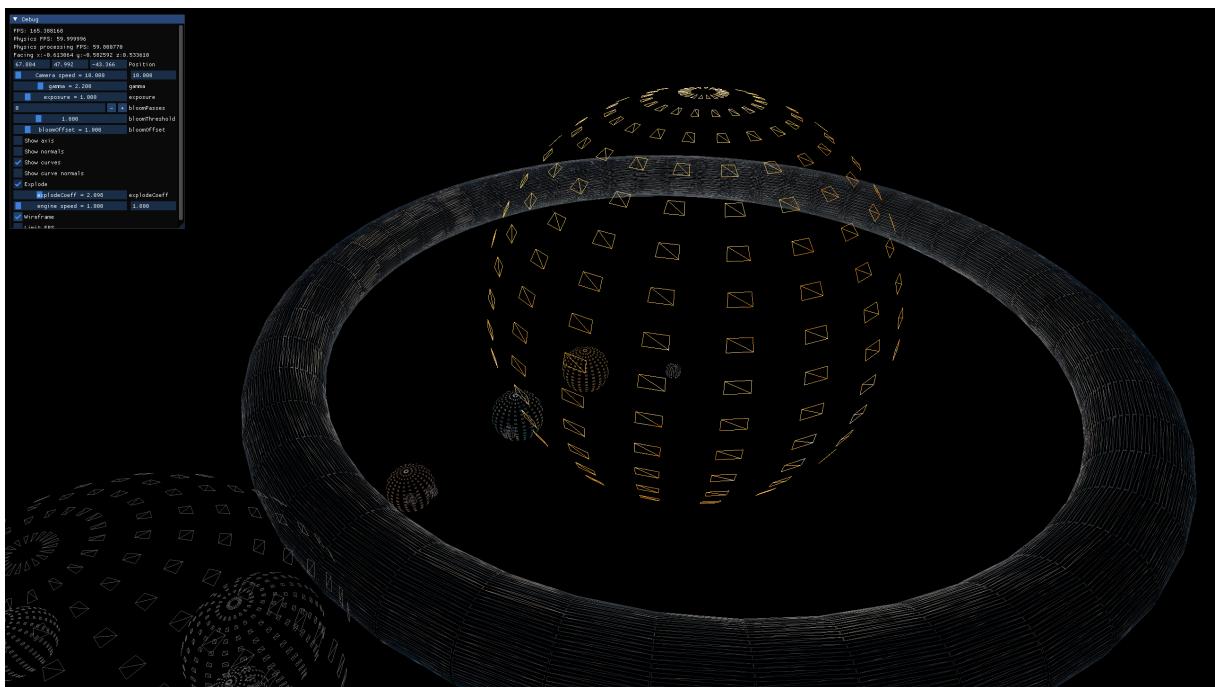
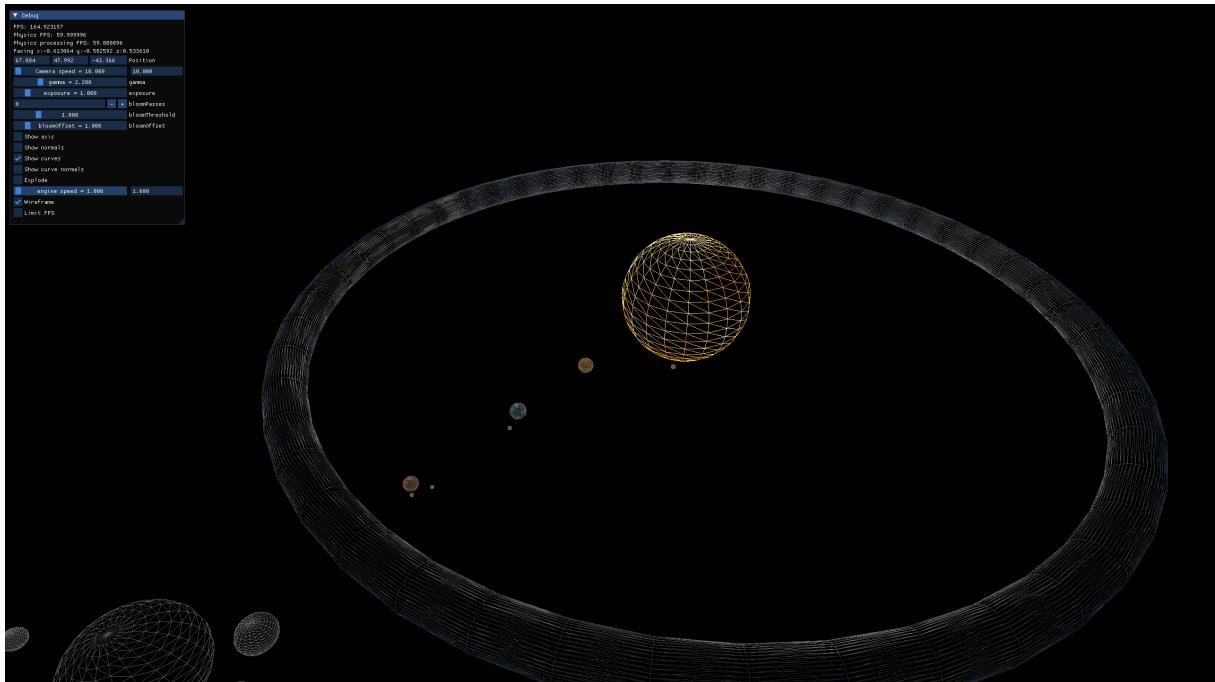
No entanto, decidimos transformar já a normal. Contudo, não é suficiente utilizar a matriz de transformação do objeto, visto transformações de escala poderem distorcer a orientação da normal. Torna-se então necessário calcular a matriz a aplicar à normal do vértice da seguinte forma:

```
vs_out.v_Normal =
    normalize(mat3(transpose(inverse(View * matriz_transf))) * normal_fornecida);
```

Além da transformação da normal pela matriz do objeto, aproveitamos já para colocar a normal em view space, visto esta ser utilizada nos cálculos referentes à iluminação.

5.1.4) Geometry shader

Este shader surgiu da ideia de fornecer um efeito de explosão aos diversos objetos, sendo por vezes utilizado como um modo de debug:



Ao invés de utilizar as normais fornecidas, as normais dos novos vértices são calculadas obtendo dois vetores entre os vértices do triângulo e calculando o seu produto externo.

```
a = gl_in[0].gl_Position - gl_in[1].gl_Position; // vetor de [1] ate [0]
b = gl_in[2].gl_Position - gl_in[1].gl_Position; // vetor de [1] ate [2]

normalize(cross(b, a)); // b,a ou a,b depende da ordem dos vertices
```

Com este valor, calculamos a transformação de cada vértice usando a sua posição inicial, a matriz de transformação do seu objeto e este novo vetor multiplicado por um coeficiente. Para obter a posição no ecrã, aplicamos ainda as matrizes View e Projection da seguinte forma:

```
explode = getAvgNormal(...) * coeff

gl_Position = u_Projection * u_View * mat_transf * explode;
```

Com estes valores, conseguimos extrair a posição em view space, a qual será usada nos cálculos da iluminação:

```
fragPos = u_View * mat_transf * explode;
```

Resumindo, este shader recebe os vértices de um triângulo e da retorna o resultado da transformação dos mesmos.

5.1.5) Fragment shader

Este é o shader que implementa a iluminação, em view space, segundo o modelo de Blinn-Phong. As luzes, tal como os materiais dos objetos e as suas respetivas matrizes de transformação, são enviadas através de 3 no texture buffers: um usado para representar as luzes ponto, outro em que se encontram as luzes direcionais e o último possui as informações das diversas luzes focais.

O cálculo da iluminação é feito da seguinte forma: primeiro é necessário calcular a direção do fragmento até à câmara através da normalização do simétrico da posição atual (como estamos a trabalhar em view space, a câmara está sempre em (0, 0, 0) pelo que a direção será $(0, 0, 0)$ - $\text{FragPos} = -\text{FragPos}$):

```
vec3 viewDir = normalize(-FragPos);
```

Em seguida, definimos a cor base de cada vértice, neste caso preto, que representa a ausência de cor:

```
vec4 color = vec4(0.0, 0.0, 0.0, 1.0);
```

A cor base é de seguida atualizada conforme os diferentes dados das luzes (ambiente, difusa, etc...) e materiais fornecidos:

```
color += calcular(luz, material, ...)
```

Por fim, adicionamos a luz emissiva do material e a textura:

```
color.rgb += material.emissive.rgb;
color *= texture(u_TextureArraySlot, vec3(fs_in.g_TexCoord.xy, material.texture_id));
```

Nota: para cada um dos cálculos, aplicamos a atenuação ao resultado final e não apenas na componente difusa, pois pareceu fornecer melhores resultados.

5.1.5.1) Point Lights

A função de cálculo da luz ponto tem o seguinte formato:

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir, Material
material)
{
    // calcular posição da luz em view space
    vec3 viewSpace_position = vec3(u_View * vec4(light.position, 1.0));

    // cálculo da direção até a luz
    vec3 fragToLight = viewSpace_position - fragPos;
    vec3 lightDir = normalize(fragToLight);
```

```

// calculo da luz difusa
vec3 diff = max(dot(normal, lightDir), 0.0) * light.diffuse * material.diffuse;

// calculo da luz especular
// halfway vector entre camera e luz, seguindo Blinn-Phong
vec3 halfwayDir = normalize(lightDir + viewDir);
vec3 spec = pow(max(dot(normal, halfwayDir), 0.0), material.shininess)
    * light.specular * material.specular;

// atenuacao
float distance = length(fragToLight);
float attenuation = 1.0 / (light.constant + light.linear * distance +
    light.quadratic * (distance * distance));

// calculo final, juntando luz ambiente e aplicando atenuacao a tudo
return ((light.ambient * material.ambient) + diff + spec) * attenuation;
}

```

5.1.5.2) Directional lights

Nestas luzes, ao invés da sua posição, é fornecida a direção até à luz, não sendo necessário, por isso, calcular este valor. No entanto, tem de ser da mesma calculado o seu equivalente para view space.

```

vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir, Material material)
{
    // transformar direção em view space
    vec3 lightDir = normalize((mat3(u_View) * light.direction));

    ...
}

```

O resto das contas é igual à Point Light, não existindo, no entanto, atenuação, visto a luz estar a distância infinita e sem um ponto de origem concreto.

5.1.5.3) Spot lights

As luzes focais foram implementadas com dois cutoffs: o cutOff representa o (cosseno do) ângulo até ao qual a luz tem iluminação máxima, e no espaço entre o cutOff e o outerCutOff, os valores são alterados de forma a fazer um efeito de fade-out, sendo o valor nulo a partir daí. A função de cálculo tem o seguinte aspeto:

```

vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir, Material material)
{
    // tudo igual a point light.....
    // calcular a direção da luz em view space, e depois cosseno entre isso e direção do
    // fragmento ate a luz
    // multiplicamos por -1 devido ao vetor direção ser interpretado como tendo o seu
    // centro na spotlight, o oposto do que precisamos
    float theta = dot(lightDir, normalize(-(mat3(u_View) * light.direction)));

    // calcular diferença entre os dois 'angulos' de cutOff (aqui ja sao cossenos)
    float epsilon = light.cutOff - light.outerCutOff;

    // intensidade de [0, 1]. 0 no outer, 1 no inner
}

```

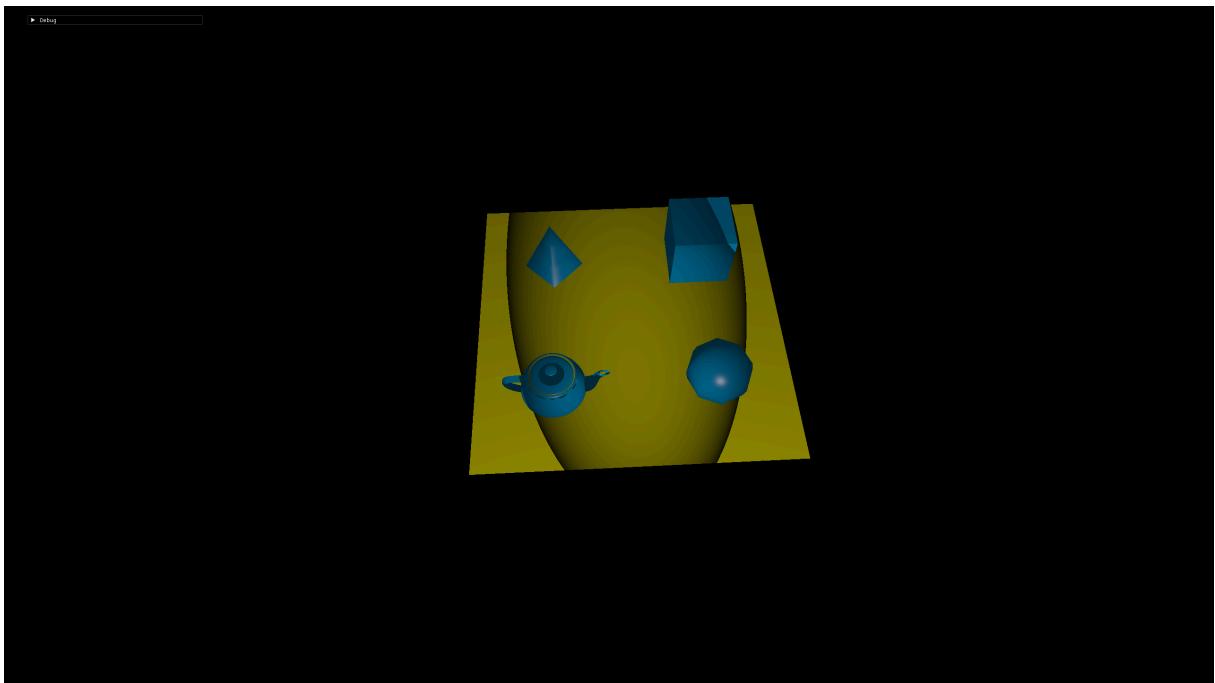
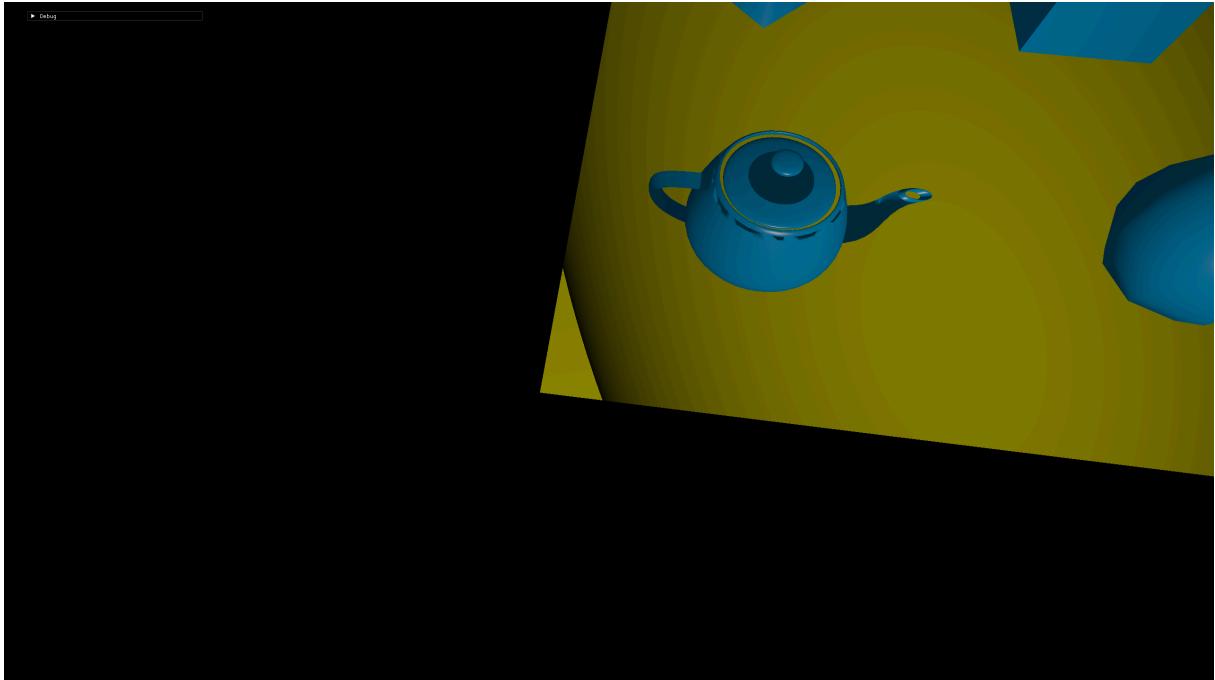
```

float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);

// nao aplicamos a intensidade a luz ambiente, de forma a ter sempre alguma luz
return ((light.ambient * material.ambient) + diff * intensity + spec * intensity) *
attenuation;
}

```

Exemplos onde se pode visualizar a intensidade a ser alterada do cutOff até ao outerCutOff:

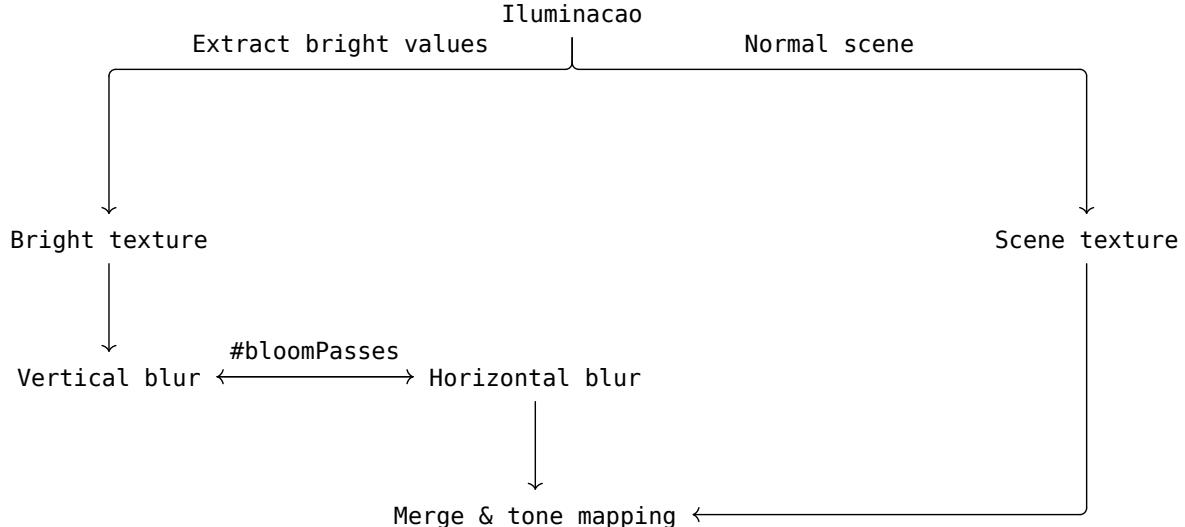


No geral, vários cálculos como o da NormalMatrix, posição em view space das luzes, etc., poderiam ter sido efetuados no CPU para evitar cálculos redundantes, porém pensamos que deste modo fica mais legível e separamos completamente estas ideias da parte do CPU, embora a performance seja objetivamente pior.

5.2) Bloom, HDR e tone mapping

Consideramos que seria interessante acentuar o brilho do sol, implementando assim um efeito de bloom. Após pesquisar sobre o assunto, vimos que este tem melhor impacto quando usado juntamente com HDR e tone mapping, pelo que resolvemos implementá-los também.

De um modo simplificado, a nossa renderização passa pelo seguinte processo:



Começando pela iluminação, a qual utiliza os shaders descritos anteriormente, usamos um framebuffer novo, de modo a usar floats de 16 bits ao invés de 8, o que permite um maior intervalo de cores (HDR). Este framebuffer tem também dois color attachments, permitindo de uma só vez colocar cores numa textura correspondente à cena e noutra correspondendo àquilo que são consideradas cores com alto brilho. Assim, no fragment shader anterior, `color` corresponde à cor normal da cena, enquanto `brightColor` escreve em outra textura.

Para decidir quais as cores consideradas brilhantes, fazemos o produto interno entre o valor da cor atual com um vetor em que cada elemento representa a intensidade da respetiva componente RGB. Se o resultado for superior ao threshold definido, então a cor é interpretada como brilhante, caso contrário é ignorada (definida como preto).

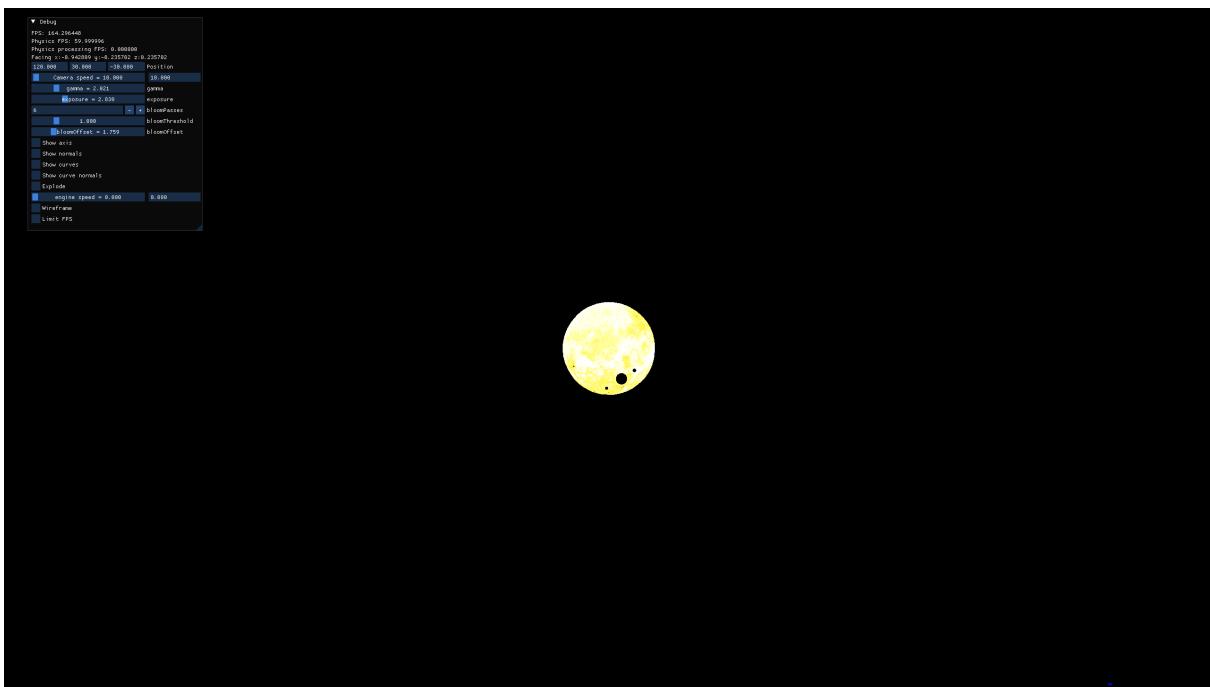
```
// números relacionados com a percepção de luminance na visão humana
float brightness = dot(color.rgb, vec3(0.2126, 0.7152, 0.0722));
if (brightness > u_BloomThreshold) {
    brightColor = vec4(color.rgb, 1.0);
} else {
    brightColor = vec4(0.0, 0.0, 0.0, 1.0);
}
```

Esta extração pode ser vista neste exemplo:

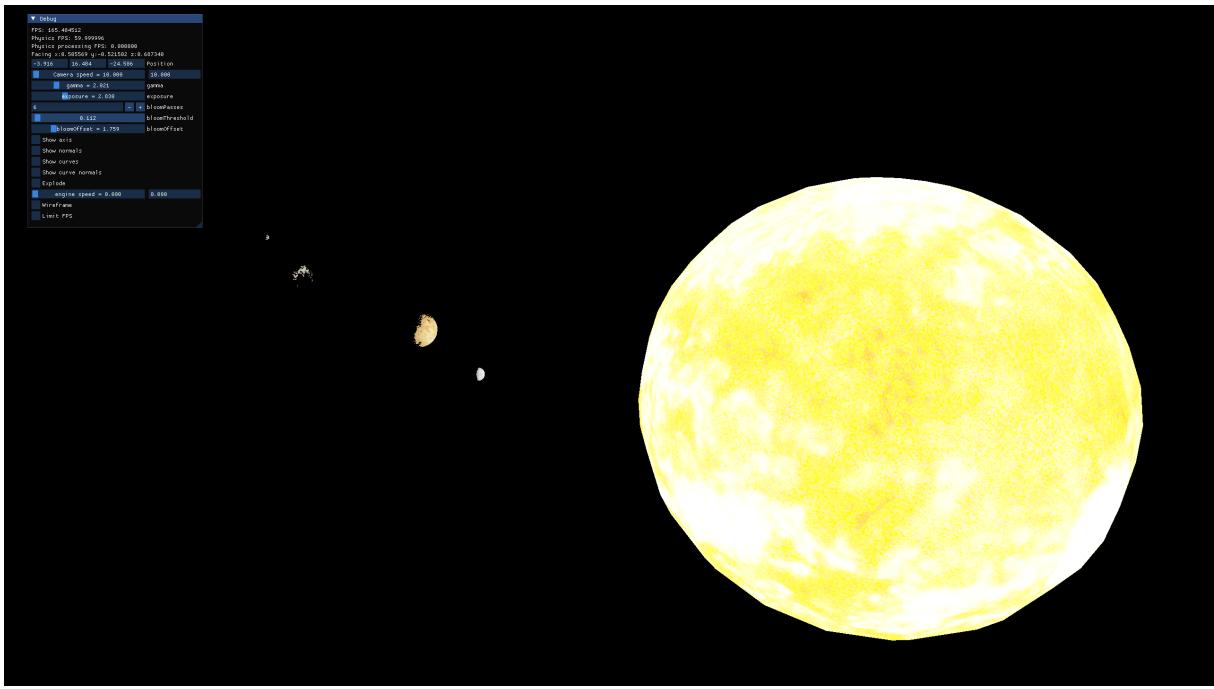
Cena normal



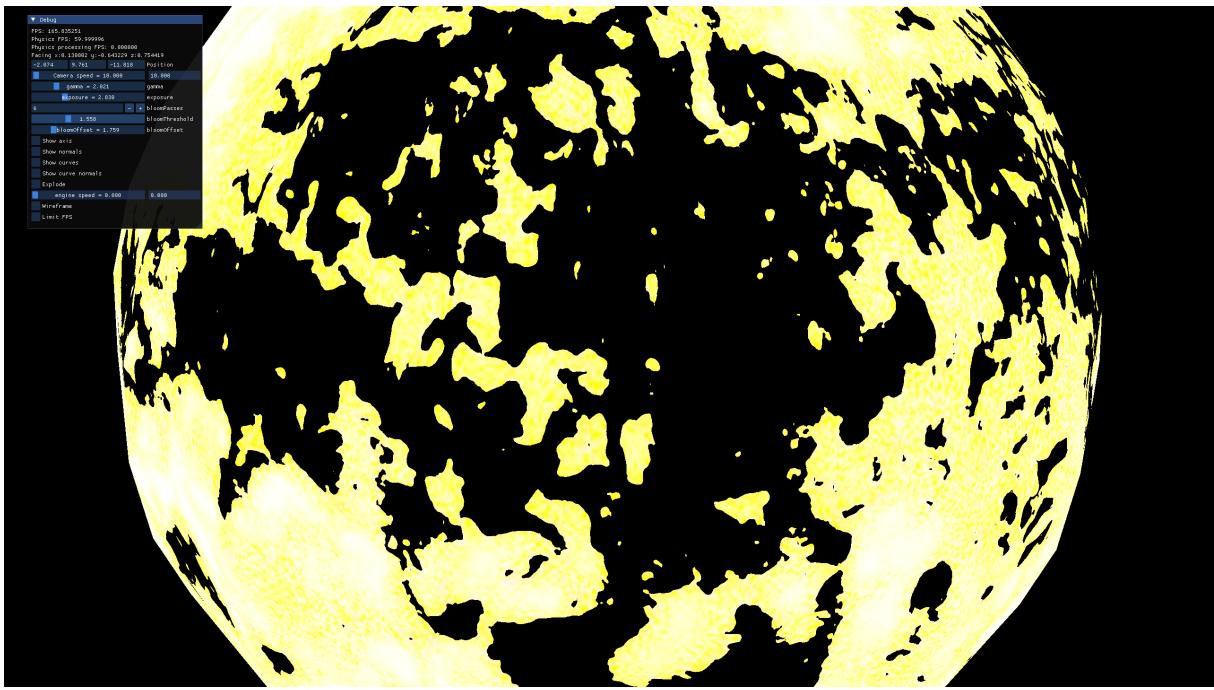
Luz extraída



Podemos observar que, ao diminuir o threshold, mais pixels começam a ser considerados brilhantes, começando a incluir os planetas:



E, fazendo o oposto, áreas do próprio sol deixam de ser consideradas brilhantes:



Agora que temos uma textura com os pixels brilhantes, basta desfocá-los. Para tal, aplicamos two-pass Gaussian blur em ‘ping-pong’, usando outros dois framebuffers, cada um com uma textura no qual o seu output é escrito para poder ser lido pelo outro: desfocamos na horizontal, depois na vertical, voltando a horizontal, e assim sucessivamente. O shader aplica a função desta forma:

```
sampler2D u_BluBuffer; // textura desfocada ate ao momento / textura com as cores  
brilhantes

bool u_Horizontal; // true: desfocar na horizontal
float u_Weight[5] = float[] (0.227027, 0.1945946, 0.1216216, 0.054054, 0.016216); //
```

```

pesos da funcao
float u_TexOffsetCoeff = 1.0;

void main()
{
    // para saber a distancia que vamos mover, usar u_TexOffsetCoeff como coeficiente
    // e juntar ao tamanho de um texel
    vec2 tex_offset = u_TexOffsetCoeff * texel_size;

    // o primeiro, weight[0], vai ser igual para horizontal ou vertical pois e sobre o
    pixel central
    vec3 result = texture(u_BluBuffer, v_TexCoord).rgb * u_Weight[0];

    if(u_Horizontal)
    {
        for(int i = 1; i < 5; i++)
        {
            // somar as contribuicoes dos pixeis em redor, diminuindo o peso consoante
            // a distancia (consoante i)
            // direita
            result += texture(u_BluBuffer, v_TexCoord + vec2(tex_offset.x * i, 0.0)).rgb
            * u_Weight[i];
            // esquerda
            result += texture(u_BluBuffer, v_TexCoord - vec2(tex_offset.x * i, 0.0)).rgb
            * u_Weight[i];
        }
        else
        {
            // igual, mas vertical (offset em y)
        }
    }

    color = vec4(result, 1.0);
}

```

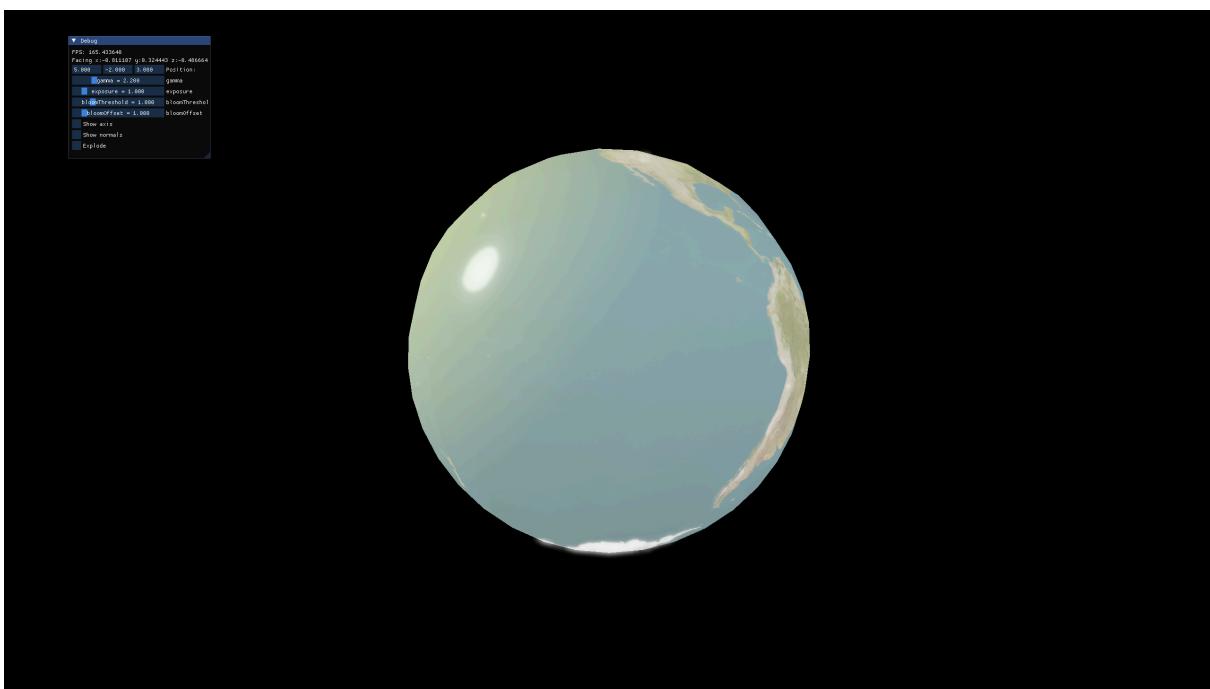
Este processo repete-se um determinado número de vezes, que pode ser alterado no menu de debug. Se o valor for 0, saltamos à frente qualquer passo de desfocagem.

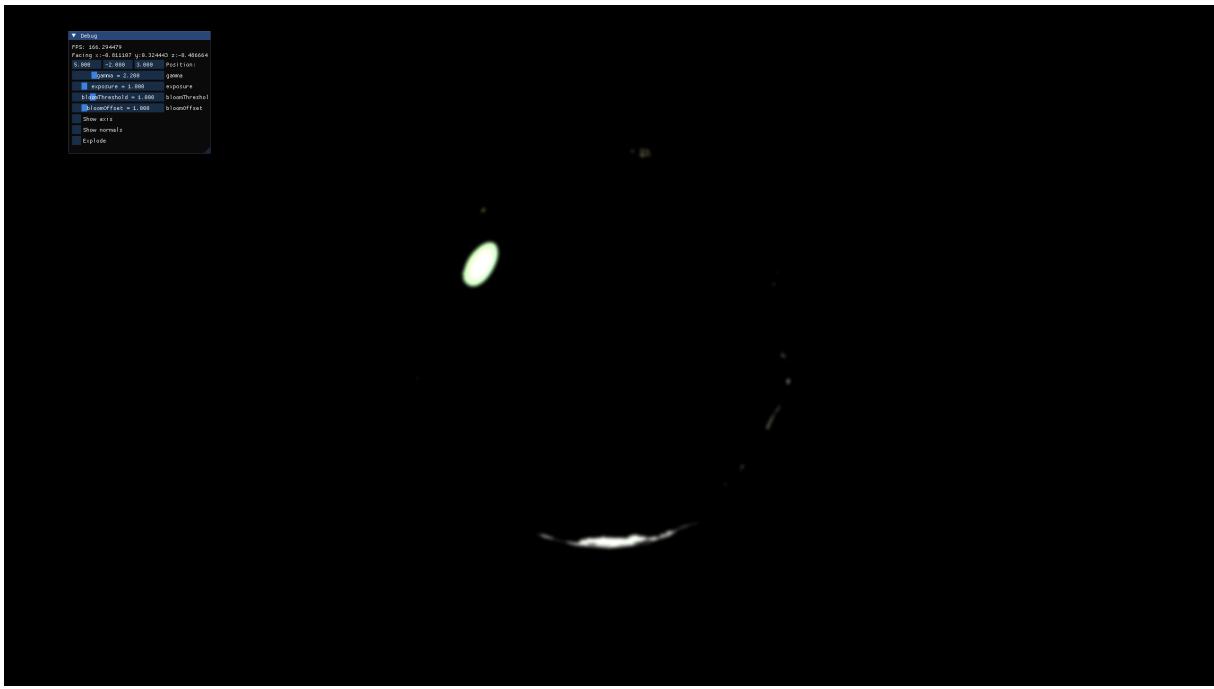
Obtendo, assim, estes resultados:

Exemplo da extração anterior, agora desfocado



Outros exemplos

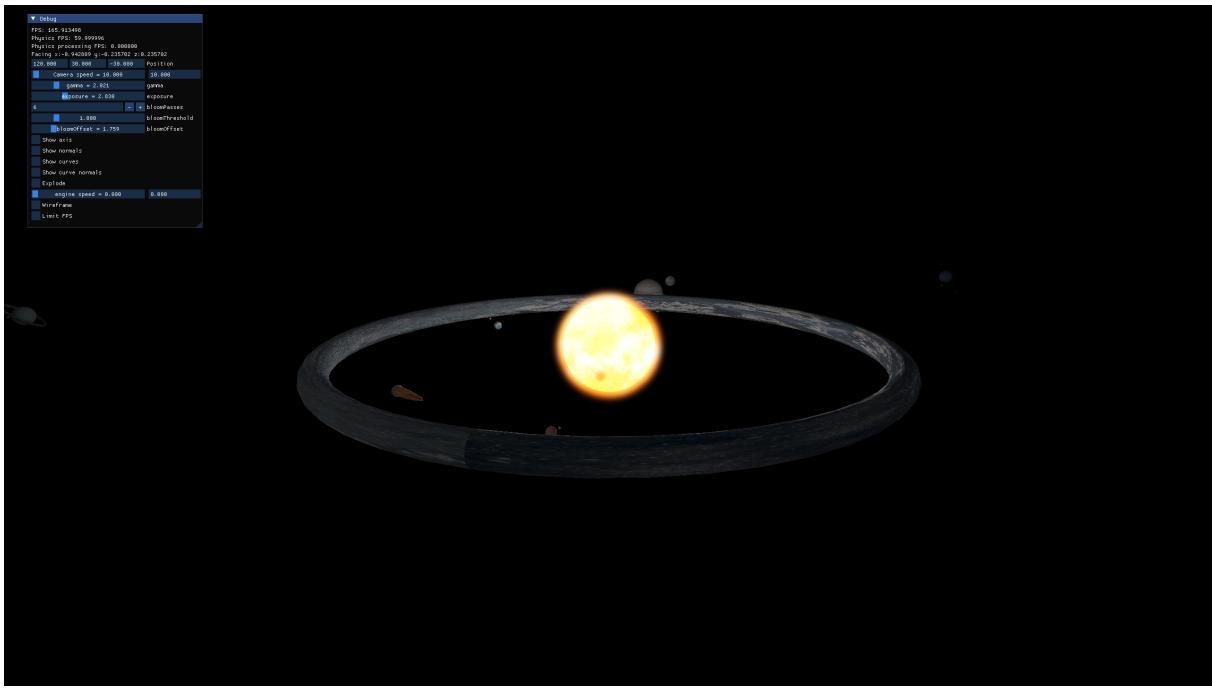




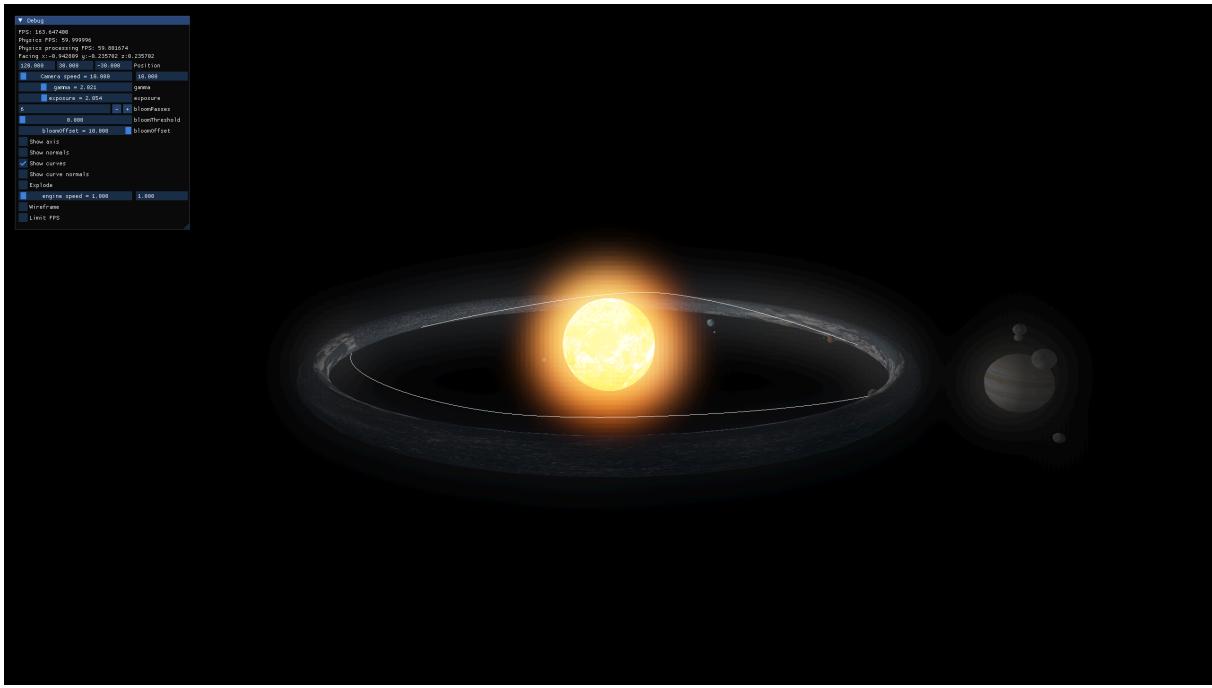
Por fim, basta juntar estes dois resultados num só e fazer *gamma correction* e *tone mapping*. Para isto, enviamos as duas texturas, da cena normal e do resultado desfocado, para um outro shader. Neste, juntamos os resultados aplicando *tone mapping* muito básico, através de *exposure*:

```
float gamma = ...  
float exposure = ...  
  
uniform float exposure;  
  
void main()  
{  
    // extract both colors  
    vec3 hdrColor = texture(textura_cena, v_TexCoord).rgb;  
    vec3 bloomColor = texture(textura_brilhante, v_TexCoord).rgb;  
  
    // blend das cores adicionando os valores  
    hdrColor += bloomColor;  
  
    // exposure tone mapping, para tornar a cor de [0, 1] (LDR) tendo em conta a exposure  
    // para traduzir HDR em LDR, fazemos um decaimento exponencial da cor  
    // a cor atual é a cor mais brilhante admissível - decaimento calculado  
    vec3 mapped = vec3(1.0) - exp(-hdrColor * exposure);  
  
    // gamma correction  
    mapped = pow(mapped, vec3(1.0 / gamma));  
  
    color = vec4(mapped, 1.0);  
}
```

No fim, obtemos resultados que consideramos ter melhorado o aspeto da cena, para além de fornecer maior controlo sobre as cores e o brilho:



Exagerando o threshold e o offset usado pela desfocagem:



6) Conclusão

A finalização desta fase indica a conclusão do trabalho. Neste foi possível colocar diversos conceitos em prática e aprender sobre diversos novos conceitos de computação gráfica. Focando no trabalho desenvolvido nesta fase, a iluminação (conceitos e implementação) e a aplicação dos materiais podem ser considerados um grande desafio, mas houve casos em que os cálculos das normais também se demonstraram verdadeiros obstáculos.

A aplicação de shaders permitiu a descoberta de novas abordagens à renderização das cenas, mudando a perspetiva do grupo sobre as fórmulas de aplicação das diversas componentes da iluminação e dos materiais, achando interessante os cálculos da iluminação em *view space*.

Ainda nos shaders, o grupo deu um pequeno passo na área de correção de cores, através da manipulação dos níveis gama e de exposição. Esta área demonstrou-se bastante interessante e inesperadamente complexa. Foi ainda necessário realizar algumas correções em relação ao código das fases anteriores, o qual não permitiria a aplicação fácil de conceitos trabalhados nesta etapa.

O grupo considera ter cumprido os objetivos pedidos nas diversas fases, esforçou-se para aplicar vários extras, desde o desenvolvimento de primitivas extra, a aplicação de alinhamentos em curvas de Catmull-Rom e finalizando com a implementação de efeitos visuais, entre os quais se destacam o bloom, gamma e exposure e efeitos de explosão dos vértices. Foi ainda desenvolvido um pequeno menu iterativo que permite o controlo sobre estes e outros extras que o grupo desenvolveu, tais como a velocidade de movimentação da nossa câmara fps, o aparecimento das normais dos objetos, etc.

Os resultados produzidos foram satisfatórios e, como trabalho futuro, o grupo acharia pertinente a implementação de, por exemplo, instancing para os asteroides, skybox, entre outros.