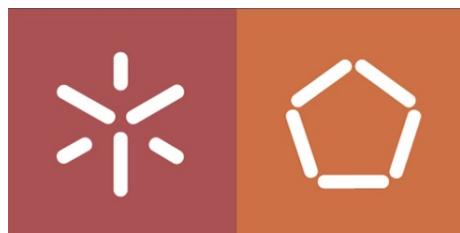


UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA



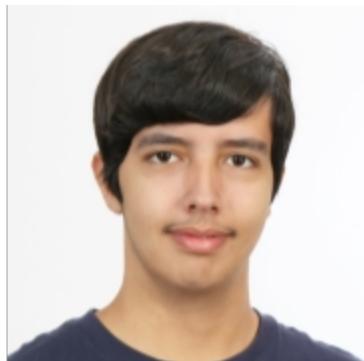
MEI - MESTRADO EM ENGENHARIA INFORMÁTICA

PERFIL EM COMPUTAÇÃO GRÁFICA

Visualização e Iluminação

GRUPO 2

PROJETO FASE 3 - FINAL



Eduardo Pereira PG53797



Filipa Rebelo PG53624



Nuno Mata PG44420

Junho 2024

Conteúdo

1	Introdução	1
2	Formatos de Imagem: JPG, PFM e OpenEXR	1
2.1	Formato PFM	1
2.2	Formato JPG	1
2.3	Formato OpenEXR	2
3	Câmaras Alternativas	2
3.1	Fish Eye	2
3.2	Múltiplas Câmaras	3
3.3	Efeito <i>Swirl</i>	4
3.4	Efeito Distorção <i>Pincushion</i>	5
4	Tone Mapping: Uncharted 2	6
5	Conclusão	7
6	Anexos	8
6.1	Formatos de Imagem	8
6.1.1	PFM	8
6.1.2	JPG	8
6.1.3	OpenEXR	9
6.2	Câmaras Alternativas	9
6.2.1	FishEye	9
6.2.2	Efeito Múltiplas Câmaras	10
6.2.3	Pinchurshion Distortion	10
6.2.4	Swirl	11
6.3	Tone Mapping Comparação: Original, Reinhard e Uncharted 2	12

1 Introdução

O presente relatório tem como objetivo abordar os temas desenvolvidos nesta fase no âmbito da Unidade Curricular de Visualização e Iluminação. Nesta fase do projeto, os temas escolhidos para implementar foram Formatos de Imagem tendo disponíveis os formatos PPM, PFM, OpenEXR e JPG, o tema de Câmaras Alternativas e ainda o de Tone Mapping. No decorrer do relatório iremos abordar em detalhe cada um destes temas e os resultados obtidos. Para consultar o desenvolvimento e evolução do projeto é possível consultar o link para o [Github do Projeto de Grupo](#) com todas as implementações efetuadas ao longo do desenvolvimento e também com os renders gerados de cada funcionalidade adicionada.

2 Formatos de Imagem: JPG, PFM e OpenEXR

2.1 Formato PFM

O formato PFM é utilizado para armazenar valores de pixels em formato float. Um ficheiro PFM é composto por duas partes principais: o cabeçalho e os dados dos pixels.

O cabeçalho inclui o tipo, indicado por "PF" para imagens coloridas (RGB) e "Pf" para imagens em escala de cinza, as dimensões, que são a largura e a altura da imagem, e a escala, um número em float que representa a escala dos valores dos pixels. Um valor negativo indica que os dados são armazenados em formato little-endian.

Os dados dos pixels são armazenados em formato float. Os pixels são armazenados na ordem das linhas de baixo para cima e da esquerda para a direita. Esse formato permite uma representação de alta precisão para as cores e a iluminação, sendo essencial para aplicações que requerem um alto alcance dinâmico.

```
//write header info to the file , negative value=>little endian
ofs << "PF\n" << W << " " << H << "\n-1.0\n";

// Write pixel data to the file
//from bottom to top, from left to right, otherwise the image would be inverted.
for (int j = this->H - 1; j >= 0; --j) {
    for (int i = 0; i < this->W; ++i) {
        float r = static_cast<float>(imageToSave[j * W + i].val[0]) / 255.0f;
        float g = static_cast<float>(imageToSave[j * W + i].val[1]) / 255.0f;
        float b = static_cast<float>(imageToSave[j * W + i].val[2]) / 255.0f;

        ofs.write(reinterpret_cast<char*>(&r), sizeof(float));
        ofs.write(reinterpret_cast<char*>(&g), sizeof(float));
        ofs.write(reinterpret_cast<char*>(&b), sizeof(float));
    }
}
```

Neste formato começamos por escrever o cabeçalho no ficheiro seguida da escrita da informação da imagem. A escrita é efetuada do fim para o inicio dado que neste formato as linhas são armazenadas de baixo para cima e caso fossem escritas do inicio para o fim a imagem ficaria invertida. Cada pixel é convertido de valores inteiros para valores de float e escrito no ficheiro em formato binário. O render deste formato esta em anexo na figura 2.

2.2 Formato JPG

De forma a implementar o suporte à geração de imagens no formato jpg, que pode ser visualizado na figura 3, recorremos à biblioteca "OpenCV". A biblioteca "OpenCV" é uma biblioteca *open source* que nos permite aplicar várias técnicas de processamento de imagem, e que nos disponibiliza uma API bastante satisfatória para converter os pixéis calculados em imagens para vários tipos de extensões.

```
cv::Mat imageJPG(H, W, CV_8UC3, cv::Scalar(0,0,0));
for (int j = 0; j < H; j++) {
    for (int i = 0; i < W; ++i) {
```

```

        imageJPG.at<cv::Vec3b>(j, i)[0] = imageToSave[j*W+i].val[2];
        imageJPG.at<cv::Vec3b>(j, i)[1] = imageToSave[j*W+i].val[1];
        imageJPG.at<cv::Vec3b>(j, i)[2] = imageToSave[j*W+i].val[0];
    }
}

cv::imwrite(filename, imageJPG);

```

Neste *snippet*, podemos ver que os nossos dados são convertidos para as estruturas de dados da biblioteca "OpenCV". Ao criar a matriz, damos a *flag* **CV_U8C3**, que faz com que a matriz receba elementos de 8-bits unsigned (8U), com 3 canais de cores (C3), que corresponde ao RGB.

Uma vantagem que podemos reparar nas imagens jpg geradas é que estas apresentam um tamanho bastante inferior relativamente a ppm, isto devido à compressão do formato. Em contrapartida, apresentam uma qualidade um pouco inferior.

2.3 Formato OpenEXR

O formato OpenEXR é usado em indústrias onde são necessárias imagens de alta qualidade, especialmente na representação de cores e luminosidade. A sua principal diferença comparativamente a outros formatos é que exibe uma gama muito mais ampla de luminosidade e contraste já que usa HDR (High Dynamic Range) para representar estas características, resultando em imagens com maior detalhe nas áreas claras e escuras.

```

bool ImageEXR::Save(std::string filename) {
    cv::Mat hdr_image(H, W, CV_32FC3);

    for (int i = 0; i < H; ++i) {
        for (int j = 0; j < W; ++j) {
            RGB &rgb = imagePlane[i * W + j];
            hdr_image.at<cv::Vec3f>(i, j) = cv::Vec3f(rgb.B, rgb.G, rgb.R);
        }
    }
}

```

Para guardarmos a imagem neste formato, começamos por recorrer ao OpenCV para criar uma matriz com dimensões de H em altura e W em largura e com o tipo **CV_32FC3**. Este último argumento indica que cada elemento da matriz (pixel) terá 3 canais (correspondentes a RGB) e cada canal contém um valor float de 32 bits, que é ideal para imagens HDR, que requerem maior precisão. Seguidamente percorremos cada pixel da imagem e obtemos a sua referência RGB que está guardada no vetor *imagePlane*. Por fim, acedemos aos elementos (pixeis) da matriz criada inicialmente (chamada *hdr-image*), na qual criamos um vetor de 3 componentes B, G e R do pixel atual, isto repete-se então para cada pixel. Desta forma garantimos que a imagem depois pode ser guardada preservando a gama HDR e as respetivas cores corretamente. Neste formato é crucial aplicarmos um algoritmo de tone mapping para que o render produza uma imagem com um bom aspeto em monitores não HDR. O render para este formato encontra-se na figura 4.

3 Câmaras Alternativas

3.1 Fish Eye

A primeira alternativa à câmera *default* que implementamos foi o efeito da lente *fish eye*, que se caracteriza pela distorção esférica das imagens que são capturadas utilizando este tipo de lentes.

```

bool Fish_Eye::GenerateRay(const int x, const int y, Ray *r, const float *cam_jitter) {
    //Camera Space
    float xc, yc;
    if (cam_jitter == NULL) {
        xc = 2.f * ((float)x + .5f) / W - 1.f;
        yc = 2.f * ((float)(H - y - 1) + .5f) / H - 1.f;
    }
}

```

```

    }
else {
    xc = 2.f * ((float)x + cam_jitter[0]) / W - 1.f;
    yc = 2.f * ((float)(H - y - 1) + cam_jitter[1]) / H - 1.f;
}

float r_norm = sqrt(xc*xc + yc*yc);

if (r_norm > 1.0f) {
    return false;
}

// Calculate the angle of the ray
float radius = r_norm * fovW * 0.5f;
float distorted_radius = tan(radius);

// Calculate the distorted direction
float ray_dir_x = sin(distorted_radius) * xc;
float ray_dir_y = sin(distorted_radius) * yc;
float ray_dir_z = cos(distorted_radius);

Vector dirc=Vector(ray_dir_x,ray_dir_y,ray_dir_z);

//Camera Setup
Vector* dir_cw=new Vector();
//ray*dir
dir_cw->X = c2w[0][0] * dirc.X + c2w[0][1] * dirc.Y + c2w[0][2] * dirc.Z;
dir_cw->Y = c2w[1][0] * dirc.X + c2w[1][1] * dirc.Y + c2w[1][2] * dirc.Z;
dir_cw->Z = c2w[2][0] * dirc.X + c2w[2][1] * dirc.Y + c2w[2][2] * dirc.Z;

// set origin and direction of the ray
r->o=Eye;
r->dir=*dir_cw;
return true;
}

```

Este algoritmo começa por converter as coordenadas do pixel para coordenadas normalizadas da câmera, e serão diferentes caso haja *camera jitter*. Após isso, calcula a distância desse ponto ao centro da imagem, e se for superior a 1 descarta o raio, visto este se encontrar fora da área de captura. Depois calcula-se o ângulo de distorção, utilizando metade do valor do *FOV* horizontal e a distância normalizada referida anteriormente através da tangente. É utilizado o *FOV* * 0.5 de forma a traduzir coordenadas normalizadas 2D em uma direção 3D que incorpora o ângulo do efeito. A tangente é utilizada pois permite simular bem o efeito, visto que para ângulos pequenos, o ângulo distorcido será quase igual, e quando maior for o ângulo, a tangente cresce mais rápido, e por isso os pontos dos lados aparentam estar mais próximos entre si, e os mais próximos do centro parecem estar mais perto da câmera. Por fim, calcula-se as componentes xyz do raio, de forma a obter o novo ponto face à distorção, convertendo estas de espaço câmera para espaço mundo. Este será a direção do raio e a origem será a câmera.

3.2 Múltiplas Câmaras

Um dos 'efeitos' que decidimos implementar foi um render que mostra 4 diferentes perspetivas do cenário, que podemos visualizar na figura 6 em anexo. Os principais pontos na implementação deste efeito são a função de adicionar câmaras, "addCamera", que nos permite produzir um render com o número de câmaras/perspetivas que for definido e obtém a resolução de cada câmera. E a função "GenerateRay", que calcula o número de câmaras e determina a sua disposição no render final, neste caso é um *grid* de 2x2. Depois calcula e verifica o índice da câmera correspondente às coordenadas fornecidas (x, y), e converte as coordenadas para coordenadas locais da

câmera selecionada. Por fim, gera um raio acedendo à classe "Ray" tendo em consideração os parâmetros passados para cada câmera.

```
// Add a camera to the list
void MultipleCams::addCamera(Camera* cam) {
    cameras.push_back(cam);
    int camWidth, camHeight;
    cam->getResolution(&camWidth, &camHeight);
    if (width == 0 && height == 0) {
        width = camWidth * 2;
        height = camHeight * 2;
    }
}
// Generate a ray from the current camera
bool MultipleCams::GenerateRay(const int x, const int y, Ray* r, const float* cam_jitter) {
    if (cameras.empty()) return false;
    int numCams = cameras.size();
    int cols = 2;
    int rows = 2;
    int subWidth = width / cols;
    int subHeight = height / rows;
    int camX = x / subWidth;
    int camY = y / subHeight;
    int camIndex = camY * cols + camX;
    if (camIndex >= numCams) return false;
    int localX = x % subWidth;
    int localY = y % subHeight;
    bool result = cameras[camIndex]->GenerateRay(localX, localY, r, cam_jitter);
    currentCamIndex = (currentCamIndex + 1) % numCams;
    return result;
}
```

Parâmetros definidos no ficheiro main.cpp, onde podemos escolher o tipo de efeito, neste caso todas as câmaras são Perspective, a sua posição e para onde está a olhar e as restantes características necessárias para construir cada câmera. O render encontra-se em anexo, na figura 6:

```
MultipleCams* multiCam = new MultipleCams();
multiCam->addCamera(new Perspective(Eye, At, Up, W / 2, H / 2, fovWrad, fovHrad));
multiCam->addCamera(new Perspective(Point(0,400,0), Point(35,450,80), Up, W, H / 2,
fovWrad, fovHrad));
multiCam->addCamera(new Perspective(Point(250,475,550), Point(420,-70,0), Up, W / 2,
H / 2, fovWrad, fovHrad));
multiCam->addCamera(new Perspective(Point(213,425,200), Point(250,30,20), Up, W / 2,
H / 2, fovWrad, fovHrad));
cam = multiCam;
```

3.3 Efeito *Swirl*

Para a implementação deste efeito inicialmente implementamos a classe Swirl, que é responsável por gerar a câmera com os atributos passados e criar as variáveis necessárias para calcular a matriz de transformação da câmera para o mundo, a matriz c2w, tal como é encontrado na implementação dos restantes efeitos. Dentro do ficheiro swirl.cpp, podemos encontrar o método GenerateRay, que como nos outros efeitos cria o espaço da câmera, calcula a direção dos raios, tendo em conta a sua origem e direção, e faz ainda a transformação da câmera para o mundo. Neste ficheiro é ainda implementado o Swirl, na função swirlEffect, que é um efeito com um aspeto de redemoinho. Esta função aplica o efeito nas coordenadas xc e yc com base no parâmetro amount. Para isto, inicialmente calcula o raio a partir da origem até o ponto (xc, yc), depois calcula o ângulo original do ponto em relação à origem e adiciona

um termo que causa a distorção em espiral. Por fim, usa o novo ângulo para calcular as novas coordenadas x e y, aplicando a transformação de volta das coordenadas polares para as cartesianas.

```

void swirlEffect(float &xc, float &yc, float amount) {
    float radius = sqrt(xc * xc + yc * yc);
    float angle = atan2(yc, xc) + radius * amount;
    xc = radius * cos(angle);
    yc = radius * sin(angle);
}

...
// Apply swirl effect
float swirl_amount = 6.0f; // Adjust this value for more or less swirl
swirlEffect(xc, yc, swirl_amount);

```

Este efeito pode ser visualizado na figura 8 em anexo.

3.4 Efeito Distorção *Pincushion*

A distorção pincushion resulta num efeito de curvatura para dentro, quando as bordas da imagem ficam mais ampliadas do que o centro.

```

bool Distortion::GenerateRay(const int x, const int y, Ray *r, const float *cam_jitter) {
    float r_norm = sqrt(xc * xc + yc * yc); // xc e yc calculados da mesma forma que em cima
                                                // Apenas retirado para o relatório
    // Apply pincushion distortion
    float distorted_r = r_norm / (1 + k1 * r_norm * r_norm + k2 * r_norm * r_norm * r_norm * r_norm);

    // Calculate the distorted direction
    float ray_dir_x = (xc / r_norm) * distorted_r;
    float ray_dir_y = (yc / r_norm) * distorted_r;
    float ray_dir_z = 1.0f; // The ray points towards the scene along the Z axis

    Vector dirc = Vector(ray_dir_x, ray_dir_y, ray_dir_z);
    dirc.normalize(); // Ensure the direction vector is normalized

    // Camera to World
    Vector dir_cw;
    dir_cw.X = c2w[0][0] * dirc.X + c2w[0][1] * dirc.Y + c2w[0][2] * dirc.Z;
    dir_cw.Y = c2w[1][0] * dirc.X + c2w[1][1] * dirc.Y + c2w[1][2] * dirc.Z;
    dir_cw.Z = c2w[2][0] * dirc.X + c2w[2][1] * dirc.Y + c2w[2][2] * dirc.Z;

    // Set origin and direction of the ray
    r->o = Eye;
    r->dir = dir_cw;
    return true;
}

```

Para aplicar a distorção começamos por converter as coordenadas de pixel para coordenadas normalizadas no espaço câmara. Estas coordenadas normalizadas são de seguida utilizadas para calcularmos a norma. De seguida usamos os coeficientes de distorção, k1 e k2, para aplicar a distorção sobre a norma calculada anteriormente. Após aplicar a distorção calculamos as novas direções das componentes do raio fixando a componente Z a 1.0 para apontar o raio na direção da cena ao longo do eixo Z. Posteriormente, normalizamos a direção do raio e transformamos essa direção do espaço da câmara para o espaço do mundo. Por fim, definimos a origem do raio como Eye, que é a posição da câmara, e a direção do raio como o vetor câmera para o mundo definido anteriormente.

4 Tone Mapping: Uncharted 2

O último tema é referente à escolha de um algoritmo de Tone Mapping melhorado comparativamente ao que é fornecido pelo professor de forma a produzir melhores resultados a nível da qualidade das imagens renderizadas. Desta forma foi escolhido o Uncharted 2 como a solução final, já que consensualmente considerámos que apresenta bons resultados. Para implementarmos este algoritmo seguimos o [recurso fornecido pelo professor \(<https://64.github.io/tonemapping/#charted-2>\)](https://64.github.io/tonemapping/#charted-2). E a sua implementação foi feita num ficheiro à parte, o Uncharted2Tonemap.hpp, para ser mais facilmente usado na implementação dos diferentes formatos de imagem.

```
inline float Uncharted2TonemapCurve(float x) {
    const float A = 0.15f;
    const float B = 0.50f;
    const float C = 0.10f;
    const float D = 0.20f;
    const float E = 0.02f;
    const float F = 0.30f;
    return ((x * (A * x + C * B) + D * E) / (x * (A * x + B) + D * F)) - E / F;
}

inline RGB Uncharted2Tonemap(RGB color) {
    // Apply the tone mapping curve to each channel
    float exposureBias = 2.0f;
    color = color * exposureBias;

    RGB mappedColor;
    mappedColor.R = Uncharted2TonemapCurve(color.R);
    mappedColor.G = Uncharted2TonemapCurve(color.G);
    mappedColor.B = Uncharted2TonemapCurve(color.B);

    // Normalize and map to 0-1 range
    float whiteScale = 1.0f / Uncharted2TonemapCurve(11.2f);
    mappedColor.R *= whiteScale;
    mappedColor.G *= whiteScale;
    mappedColor.B *= whiteScale;

    return mappedColor;
}
```

Para implementarmos este algoritmo inicialmente é necessário definir os valores/coeficientes usados na curva de tone mapping, denominados pelas variáveis A, B, C, D, E e F. Com estes valores estabelecidos e com o argumento "x", que representa o valor de luminância de entrada, podemos então calcular com a formula de tone mapping o novo valor de "x" ajustado para uma faixa dinâmica mais baixa. Este vai ser o valor retornado então pela função "Uncharted2TonemapCurve". Já, a segunda função "Uncharted2Tonemap" começa por definir o valor para "exposureBias" para que este seja usado no ajuste da luminosidade da imagem, multiplicando cada canal de cor por este valor. Isso ajuda então a ajustar o brilho geral da imagem antes de aplicar o tone mapping no passo seguinte. Depois é aplicada a curva de tone mapping para cada canal de cor R, G e B. E por fim, faz-se a normalização e mapeamento para garantir que os valores finais estão numa gama entre 0-1 que é a gama indicada para os dispositivos LDR (Low Dynamic Range). Estas funções de tone mapping são depois usadas em cada um dos formatos de imagem implementados. O resultado e comparação da implementação está em anexo, na figura 9.

Em 6.3 podemos ver uma comparação entre o tone mapping original, o tone mapping final (charted 2), e um outro tipo de tone mapping que decidimos substituir pela técnica de uncharted 2 (Reinhard). O que é possível reparar logo é a diferença no contraste e saturação das cores. No inicial, podemos ver que o contraste entre as cores e as sombras é muito alto, e as cores estão muito saturadas. No intermediário, o contraste é melhor, mas a imagem parece escura, o que resulta numa visibilidade do ruído muito mais acentuada. No final, podemos ver um balanço em termos de brilho, o que nos parece ter qualidade bastante superior.

5 Conclusão

Dado por concluído este projeto, consideramos importante realizar uma análise crítica, elaborar possibilidades para trabalho futuro, e ainda, realizar uma avaliação final do trabalho realizado.

Um ponto positivo do nosso trabalho é o facto de termos conseguido implementar corretamente a totalidade do trabalho ao longo de cada uma das fases e os 3 temas escolhidos nesta última fase.

Por outro lado, existem melhorias que podiam ser feitas tais como, implementar técnicas de multi-processamento para melhorar a eficiência e a performance do pipeline de renderização dado que a implementação sequencial revela um tempo de execução alto.

Para concluir, consideramos que foi obtido um balanço positivo na totalidade do trabalho, apesar das melhorias que poderiam ser efetuadas.

6 Anexos

6.1 Formatos de Imagem

6.1.1 PFM

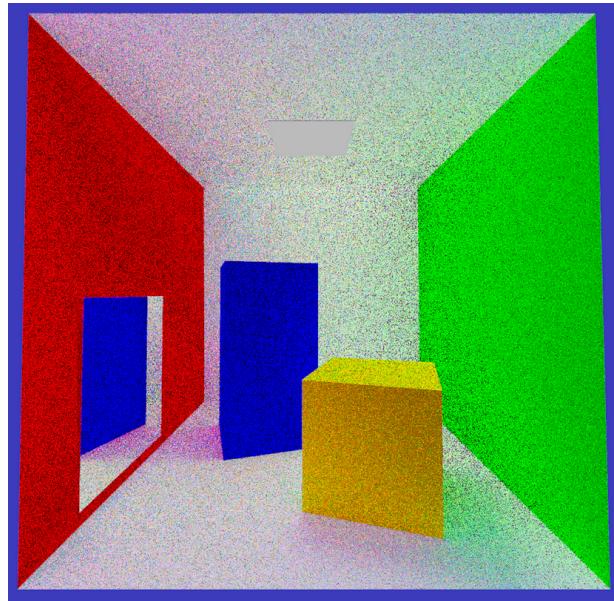


Figura 2: Formato PFM

6.1.2 JPG

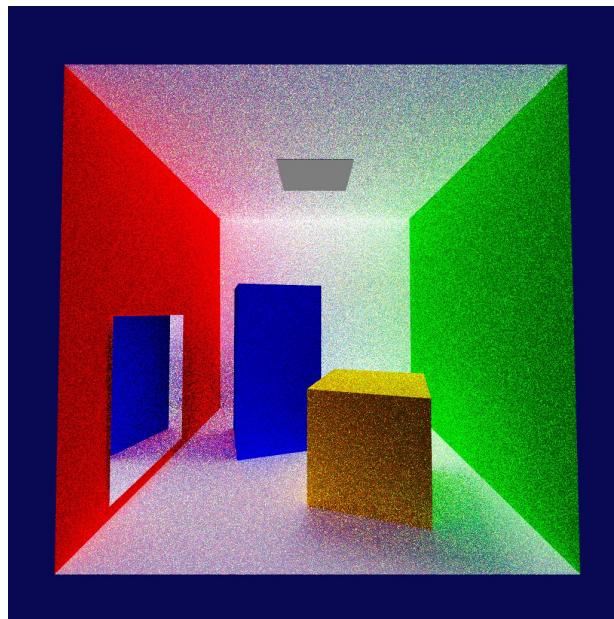
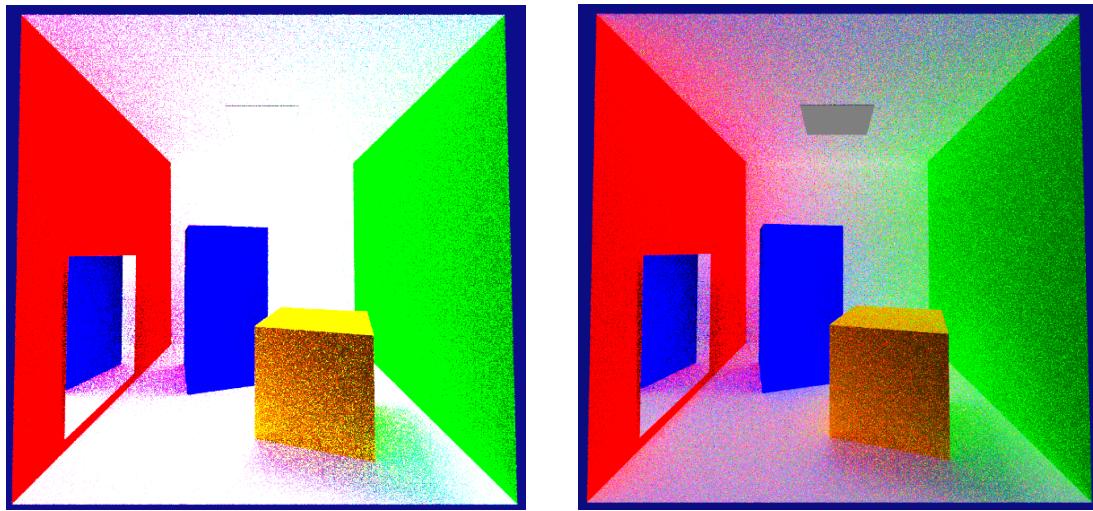


Figura 3: Formato JPG

6.1.3 OpenEXR



(a) Formato OpenEXR sem Tone Mapping

(b) Formato OpenEXR Tone Mapped

Figura 4: Comparaçāo dos Renders OpenEXR

6.2 Cāmaras Alternativas

6.2.1 FishEye

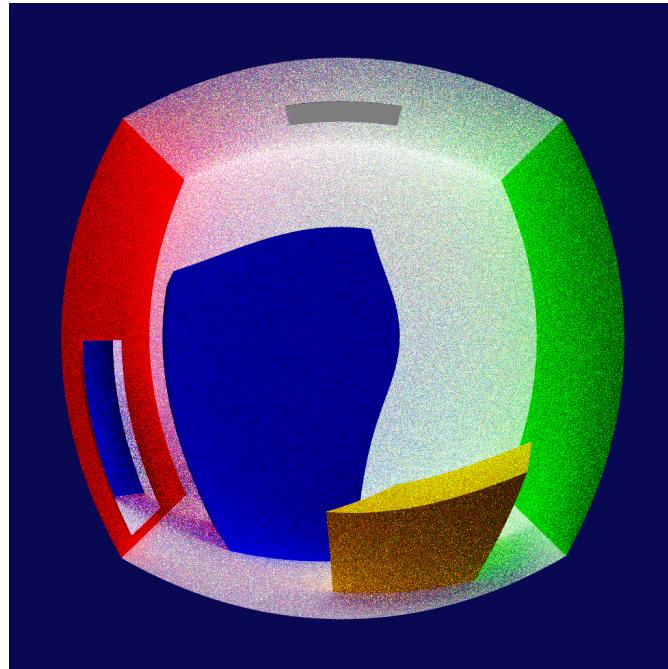


Figura 5: FishEye

6.2.2 Efeito Múltiplas Câmaras

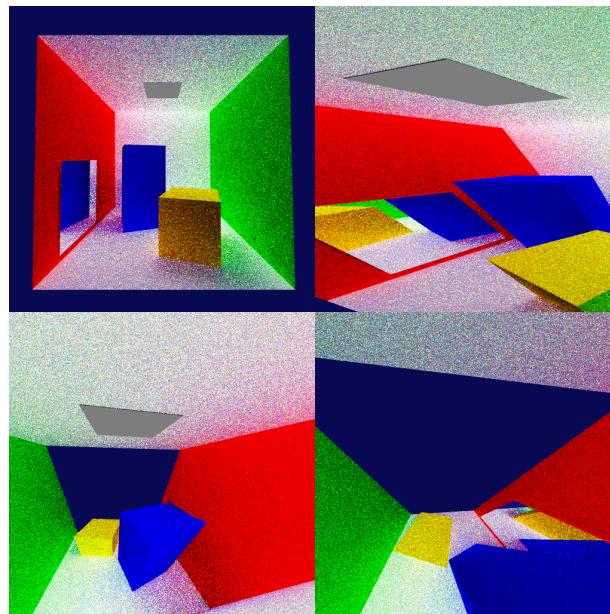


Figura 6: Render com Múltiplas Perspetivas em Simultâneo

6.2.3 Pinchurshion Distortion

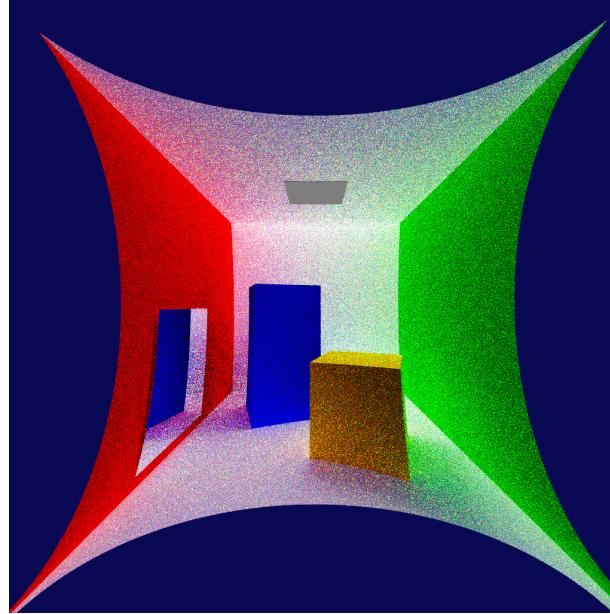


Figura 7: Render com lente Pincushion Distortion

6.2.4 Swirl

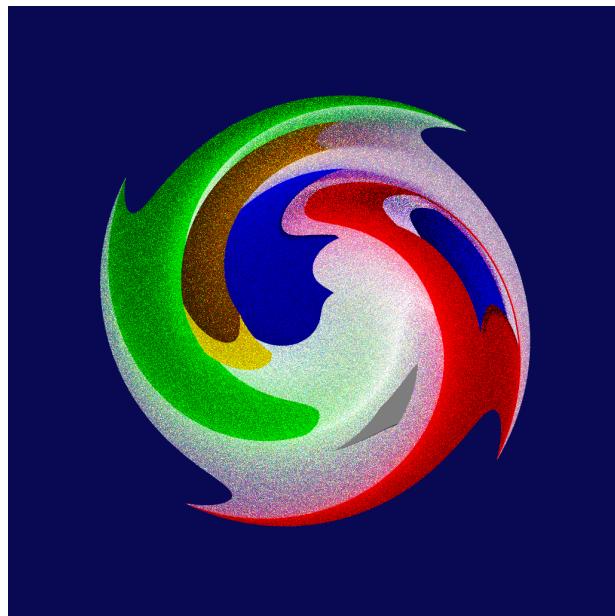


Figura 8: Render com efeito de Swirl

6.3 Tone Mapping Comparaçāo: Original, Reinhard e Uncharted 2

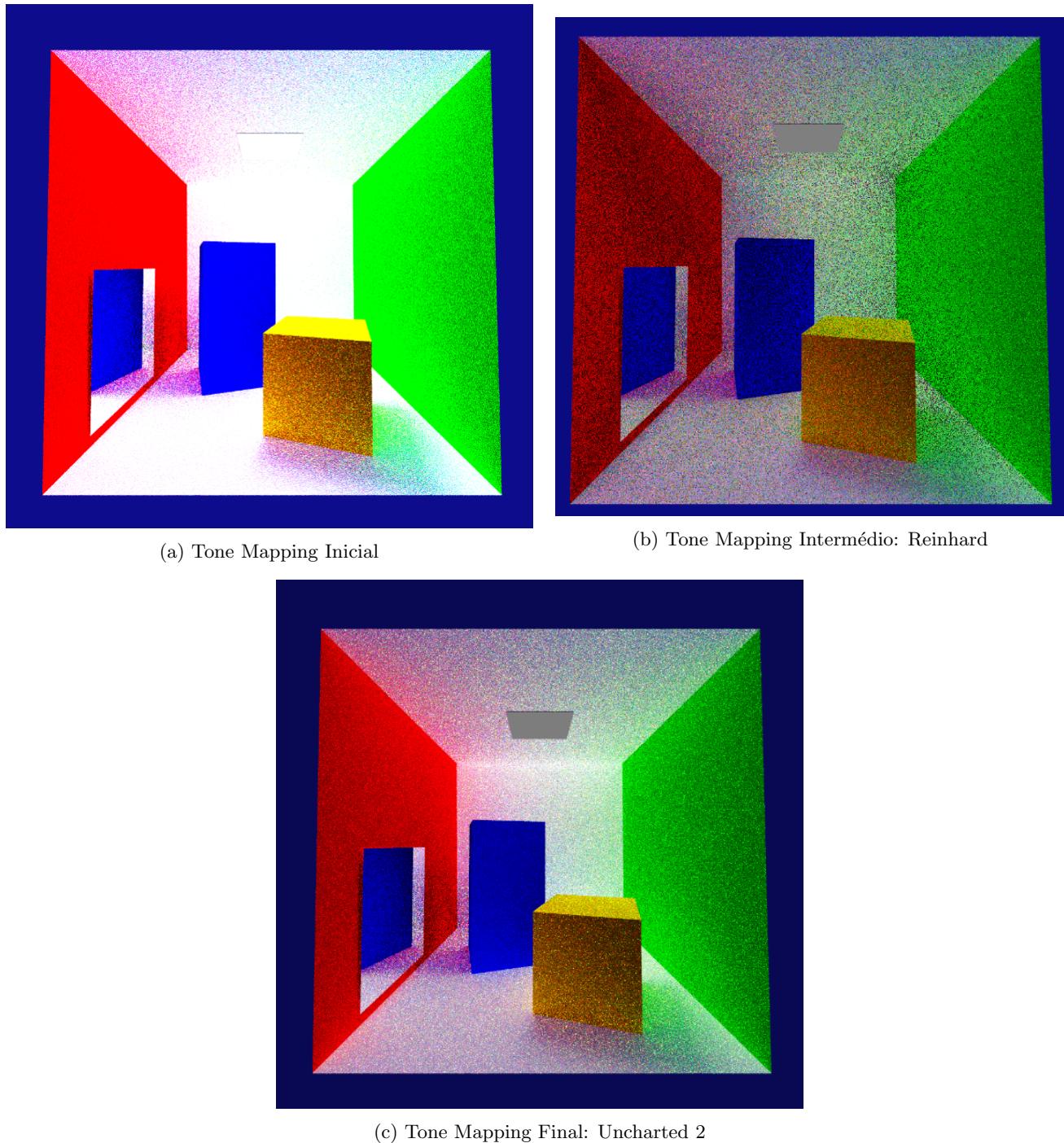


Figura 9: Comparação dos Algoritmos de Tone Mapping Implementados