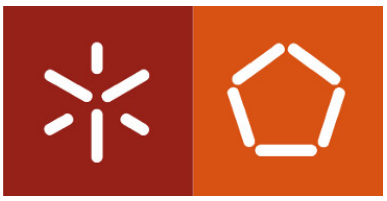
Universidade do Minho

Escola de Engenharia



# Visualização e Iluminação

Mestrado em Engenharia Informática

## Cornell Box

Diogo Vieira - [PG50518]
Laura Rodrigues - [PG50542]
Mariana Marques - [PG50633]

Julho, 2023

# 1. Introdução

Para esta fase houve a tentativa de implementar 3 dos temas propostos pelo docente.

Optámos assim por implementar *Environment Cameras*, na qual se dá a sensação de visualizar a Cornell Box em 360º e *Output Images*, tendo agora disponíveis os formatos PPM, JPG, PFM e OpenEXR.

Houve a tentativa de implementar o tema *Parallel multithreading*, no qual não se obteve sucesso. Este será também elaborado adiante.

Por fim, o terceiro tema implementado com sucesso foi o de *Tone Mapping*.

# 2. Environment Cameras

A **Environment Camera** traça raios em todas as direções ao redor de um ponto na cena, escolhendo pontos em uma esfera que circunda a posição da câmara. Esses pontos na esfera estão associados a coordenadas esféricas ($\theta$,$\phi$), em que $\theta$ varia de 0 a $\pi$ e $\phi$ varia de 0 a $2\pi$. Deste modo, irá ter uma visão 360º horizontalmente e 180º verticalmente.

## 2.1 Coordenadas esféricas

O cálculo das coordenadas esféricas é feito da seguinte forma:

```
float theta = M_PI * (y + cam_jitter[1]) /H;
float phi = 2 * M_PI * (x + cam_jitter[0]) / W;
Vector dir(std::sin(theta) * std::cos(phi), std::cos(theta),
            std::sin(theta) * std::sin(phi));
```

Sendo H e W, a altura e a largura da imagem resultante, respetivamente. E (x,y) as coordenadas do pixel atual.

## 2.2 Resultados

Resultado de utilizar $\phi$ a variar apenas de 0 a $\pi$.

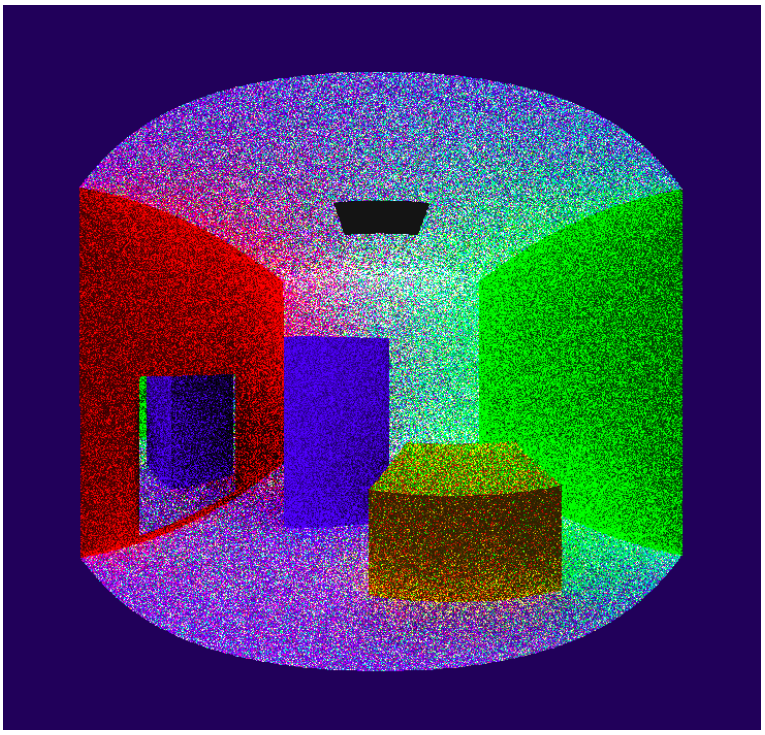


Figura 2.1: Resultado de utilizar $\phi$ a variar apenas de 0 a $\pi$.

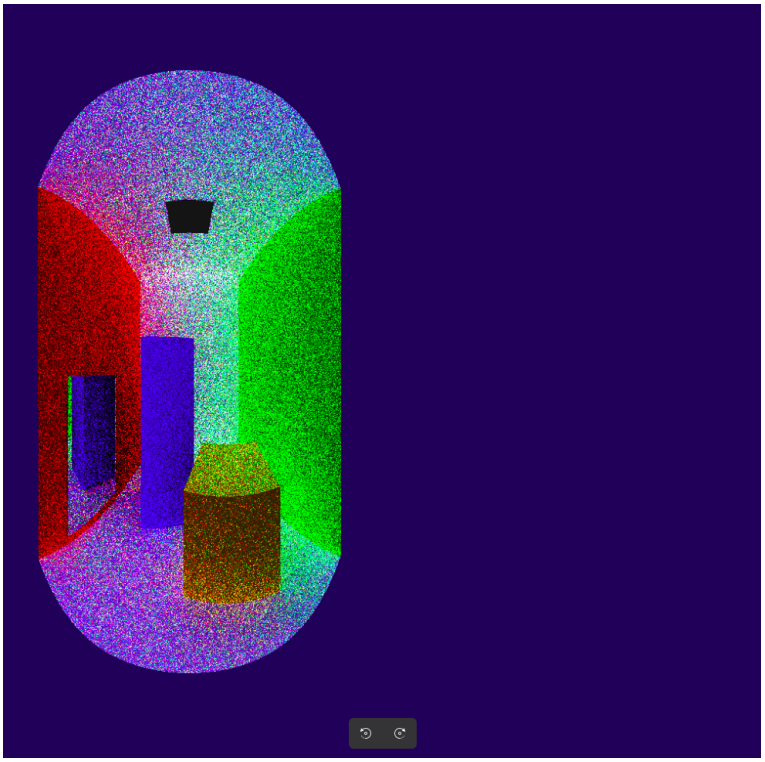


Figura 2.2: Resultado de utilizar $\phi$ a variar de 0 a $2\pi$.

# 3. Output JPG / PFM / OpenEXR Images

## 3.1 JPG

O JPG é um formato de compressão com perda bastante usado para fotografias e imagens com gradientes suaves. Elementos principais:

- Utiliza a biblioteca libjpeg para compressão e descompressão.
- Os dados dos píxeis são transformados numa sequência de bytes comprimida usando o algoritmo JPEG.
- A compressão com perda permite reduzir o tamanho do arquivo ao custo de alguma qualidade de imagem.

A parte principal do código está presente na função ImagePPM::SaveJPG:

```
ToneMap();
...
struct jpeg_compress_struct cinfo;
struct jpeg_error_mgr jerr;

cinfo.err = jpeg_std_error(&jerr);
jpeg_create_compress(&cinfo);
...
jpeg_start_compress(&cinfo, TRUE);
...
while (cinfo.next_scanline < cinfo.image_height) {
    row_pointer[0] = (JSAMPLE *)&imageToSave[cinfo.next_scanline * this->W];
    (void)jpeg_write_scanlines(&cinfo, row_pointer, 1);
}
jpeg_finish_compress(&cinfo);
jpeg_destroy_compress(&cinfo);
...
```

À semelhança do formato PPM começa-se for fazer o ToneMap(). Recorrendo a libjpeg, iniciam-se as estruturas e é iniciado o processo de compressão.

## 3.2 PFM (Portable FloatMap)

O PFM é um formato simples de arquivo que armazena valores de píxeis em formato float, frequentemente usado para imagens de alto alcance dinâmico (HDRI) e aplicações de computação

gráfica. Elementos principais:

- Armazena os dados dos píxeis como decimais, proporcionando alta precisão.
- Suporta imagens em escala de cinza e RGB.
- O cabeçalho é mínimo, contendo informações sobre as dimensões da imagem e a ordem dos bytes.

Assim temos:

```
...
ofs << "PF\n";
ofs << this->W << " " << this->H << "\n";
ofs << "-1.0\n";
...
for (int j = this->H - 1; j >= 0; --j) {
    for (int i = 0; i < this->W; ++i) {
        const RGB &rgb = imagePlane[j * this->W + i];
        ofs.write(reinterpret_cast<const char *>(&rgb.R), sizeof(float));
        ofs.write(reinterpret_cast<const char *>(&rgb.G), sizeof(float));
        ofs.write(reinterpret_cast<const char *>(&rgb.B), sizeof(float));
    }
}
```

Neste formato temos assim em primeiro lugar a escrita do cabeçalho do ficheiro, seguida da informação da imagem. Esta informação é escrita em binário e por ordem inversa (do fim para o ínicio), caso contrário a imagem ficaria invertida.

## 3.3 OpenEXR

O OpenEXR é um formato de arquivo de imagem de alto alcance dinâmico que suporta compressão sem perda e com perda. Elementos principais:

- Oferece suporte a imagens de alto alcance dinâmico e cores profundas, tornando-o adequado para as indústrias de efeitos visuais e animação.
- Utiliza a biblioteca libOpenEXR para ler e gravar arquivos OpenEXR.
- Pode armazenar vários canais de imagem com diferentes tipos de píxeis e configurações de compressão.

Este formato foi implementado da seguinte forma:

```
EXRAux();

try {
    Imf::RgbaOutputFile file(filename.c_str(), W, H, Imf::WRITE_RGBA);
```

```
        // Write the image data to the OpenEXR file
        file.setFrameBuffer(imageToSaveEXR, 1, W);
        file.writePixels(H);

        return true;
    } catch (const std::exception &ex) {
        std::cerr << "Error: " << ex.what() << std::endl;
        return false;
    }
```

Sendo que em EXRAux() temos:

```
    imageToSaveEXR = new Imf::Rgba[W * H];
    for (int j = 0; j < H; j++) {
        for (int i = 0; i < W; ++i) {
            imageToSaveEXR[j * W + i].r = imagePlane[j * W + i].R;
            imageToSaveEXR[j * W + i].g = imagePlane[j * W + i].G;
            imageToSaveEXR[j * W + i].b = imagePlane[j * W + i].B;
            imageToSaveEXR[j * W + i].a = 1.0f;
```

Começamos assim por adicionar o parâmetro *alpha* a todos os píxeis na função auxiliar. Posteriormente cria-se um ficheiro RGBA recorrendo à biblioteca libOpenEXR. Esta fornece métodos já capazes de guardar as informações dos píxeis corretamente.

## 3.4 Resultados

Após iniciar o programa é possível escolher um dos quatro formatos disponíveis (PPM, JPG, PFM e OpenEXR). Todos os formatos foram gerados corretamente.

Como seria de esperar, ao guardar a imagem em JPG é perdida alguma qualidade de imagem.

# 4. Tone Mapping

O tone mapping envolve a conversão de imagens High Dynamic Range (HDR) em imagens de Low Dynamic Range (LDR), preservando a aparência visual e os detalhes. O objetivo é fazer com que a imagem pareça mais natural em monitores padrão que uma têm alcance limitado.

Uma fórmula de tone mapping é a de tone mapping de Reinhard, que é relativamente simples. A referente equação é a seguinte:

$L_{\text{out}} = L_{\text{in}}/(1 + L_{\text{in}})$

Onde $L_{\text{in}}$ é o valor de luminância do píxel na imagem HDR de entrada e $L_{\text{out}}$ é o valor de luminância resultante após o tone mapping.

Para aplicar o tone mapping de Reinhard, é necessário calcular a luminância de cada píxel na imagem HDR, aplicar a equação acima e, em seguida, converter a luminância obtida novamente para o espaço de cores RGB.

```
for (int j = 0; j < H; j++) {
    for (int i = 0; i < W; ++i) {
        // Calculate luminance from RGB values (using approximate luminance
            weights)
        float luminance = 0.2126f * imagePlane[j*W+i].R + 0.7152f *
            imagePlane[j*W+i].G + 0.0722f * imagePlane[j*W+i].B;

        // Apply Reinhard tone mapping
        float toneMappedLuminance = luminance / (1.0f + luminance);

        imageToSave[j*W+i].val[0] = (unsigned char)(toneMappedLuminance *
            imagePlane[j*W+i].R / luminance * 255);
        ...
    }
}
```

## 4.1 Resultados

Tal como seria de esperar, após aplicar a fórmula de Reinhard a imagem ficou mais acinzentada e os brancos ficaram menos intensos.

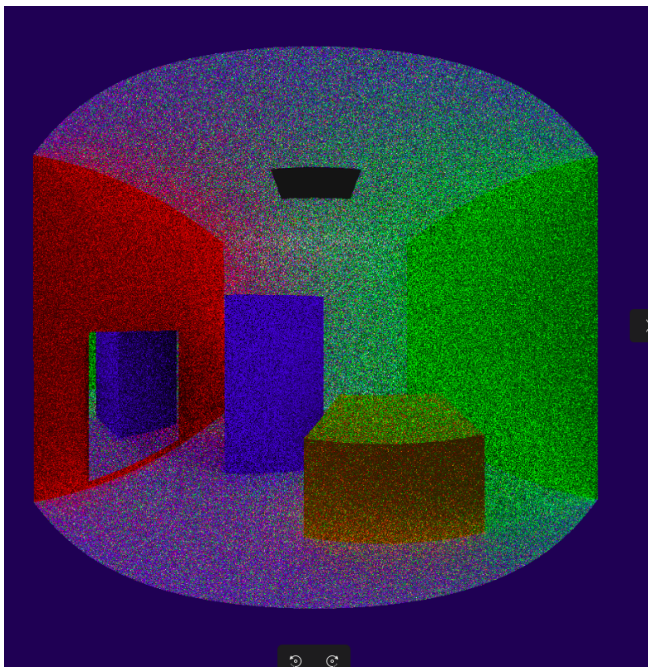


Figura 4.1: Tone Mapping

# 5. Parallel Multithreading

## 5.1 Estratégia Implementada

Para a realização deste projeto realiza-se uma tentativa de implementação de *threads* no *render*. Porém, foram encontrados dois problemas difíceis de resolver, nomeadamente:

- **Serialização da função rand()** - De forma a resolver este problema, foram desenvolvidas duas funções, dadas por omp_srand() e omp_rand, responsáveis por gerar números aleatórios com a atualização do valor da *seed* por iteração. É de salientar que o valor da *seed* é devolvido em vez de estar presente uma *seed* escondida como a função srand().

```
typedef unsigned long long rand_state;
#define OMP_RAND_MAX 0xffffffff
static const int multiplier = 314159269;
static const int addend = 1;
static const int modulus = 0xffffffff;
rand_state omp_srand() {
    rand_state state = time(NULL);
    state ^= (unsigned long long)omp_get_thread_num() << 32;
    return state; }
int omp_rand(rand_state *state) {
    *state = *state * multiplier + addend;
    return *state & modulus; }
void StandardRenderer::Render () {
    int W,H; // resolution
```

```
    cam->getResolution(&W,&H);
#pragma omp declare reduction(+: RGB: omp_out += omp_in)
    RGB color;
    const int spp=16;
#pragma omp parallel for reduction(+:color)
    for (int y=0; y < H; y++) { // loop over rows
        for (int x=0; x< W; x++) { // loop over columns
            color = RGB(0.,0.,0.);
            for (int ss = 0; ss < spp; ss++) {
                ...
                float jitterV[2];
                rand_state seed = omp_srand();
                jitterV[0] = (float)omp_rand(&seed) / OMP_RAND_MAX;
                jitterV[1] = (float)omp_rand(&seed) / OMP_RAND_MAX;
                ...
            }
...
```

- **Data Races** - Em relação aos *dataraces* foi difícil detetar, visto que, como é um projeto com bastantes classes, não se conseguiu detetar a origem do erro. É de salientar o uso da diretiva **reduction(+ color)**, visto que é uma operação atómica que causa concorrência entre threads.

## 5.2 Resultados

Como não foi possível resolver as *dataraces*, ou talvez a serialização, o resultado é, de facto, melhor ao utilizar apenas uma thread.

1. **Sem Multithreading** - 19.879 segundos
2. **Com Multithreading** - 25.737 segundos

# 6. Conclusão

Em suma, conseguimos fazer com sucesso **3** dos temas propostos, como foi explicado anteriormente, porém o tema Parallel MultiThreading não foi conseguido, mas também foi explicada a nossa ideia e as dificuldades que enfrentámos.

Como trabalho futuro gostaríamos de implementar corretamente o tema mencionado anteriormente, porque a implementação sequencial é muito demorada.

Em suma, foi um projeto que enriqueceu todo o nosso conhecimento na área de computação gráfica mais concretamente Ray-Tracing, tanto em termos teóricos como práticos.