



# 图形绘制技术

## 大作业实验报告

作者：顾秋涵

组织：南京大学计算机科学与技术系

时间：2023.7

学号：201830204



让科技和艺术保持平衡。——帕特里克·汉拉汉

# 目录

<b>第1章 光线追踪降噪</b>	<b>1</b>
1.1 问题描述 . . . . .	1
1.1.1 求解渲染方程 . . . . .	1
1.1.2 蒙特卡洛积分方法 . . . . .	1
1.1.3 产生的问题 . . . . .	2
1.2 解决思路 . . . . .	2
1.3 实验内容 . . . . .	3
1.3.1 修改 Moer-lite 框架 . . . . .	3
1.3.2 图像噪声模型 . . . . .	8
1.3.2.1 高斯噪声 . . . . .	8
1.3.2.2 椒盐噪声 . . . . .	10
1.3.3 传统滤波器降噪 . . . . .	11
1.3.3.1 中值滤波 . . . . .	11
1.3.3.2 均值滤波 . . . . .	13
1.3.3.3 高斯滤波 . . . . .	14
1.3.3.4 双边滤波 . . . . .	16
1.3.4 Intel Open Image Denoise 工业降噪库 . . . . .	18
1.4 实验结果与分析 . . . . .	19
1.4.1 添加噪声 . . . . .	19
1.4.2 中值滤波 . . . . .	20
1.4.3 均值滤波 . . . . .	22
1.4.4 高斯滤波 . . . . .	25
1.4.5 双边滤波 . . . . .	26
1.4.6 Intel Open Image Denoise 降噪库 . . . . .	28
<b>第2章 实验总结与注意点</b>	<b>31</b>
2.1 去噪总结 . . . . .	31
2.2 实验注意点 . . . . .	31
<b>第3章 思考创新与展望</b>	<b>32</b>
<b>第4章 课程与实验收获</b>	<b>33</b>
<b>参考文献</b>	<b>34</b>

# 第1章 光线追踪降噪

## 1.1 问题描述

### 1.1.1 求解渲染方程

首先给出渲染方程：

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\omega_i \in \Omega_+} f_r(\omega_i, x, \omega_o) L_i(x, \omega_i) \cos \theta_i d\omega_i \quad (1.1)$$

求解渲染方程即求解一个积分函数。由于渲染方程中的被积函数的原函数很难找到，利用经典积分方法得不到积分结果。因此使用蒙特卡洛积分方法 (Monte Carlo Integration)，利用一个随机变量对被积函数进行采样，并将采样值进行一定的处理，当采样数量很高时，得到的结果近似原积分的结果。

### 1.1.2 蒙特卡洛积分方法

#### 1. 前提知识

由概率论基本知识，假设一连续型随机变量  $X$  的样本空间为  $D$ ，其概率密度分布函数为  $p(x)$ ，则其数学期望为

$$E[X] = \int_D p(x) dx \quad (1.2)$$

若另一连续随机变量  $Y$  满足  $Y = f(X)$ ，则  $Y$  的数学期望  $E[Y]$  可由下式给出

$$E[Y] = \int_D f(x)p(x) dx \quad (1.3)$$

#### 2. 蒙特卡洛积分与重要性采样

根据以上叙述，假设需要计算一个一维积分式

$$A = \int_a^b f(x) dx \quad (1.4)$$

借助蒙特卡洛积分方法，可以构造

$$F_N = \frac{b-a}{N} \sum_{i=1}^N f(X_i) \quad (1.5)$$

其中的每一个  $X_i (i = 1, 2, 3, \dots, N)$  为  $[a, b]$  之间的均匀连续随机变量，所有的  $X_i$  组成一个随机变量集合。可以证得， $F_N$  的数学期望  $E[F_N]$  即为要求的结果  $A$ ，随着  $N$  的增加， $F_N$  就越逼近理论上  $A$  的值。

对于与原积分区间相同，但却不是均匀分布的一般随机变量，上述结论仍成立。假设随机变量  $X$  的概率密度分布函数已知，为  $p(x)$ ，构造

$$F_N = \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)} \quad (1.6)$$

再构造随机变量

$$Y = \frac{f(X)}{p(X)} \quad (1.7)$$

这里仅要求随机变量  $X$  的概率密度分布函数  $p(X)$  已知且在  $X$  的样本空间内  $p(X) \neq 0$ 。同样可以证得， $F_N$  的数学期望  $E[F_N]$  即为要求的结果  $A$ 。综合上述叙述，蒙特卡洛积分方法如下

$$\int_D f(X) dx = \lim_{n \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)} \quad (1.8)$$

要准确得出蒙特卡洛积分方法的收敛速度特性，需要分析随着样本数量  $N$  的增加，估计值  $F_N$  的方差  $\sigma^2[F_N]$  的变化情况，计算如下 [3]:

$$\begin{aligned} \sigma^2[F_N] &= \sigma^2\left[\frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)}\right] \\ &= \frac{1}{N^2} \sum_{i=1}^N \sigma^2\left[\frac{f(X_i)}{p(X_i)}\right] \\ &= \frac{1}{N^2} \sum_{i=1}^N \sigma^2[Y_i] \\ &= \frac{1}{N^2}(N\sigma^2[Y]) \\ &= \frac{1}{N}\sigma_2[Y] \end{aligned}$$

所以

$$\sigma^2[F_N] = \frac{1}{\sqrt{N}}\sigma_2[Y] \quad (1.9)$$

由式(1.9)可以得出，估计值的不稳定来源于随机变量  $Y$  的取值不稳定， $\frac{f(X_i)}{p(X_i)}$  因不同  $X_i$  的取值变化地越剧烈，则  $Y$  的方差越大，估计值的收敛速度越慢。因此， $p(x)$  的形状越接近  $f(x)$ ，越有益于最终结果的收敛。上述思想即为“重要性采样”方法，即对积分值有重要贡献( $f(x)$  较大)的被积函数区间，需要以较大概率生成处于这个区间附近的随机变量，用于快速逼近理论值。这也是引入任意分布随机变量的蒙特卡洛积分方法，而不满足于利用均匀分布随机变量来求蒙特卡洛积分的原因。

### 1.1.3 产生的问题

上述经典的蒙特卡洛积分方法有两大优势：

1. 提供了一个统一的框架来渲染几乎所有基于物理的渲染模型。
2. 大多数 MC 方法都能保证数学收敛到真实值，这是高质量渲染的关键。

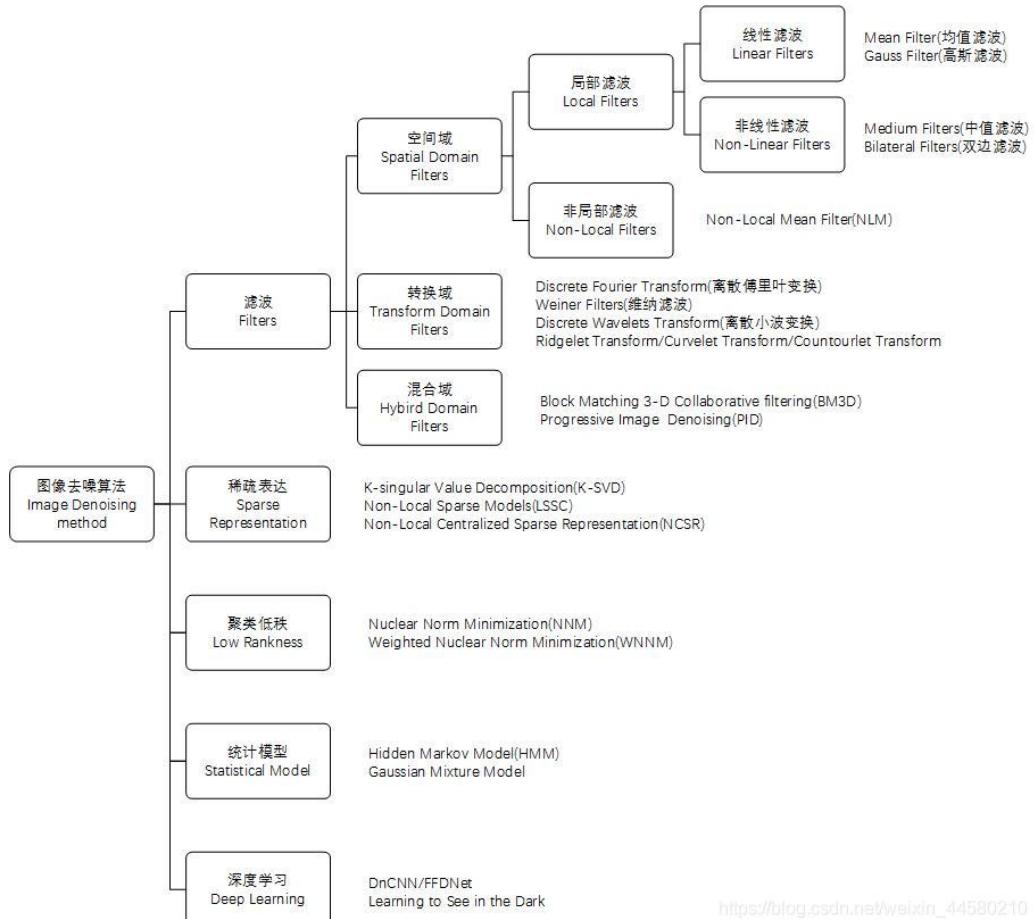
然而问题在于，虽然蒙特卡洛积分方法得到的结果估计值与理论值之间的误差可以通过增加样本数来减小，但收敛速率仅为  $O(\sqrt{N})$ ，因此需要大量样本才能实现可靠的收敛。尽管计算能力不断增加，现实渲染的成本仍然是一个限制和实际约束，因为它需要几个小时来渲染一个高质量的图像或帧。当使用较少的样本时，蒙特卡洛积分结果往往会受到估计量方差的影响，产生视觉干扰。同时，光线追踪的随机性，也可能导致渲染图像中出现噪点。大量的计算消耗是阻碍蒙特卡洛积分更广泛应用的主要因素之一。

## 1.2 解决思路

为了解决这一问题，现有的解决方案分为两类：过程中采样(in-process sampling)方案和后处理重建(post-processing reconstruction)方案[8]。后处理方案被称为蒙特卡洛降噪，是渲染界研究最多的领域之一。

蒙特卡洛降噪旨在通过使用具有低样本计数的噪声数据以及辅助特征，将噪点较多的输入  $x$  映射到一个更加真实的渲染结果  $r$ 。这种映射过程通过对样本数目的增加和利用附加信息来减少噪声，并提高渲染图像的质量和真实感[8]。

降噪的方法大致可分为传统滤波方法和深度学习方法。空间域滤波器方法是传统图片降噪的最基础的一类方法，包括均值滤波、高斯滤波、双边滤波等。而在深度学习引入图像处理领域后，神经网络方法逐渐取代了传



[https://blog.csdn.net/weixin\\_44580210](https://blog.csdn.net/weixin_44580210)

图 1.1: 图像降噪算法分类

统的滤波方法。早期神经网络图像降噪方法大多通过引用图像空间的辅助特征，如：albedo、normal 等信息，提供给神经网络，让网络进行降噪处理，如：KPCN 方法 [1]。近年来基于光线追踪采样样本的降噪方法被证实具有更优的降噪效果，如：SBMC 方法 [5]，该方法除了使用图像空间的信息，还保存了光线追踪过程中，采样的信息，如：光源采样的方向等。

CSDN 上一位博主 [2] 对图像降噪算法进行了总结分类。如图 1.1 所示。

## 1.3 实验内容

本次实验一共完成了四项内容：

- 实现了添加两类噪声的功能，并集成到 Moer-lite 系统中。
- 实现了四种传统滤波器降噪方法，并集成到 Moer-lite 系统中。
- 集成 Intel Open Image Denoise 降噪库到 Moer-lite 系统中。
- 对比不同滤波器的优缺点和过滤不同噪声的性能。

### 1.3.1 修改 Moer-lite 框架

。在 Moer-lite 框架中，为了实现加噪声、去噪的功能，需要修改框架代码，主要修改部分如下：

- 在 More-lite 中，创建一个 Noise 类用于添加噪声，代码如下：

```

1 // FunctionLayer/Noise/Noise.h
2 #pragma once
3 #include <ResourceLayer/Image.h>

```

```

4 #include <memory>
5 #include <vector>
6
7 // 实现增加噪声
8 class Noise {
9 public:
10    //参数可以手动修改
11    void AddSaltNoise(const std::shared_ptr<Image>& img, int n=500);
12    void AddGaussianNoise(const std::shared_ptr<Image>& img, double mu=0, double sigma
13                           =25);
14 };

```

- 在 More-lite 中，创建一个 Filter 类集成传统滤波器，代码如下：

```

1 //FunctionLayer/Filter/Filter.h
2 #pragma once
3 #include <ResourceLayer/Image.h>
4 #include <memory>
5 #include <vector>
6
7 // 实现传统滤波器
8 class Filter {
9 public:
10    //参数可以手动修改
11    void MedianFilter(const std::shared_ptr<Image>& src, std::shared_ptr<Image>& dst,
12                      double size=3);
13    void MeanFilter(const std::shared_ptr<Image>& src, std::shared_ptr<Image>& dst,
14                     double size=3);
15    void GaussianFilter(const std::shared_ptr<Image>& src, std::shared_ptr<Image>& dst
16                         ,double size=23, double sigma=0.8);
17    std::vector<std::vector<double>> GaussianTemplate(double size, double sigma);
18    void BilateralFilter(const std::shared_ptr<Image>& src, std::shared_ptr<Image>&
19                          dst, int size=3, float sigmaD=10, float sigmaR=35);
20    std::vector<float> RangeTemplate(int range, float sigmaR);
21 };

```

- 在 Film 类中调用 Filter 类和 Noise 类，添加存储降噪后图片的功能，代码如下：

```

1 //FunctionLayer/Film/Film.h
2 #pragma once
3 #include <CoreLayer/ColorSpace/Spectrum.h>
4 #include <ResourceLayer/Image.h>
5 #include <ResourceLayer/JsonUtil.h>
6 #include <FunctionLayer/Filter/Filter.h>
7 #include <FunctionLayer/Noise/Noise.h>
8
9 class Film {
10 public:
11    Film() = delete;
12    Film(const Json &json) {
13        size = fetchRequired<Vector2i>(json, "size");

```

```

14     image = std::make_shared<Image>(size);
15     deimage = std::make_shared<Image>(size);
16 }
17
18 void deposit(const Vector2i xy, const Spectrum &spectrum) {
19     /* 无论光谱内部实现如何，写入图片时均转为3通道格式
20     Vector3f v = toVec3(spectrum);
21     image->setValue(xy, v);
22 }
23
24 void savePNG(const char *filename) { image->savePNG(filename); }
25 void saveHDR(const char *filename) { image->saveHDR(filename); }
26 void savePFM(const char *filename) { image->savePFM(filename); }
27
28 //滤波
29 void filter(const int type){
30     if(type==1) Filter().MedianFilter(image,deimage);
31
32     if(type==2) Filter().MeanFilter(image,deimage);
33
34     if(type==3) Filter().GaussianFilter(image,deimage);
35
36     if(type==4) Filter().BilateralFilter(image,deimage);
37 }
38
39 //加噪声
40 void addNoise(const int type){
41     if(type==1) Noise().AddSaltNoise(image);
42
43     if(type==2) Noise().AddGaussianNoise(image);
44 }
45
46 //存储滤波后的图片
47 void save_dePNG(const char *filename) { deimage->savePNG(filename); }
48 void save_deHDR(const char *filename) { deimage->saveHDR(filename); }
49
50 public:
51     Vector2i size;
52
53 protected:
54     std::shared_ptr<Image> image = nullptr;
55     std::shared_ptr<Image> deimage = nullptr; //降噪后的image
56 };

```

- 修改 main.cpp 文件，在完成渲染后，调用 Film 类中的加噪声函数和滤波函数，并进行耗时统计，添加的代码如下：

```

1 //main.cpp
2 //main函数最后增加对含有三个参数的命令的判断：
3 //需要加噪声/滤波

```

```

4   if (argc == 4) {
5     // 命令格式为 "./Moer <样例路径> <加噪声> <去噪>"
6     const std::string noise = std::string(argv[2]); // 获取第三个参数的值, 噪声类型
7     const std::string denoise = std::string(argv[3]); // 获取第四个参数的值, 去噪类型
8     // 存储加噪声的图片
9     std::string outputName1 = fetchRequired<std::string>(json["output"], "filename")
10    ;
11    std::size_t extensionPos1 = outputName.find_last_of(".");
12    // 存储去噪的图片
13    std::string outputName2 = fetchRequired<std::string>(json["output"], "filename")
14    ;
15    std::size_t extensionPos2 = outputName.find_last_of(".");
16
17    // 加噪声
18    printf("\n*****Begin Add Noise *****\n");
19    // 加椒盐噪声
20    if(noise=="add_salt_noise") {
21      camera->film->addNoise(1);
22      if (extensionPos1 != std::string::npos) {
23        outputName1.insert(extensionPos1, "_add_salt_noise");
24      }
25    }
26    // 加高斯噪声
27    else if(noise=="add_g_noise") {
28      camera->film->addNoise(2);
29      if (extensionPos1 != std::string::npos) {
30        outputName1.insert(extensionPos1, "_add_g_noise");
31      }
32    }
33    // 不加噪声
34    else if(noise=="none") {
35    }
36    std::cout<<"Only support add_salt_noise , add_g_noise , none\n";
37  }
38
39  // 存储加过噪声的图片
40  if (std::regex_match(outputName1, std::regex("(.*)(\\.png)"))) {
41    camera->film->savePNG(outputName1.c_str());
42  } else if (std::regex_match(outputName1, std::regex("(.*)(\\.hdr)"))) {
43    camera->film->saveHDR(outputName1.c_str());
44  } else {
45    std::cout << "Only support output as PNG/HDR\n";
46  }
47
48  std::cout<< "Save noise image to "<<outputName1<<std::endl;
49

```

```

50     //去噪
51     auto s = std::chrono::system_clock::now();
52     auto e = std::chrono::system_clock::now();
53     printf("\n*****Begin Denoise *****\n");
54
55     if(denoise=="median_denoise") {
56         camera->film->filter(1);
57         e = std::chrono::system_clock::now();
58         if (extensionPos2 != std::string::npos) {
59             outputName2.insert(extensionPos2, "_median_denoise");
60         }
61     }
62
63     //均值去噪
64     else if(denoise=="mean_denoise") {
65         camera->film->filter(2);
66         e = std::chrono::system_clock::now();
67         if (extensionPos2 != std::string::npos) {
68             outputName2.insert(extensionPos2, "_mean_denoise");
69         }
70     }
71     //高斯去噪
72     else if(denoise=="g_denoise") {
73         camera->film->filter(3);
74         e = std::chrono::system_clock::now();
75         if (extensionPos2 != std::string::npos) {
76             outputName2.insert(extensionPos2, "_g_denoise");
77         }
78     }
79     //双边去噪
80     else if(denoise=="bi_denoise") {
81         camera->film->filter(4);
82         e = std::chrono::system_clock::now();
83         if (extensionPos2 != std::string::npos) {
84             outputName2.insert(extensionPos2, "_bi_denoise");
85         }
86     }
87     //Intel降噪库去噪
88     else if(denoise=="intel_denoise") {
89         //构造输入pfm图片
90         camera->film->savePFM("tmp.pfm");
91         //执行Intel降噪库的sh命令
92         const char* command = "./oidnDenoise -hdr tmp.pfm -o re.pfm";
93         //使用system函数执行Shell命令
94         system(command);
95         std::cout<< "Save denoised image to re.pfm"<<std::endl;
96         printf("\nIntel Denoising costs %.2fs\n", (std::chrono::duration_cast<std::
97             chrono::milliseconds>(e - s)).count() / 1000.f);
98         return 0;

```

```

98     }
99     // 不去噪声
100    else if(denoise=="none") {
101        printf("\nIntel Denoising costs %.2fs\n", (std::chrono::duration_cast<std:::
102            chrono::milliseconds>(e - s)).count() / 1000.f);
103        return 0;
104    }
105    else {
106        std::cout<<"Only support median_denoise, mean_denoise, g_denoise, bi_denoise,
107        intel_denoise, none\n";
108    }
109
110    // 输出降噪后的图片
111    if (std::regex_match(outputName2, std::regex("(.*)(\\.png)"))) {
112        camera->film->save_dePNG(outputName2.c_str());
113    } else if (std::regex_match(outputName2, std::regex("(.*)(\\.hdr)"))) {
114        camera->film->save_deHDR(outputName2.c_str());
115    } else {
116        std::cout << "Only support output as PNG/HDR\n";
117    }
118
119    std::cout<< "Save denoised image to "<<outputName2<<std::endl;
120    // 计算去噪用时
121    printf("\nIntel Denoising costs %.2fs\n", (std::chrono::duration_cast<std::chrono
122        ::milliseconds>(e - s)).count() / 1000.f);
123

```

- 增加执行 Moer 命令的参数，调用 main.cpp 中的相应代码

```

1 // 1. 不进行滤波和降噪，只传入一个参数
2 ./Moer <场景描述文件的目录>
3
4 // 2. 进行滤波/降噪，需要传入三个参数
5 ./Moer <场景描述文件的目录> <add_salt_noise/add_g_noise/none> <mean_denoise/
  median_denoise/g_denoise/bi_denoise/none>

```

## 1.3.2 图像噪声模型

在降噪前，先对不同的图像噪声进行建模，以便对比同一滤波器对不同噪声的滤波效果。本次实验中主要实现椒盐噪声和高斯噪声的建模。

图像降噪模型可以建模为：

$$y = x + n \quad (1.10)$$

其中， $y$  是观察到的噪声图像， $x$  是图像真值， $n$  是图像噪声，图像降噪过程就是通过  $y$  获取  $x$ ，在许多论文中将这个过程描述为不可逆过程 [7]，这也就是为什么图像降噪难度较大。

### 1.3.2.1 高斯噪声

#### 1. 定义：

高斯噪声是最常见也是最重要的的一种噪声，众多的图像降噪算法都以降低高斯噪声为目标设计，其概率密度函数为

$$p(z) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(z-\mu)^2/2\sigma^2} \quad (1.11)$$

其中， $\sigma$  是标准偏差， $\mu$  是灰度值的平均值，这个公式说明的是灰度值为  $z$  的概率为多少。

## 2. 代码思路：

随机生成两个随机数，利用 Box-Muller 公式将两个均匀分布的随机数转换为服从均值为  $\mu$ ，标准差为  $\sigma$  的高斯分布的随机数，并且控制添加噪声的数据范围，使之适合降噪 [10]。在数据范围内给像素点加上符合高斯分布的值。

## 3. 代码集成：

在 More-lite 中，集成生成高斯噪声的代码如下：

```

1 // FunctionLayer/Noise/Noise.cpp
2 // 添加高斯噪声
3 void Noise::AddGaussianNoise(const std::shared_ptr<Image>& img, double mu, double sigma)
4 {
5     int width = img->size[0];
6     int height = img->size[1];
7     int channels = Image::channels;
8
9     int RandomArray[width+1][height+1];
10    for (int i = 0; i < width; i++) {
11        for (int j = 0; j < height; j++) {
12            RandomArray[i][j] = (rand() % (255 - 0 + 1)) + 0;
13            //要取得[a,b]的随机整数，使用(rand() % (b-a))+ a;
14            //要取得[a,b]的随机整数，使用(rand() % (b-a+1))+ a;
15            //要取得(a,b]的随机整数，使用(rand() % (b-a))+ a + 1;
16        }
17    }
18    //计算要添加噪声的数据范围小于20%
19    int down1 = 0, up1 = 255 * 1 / 20; //0-0.05
20
21    for (int i = 0; i < width; i++) {
22        for (int j = 0; j < height; j++) {
23            if (RandomArray[i][j] < up1 && RandomArray[i][j] > down1) {
24                Vector2i xy(i, j);
25                if (channels == 1) { //只有一个通道，是灰度图
26                    // 构建高斯噪声
27                    double u1, u2;
28                    do {
29                        u1 = rand() * (1.0 / RAND_MAX);
30                        u2 = rand() * (1.0 / RAND_MAX);
31                    } while (u1 <= std::numeric_limits<double>::min()); // u1不能为0
32
33                    double z = sigma * sqrt(-2.0 * log(u1)) * cos(2 * PI * u2) + mu;
34
35                    Vector3f pixelVal = img->getValue(xy);
36                    int val = static_cast<int>(pixelVal[0] + z * 32);
37                    val = (val < 0) ? 0 : val;

```

```

38         val = (val > 255) ? 255 : val;
39
40         Vector3f newPixelVal(val, val, val);
41         img->setValue(xy, newPixelVal);
42     } else { // 多个通道，是彩色图
43
44         for (int k = 0; k < channels; k++) {
45             // 构建高斯噪声
46             // 产生随机数
47             double u1, u2;
48             do {
49                 u1 = rand() * (1.0 / RAND_MAX);
50                 u2 = rand() * (1.0 / RAND_MAX);
51             } while (u1 <= std::numeric_limits<double>::min()); // u1 不能为 0
52
53             // 利用 Box-Muller 公式将两个均匀分布的随机数转换为服从均值为 mu,
54             // 标准差为 sigma 的高斯分布的随机数 z
55             double z = sigma * sqrt(-2.0 * log(u1)) * cos(2 * PI * u2) + mu;
56
57             Vector3f pixelVal = img->getValue(xy);
58
59             int val = static_cast<int>(pixelVal[k] + z*32);
60             val = (val < 0) ? 0 : val;
61             val = (val > 255) ? 255 : val;
62
63             pixelVal[k] = val;
64             img->setValue(xy, pixelVal);
65         }
66     }
67 }
68 }
69 }

```

### 1.3.2.2 椒盐噪声

#### 1. 定义：

椒盐噪声又称脉冲噪声、尖峰噪声，在图像上表现为随机分布的黑白点，其概率密度函数为

$$p(z) = \begin{cases} P_a, & z = a \\ P_b, & z = b \\ 1 - P_a - P_b, & otherwise \end{cases} \quad (1.12)$$

椒盐噪声可以通过中值滤波器进行消除。

#### 2. 代码思路：

在代码中利用一个变量 n 来控制添加的椒盐噪声的数量。各生成 n 个黑点和 n 个白点随机分布。

#### 2. 代码集成：

在 More-lite 中，集成生成椒盐噪声的代码如下：

```
// FunctionLayer/Noise/Noise.cpp
```

```

2 // 添加椒盐噪声
3 void Noise::AddSaltNoise(const std::shared_ptr<Image>& img, int n) {
4     int width = img->size[0];
5     int height = img->size[1];
6
7     for (int k = 0; k < n; k++) {
8         int x = rand() % width;
9         int y = rand() % height;
10
11         Vector3f whitePixel(255, 255, 255); // White color
12
13         img->setValue(Vector2i(x, y), whitePixel);
14     }
15
16     for (int k = 0; k < n; k++) {
17         int x = rand() % width;
18         int y = rand() % height;
19
20         Vector3f blackPixel(0, 0, 0); // Black color
21
22         img->setValue(Vector2i(x, y), blackPixel);
23     }
24 }
```

### 1.3.3 传统滤波器降噪

空间域滤波器可以分为局部滤波器和非局部滤波器，其中局部滤波器又分为线性滤波器和非线性滤波器。经典的局部滤波器如中值滤波、均值滤波、高斯滤波、双边滤波，其中均值滤波和高斯滤波为线性滤波器，中值滤波和双边滤波为非线性滤波。

#### 1.3.3.1 中值滤波

##### 1. 定义：

如图 1.3(a)所示，以目标像素周围  $3 \times 3$  的邻域为例，就是将  $3 \times 3$  邻域中九个像素灰度值进行排序，将中间灰度值作为目标像素的灰度值。由于椒盐噪声影响的像素的灰度值通常都非常大或者非常小，因此通过排序会被消除掉。

相比于其他三种传统滤波器，中值滤波的特点有三：

- 没有引入新的像素值
- 有利于消除脉冲噪声、椒盐噪声
- 保边

##### 2. 代码思路：

- 传入的参数：保存原图的 Image 对象、保存输出图的 Image 对象、窗口大小（默认为 3）。
- 图片边缘处理：一般来说，在对图像应用滤波器进行过滤时，边界问题是一个需要处理的问题。有 3 种处理的方法，如图 1.2 所示。
  - 不做边界处理不对图像的边界作任何处理，在对图像进行滤波时，滤波器没有作用到图像的四周，因此图像的四周没有发生改变。

1	2	3	4	0	0
5	6	7	8	0	0
9	10	11	12	0	0
13	14	15	15	0	0

原矩阵

0	0	0	0	0	0
0	1	2	3	4	0
0	5	6	7	8	0
0	9	10	11	12	0
0	13	14	15	16	0
0	0	0	0	0	0

填充 0

1	1	2	3	4	4
1	1	2	3	4	4
5	5	6	7	8	8
9	9	10	11	12	12
13	13	14	15	16	16
13	13	14	15	16	16

填充最近值

图 1.2: 边缘处理的三种方式

- 填充 0 对图像的边界做扩展，在扩展边界中填充 0，对于边长为  $2k+1$  的方形滤波器，扩展的边界大小为  $k$ ，若原来的图像为  $[m, n]$ ，则扩展后图像变为  $[m+2k, n+2k]$ 。进行滤波之后，图像会出现一条黑色的边框。
- 填充最近像素值扩展与填充 0 的扩展类似，只不过填充 0 的扩展是在扩展部分填充 0，而这个方法是填充距离最近的像素的值。

注：在本实验中，所有滤波都采取对边缘填充 0 的策略。

- 求中值：通过遍历窗口中的所有像素值并进行排序，得到中值。

### 3. 代码集成：

在 More-lite 中，集成中值滤波的代码如下：

```

1 // FunctionLayer/Filter/Filter.cpp
2 // 中值滤波
3 void Filter::MedianFilter(const std::shared_ptr<Image>& src, std::shared_ptr<Image>& dst
, double size)
4 {
5     //边缘像素默认填充0
6     int start = size / 2;
7     for (int i = start; i < dst->size[1] - start; i++)
8     {
9         for (int j = start; j < dst->size[0] - start; j++)
10        {
11            // 收集邻域内的像素值
12            std::vector<Vector3f> model;
13            for (int m = i - start; m <= i + start; m++)
14            {
15                for (int n = j - start; n <= j + start; n++)
16                {
17                    model.push_back(src->getValue(Vector2i(n, m)));
18                }
19            }
20            // 对邻域内的像素值进行排序
21            std::sort(model.begin(), model.end(), [] (const Vector3f& a, const Vector3f& b)
22            {
23                return a.length() < b.length();
24            });
25            // 取中值作为输出图像中的像素值

```

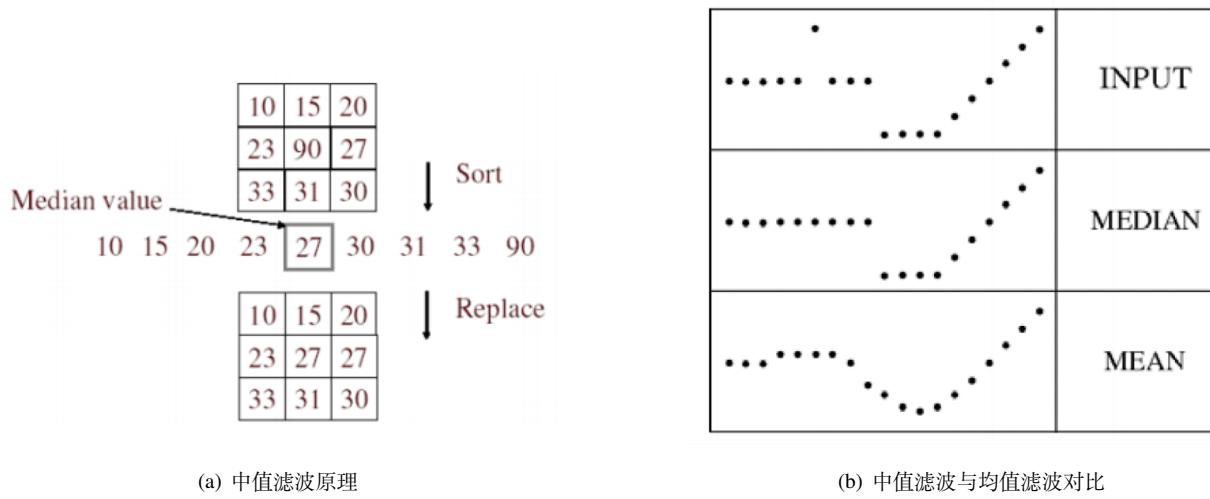


图 1.3: 中值滤波与均值滤波

```

26         dst->setValue(Vector2i(j, i), Vector3f(model[size * size / 2]));
27     }
28 }
29 }
```

### 1.3.3.2 均值滤波

**1. 定义:** 同样以目标像素周围  $3 \times 3$  的邻域为例，就是将  $3 \times 3$  邻域中九个像素灰度值的平均值作为目标像素的灰度值。

如图 1.3(b)所示，相比于中值滤波，均值滤波会模糊边缘。但是均值滤波耗时较少。

#### 2. 代码思路:

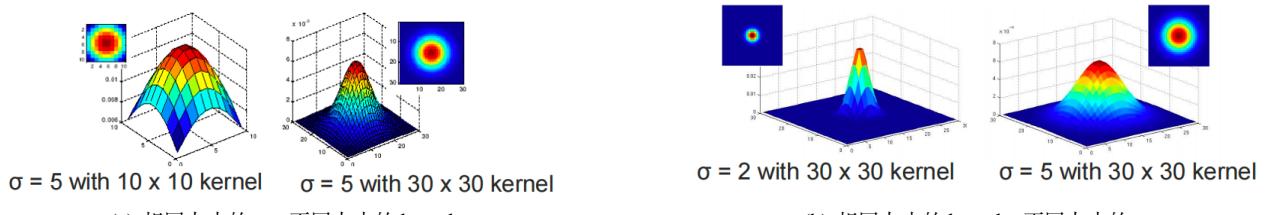
代码思路与中值滤波相似，通过控制窗口大小，求窗口中所有像素值的均值赋予中间像素。

#### 3. 代码集成:

在 More-lite 中，集成均值滤波的代码如下：

```

1 // FunctionLayer/Filter/Filter.cpp
2 // 均值滤波
3 void Filter::MeanFilter(const std::shared_ptr<Image>& src, std::shared_ptr<Image>& dst,
4                         double size)
5 {
6     int start = size / 2;
7     for (int i = start; i < dst->size[1] - start; i++)
8     {
9         for (int j = start; j < dst->size[0] - start; j++)
10        {
11            Vector3f sum(0.0f, 0.0f, 0.0f);
12            for (int m = i - start; m <= i + start; m++)
13            {
14                for (int n = j - start; n <= j + start; n++)
15                {
16                    Vector3f value = src->getValue(Vector2i(n, m));
17                    sum += value;
```



**图 1.4:** 高斯滤波的影响因素。一是核 (kernel) 的大小，二是  $\sigma$  的取值。kernel 的尺寸越大，越完整，但是计算开销越大；标准差代表着数据的离散程度，若  $\sigma$  较小，则生成的模板的中心系数较大，周围的系数较小，图像的平滑效果不明显；反之， $\sigma$  较大，则生成的模板的各个系数相差不大，与均值模板类似，对图像的平滑效果更明显。

```

17         }
18     }
19     Vector3f average = sum / (size * size);
20     dst->setValue(Vector2i(j, i), average);
21 }
22 }
23 }
```

### 1.3.3.3 高斯滤波

#### 1. 定义：

高斯滤波器是一种线性滤波器，其作用原理和均值滤波器类似，都是取滤波器窗口内的像素的均值作为输出。其窗口模板的系数和均值滤波器不同，均值滤波器的模板系数都是相同的为 1；而高斯滤波器的模板系数随着距离模板中心的增大而系数减小。所以，高斯滤波器相比于均值滤波器对图像个模糊程度较小。

高斯滤波主要可以使用两种方法实现。一种是离散化窗口滑窗卷积，另一种方法是通过傅里叶变化。只有当离散化的窗口非常大，用滑窗计算量非常大的情况下，会考虑基于傅里叶变化的实现方法。本次试验中选择使用滑窗实现的卷积。

离散化窗口滑窗卷积时主要利用大小为奇数的高斯核，常用的高斯模板的计算公式为

$$G(x, y) \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (1.13)$$

这样输出的模板有两种形式：

- 小数类型：直接计算得到的值，没有经过任何处理。
- 整数类型：将得到的值进行归一化处理，即将左上角的值归一化为 1，其他每个系数都除以左上角的系数，然后取整。在使用整数模板时，需要在模板的前面加一个系数，该系数为模板系数之和的倒数。

#### 2. 参数理解：

影响高斯滤波器效果的参数有两个，一是核 (kernel) 的取值，即窗口的大小，二是  $\sigma$  的取值。如图 1.4 所示，图 1.4(a)展示了相同大小的  $\sigma$ ，不同大小的 kernel 对高斯滤波的影响，表明 kernel 的尺寸越大，越完整，但是计算开销越大；图 1.4(b)展示了相同大小的 kernel，不同大小的  $\sigma$  对高斯滤波的影响。标准差代表着数据的离散程度，若  $\sigma$  较小，则生成的模板的中心系数较大，周围的系数较小，图像的平滑效果不明显；反之， $\sigma$  较大，则生成的模板的各个系数相差不大，与均值模板类似，对图像的平滑效果更明显。

#### 3. 代码思路：

在窗口数组中遍历，使用单独的高斯模版计算函数求得高斯掩模的值，作为权重与数组相乘。核中间的像素值为加权和。

实际上，高斯掩膜的求解与位置  $(x, y)$  无关，因为在计算过程中  $x, y$  被抵消掉了，因此只要计算一次就可以了。

#### 4. 代码集成：

在 More-lite 中，集成高斯滤波的代码如下：

```

1 // FunctionLayer/Filter/Filter.cpp
2 // 高斯滤波
3 void Filter::GaussianFilter(const std::shared_ptr<Image>& src, std::shared_ptr<Image>&
4     dst, double size, double sigma)
5 {
6     std::vector<std::vector<double>> gaussianTemplate = GaussianTemplate(size, sigma);
7     int start = size / 2;
8     for (int i = start; i < dst->size[1] - start; i++)
9     {
10         for (int j = start; j < dst->size[0] - start; j++)
11         {
12             Vector3f sum(0.0f, 0.0f, 0.0f);
13             for (int m = i - start; m <= i + start; m++)
14             {
15                 for (int n = j - start; n <= j + start; n++)
16                 {
17                     Vector3f value = src->getValue(Vector2i(n, m));
18                     float weight = gaussianTemplate[m - i + start][n - j + start];
19                     sum += value * weight;
20                 }
21             }
22             dst->setValue(Vector2i(j, i), sum);
23         }
24     }
25 }
26 // 高斯掩膜的求解与位置(x,y)无关，因为在计算过程中x,y被抵消掉了，因此只要计算一次就可以了
27 // 需要 size, sigma
28 // 同时也是双边滤波的空间域模板
29 std::vector<std::vector<double>> Filter::GaussianTemplate(double size, double sigma)
30 {
31     std::vector<std::vector<double>> temp;
32     double base = 1.0 / 2.0 / PI / sigma / sigma;
33     for (int i = 0; i < size; i++)
34     {
35         std::vector<double> vec;
36         for (int j = 0; j < size; j++)
37         {
38             double a = (pow(i - size/2, 2) + pow(j - size/2, 2)) / 2.0 / sigma / sigma;
39             double b = base * exp(-a);
40             vec.push_back(b);
41         }
42         temp.push_back(vec);
43     }
44     return temp;
45 }
```

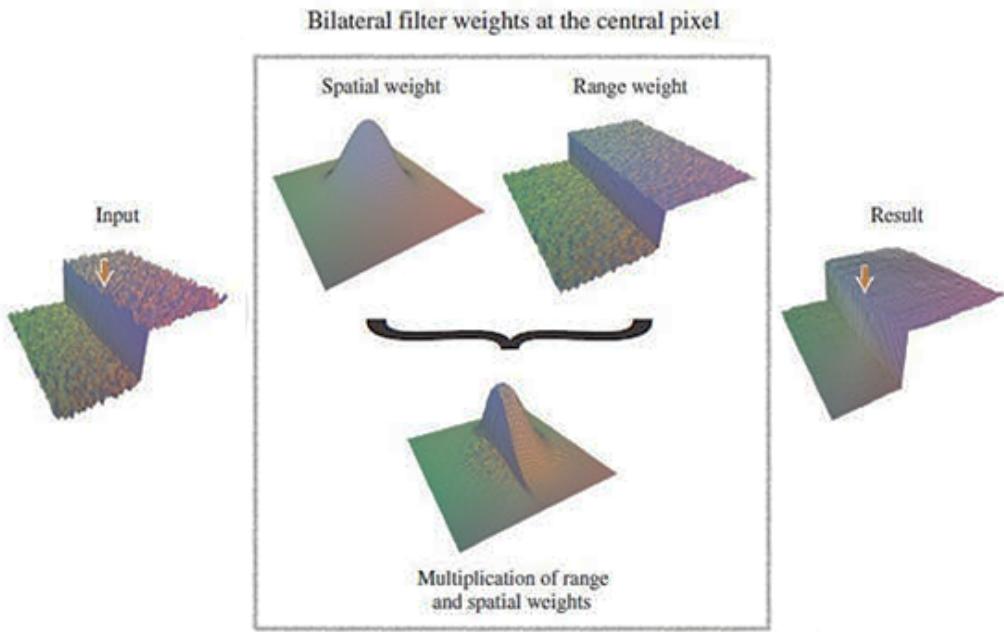


图 1.5: 双边滤波示意图

### 1.3.3.4 双边滤波

#### 1. 定义:

高斯滤波去降噪，会较明显地模糊边缘，对于高频细节的保护效果并不明显，因此引入双边滤波。双边滤波(Bilateral filter)是一种非线性的滤波方法，是结合图像的空间邻近度和像素值相似度的一种折衷处理，同时考虑空域信息和灰度相似性，达到保边去噪的目的，如图 1.5 所示。双边滤波器比高斯滤波多了一个高斯方差，它是基于空间分布的高斯滤波函数，所以在边缘附近，离的较远的像素不会太多影响到边缘上的像素值，这样就保证了边缘附近像素值的保存。但是由于保存了过多的高频信息，对于彩色图像里的高频噪声，双边滤波器不能够干净的滤掉，只能能够对于低频信息进行较好的滤波。

双边滤波器的核由两个函数生成：空间域核和值域核。

- 空间域核：由像素位置欧式距离决定的模板权值  $w_d$ :

$$w_d(i, j, k, l) = e^{-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2}} \quad (1.14)$$

其中， $q(i, j)$  为模板窗口的其他系数的坐标； $p(k, l)$  为模板窗口的中心坐标点； $\sigma_d$  为高斯函数的标准差。使用该公式生成的滤波器模板和高斯滤波器使用的模板是没有区别的。

- 值域核：由像素值的差值决定的模板权值  $w_r$ :

$$w_r(i, j, k, l) = e^{-\frac{\|f(i, j) - f(k, l)\|^2}{2\sigma_r^2}} \quad (1.15)$$

其中， $q(i, j)$  为模板窗口的其他系数的坐标， $f(i, j)$  表示图像在点  $q(i, j)$  处的像素值； $p(k, l)$  为模板窗口的中心坐标点，对应的像素值为  $f(k, l)$ ， $\sigma_r$  为高斯函数的标准差。

将上述两个模板相乘就得到了双边滤波器的模板权值：

$$w(i, j, k, l) = w_d(i, j, k, l) * w_r(i, j, k, l) = e^{-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2} - \frac{\|f(i, j) - f(k, l)\|^2}{2\sigma_r^2}} \quad (1.16)$$

因此，双边滤波器的数据公式可以表示如下：

$$g(i, j) = \frac{\sum_{k, l} f(k, l) w(i, j, k, l)}{\sum_{k, l} w(i, j, k, l)} \quad (1.17)$$

#### 2. 参数理解：

空域权重  $w_d$  衡量的是  $p, q$  两点之间的距离，距离越远权重越低；值域权重  $w_r$  衡量的是  $p, q$  两点之间的像素值相似程度，越相似权重越大。

从图像的平坦区域和边缘区域定性分析双边滤波的降噪效果：在平坦区域，临近像素的像素值的差值较小，对应值域权重  $w_r$  接近于 1，此时空域权重  $w_d$  起主要作用，相当于直接对此区域进行高斯模糊。因此，平坦区域相当于进行高斯模糊；在边缘区域，临近像素的像素值的差值较大，对应值域权重  $w_r$  接近于 0，导致此处核函数下降（因  $w = w_r w_d$ ），当前像素受到的影响就越小，从而保持了原始图像的边缘的细节信息。

### 3. 代码思路：

与高斯滤波相似，在窗口数组中遍历，使用单独的空间域模板（高斯模板）计算函数求得高斯掩模的值，使用单独的值域模板计算函数求得值域掩模的值，两值相乘作为权重与数组相乘。核中间的像素值为加权和。

### 4. 代码集成：

在 More-lite 中，集成高斯滤波的代码如下：

```

1 // FunctionLayer/Filter/Filter.cpp
2 // 双边滤波，去噪保边
3 void Filter::BilateralFilter(const std::shared_ptr<Image>& src, std::shared_ptr<Image>&
4     dst, int size, float sigmaD, float sigmaR)
5 {
6     std::vector<std::vector<double>> spatialTemplate = GaussianTemplate(size, sigmaD);
7     std::vector<float> rangeTemplate = RangeTemplate(256, sigmaR);
8
9     int start = size / 2;
10
11    for (int y = start; y < src->size[1] - start; ++y)
12    {
13        for (int x = start; x < src->size[0] - start; ++x)
14        {
15            Vector3f sum(0.0f, 0.0f, 0.0f);
16            float weightSum = 0.0f;
17
18            for (int i = y - start; i <= y + start; ++i)
19            {
20                for (int j = x - start; j <= x + start; ++j)
21                {
22                    Vector3f value = src->getValue(Vector2i(j, i));
23                    float spatialWeight = spatialTemplate[i - y + start][j - x + start];
24                    float rangeWeight = rangeTemplate[static_cast<int>(std::abs(value.
25                        length()))];
26                    float weight = spatialWeight * rangeWeight;
27
28                    sum += value * weight;
29                    weightSum += weight;
30                }
31            }
32            dst->setValue(Vector2i(x, y), sum / weightSum);
33        }
34    }
35 // 空间域模板使用高斯模板
36 // 值域模板

```

```

36 std::vector<float> Filter::RangeTemplate(int range, float sigmaR)
37 {
38     std::vector<float> mask;
39     float base = 1.0f / (2.0f * sigmaR * sigmaR);
40
41     for (int i = 0; i < range; ++i)
42     {
43         float distance = i * i;
44         float weight = std::exp(-distance / (2.0f * sigmaR * sigmaR));
45         mask.push_back(weight);
46     }
47
48     return mask;
49 }
```

### 1.3.4 Intel Open Image Denoise 工业降噪库

#### 1. 介绍:

英特尔开放图像去噪库 [9] 的核心是一系列基于高效深度学习的去噪滤波器，这些滤波器经过训练可以处理各种 spp 样本。因此它适用于预览和最终帧渲染。过滤器可以仅使用带噪声的 color 缓冲区对图像进行去噪，也可以为了保留尽可能多的细节，选择使用辅助特征缓冲区（例如反照率、法线）。

#### 2. 代码集成:

- 参考 Open Image Denoise 的 GitHub 仓库 [6] 中的步骤，下载源码并编译，获得 bin、include、lib 文件，将得到的这三个文件夹复制合并到 Moer-lite 编译后生成的 target 文件夹中，如图 1.6 所示。

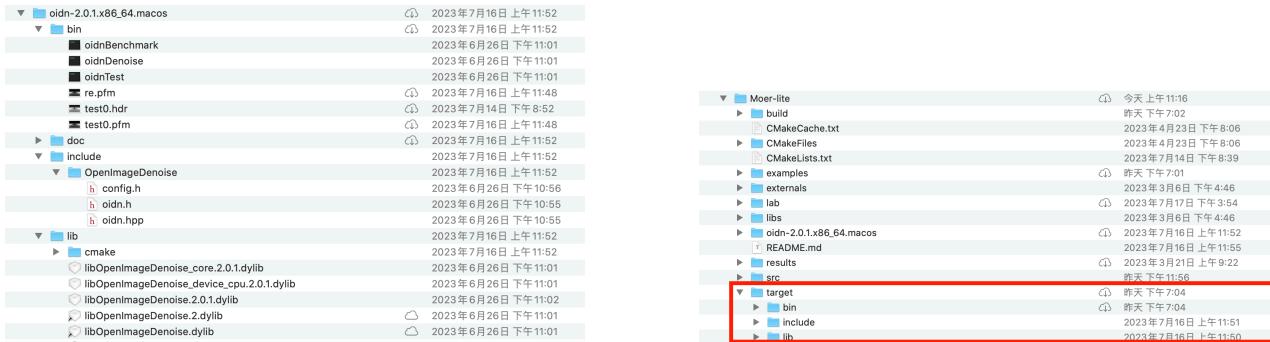


图 1.6: Moer-lite 集成 Intel Open Image Denoise 降噪库

- 由于降噪库滤波的输入图片需要为 pfm 格式，因此在 Image 类中增加 Image::savePFM 函数：

```

1 //ResourceLayer/Image.cpp
2 void Image::savePFM(const char* filename) const {
3     std::ofstream f(filename, std::ios::binary);
4     if (!f.is_open()) {
5         std::cerr << "Failed to open file: " << filename << std::endl;
6         return;
7     }
8
9     f << "PF\n";
10    f << size[0] << " " << size[1] << "\n";
```

```

11   f << "-1.0\n";
12
13   for (int y = size[1] - 1; y >= 0; --y) {
14     for (int x = 0; x < size[0]; ++x) {
15       const int offset = (x + y * size[0]) * channels;
16       for (int c = 0; c < channels; ++c) {
17         const float value = data[offset + c];
18         f.write(reinterpret_cast<const char*>(&value), sizeof(float));
19     }
20   }
21 }
22
23 f.close();
24 }
```

- Intel Open Image Denoise 降噪库的执行命令为 `./oidnDenoise -hdr < 输入图片路径 > -o < 输出图片路径 >`, 需要在 Moer-lite 代码中执行过程中执行这一行 shell 命令, 集成结果参见1.3.1 修改 Moer-lite 框架一节。
- 增加执行 Moer 命令的参数, 调用 main.cpp 中的相应代码。集成结果参见1.3.1 修改 Moer-lite 框架一节。

## 1.4 实验结果与分析

**注:** 由于 Latex 不支持 **hdr** 格式的图片导入, 所以实验报告所有出现的图片均为截图, 截图的方式也许会导致图片边缘不完全准确, 但是尽量对准了边缘进行截图并放在实验报告中。

### 1.4.1 添加噪声

#### 1. 椒盐噪声

分别设置噪声点个数为 500、2000、10000, 添加噪声后结果如下:

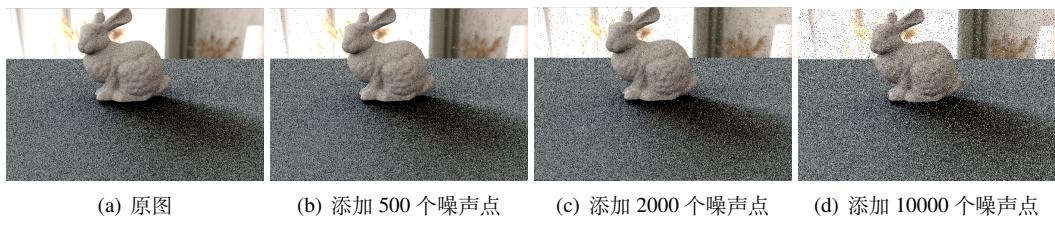


图 1.7: 添加椒盐噪声

**分析:** 显而易见, 噪声点  $n$  值越大, 图片上的黑白噪点越多。

#### 2. 高斯噪声

生成高斯噪声的函数主要由两个变量需要控制(参见1.3.2.1 高斯噪声一节), 一是要添加噪声的数据范围, 二是  $\sigma$ ,  $\mu$  默认为 0。

分别设置不同的参数值, 添加噪声后结果如 1.8 所示。

#### 分析:

- 较大的  $\sigma$  值会生成更强烈的噪声, 噪声点与原始像素值的差异更大, 使图像中的噪声更加明显。同时, 较大的  $\sigma$  值会使噪声点分布更广泛, 图像中的噪声更加扩散和分散。较小的  $\sigma$  值则会生成集中在均值附近的细小噪声点, 使噪声更集中。

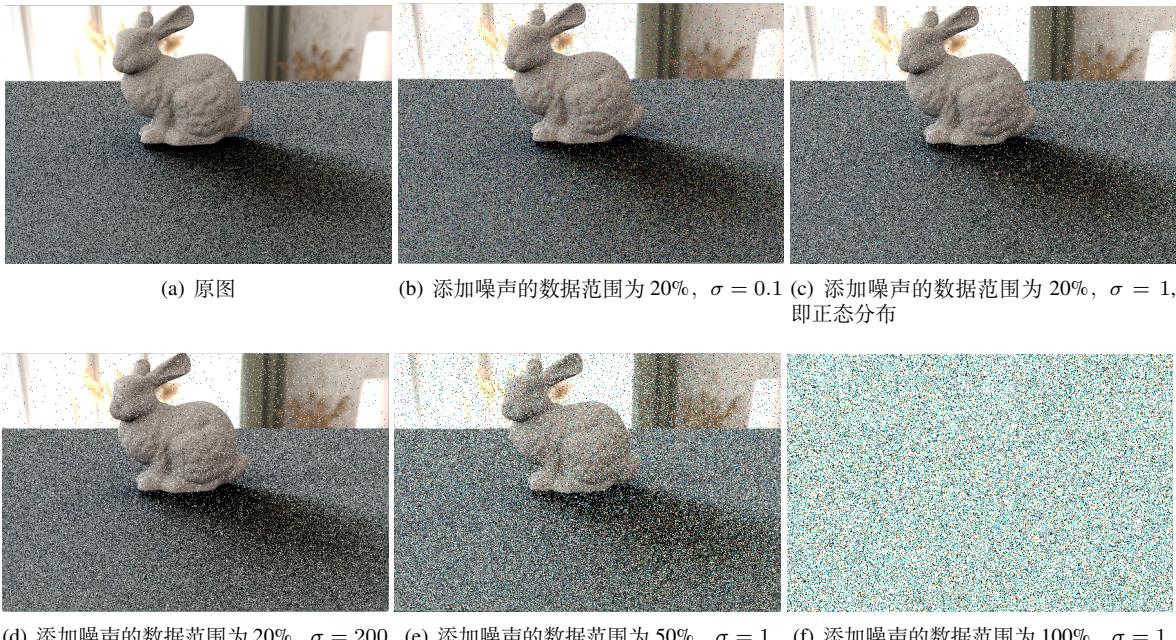


图 1.8: 添加高斯噪声

- 需要合理控制添加噪声的数据范围，例如当数据范围为 100%，即 RGB 值在 [0, 255] 之间的像素都需要添加噪声时，就会出现如图 1.7(f) 所示情况，这种图片不适合用于做降噪。

### 1.4.2 中值濾波

在分析中值滤波的效果时，一共做了四组对比实验：

- 对照组：使用中值滤波过滤未添加噪声的图片。如图 1.9 所示。
  - 同样大小的窗口对不同程度的椒盐噪声的去噪效果：使用窗口大小为 3 的中值滤波过滤添加椒盐噪声的图片， $n$  的取值包含 500、10000。如图 1.10 和 1.11 所示。
  - 同样大小的窗口对不同程度的高斯噪声的去噪效果：使用窗口大小为 3 的中值滤波过滤添加高斯噪声的图片， $\sigma$  取值包含 0.1、1、100，数据处理范围为 20%。如图 1.13、1.14 和 1.15 所示。
  - 不同大小的窗口对同样程度的椒盐噪声的去噪效果：使用窗口大小为 3、21 的中值滤波过滤添加椒盐噪声的图片。如图 1.11 和 1.12 所示。



图 1.9: 不添加噪声 进行中值滤波



(a) 渲染得到的原图

(b) 添加椒盐噪声 ( $n=500$ )

(c) 进行中值滤波

图 1.10: 添加椒盐噪声 ( $n=500$ ), 进行中值滤波, 窗口大小为 3

(a) 渲染得到的原图

(b) 添加椒盐噪声 ( $n=10000$ )

(c) 进行中值滤波, 窗口大小为 3

图 1.11: 添加椒盐噪声 ( $n=10000$ ), 进行中值滤波, 窗口大小为 3

(a) 渲染得到的原图

(b) 添加椒盐噪声 ( $n=10000$ )

(c) 进行中值滤波, 窗口大小为 21

图 1.12: 添加椒盐噪声 ( $n=10000$ ), 进行中值滤波, 窗口大小为 21

(a) 渲染得到的原图

(b) 添加高斯噪声 ( $\sigma = 0.1$ )

(c) 进行中值滤波, 窗口大小为 3

图 1.13: 添加高斯噪声 ( $\sigma = 0.1$ ), 进行中值滤波, 窗口大小为 3

(a) 渲染得到的原图 (b) 添加高斯噪声 ( $\sigma = 1$ ) (c) 进行中值滤波, 窗口大小为 3图 1.14: 添加高斯噪声 ( $\sigma = 1$ ), 进行中值滤波, 窗口大小为 3(a) 渲染得到的原图 (b) 添加高斯噪声 ( $\sigma = 100$ ) (c) 进行中值滤波, 窗口大小为 3图 1.15: 添加高斯噪声 ( $\sigma = 100$ ), 进行中值滤波**分析:**

- 滤波后的图片四周出现黑边, 是由于在滤波时, 在图片的扩展边界填充 0。中值滤波的窗口越大, 则黑边越宽。
- 椒盐噪声点越少, 中值滤波的平滑效果越好。
- 高斯噪声中  $\sigma$  越大, 中值滤波的平滑效果越好。
- 中值滤波对椒盐噪声表现较好, 对高斯噪声表现较差。

### 1.4.3 均值滤波

在分析均值滤波的效果时, 一共做了四组对比实验:

1. 对照组: 使用均值滤波过滤未添加噪声的图片。如图 1.16 所示。
2. 同样大小的窗口对不同程度的椒盐噪声的去噪效果: 使用窗口大小为 3 的均值滤波过滤添加椒盐噪声的图片,  $n$  的取值包含 500、10000。如图 1.17 和 1.18 所示。
3. 同样大小的窗口对不同程度的高斯噪声的去噪效果: 使用窗口大小为 3 的均值滤波过滤添加高斯噪声的图片,  $\sigma$  取值包含 0.1、1、100, 数据处理范围为 20%。如图 1.19、1.20 和 1.21 所示。
4. 不同大小的窗口对同样程度的高斯噪声的去噪效果: 使用窗口大小为 3、21 的均值滤波过滤添加椒盐噪声的图片。如图 1.19 和 1.22 所示。

**分析:**

- 均值滤波主要起模糊的作用, 对于灰度图 (channel=1) 来说, 效果较好。但是对于彩色图 (channel=3), 模糊的效果使图片噪点更多。同时模糊的效果会导致细节丢失。
- 窗口尺寸越大, 均值滤波模糊效果越明显。
- 均值滤波对于轻度的高斯噪声有一定的去噪效果, 但对于椒盐噪声不太适用。



(a) 渲染得到的原图

(b) 进行均值滤波

图 1.16: 不添加噪声, 进行均值滤波



(a) 渲染得到的原图

(b) 添加椒盐噪声 ( $n=500$ )

(c) 进行均值滤波, 窗口大小为 3

图 1.17: 添加椒盐噪声 ( $n=500$ ), 进行均值滤波, 窗口大小为 3

(a) 渲染得到的原图

(b) 添加椒盐噪声 ( $n=10000$ )

(c) 进行均值滤波, 窗口大小为 3

图 1.18: 添加椒盐噪声 ( $n=10000$ ), 进行均值滤波, 窗口大小为 3

(a) 渲染得到的原图

(b) 添加高斯噪声 ( $\sigma = 0.1$ )

(c) 进行均值滤波, 窗口大小为 3

图 1.19: 添加高斯噪声 ( $\sigma = 0.1$ ), 进行均值滤波, 窗口大小为 3



(a) 渲染得到的原图

(b) 添加高斯噪声 ( $\sigma = 1$ )

(c) 进行均值滤波, 窗口大小为 3

图 1.20: 添加高斯噪声 ( $\sigma = 1$ ), 进行均值滤波, 窗口大小为 3

(a) 渲染得到的原图

(b) 添加高斯噪声 ( $\sigma = 100$ )

(c) 进行均值滤波, 窗口大小为 3

图 1.21: 添加高斯噪声 ( $\sigma = 100$ ), 进行均值滤波, 窗口大小为 3

(a) 渲染得到的原图

(b) 添加高斯噪声 ( $\sigma = 0.1$ )

(c) 进行均值滤波, 窗口大小为 21

图 1.22: 添加高斯噪声 ( $\sigma = 0.1$ ), 进行均值滤波, 窗口大小为 21

#### 1.4.4 高斯滤波

在分析高斯滤波的效果时，一共做了四组实验：

1. 对照组：使用高斯滤波 ( $\text{size}=9, \sigma = 0.8$ ) 过滤未添加噪声的图片。如图 1.23 所示。
2. 对椒盐噪声的去噪效果：使用高斯滤波 ( $\text{size}=9, \sigma = 0.8$ ) 过滤添加椒盐噪声 ( $n=500$ ) 的图片。如图 1.24 所示。
3. 不同大小的窗口对高斯噪声的去噪效果：使用窗口大小为 9 和为 21 的高斯滤波 ( $\sigma = 0.8$ ) 过滤添加高斯噪声 ( $\sigma$  取值 1，数据处理范围为 20%) 的图片。如图 1.25 和 1.26 所示。
4. 不同大小的  $\sigma$  对高斯噪声的去噪效果：使用  $\sigma$  为 0.8、2.8，窗口大小为 9 的高斯滤波过滤添加高斯噪声 ( $\sigma$  取值 1，数据处理范围为 20%) 的图片。如图 1.25 和 1.27 所示。



图 1.23: 不添加噪声，进行高斯滤波



图 1.24: 添加椒盐噪声，进行高斯滤波



图 1.25: 添加高斯噪声，进行高斯滤波，窗口大小为 9



(a) 渲染得到的原图 (b) 添加高斯噪声 (c) 进行高斯滤波, 窗口大小为 21

图 1.26: 添加高斯噪声, 进行高斯滤波, 窗口大小为 21

(a) 渲染得到的原图 (b) 添加高斯噪声 (c) 进行高斯滤波,  $\sigma = 2.8$ 图 1.27: 添加高斯噪声, 进行高斯滤波 ( $\sigma = 2.8$ )**分析:**

- 高斯滤波主要起平滑的作用,  $\sigma$  越大, 平滑作用越明显。但是边缘也越模糊, 细节丢失更多。
- 窗口尺寸越大, 高斯滤波平滑效果越明显。

#### 1.4.5 双边滤波

在分析双边滤波的效果时, 一共做了四组实验:

1. 对照组: 使用双边滤波 ( $\text{size}=9, \sigma_D = 0.8, \sigma_{\text{sigma}} = 35$ ) 过滤未添加噪声的图片。如图 1.28 所示。
2. 对椒盐噪声的去噪效果: 使用双边滤波 ( $\text{size}=9, \sigma_D = 0.8, \sigma_{\text{sigma}} = 35$ ) 过滤添加椒盐噪声的图片。如图 1.29 所示。
3. 不同大小的窗口对高斯噪声的去噪效果: 使用窗口大小为 9 和为 21 的高斯滤波 ( $\sigma_D = 0.8, \sigma_{\text{sigma}} = 35$ ) 过滤添加高斯噪声 ( $\sigma$  取值 1, 数据处理范围为 20%) 的图片。如图 1.30 和 1.31 所示。
4. 不同大小的  $\sigma_R$  对高斯噪声的去噪效果: 使用  $\sigma_R$  为 35、65 的双边滤波 ( $\text{size}=9, \sigma_D = 0.8$ ) 过滤添加高斯噪声 ( $\sigma$  取值 1, 数据处理范围为 20%) 的图片。如图 1.31 和 1.32 所示。

**分析:**

- $\sigma_D$  与  $\sigma_R$  的值为 kernel 的方差, 方差越大, 说明该项对于权重的影响越大。
- 两个方面的某个的  $\text{sigma}$  相对变大, 表示这一方面相对较重要, 得到强调。如  $\sigma_D$  变大, 表示更多采用近邻的值作平滑, 说明图像的空间信息更重要, 即相近相似。如果  $\sigma_D$  变小 ( $\sigma_R$  相对更大), 则更强调值域的相似性, 像素值的影响更大。
- 相比于同参数的高斯滤波, 图像的平滑效果更明显, 边缘更清晰, 双边滤波的  $\sigma_R$  小于 10 时, 平滑效果不是很明显。



(a) 渲染得到的原图

(b) 进行双边滤波,  $\text{size}=9, \sigma_D = 0.8, \sigma_R = 35$ 图 1.28: 不添加噪声, 进行双边滤波 ( $\text{size}=9, \sigma_D = 0.8, \sigma_R = 35$ )

(a) 渲染得到的原图

(b) 添加椒盐噪声,  $n=500$ (c) 进行双边滤波  $\text{size}=9, \sigma_D = 0.8, \sigma_R = 35$ 图 1.29: 添加椒盐噪声, 进行双边滤波 ( $\text{size}=9, \sigma_D = 0.8, \sigma_R = 35$ )

(a) 渲染得到的原图

(b) 添加高斯噪声,  $\sigma = 0.01$ (c) 进行双边滤波,  $\text{size}=9, \sigma_D = 0.8, \sigma_R = 35$ 图 1.30: 添加高斯噪声, 进行双边滤波 ( $\text{size}=9, \sigma_D = 0.8, \sigma_R = 35$ )

(a) 渲染得到的原图

(b) 添加高斯噪声,  $\sigma = 0.01$ (c) 进行双边滤波,  $\text{size}=21, \sigma_D = 0.8, \sigma_R = 35$ 图 1.31: 添加高斯噪声, 进行双边滤波 ( $\text{size}=21, \sigma_D = 0.8, \sigma_R = 35$ )



图 1.32: 添加高斯噪声, 进行双边滤波 ( $\text{size}=9, \sigma_D = 0.8, \sigma_R = 65$ )

#### 1.4.6 Intel Open Image Denoise 降噪库

在分析 Intel Open Image Denoise 降噪库的效果时, 一共做了三组实验:

1. 对照组: 使用降噪库过滤未添加噪声的图片。如图 1.33 所示。
2. 对椒盐噪声的去噪效果: 使用降噪库过滤添加椒盐噪声 ( $n=500$  和  $n=10000$ ) 的图片。如图 1.34 和 1.35 所示。
3. 对高斯噪声的去噪效果: 使用降噪库过滤添加高斯噪声 ( $\sigma = 1$  和  $\sigma = 100$ ) 的图片。如图 1.36 和 1.37 所示。



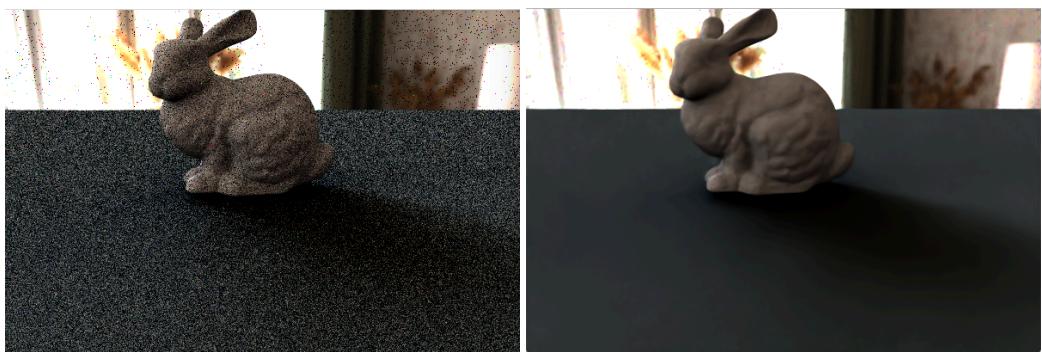
图 1.33: 不添加噪声, 使用降噪库降噪



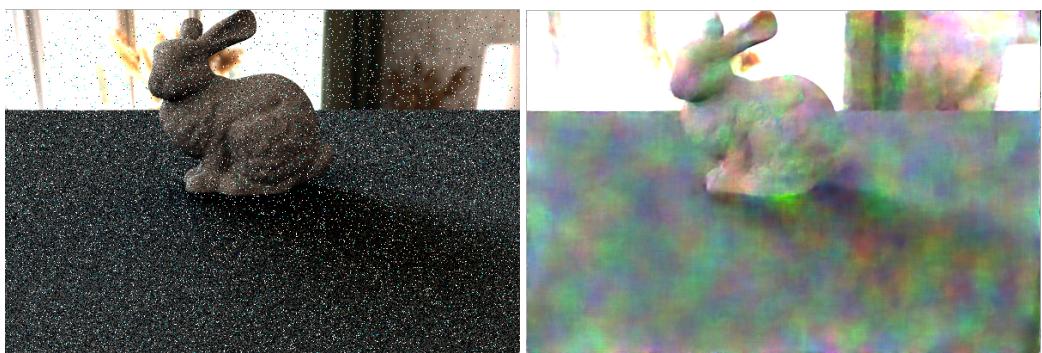
图 1.34: 添加椒盐噪声 ( $n=50$ ), 使用降噪库降噪

(a) 输入 (添加椒盐噪声, $n=200$ )

(b) 降噪库输出

图 1.35: 添加椒盐噪声 ( $n=200$ ), 使用降噪库降噪(a) 输入 (添加高斯噪声, $\sigma = 0.01$ )

(b) 降噪库输出

图 1.36: 添加高斯噪声 ( $\sigma = 0.01$ ), 使用降噪库降噪(a) 输入 (添加高斯噪声, $\sigma = 1$ )

(b) 降噪库输出

图 1.37: 添加高斯噪声 ( $\sigma = 1$ ), 使用降噪库降噪

### 分析：

- 相比于自己实现的滤波器，intel 降噪库的用时更长，如图 1.38 所示。
- 降噪库对于椒盐噪声、高斯噪声的过滤效果都比较好，但是到高斯噪声中的  $\sigma$  较大时，会出现奇怪的晕影。
- 降噪库只支持 pfm 格式的输入，限制了一些应用。

```
[pro@probeMacBook-Pro-2 bin % ./Moer /Users/pro/Desktop/Rendering_labs/Moer-lite/]
examples/bunny add_g_noise g_denoise
Using acceleration type embree
100% [|||||]
Rendering costs 16.53s

*****Begin Add Noise*****
Save noise image to bunny-2_add_g_noise.hdr

*****Begin Denoise*****
Save denoised image to bunny-2_g_denoise.hdr

Intel Denoising costs 0.08s
```

(a) 高斯滤波用时 0.08s

```
[pro@probeMacBook-Pro-2 bin % ./Moer /Users/pro/Desktop/Rendering_labs/Moer-lite/]
examples/bunny add_g_noise bi_denoise
Using acceleration type embree
100% [|||||]
Rendering costs 16.12s

*****Begin Add Noise*****
Save noise image to bunny-2_add_g_noise.hdr

*****Begin Denoise*****
Save denoised image to bunny-2_bi_denoise.hdr

Intel Denoising costs 0.15s
```

(b) 双边滤波用时 0.15s

```
[pro@probeMacBook-Pro-2 bin % ./Moer /Users/pro/Desktop/Rendering_labs/Moer-lite/]
examples/bunny add_g_noise intel_denoise
Using acceleration type embree
100% [|||||]
Rendering costs 16.11s

*****Begin Add Noise*****
Save noise image to bunny-2_add_g_noise.hdr

*****Begin Denoise*****
Initializing device
  device=CPU, version=2.0.1, msec=6.65129
Loading input
Resolution: 600x400
Initializing filter
  filter=RT, msec=16.8408
Denoising 100%
  msec=232.896
Saving output
Save denoised image to re.pfm

Intel Denoising costs 0.36s
```

(c) 降噪库滤波用时 0.36s

**图 1.38:** 对于同一张添加了高斯噪声 ( $\sigma = 0.1$ ) 的图片，不同降噪器的降噪用时

## 第2章 实验总结与注意点

### 2.1 去噪总结

在传统滤波器中，没有一种滤波器能够适用于所有类型的噪声。选择合适的滤波器取决于噪声的类型和强度，以及对图像细节的要求。在实际应用中，常常需要根据具体情况进 行实验和调整，甚至采用多种滤波器的组合来达到最佳的去噪效果。

1. 中值滤波：中值滤波主要对椒盐噪声有较好的去噪效果。由于中值滤波采用像素值的中值替代当前像素值，它对于异常像素（噪声点）有很好的响应。但它在去除高斯噪声等其他类型的噪声时效果较差。同时，中值滤波可能导致图像略微模糊，特别是对于噪声密集的图像。
2. 均值滤波：均值滤波对于轻度的高斯噪声有一定的去噪效果，但对于椒盐噪声不太适用。同时均值滤波会导致图像细节的模糊。
3. 高斯滤波：  
高斯滤波主要对高斯噪声有良好的去噪效果。高斯噪声是一种符合高斯分布的随机噪声，常见于图像传感器等设备。高斯滤波通过对图像像素周围的邻域进行加权平均，可以有效地模糊图像中的噪声，但同时也可能会导致图像的细节丢失。
4. 双边滤波：双边滤波对于高斯噪声和椒盐噪声都有较好的去噪效果。椒盐噪声是一种在图像中随机出现的黑白像素点，类似于盐和胡椒的分布，可能由于传感器故障或传输中的错误引起。双边滤波在进行像素平均时，除了考虑距离上的邻近性，还考虑了像素值之间的相似性。这使得双边滤波能够有效地去除噪声，同时保留图像的边缘和细节。

在工业降噪库中，也存在诸多限制。比如 NVIDIA OptiX Denoiser 降噪库需要硬件支持、有些降噪库不支持 MacOS 操作系统等。同时，在使用 Intel Image Open Denoise 降噪库时，发现这个降噪库只支持 pfm 格式的图片输入，在将 hdr 图片转成 pfm 图片时，会导致像素值的变化。

### 2.2 实验注意点

本次实验中，一共遇到了以下几点需要注意的问题：

1. 在进行滤波时，需要对边缘进行处理，填充 0 或者填充最近像素值。
2. 在添加高斯时，需要规定处理数据的范围，如果对 [0, 255] 值域内的像素全部添加高斯噪声，则会出现如图 1.7(f) 所示情况，是无法进行滤波的。
3. 滤波时核（窗口）的大小需要为奇数。
4. 本次实验中涉及的传统滤波算法均为通用的滤波算法，应用广泛且不依赖于图像的具体内容，通常用于常规的去噪和平滑任务。因此没有设计基于不同材质的渲染图像进行滤波的对比。

## 第3章 思考创新与展望

随着深度学习的研究深入，深度学习技术在 MC 去噪和其他领域的应用表现出了巨大的潜力，并在学术界和工业界都受到了广泛的关注和应用。深度学习在 MC 降噪领域的研究可以被分为三个方面 ??：像素降噪、非凡域降噪、高维度降噪。

由于光的分布是多模态的，当样本特征的分布是多模态的时候，简单地使用统计量来描述整个像素或局部区域的属性可能会导致信息丢失或模糊化。这是因为统计量通常对数据进行了平均化或整体化处理，无法准确地反映多个模态之间的差异和复杂的光传输现象。因此传统的基于像素的去噪丢失了很多的采样时的信息。所以，研究人员提出了基于采样信息的去噪 ??。

在这项研究中，为了解决传统 MC 去噪器的问题，作者提出了“kernel-spatting network”方法。该方法学习样本与图像之间的映射关系，并采用了一种新颖的神经网络架构设计，以应对样本空间中的多个挑战。由于样本的顺序在样本空间中是任意的，而且这些样本需要以排列不变的方式进行处理。这意味着样本的排列顺序不应该影响去噪结果。作者通过“kernel-spatting network”实现了样本排列不变性。传统方法通常使用传统的“gathering kernels”来处理样本。而作者提出了使用卷积神经网络来预测“spatting kernels”，并将个体样本投影到附近的像素上。这样的方法更加自然，尤其适用于包含运动模糊、景深和光传输等问题的情况，因为它更容易预测每个样本对哪些像素有贡献，而不是预测“gathering kernels”需要确定相关像素之间的信息关系。

相比于传统 MC 去噪方法，这项研究中新的架构在视觉上和数值上都产生了更高质量的结果，尤其对于低样本计数图像和分布效果图像。

然而，随着样本数量的增加，该算法的计算成本会大大增加。性能扩展性能也会受到影响，该算法仍然存在改进空间以提高效率。以下是我的一些思考：

- 是否可以借鉴 Delbracio 等人提出的直方图思想 ??，让模型学习预先聚合相似的样本，以便一起进行去噪？通过预先聚合相似的样本，可以降低计算复杂性，从而提高算法的可扩展性。
- 在样本数很大时，存储所有样本的 radiance 和特征可能会变得不便。是否可以使用递归式的模型来避免在计算上下文特征时存储所有样本？即逐步增加样本数量来改善图像质量。
- 同时，递归式模型可以更好地利用数据之间的相关性和结构信息。这也涉及到另一个问题，即去噪时利用图像的时间连续性。去噪算法中，采样时通常都关注全局几何信息、局部空间信息，但是较少关注时间连续性，当有一段图像帧进行去噪时，如何利用时间连续性信息和保证去噪后保持视频在时间维度上的连贯性和自然性是很重要的。如何将这两个方向上的研究结合起来我觉得是值得研究的。基于上述 sample-based denoise 算法，我认为可以从采样入手，选择的样本中可以包含时间序列的信息。

基于以上想法，如何设计一个统一的算法，既保证了空间维度上的准确性，又保证了时间维度上的准确性，我认为也许会是未来的研究方向。

## 第4章 课程与实验收获

这门课是我继《图形学》后修读的第二门CG课程，如果说《图形学》为我提供了CG的基础，那么这门《图形绘制技术》则将我引入了现代图形学的大门。在第一堂课上，我就被课程PPT上展示的像照片一样真实的渲染图片深深吸引。如何将二维扩展为二点五维、再扩展为三维，是我对图形渲染的直观认知。随着课程深入，我发现这个领域并不是像第一节课上展示出来的那么有趣，而是需要大量的数学物理基础、深度学习的知识储备，我很庆幸没有在第一次看见蒙特卡洛积分和概率密度函数的时候就退课，而是坚持学到最后。事实证明，没有学不会的数学，也没有理解不了的光线追踪原理。

在这次大实验中，我详细研究了蒙特卡洛积分与渲染方程，对光线追踪原理的掌握比学期中的任何一堂课都更熟练。在前两次的小实验中，我只是按着实验指导pdf中的步骤按部就班地填代码，但是在完成大实验的过程中，我详读了Moer-lite框架，对其中的每一部分及其功能都烂熟于心，从而也更加深入理解了应该如何编写一个光追渲染引擎。

在学习蒙特卡洛降噪的过程中，我掌握了关于图形渲染的一些常见的思路，例如基于像素出发、基于采样点出发、考虑采样点的重要性、收集采样过程中产生的辅助信息、邻域思想等。这对我以后学习CG都有很大帮助。

最后，我很想感谢这门课的助教学长们，在三次实验中，我都产生了很多问题和bug，并且不断请教助教，他们不厌其烦地为我解答、帮助我检查源码、帮助我理解框架代码。十分感谢助教们的帮助，让我借助实验顺利完成这门课的学习。

十分感谢过洁老师在学期中对我的教学与指导。

## 参考文献

- [1] Steve Bakó et al. “Kernel-predicting convolutional networks for denoising Monte Carlo renderings”. In: *ACM Trans. Graph.* 36.4 (2017), pp. 97–1.
- [2] CSDN. 图像降噪算法——图像降噪算法总结. CSDN Blog. URL: [https://blog.csdn.net/weixin\\_44580210/article/details/104850444](https://blog.csdn.net/weixin_44580210/article/details/104850444).
- [3] CSDN. 蒙特卡洛积分与重要性采样详解.<https://www.cnblogs.com/time-flow1024/p/10094293.html>. 2023.
- [4] M. Gharbi et al. “Sample-based Monte Carlo denoising using a kernel-splatting network”. In: *ACM Transactions on Graphics* 38.4 (2019), p. 125.
- [5] Michaël Gharbi et al. “Sample-based Monte Carlo denoising using a kernel-splatting network”. In: *ACM Transactions on Graphics (TOG)* 38.4 (2019), pp. 1–12.
- [6] Github. *Open Image Denoise GitHub Repository*. <https://github.com/OpenImageDenoise/oidn>.
- [7] Shuhang Gu and Radu Timofte. “A brief review of image denoising algorithms and beyond”. In: *Inpainting and Denoising Challenges* (2019), pp. 1–21.
- [8] Yuchi Huo and Sung-eui Yoon. “A survey on deep learning-based Monte Carlo denoising”. In: *The Author(s)* (2021).
- [9] Intel Open Image Denoise. <https://www.openimagedenoise.org>.
- [10] zhihu. c++ 自设概率叠加高斯噪声. <https://zhuanlan.zhihu.com/p/97905713>.