图形绘制技术
Project2(Gabor Noise)

设计者 陈盛恺

学号 181860007

CONTANTS

项目背景

系统分析与设计

系统实现

效果展示

参考资料

项目背景

项目背景

Gabor Noise通过 Gabor Kernel 与随机点的卷积生成纹理，这种纹理的生成方法可以通过调整频谱中的参数对纹理进行修改，快速生成纹理。

实验中实现了通过调整频谱生成二维平面纹理以及利用OpenGL的功能实现了纹理的实时映射。

系统分析与设计

底层数据结构
——函数与图像

Func采样生成Image

MathFun2d

Image

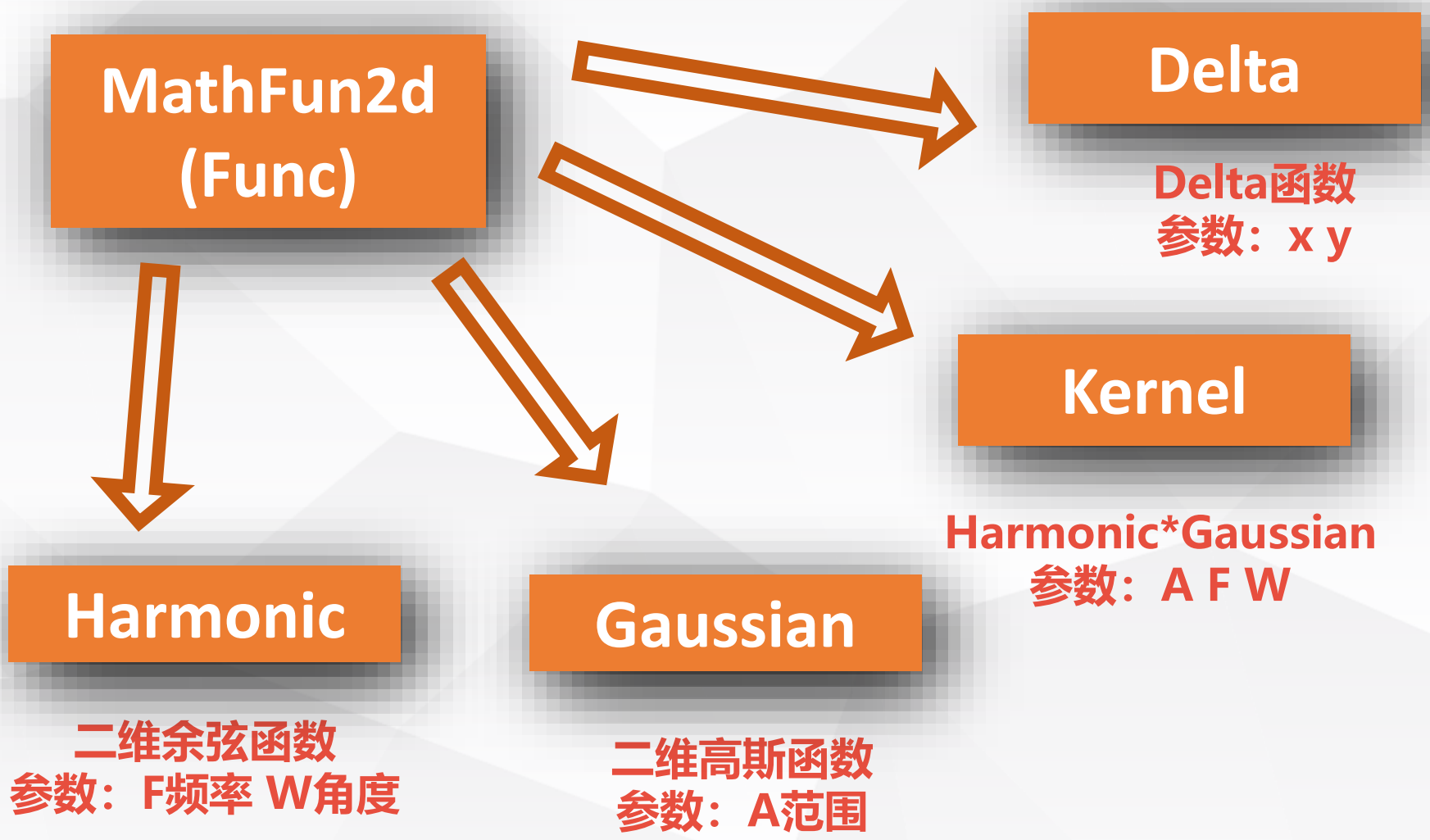
双变量函数处理类

功能与接口：
1.双变量函数的基类Func
2.Func的基本运算与复合
3.重载Func(x,y)在(x,y)处取值
4.Range类维护Func的值域
5.Range类实现区间的基本操作

2D图像类

功能与接口：
1.二维图像的RGBA存储
2.对Func对象采样生成二维图像
3.对两个Image对象卷积

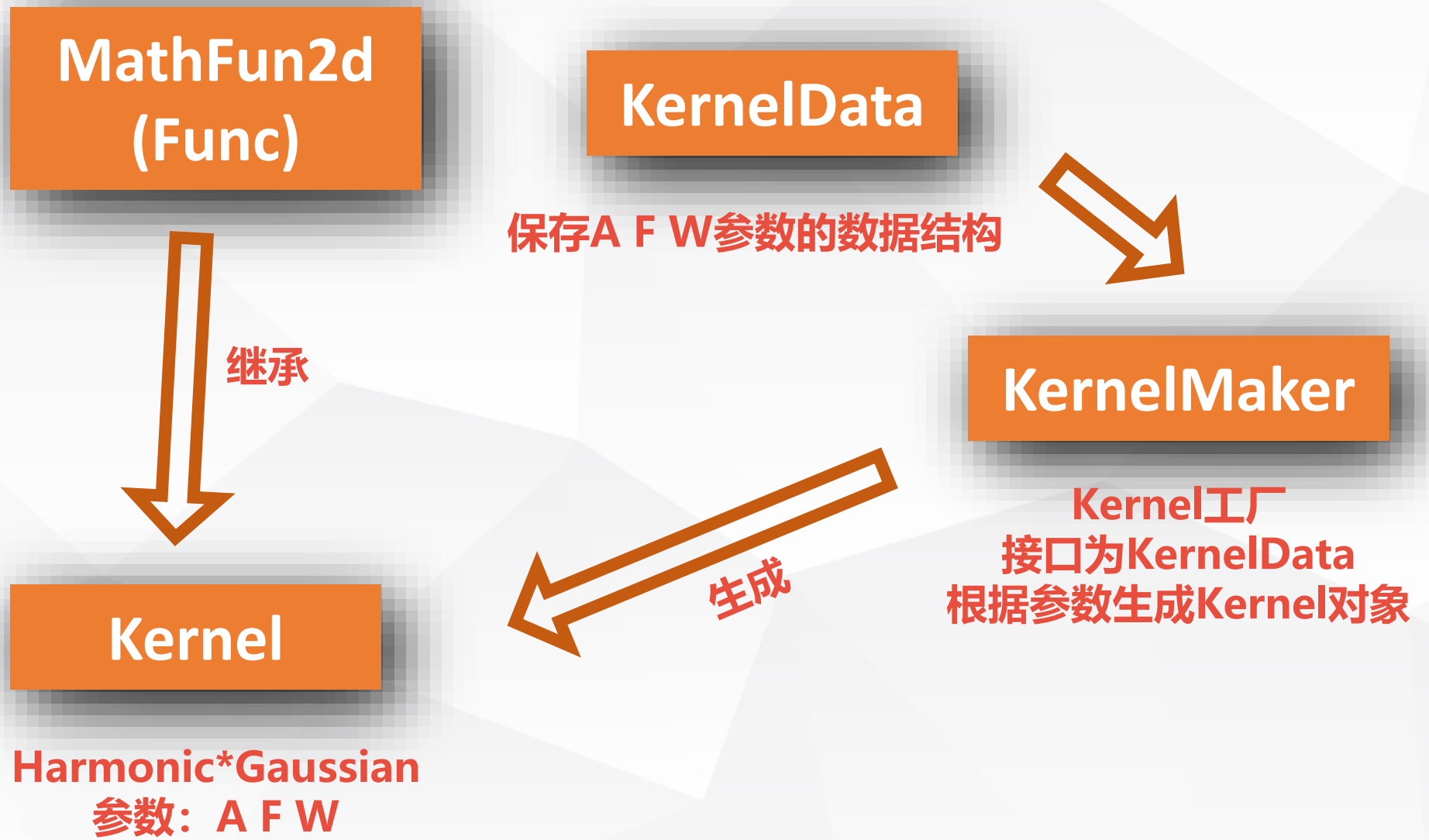
函数模块扩展(1)


MathFun2d
(Func)
Delta
Delta函数
参数：x y
Kernel
Harmonic*Gaussian
参数：A F W
Harmonic
二维余弦函数
参数：F频率 W角度
Gaussian
二维高斯函数
参数：A范围

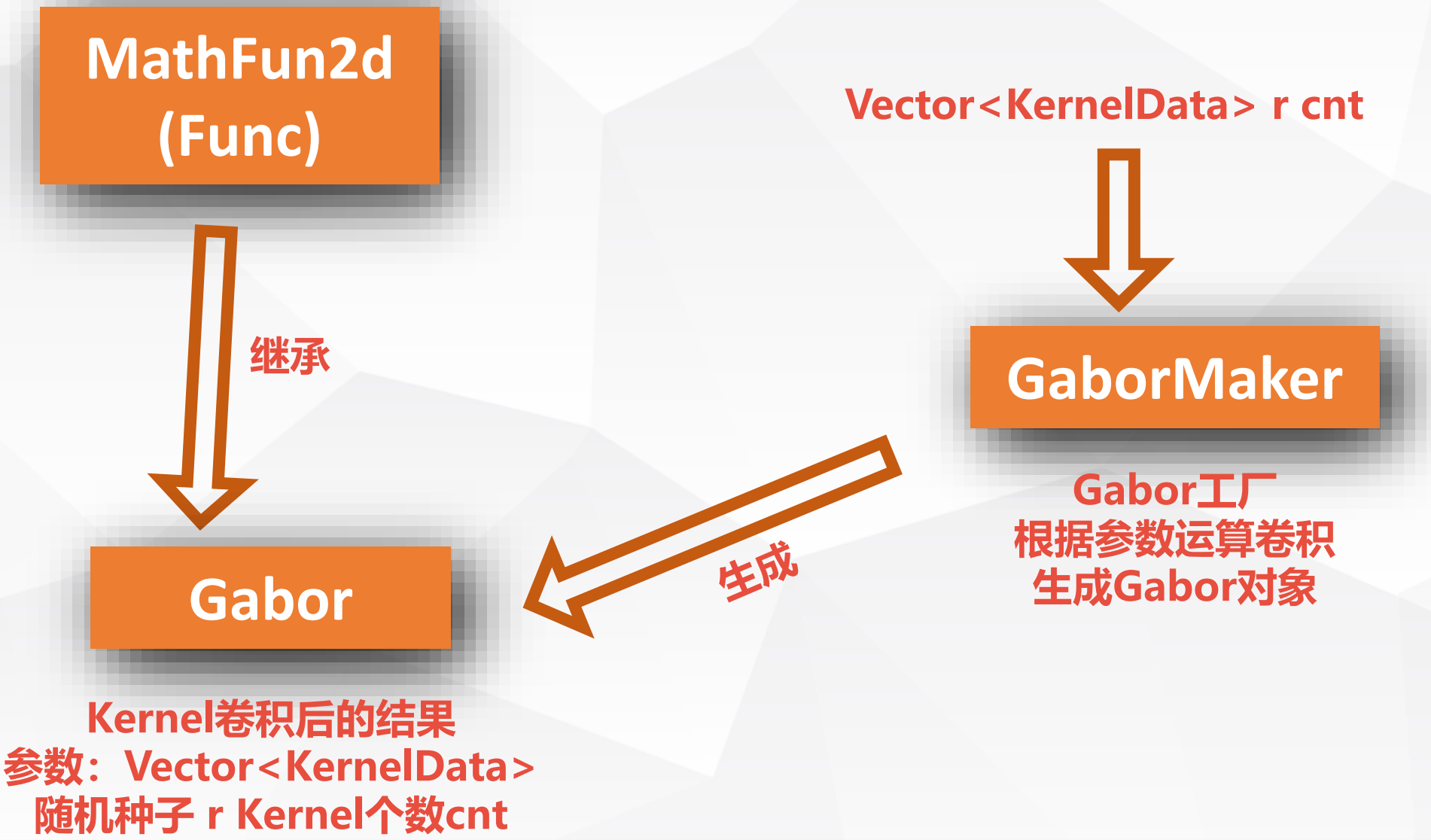
函数模块扩展(1)

在第一版的函数扩展中，对于最终Gabor Noise的生成方案是先实现Kernel以及Delta函数然后分别利用Image采样生成两张图像，然后对图像进行卷积运算得到纹理。但是在实现之后碰到的问题是Delta函数的采样过程以及Image的卷积速度很慢，运算效率很低。考虑到Delta函数为离散的点，参考论文中的实现算法，将Kernel函数直接通过随机参数完成卷积运算生成Gabor函数，最后对Gabor函数进行采样，效率得到大幅提高，扩展性也得到了增强。

函数模块扩展(2)


MathFun2d
(Func)
KernelData
保存A F W参数的数据结构
继承
KernelMaker
Kernel工厂
接口为KernelData
根据参数生成Kernel对象
生成
Kernel
Harmonic*Gaussian
参数：A F W

函数模块扩展(2)

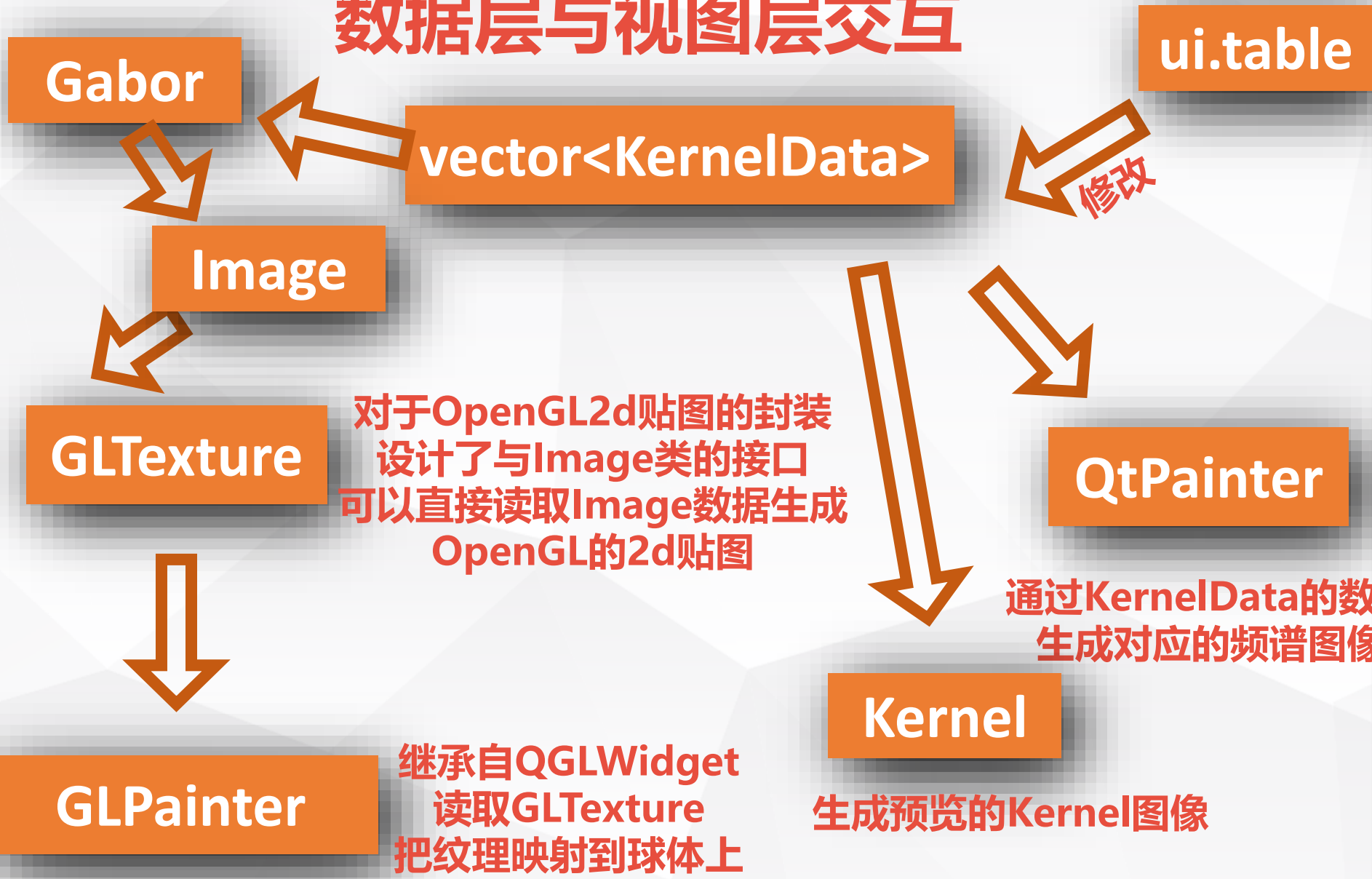

MathFun2d
(Func)
继承
Gabor
Kernel卷积后的结果
参数：Vector<KernelData>
随机种子 r Kernel个数cnt
生成
Vector<KernelData> r cnt
GaborMaker
Gabor工厂
根据参数运算卷积
生成Gabor对象

函数模块扩展(2)

在第二版的函数中决定最终纹理的核心的数据结构是：vector<KernelData>，随机种子r，Kernel的个数cnt，以及在后续的实验中扩展添加的对比度contrast

在KernelData中，除了一开始的设定的三个参数：A W F以外，后续的实验中扩展了权重P，以及为了方便设计参数增加了F range ， W range实现了Kernel的范围随机生成

数据层与视图层交互


Gabor
vector<KernelData>
ui.table
修改
Image
GLTexture
对于OpenGL2d贴图的封装
设计了与Image类的接口
可以直接读取Image数据生成
OpenGL的2d贴图
QtPainter
通过KernelData的数
生成对应的频谱图像
Kernel
GLPainter
继承自QGLWidget
读取GLTexture
把纹理映射到球体上
生成预览的Kernel图像

系统实现

开发工具与环境

开发环境：

Windows10 Visual Studio 2017

开发工具：

c++(MSVC 2014)

Qt 5.12

OpenGL

OpenMP

运行环境：

Window系统(x86)

核心代码

```
class GaborMaker {
public:
    map<double, KernelMaker> maker;
    int N, cnt,r;
    GaborMaker() {}
    GaborMaker(vector<KernelData> &data,int N = 10,int cnt = 20,int r = 0):N(N),cnt(cnt),r(r) {
        double sum = 0, t = 0;
        for (auto &i : data) sum += i.weight;
        for (auto &i : data) maker[t] = KernelMaker(i), t += i.weight / sum;
    }
    GaborCube Cube(double N, double con = 1, int r = 0) {
        return GaborCube(maker, N, cnt, con, r);
    }
    Gabor operator()(double con = 1,int x=0,int y = 0,int r = 0) {
        return Gabor(maker,N,cnt,con,x,y ,r);
    }
};
```

核心代码

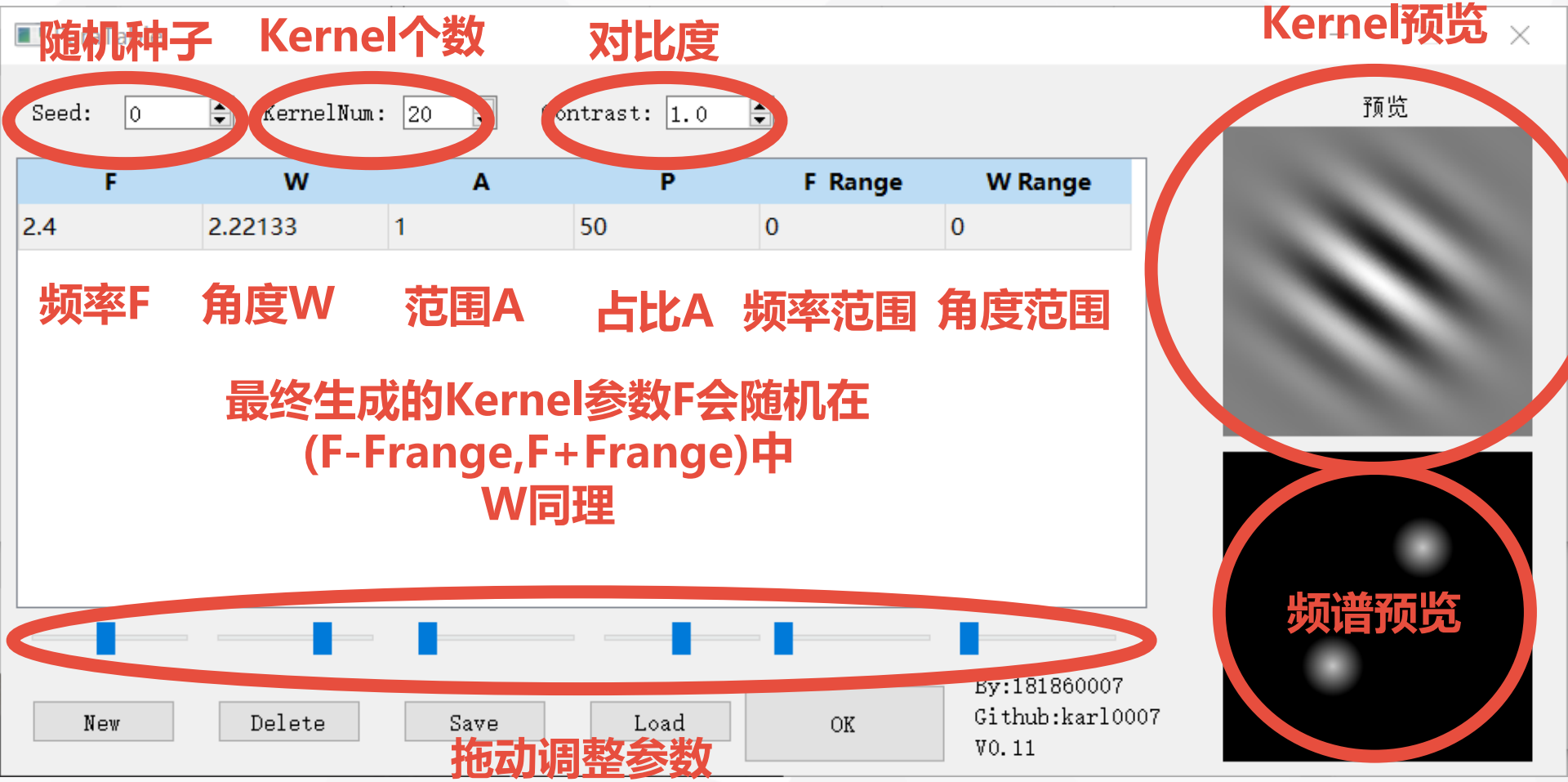
```
class KernelData {
public:
    Range W, F;
    double A;
    int weight;
    KernelData(Range &f = Range(5, 5), Range &w = Range(0, 0),double A = 1,int weight = 1) :
        F(f), W(w), weight(weight),A(A) {
        F = F.Fix(Setting::FMIN, Setting::FMAX);
        W = W.Fix(Setting::WMIN, Setting::WMAX);
    }
    string ToStr() {}
};


class KernelMaker {
    uniform_real_distribution<double> F, W;
    double A;
public:
    KernelMaker() {}
    KernelMaker(KernelData &Data):
        W(Data.W.l, Data.W.r), F(Data.F.l, Data.F.r),A(Data.A){
    };

    auto Make(int seed = 0) {
        minstd_rand e(seed);
        double w = W(e), f = F(e);
        return Gaussian(1.0/DR*A) * Harmonic(F(e)/DR, W(e));
    }
    double operator ()(int seed, double x0, double y0) {
        minstd_rand e(seed);
        static uniform_real_distribution<double> d(-DR, DR);
        double w = W(e), f = F(e), x = -d(e) + x0, y = -d(e) + y0;
        return  std::exp(-Const::pi*(1.0 / (DR*DR )* A * A)*(x*x + y * y)) * std::cos(2 * Const::pi*f / DR * (x*std::cos(w) + y * std::sin(w)));
    }
};
```

核心代码

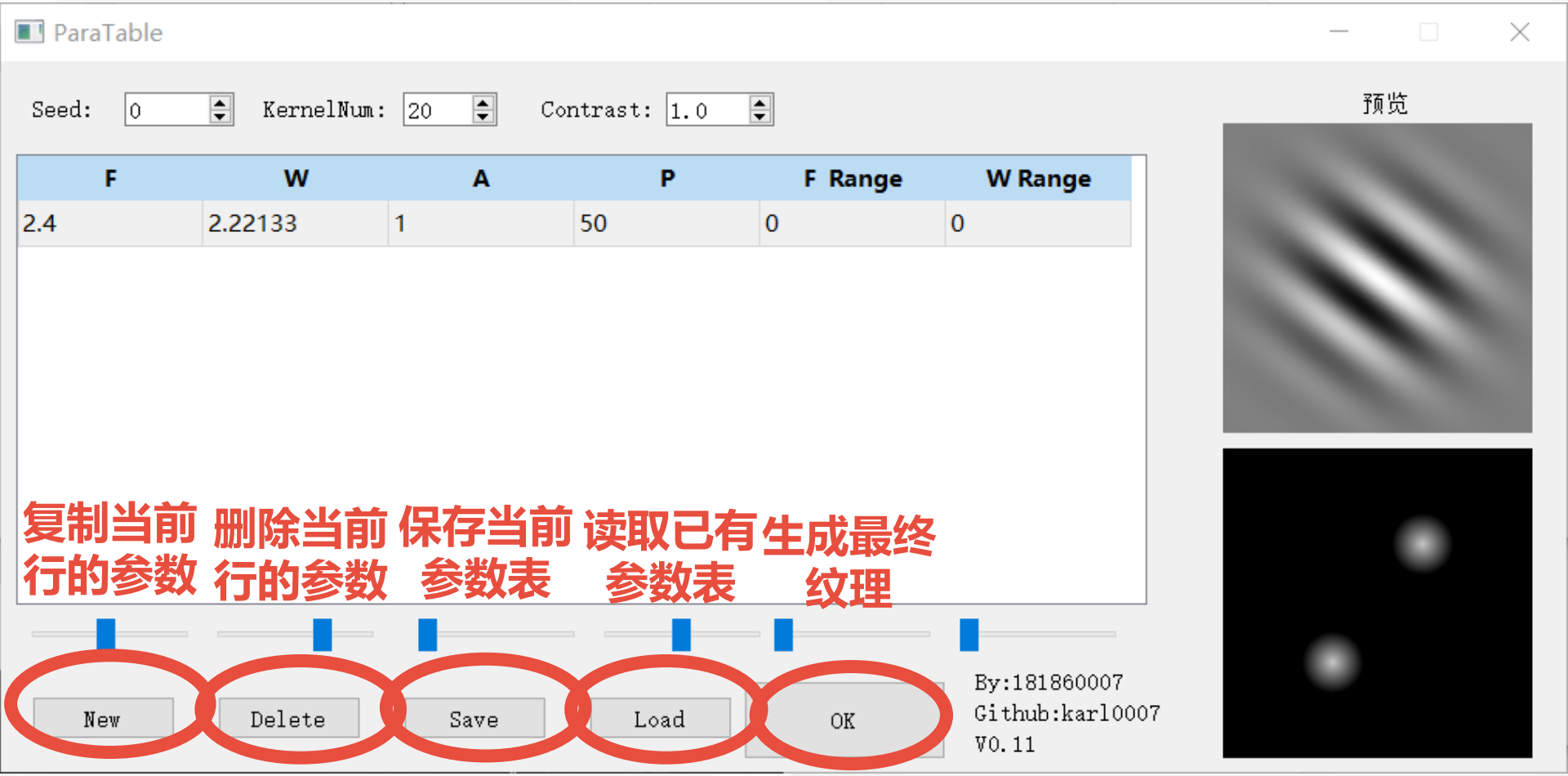
```
class Gabor : public Func {
public:
    map<double, KernelMaker> maker;
    int N,mx,my,cnt,r;
    double con;
public:
    Gabor() : Func(-1, 1) {}
    Gabor(map<double, KernelMaker> &maker, int N, int cnt,double con,int mx=0,int my=0,int r = 0)
        :Func(-1, 1), maker(maker), N(N), mx(mx),my(my),cnt(cnt),r(r),con(con) {}

    double operator()(double x, double y) {
        double res = 0;
        static uniform_real_distribution<double> d(-DR, DR);
        static uniform_real_distribution<double> d01(0, 1);
        static uniform_real_distribution<double> d11(-1, 1);

        int posx = Range(0, N).Reflect(Range(-DR, DR).Normalize(x));
        int posy = Range(0, N).Reflect(Range(-DR, DR).Normalize(y));
        for (int i = -1; i <= 1; i++) {
            for (int j = -1; j <= 1; j++) {
                minstd_rand e((posx+i+N)%N+ ((posy + j+N)%N)*N + r);
                for (int k = 0; k < cnt; k++) {
                    res += con * d11(e) * (--maker.upper_bound(d01(e)))->second(e(),
                        -(x*N - (posx - N / 2)*DR * 2) + i * DR * 2,
                        -(y*N - (posy - N / 2)*DR * 2) + j * DR * 2);
                }
            }
        }
        return res;
    }
```

效果演示

界面简介


随机种子
Kernel个数
对比度
Kernel预览
Seed: 0
KernelNum: 20
ontrast: 1.0
预览
F
W
A
P
F Range
W Range
2.4
2.22133
1
50
0
0
频率F
角度W
范围A
占比A
频率范围
角度范围
最终生成的Kernel参数F会随机在
(F-Frange,F+Frange)中
W同理
频谱预览
New
Delete
Save
Load
OK
By:181860007
Github:karl0007
V0.11
拖动调整参数

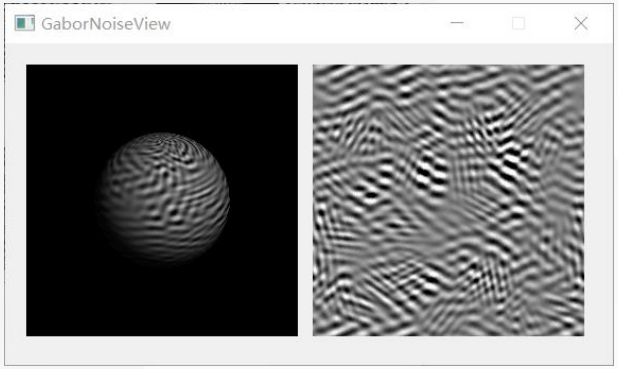
界面简介

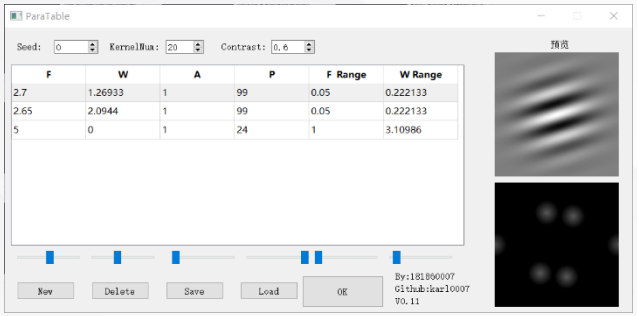

ParaTable
Seed: 0
KernelNum: 20
Contrast: 1.0
F
W
A
P
F Range
W Range
2.4
2.22133
1
50
0
0
预览
复制当前行的参数
删除当前行的参数
保存当前参数表
读取已有参数表
生成最终纹理
New
Delete
Save
Load
OK
By:181860007
Github:karl0007
V0.11

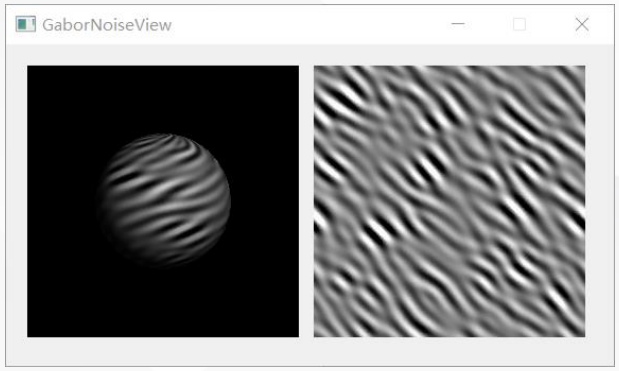
效果演示

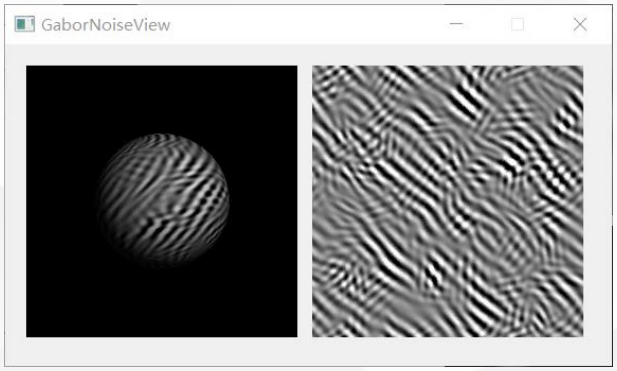
Demo展示

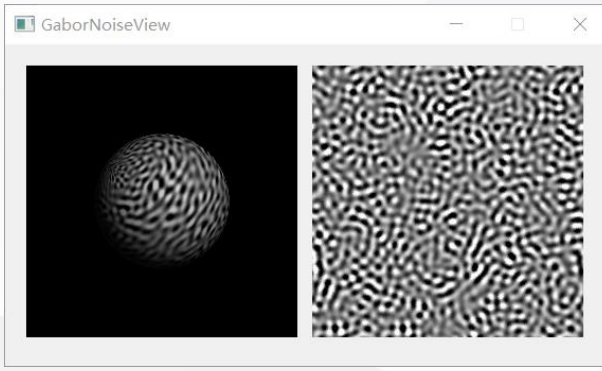
所有的参数表已保存在Demo文件夹中

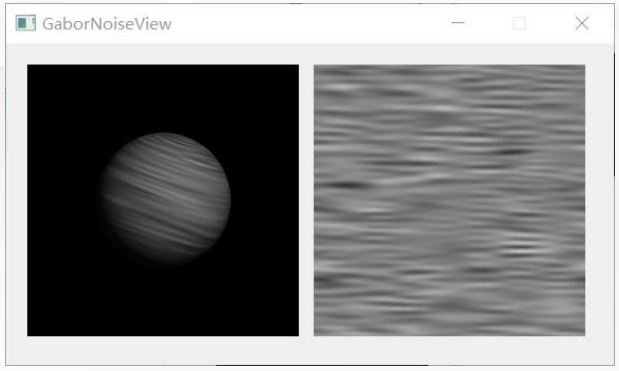


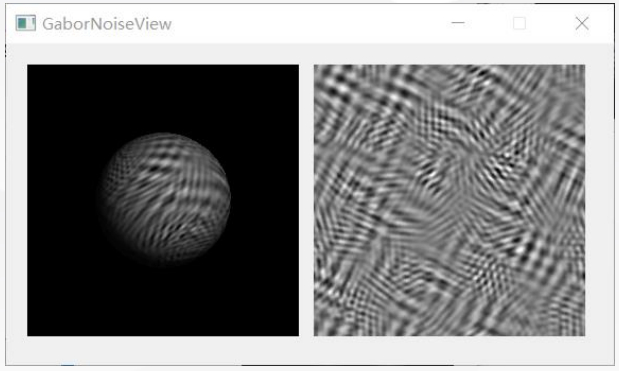


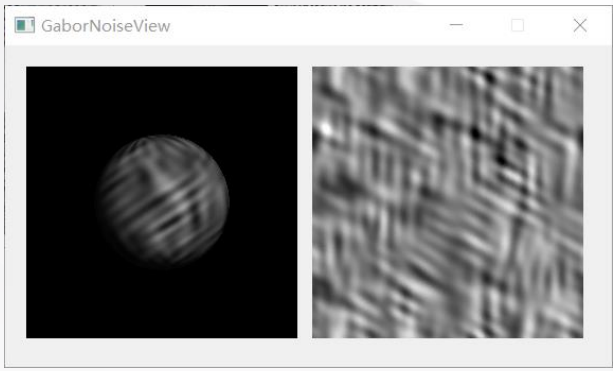












总结与展望

总结与展望

已解决的问题：

在开发过程中由于对于开发工具（Opengl QT）不熟悉，所以在学习过程中花费了不少时间。

在底层的二维函数库实现方面功能比较齐全，但是对于项目本身来说过于臃肿。

一开始没有理解论文中的算法导致卷积效率低下，在改进算法之后性能得到了明显的提升。

总结与展望

待解决的问题：

由于Range参数是后期为了方便设置加入的，考虑到效率以及实时的预览，并没有能够在预览界面与频谱中表现出来。在大量使用Range时不够直观，而如果重复的复制工作量就会很大。可以采取的解决方案是把Range拆解成多个Kernel Data。

提升效率方面由于对GPU编程不了解，所以改用OpenMP对一些计算进行了CPU的并行优化。

纹理映射方面由于调用了OpenGL的自带函数没有实现论文中原生的曲面纹理映射。

可以加入色彩映射使纹理更加丰富，但是受限于找不到合适的UI控件，只能读取参数表，与界面不符就舍弃了。

THANKS