1 Parallel Coordinates Visualization

这一部分其实就是两部分，一个是图的必要元素以及它们的定位（包括七个轴，标签以及上下限），另一个就是数据线的绘制。

首先在图上定出7个距离一样的坐标轴，并调用 `DrawLine` 函数绘制：

```
float x[7]={0.05,0.2,0.35,0.5,0.65,0.8,0.95};
for(int i=0;i<7;++i){

 DrawLine(input,glm::vec4(0,0,0,1),glm::vec2{x[i],0.1},glm::vec2{x[i],
0.9},3.0f);

 DrawLine(input,glm::vec4(0.5),glm::vec2{x[i],0.1},glm::vec2{x[i],0.9}
,20.0f);

 DrawLine(input,glm::vec4(0.5),glm::vec2{x[i]+0.016,0.1},glm::vec2{x[i
]+0.016,0.9},20.0f);
}
```

这里我采用了和参照图一样的绘制，在一条深色且比较细的坐标线两旁拓展出一个灰色矩形以获得更好的视觉效果。然后把标签以及上下限在图上的相应位置画出来：

```
std::string data_types[7]=
{"cylinders","displacement","weight","horsepower","acceleration(0-
60mph)","mileage","year"};
std::string data_num_upper[7]={"9","494","5493","249","27","51","84"};
std::string data_num_lower[7]={"2","29","1260","27","6","5","68"};
for(int i=0;i<7;++i){

 PrintText(input,glm::vec4{0,0,0,1},glm::vec2{x[i],0.04},0.02f,data_ty
pes[i]);

 PrintText(input,glm::vec4{0,0,0,1},glm::vec2{x[i]-0.007f,0.07},0.02f,
data_num_upper[i]);

 PrintText(input,glm::vec4{0,0,0,1},glm::vec2{x[i]-0.007f,0.93},0.02f,
data_num_lower[i]);
 }
```

接下来就是数据线的绘制。我们先定出每个数据在图上的百分比y坐标。由于我们的数据轴是线性的，所以得到这个坐标并不难。我们的坐标轴的y坐标从0.1到0.9，于是一个数据点在图上的y坐标其实就是

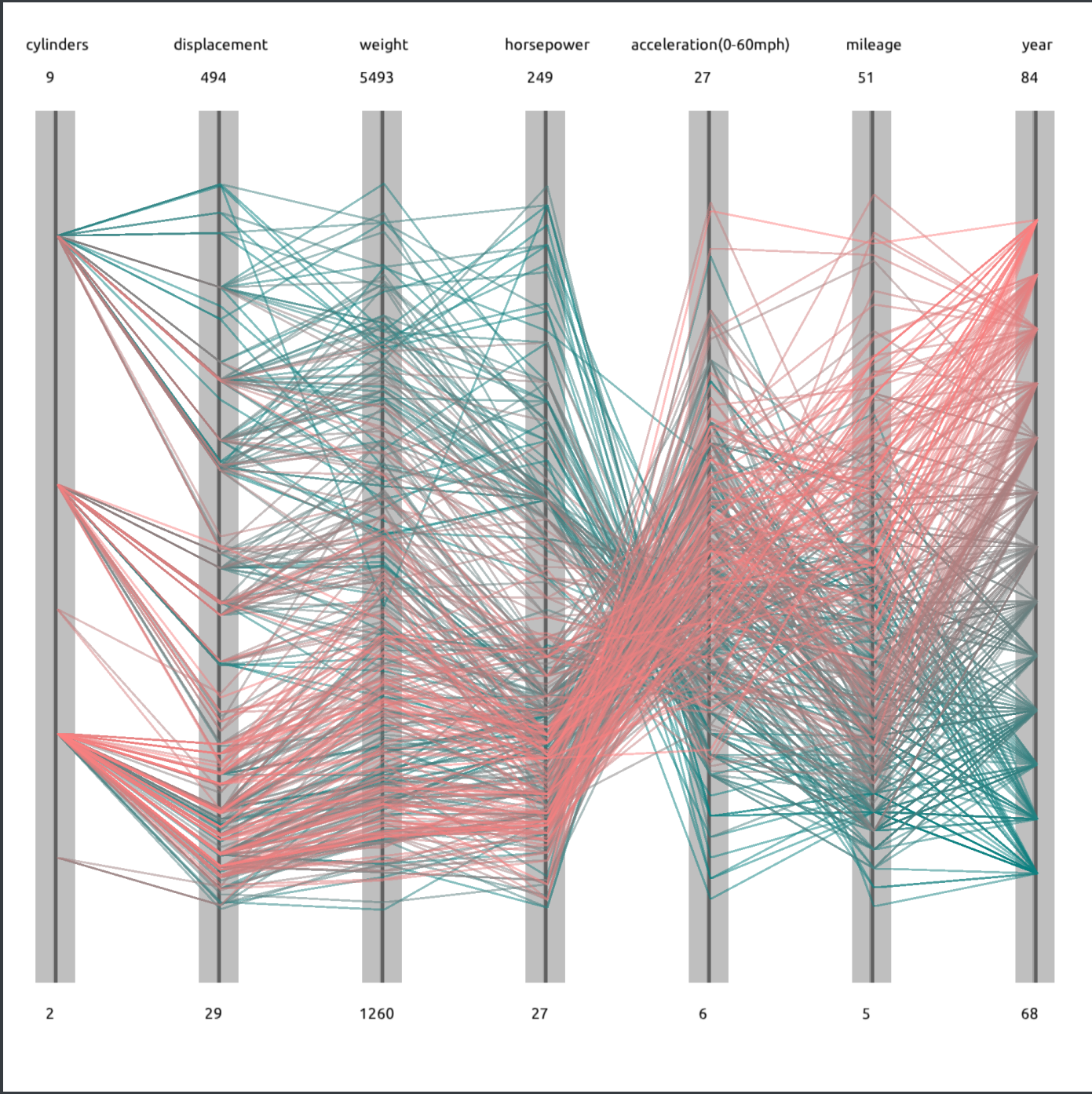
$$y = 0.8 \cdot \frac{up - data}{up - down} + 0.1 \tag{1}$$

其中up为上面端点的数据，$down$就是下面端点的数据。按照这个式子把每辆车的7个数据点画出来，然后用线连接起来就可以了。这里才用了渐变的颜色，使得效果看起来比较炫酷：

```
for(auto car:data){
    ++i;
    float color=(i*1.0f/data.size());
    glm::vec4 line_color=glm::vec4{color,0.5f,0.5f,0.5f}; // change
color gradually
    float pos_y[7]={
        0.1f+0.8f*(9-car.cylinders)/(9-2),
```

```
            0.1f+0.8f*(494-car.displacement)/(494-29),
            0.1f+0.8f*(5493-car.weight)/(5493-1260),
            0.1f+0.8f*(249-car.horsepower)/(249-27),
            0.1f+0.8f*(27-car.acceleration)/(27-6),
            0.1f+0.8f*(51-car.mileage)/(51-5),
            0.1f+0.8f*(84-car.year)/(84-68)
        };
        for(int j=0;j<6;++j)

    DrawLine(input,line_color,glm::vec2{x[j],pos_y[j]},glm::vec2{x[j+1],po
    s_y[j+1]},1.0f);
```

效果如下：



接下来，我想实现Protovis中拖动鼠标就可以让被选中的区域高亮，其他不在范围里面的数据变为灰色。具体来说：我们想要实现的效果是：

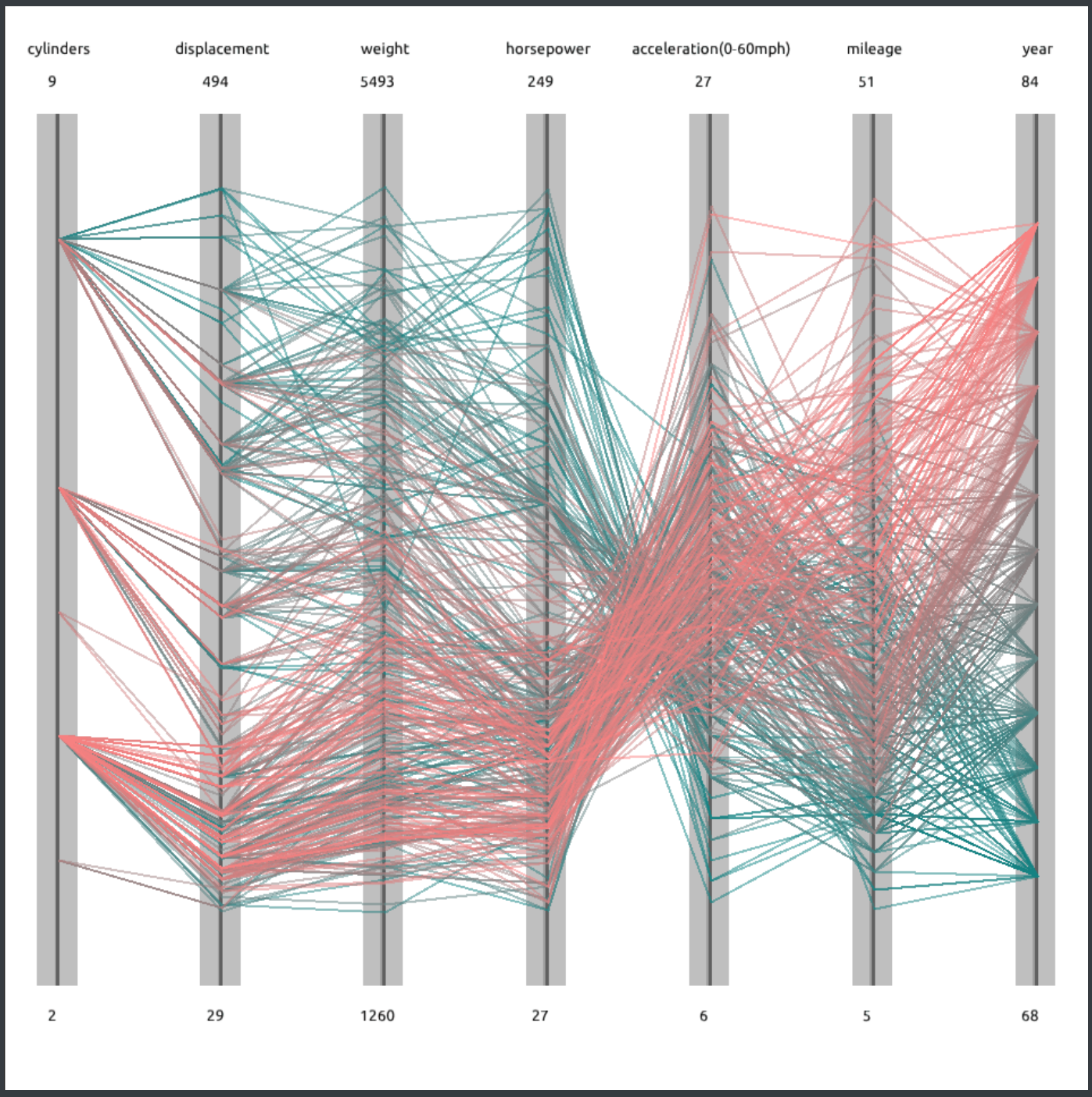
当鼠标在某一个坐标轴附近开始拖动后，不论之后它移动到什么x坐标，在原先的坐标轴上数据范围**在一开始的y坐标以及现在的y坐标之间的数据点对应的数据线**都高亮显示，其余数据线变为半透明灰色。

为此我们需要在绘制之前加一个判断：

```
if(!proxy.IsDragging())
    for(int j=0;j<6;++j)

DrawLine(input,line_color,glm::vec2{x[j],pos_y[j]},glm::vec2{x[j+1],po
s_y[j+1]},1.0f);
else{
    glm::vec2 start_pos=proxy.DraggingStartPoint();
    glm::vec2 end_pos=proxy.MousePos();
    float start_x=start_pos.x;
    float y_lower=std::min(start_pos.y,end_pos.y);
    float y_upper=std::max(start_pos.y,end_pos.y);
    bool selected_an_axis=false;
    for(int j=0;j<6;++j){
        if(start_x>=x[j]-0.016f&&start_x<=x[j]+0.016f){
            selected_an_axis=true;
            if(pos_y[j]>=y_lower && pos_y[j]<=y_upper)
                for(int k=0;k<6;++k)

DrawLine(input,line_color,glm::vec2{x[k],pos_y[k]},glm::vec2{x[k+1],po
s_y[k+1]},1.0f);
            else
                for(int k=0;k<6;++k)

 DrawLine(input,glm::vec4{0.5f,0.5f,0.5f,0.1f},glm::vec2{x[k],pos_y[k]
},glm::vec2{x[k+1],pos_y[k+1]},1.0f);
        }
    }
    if(!selected_an_axis)
        for(int j=0;j<6;++j)

 DrawLine(input,line_color,glm::vec2{x[j],pos_y[j]},glm::vec2{x[j+1],p
os_y[j+1]},1.0f);
}
```

相当于如果鼠标开始拖拽，我们就记录下开始拖拽时的坐标 `start_pos` ，以及当前鼠标位置 `end_pos` ，然后计算出y坐标的上下限 `y_lower` 以及 `y_upper` 。接着判断开始拖拽时的x坐标是否在某个坐标轴附近，如果是的话就判断该数据点的y坐标是否在上下限之间，如果在的话那么这条数据线就高亮显示，否则就变为灰色半透明。如果开始拖拽时的x坐标并不在任何一个坐标轴附近，那么就和没有拖拽时一样显示所有数据线。

效果如下：

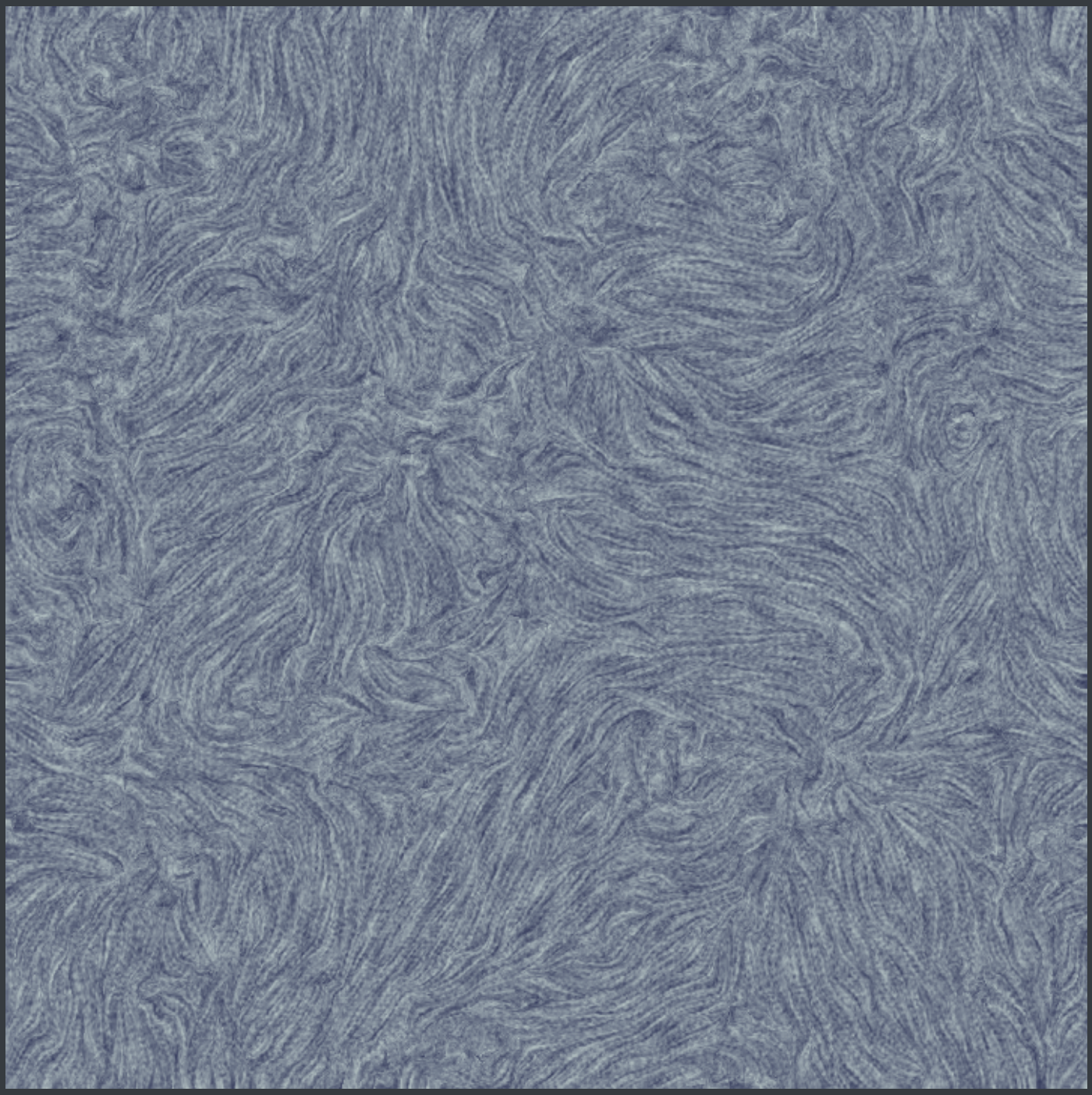


*给出的 `IsDragging` 函数很不好用，因为它需要鼠标**正在移动**才能判定为 `True` ，这对我们的眼睛来说不太好，因为拖动鼠标的时候总有那么一些时刻鼠标按下但是静止，会出现拖动过程中一闪一闪的情况*

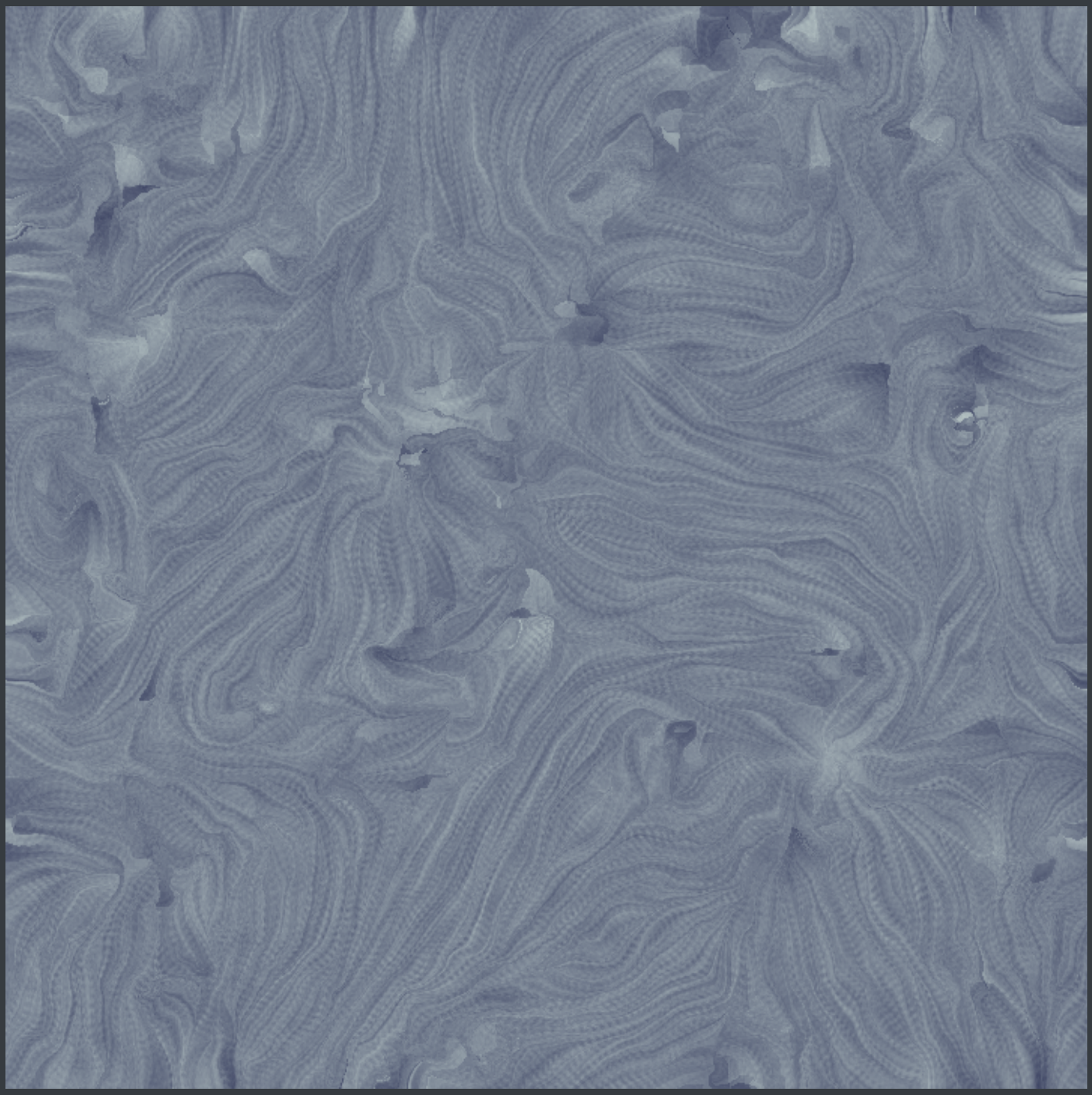
2 Flow Visualization

这一部分我基本上就直接根据给出的lic.py来实现，也就是把这个代码翻译成C++代码。原理上我理解了，基本上就是对于每一个点，每一步沿着向量场走到最近的像素网格，然后将每一步得到的颜色加权平均即可。

`Turbulence` 场在 `step=10` 的实现效果如下：



`step=50` 时效果如下：



可以看到， `step` 越大，得到的结果越平滑。