基于GraphCut的纹理合成

2017013599 从业臻

一、运行环境

系统：Windows10

语言：Python 3.7

依赖：PyMaxflow, NumPy, opencv-python

二、功能实现

- 所有基础要求
- **切割算法改进----处理 surrounded regions，1'**
- **切割算法改进----考虑旧的切割，1'**
- **切割算法改进----能量函数引入梯度信息，1'**
- **偏移算法加速----快速傅里叶变换，3'**

加分项**总计6'**。

三、代码思路

1 块切割算法

1.1 建立图结构

设最终的合成图片大小为(H, W)，纹理图片大小为(h, w)。合成图片初始为全0的$H \times W \times 3$数组，下面成为***画布***。使用了一个二维数组$filled$记录画布每个像素点是否已经被纹理覆盖。

我考虑了新旧区域重合区域为**任意可能（包括surrounded regions的情况）**，但新区域一定是矩形的情况。这种情况下，需要构造三种边：

- 遍历新区域，根据$filled$得到重合区域，并产生重合区域内部相邻点之间的边（双向）。
- 遍历新区域的边界，如果属于重合区域，且向外延伸一个像素点（比如上边界，向外延伸就是向上一个像素；右边界的延伸就是向右一个像素）属于旧区域，那么该像素点落在旧区域和重合区域的交汇处，应有一条以该像素点为终点，以源为始点，流量为∞的边。
- 遍历新区域，如果属于重合区域，且周围任意一格满足：

  1. 仍在新区域内
  2. 不属于旧区域

  那么该像素点落在新区域和重合区域的交汇处，应有一条以该像素点为始点，以汇为终点，流量为∞的边。

边的添加使用`PyMaxflow`库的`maxflow.Graph.add_edge`和`maxflow.Graph.add_tedge`实现。

特别地，如果新区域被完全包围，则选择其中心像素点与汇相连。

1.2 求解最小割

由最小割最大流定理，求最小割即是求最大流，使用`PyMaxflow`库的`maxflow.Graph.maxflow`实现。

1.3 像素拷贝

`maxflow.Graph.get_grid_segments`则给出了求解最大流之后，对图中端点的分类（即源和最小割之间还是最小割和汇之间）。根据该分类矩阵，可以选择性的将纹理的像素点拷贝到画布上。

1.4 能量函数

实现了**引入梯度信息的能量函数。**由于计算能量函数总开销很小，我直接对每个边单独调用该函数。

1.5 考虑旧切割

将维护两个矩阵，分别记录纵向和横向的切割，矩阵中的数据包括边的能量函数，边两端点的像素值（新旧patch），以实现后续**添加seam node**时能量函数的计算。

2 块偏移生成算法

2.1 随机生成

行列分别产生一个$[0, H-h]$和$[0, W-w]$范围内的随机数作为偏移即可。

2.2 最佳位置匹配

遍历所有可能情况，计算SSD本身很简单，但是实验发现对于稍大一些的图（比如🤔）开销就有点过高了。因此，实现了基于$summed\ table$和FFT的加速。

首先，暴力遍历的方法，时间复杂度为$O((H-h)(W-w)hw)$，对于大多数情况，H, W分别是h, w的若干倍（下面也都默认这样的情况分析，用h, w替代H, W），那么时间复杂度约为$O(h^2w^2)$，对于200×200的纹理来说，已经是一个很高的复杂度了。参考课件和项目说明：

$$C(t) = \sum_p \mathbf{I}^2(p-t) + \sum_p \mathbf{O}^2(p) - 2\sum_p \mathbf{I}(p-t)\mathbf{O}(p)$$

其中第二项对于画布所有位置都一样，开销为$O(hw)$；第一项可以使用预积分表计算（使用`numpy.cumsum`高效计算），开销为$O(HW) = O(hw)$；第三项则可以使用快速傅里叶变换计算（使用`cv2.filter2D`），开销为$O(hw\ log(hw))$。总复杂度为$O(hw\ log(hw))$，理论上应该比暴力方法快很多。

注意到这里其实有个小细节，即对画布某个位置，代价函数应为

$$\frac{\sum_p [filled(p-t)((I(p-t)-O(p))]^2}{\sum_p filled(p-t)},$$

这样实际上原本的第二项应该是$\sum_p O(p)^2 * filled(p-t)^2$，不过这不是问题，只不过是另一个快速傅里叶变换计算；此外，上式分母也使用预积分表计算。总的复杂度仍是$O(hw\ log(hw))$。

2.3 生成概率分布

得到所有位置的代价函数值后，使用一系列`numpy`操作生成概率密度，进行`numpy.cumsum`得到分布数组，然后`for`循环遍历判断即可。

2.4 子块最佳位置匹配

匹配原理同2.2。

2.5 额外规则

为了更快地填充整个画布的同时尽可能保证接缝的平滑，我规定了单次操作，新旧区域重合面积不小于某个阈值（新区域的10%），也不大于某个阈值（新区域的70%）；如果重合面积一定会大于某个阈值，那么优先选取重合面积小的。

四、实验结果

3.1 拼接策略

我设计了几种拼接策略：

1. 一行行铺满，每行使用最佳位置匹配（先铺满一行，然后寻找全局最佳位置，再铺满一行，如此循环直至完全铺满）
2. 全局子块最佳位置匹配，子块面积为原来的1/4（等比例），子块为随机截取
3. 一行行铺满，每行使用子块最佳位置匹配，子块面积为原来的1/4

3.1 合成图片展示

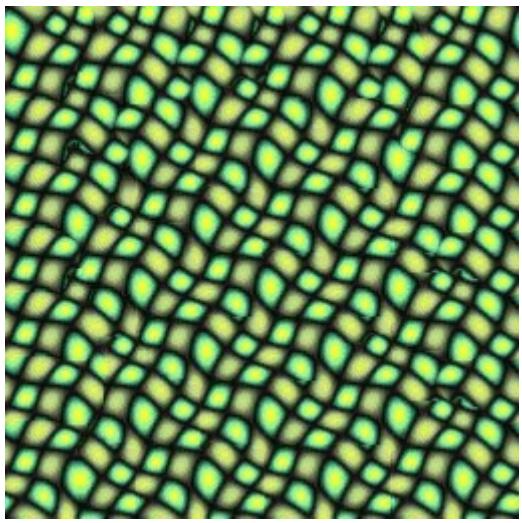
以下图片，位置匹配的超参数k=100，随机性较低。

（4倍面积）：

策略3

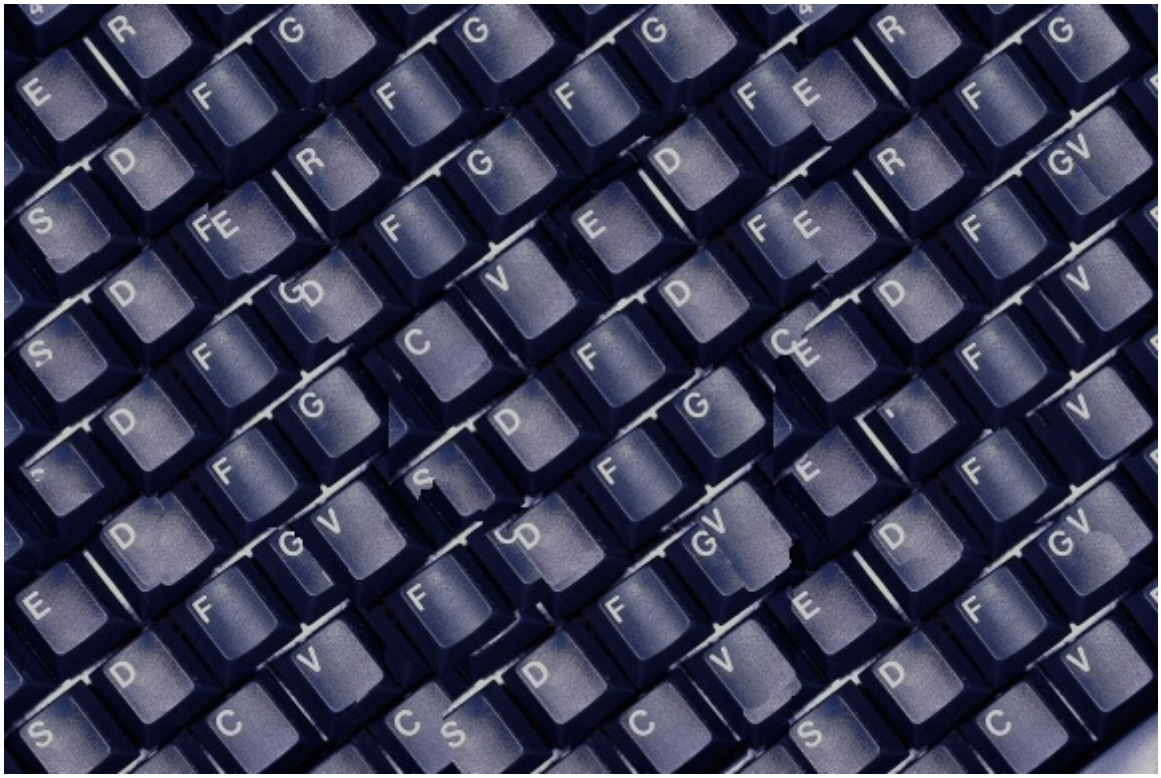
绿色纹理（16倍面积）：

策略1



键盘（9倍面积）：

策略1



3.3 能量函数引入梯度前后比较

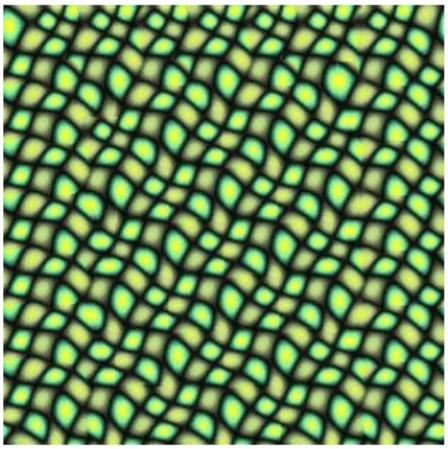
虽然不是非常明显，但是引入了梯度之后似乎效果略好一点。比如，对于键盘，左图左下方不连续处较多，整体质量略差于右图。对于草莓，右图看上去更自然，作图中间偏上有一块明显的不连续。

不使用

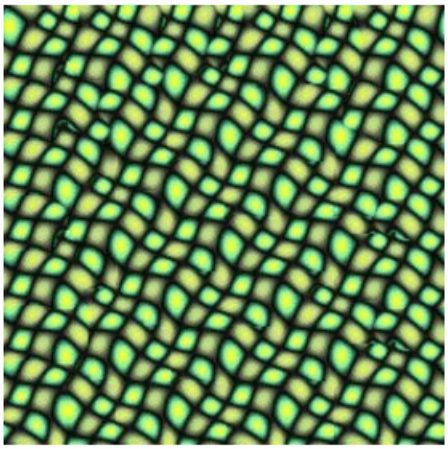
使用

不使用

使用



不使用



使用

3.4 考虑old seam前后比较

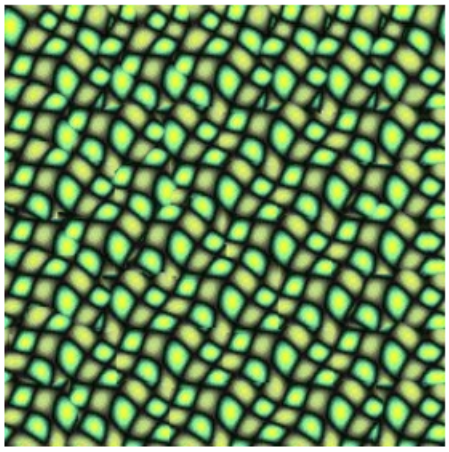
同样考虑old seam前后效果差别不是很大。对于绿色花纹，作图上半部分有些地方接缝较为明显，而右图则非常平滑。

不使用

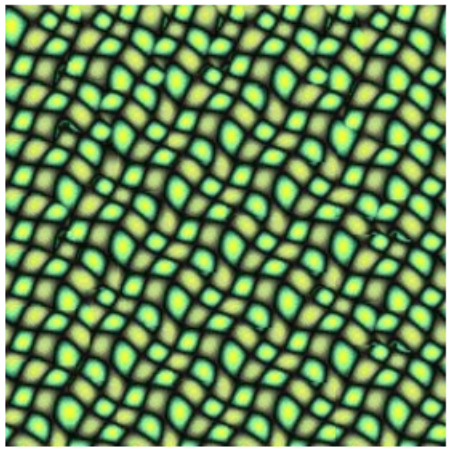
使用

不使用

使用



不使用



使用

3.5 使用预积分表与FFT加速前后比较

可以看出，加速效果非常好，快了几百倍。即使将画布大小设为花纹的100倍（面积），加速后一次全局最优位置匹配的时间依然较低，可以看出时间的增长和画布面积的关系大约是线性的，与理论时间复杂度较为一致。

	画布大小	花纹大小	加速前时间开销（一次全局最优位置匹配）(s)	加速后时间开销（一次全局最优位置匹配）(s)
绿色	256×256	64×64	6.82	0.04
键盘	576×384	192×128	99.00	0.18
	360×270	240×180	25.78	0.14
绿色	640×640	64×64	/	0.27
键盘	1920×1280	192×128	/	1.65
	2400×1800	240×180	/	2.99

五、总结

这个项目收获丰富，包括但不限于：

- 阅读论文和课件动手实现一个经典算法
- 学习预积分表、FFT这两个优化技巧
- 更加熟悉`opencv-python`&`NumPy`，且学习了`PyMaxflow`
- 锻炼编程能力

感谢老师与助教的指导！