基于 GraphCut 的纹理生成

计 75 班 赵成钢 2017011362

2020 年 12 月

目录

1 目录结构和采分点

1.1 目录结构

1. images/originals/*：纹理原图
2. images/outputs/*：程序输出
3. stb/stb_image.h：STB 库图片读取（来源 GitHub）
4. stb/stb_image.h：STB 库图片写入（来源 GitHub）
5. stb/stb_lib.cpp：STB 库链接主文件
6. cherry.hpp：自己实现的一个 C++ 扩展库（也经常在其他项目中用）
7. CMakeLists.txt：CMake 文件
8. compile.sh：通过 CMake 编译
9. dft.hpp：2D FFT 的具体实现
10. dft_test.cpp：2D FFT 的简单测试
11. graph.hpp：点边为基础的图相关定义实现，包括 Dinic 网络流最小割实现
12. image.hpp：图片相关逻辑实现，包括像素（Pixel）、图像（Image）、贴图（Texture）和画布（Canvas）的实现
13. main.cpp：程序入口，负责整个控制流
14. placer.hpp：几种匹配方式（Random、Entire matching 和 Subpatch matching）的具体实现
15. run_all.sh：运行全部纹理

1.2 编译运行方式

在代码目录运行 compile.sh 即可运行编译，编译后在代码目录运行 run_all.sh 即可运行全部提供的测例。

程序参数：graph_cut <input> <output> <canvas_size>，其中 input 和 output 分别是输入纹理路径和输出路径，canvas_size 是画布大小，示例：graph_cut peas.png peas_output.png 512x512。

1.3 采分点

1. 基本算法：主要对应 Canvas 类中对 Patch 类的 apply() 函数（image.hpp），apply() 函数将会调用 Graph 类中的结构建图（graph.hpp），随后通过 Dinic 算法求解最小割（graph.hpp）
2. Old cuts：在 Canvas 类中的 apply() 函数（image.hpp）中，建图考虑了 Seam nodes
3. 最佳接缝：在 Placer 类中实现了 Random、Entire Matching 和 Sub-patch Matching 三个算法（placer.hpp）
4. FFT 加速：在 Placer 中的 Entire matching 算法中调用了 FFT 算法，FFT 的具体实现在 dft.hpp 中

2 代码思路

2.1 整体思路

首先，对于整体贴图生成的方式，我认为这个概念类似于 PhotoShop 中的画布上打补丁的概念，所以首先我定义了 Image 基类来表示全部的图片，后面定义了画布类 Canvas 继承自 Image，又定义了包含 Image 指针和 offset 的 Patch 类。随后，我们需要做的只是把生成不同 Offset 的 Patch，并把这些 Patch apply 到最后的 Canvas 上。整体架构如下图所示。

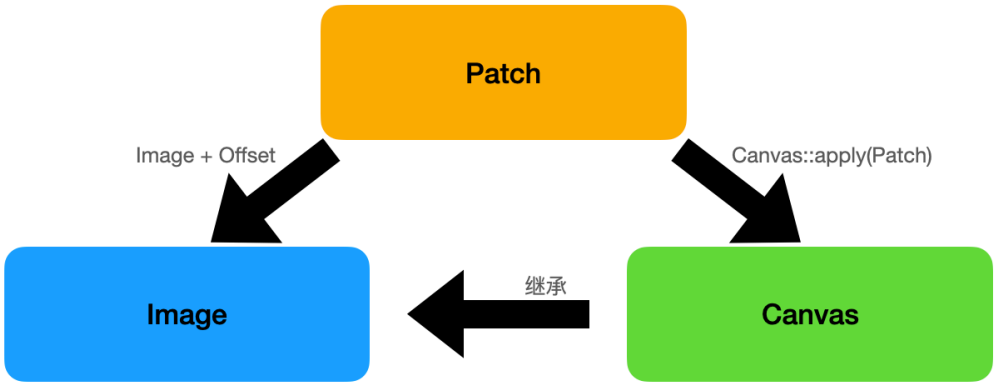

图 1: 整体架构

2.2 Apply 函数实现

对于 Apply 函数，我们需要做的是先找到 overlapped 的区域，这个直接通过每个像素判断即可。那么没有被 overlapped 的区域，就直接把 Patch 的像素直接赋值到 Canvas 的对应位置上。

对于 overlapped 的区域，我们枚举每个这样的像素，循环它的上下左右四个点进行建图，如果它的某个邻居是边界外的来自原图或者来自 Patch 的像素，那么我们就像论文中一样从 S 到该

点连边或者从 T 从该点连边；如果某个邻居不是边界外的点，那我们就通过文中的 $M(s,t,A,B)$ 代价连边，我对于该代价的理解是，要替换的像素（新耦合的边界点）和原来没有缝隙的边界点的像素距离和。这样，我们就完成了基础的建图。

对于进阶的考虑 Old cuts 的情况，首先，我们不需要存储具体的 Old cuts，我们只需要记住每个像素来自哪个 Patch，如果一对邻居来自不同的 Patch 那自然是一对 Old cuts（而且这样把之前直接覆盖的情况也纳入的边界，考虑更加周全），之后，我们再按照文中的方式建图，分别连接邻居和中间的 Seam node，同时 Seam node 连到 T 上把 Old cuts 的代价作为边权。我个人觉得这个连接方式非常精妙，首先，这样连有一个大前提，就是代价函数满足三角形不等式，这意味着在三者中只会选择一个边来切（选择一个以上代价只会更大），那么我们需要反着考虑每个边的代价（也就是连了剩下两条边的连通性），而不是顺着边考虑代价。

2.3 最小割实现

这里采用了基于 BFS 增广和 DFS 的 Dinic 算法，具体细节不在赘述，详见 graph.hpp 文件。

2.4 Matching 实现

对于 Matching，我在 placer.hpp 中实现了三种方法：Random、Entire Matching 和 Sub-patch Matching。

在画布是空的时候，我先使用了 Random 的方式每次控制随机的范围先保证把整个画布铺满，这样虽然在后面导致后面收敛的效果没有那么好，但是基于 FFT 的匹配必须保证中间没有空洞，所以目前实现为了简单也只是做到了前面随机铺满再后面修补。

对于 Entire Matching，一种简单的实现是随机 offset，然后计算 SSD，随机多次取最好的结果，在下个小节中，我会介绍基于 FFT 的做法。

同样对于 Sub-patch Matching，我们先取画布上的一小块，然后在 Patch 中去 match，这样计算量相对较少，同时效果也比较好，一种简单的实现也是进行随机（目前实现），后续也可以用 FFT 来加速。

2.5 FFT 加速

对于 offset (x,y)，我们可以计算 SSD 为 $SSD_{(x,y)}=\sum_{i=0}^{w}\sum_{j=0}^{h}[Patch(i,j)-Image(i+x,i+y)]^2$，进一步展开为 $\sum\sum Patch(i,j)^2+Image(i+x,i+y)^2-2Patch(i,j)Image(i+x,i+y)$，对于前两项，我们可以直接用二维的前缀和 $O(wh)$ 预处理，$O(1)$ 来查询；对于第三项，我们将 Patch 进行翻转，令 $Flipped(i,j)=Patch(w-i-1,h-j-1)$，这样第三项可以写作 $-2Flipped(w-i-1,h-j-1)Image(i+x,i+y)$，注意到每一维的下标之和与 i 无关，这是一个卷积的形式，那么我们只需对 Flipped 和 Image 进行 DFT，在频域相乘再进行 IDFT，可以在 $O(wh(\log w+\log h))$ 的复杂度内完成运算。

对于 DFT 的具体实现，其实就是对着两维分别做一维 DFT（另一维循环），具体可以这样做的原因是 DFT 式子中 e 上关于 i 和 j 的指数是加法，可以拆成两个一维的 DFT，具体实现可以参考 dft.hpp。

最后还需要注意的是，我们一下得到了全部的 SSD 值，如果只取最小的那一个，那么最后非常有可能每次迭代都选择同一个（不停地贴一个地方）。对应这种情况，我们可以用文章提供的概率方法，定义 $P(t) = e^{-\frac{C(t)}{k\sigma^2}}$，然后我们把所有 offset t 的概率全部求出来，然后根据这个分布随机进行选择，为了让结果随机性更强，我们可以把 k 调大，这样分布会更加平滑，因为 SSD 的大小差距带来的概率差异会减少，随机性会更强。

3 实验结果和分析

下面分别是提供的四张图片的结果：

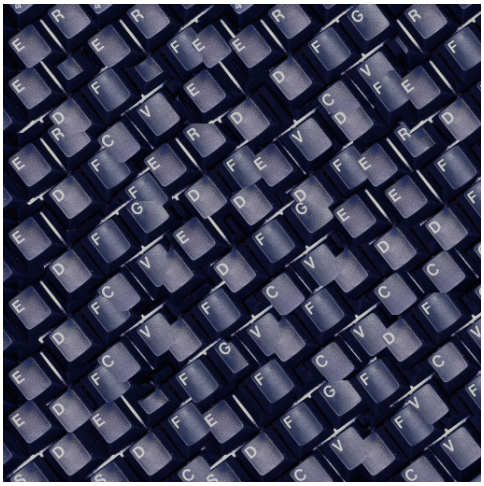

图 2: 键盘

图 3: 鹰嘴豆

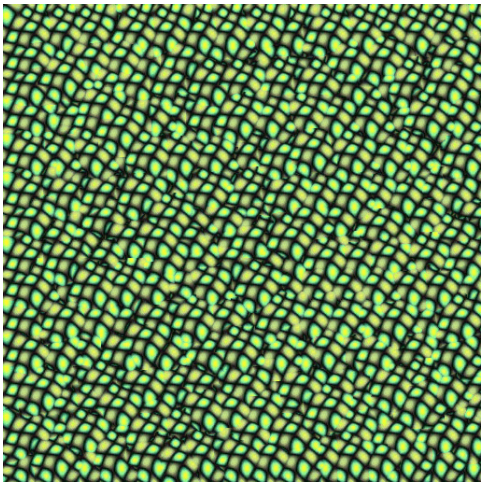

图 4: 绿色纹理

图 5: 草莓

可以看到其中草莓和鹰嘴豆的效果相对较好，在远处看几乎和真实的一样，这也得益于本身纹理的不规则性。而相对规则的键盘和绿色纹理因为我在开始的时候是随机铺满的画布，后面再进行的 refinement，所以最后的效果很大程度上受到随机铺满的影响，效果没有那么好。

4 感想和感谢

本次作业中我认为非常有意思，不同于其他课程的作业，本课程可以真正看到自己在做什么，让我感受到了以媒体为载体和目标的计算的乐趣，也感慨前辈们在目前生活中电影、影像等技术中的贡献。

感谢徐老师的讲解和助教的批改！