机器视觉第四次作业

1. 写代码实现算法，并分析合成纹理窗口大小（size）、形状 （shape），和不同匹配误差度量（not ssd）对合成效果的影响。

该非参数纹理合成算法实现的步骤为：

首先，从输入图像中随机选择一个像素块作为合成图像的构建单元；随后将选中的方块放置到另一个图像（通常为全 0 图像）中的一个随机位置；对于每个未合成的位置，计算其邻域与示例图像中所有可能位置的领域之间的匹配度，当找到与目标位置匹配度较高的区域，则从示例图像中对应的位置随机选取一个值填充到目标图像中的位置，重复步骤直到素有位置都填充完毕。

在代码中，首先设置随机选择的像素块的大小以及目标生成图像的大小。定义了随机选择像素块的函数 get_patch，给函数从给定的输入图像中随机选择一个规定大小的像素块，如图 1 所示，函数首先获取图像的高度和宽度，并从中选择像素块的左上角坐标，随后返回该像素块和左上角坐标。

```python
def get_patch(image, patch_size):
    """从示例图像中随机选择一个小方块"""
    h, w = image.shape[:2]
    #随机生成一个图像块的上边界和左边界坐标，即左上角坐标
    top_left_y = random.randint(0, h - patch_size)
    top_left_x = random.randint(0, w - patch_size)
    #计算得到图像块
    patch = image[top_left_y:top_left_y + patch_size, top_left_x:top_left_x + patch_size]
    return patch, (top_left_x, top_left_y)
```

图 1 随机选择像素块代码

随后，为了匹配邻域，定义了 match_patch 函数在原输入图像中搜索最匹配的区域并返回。首先从待生成图像中获取当前位置的坐标以及当前待填充位置，通过遍历原输入图像中的每一个可能的位置，计算该位置的像素块与待生成图像位置邻域之间的均方误差，当误差越小时，匹配度越高，所以这个像素块则为最佳的匹配块。此处也可以将均方误差改为余弦相似度，由于余弦相似度与 MSE 相反，它是与匹配度成正比，但是计算相似度的指标是余弦距离，其是与匹配度呈反比，即余弦距离越小，余弦相似度越高，匹配度越高，所以初始标准仍可以定为 inf，并将邻域和原始输入图像扁平化为向量，最后输入到计算函数中计算两个扁平化向量的余弦距离再选择最小的向量较小输出，两种误差度量方式如图 2(a)(b)所示。

```python
def match_patch(target_image, example_image, patch_size, target_pos):
    """在示例图像中匹配目标图像当前位置的邻域"""
    t_y, t_x = target_pos
    target_neighborhood = target_image[t_y:t_y + patch_size, t_x:t_x + patch_size]
    # 计算匹配度
    best_score = float('inf')
    best_patch = None
    h, w = example_image.shape[:2]
    #由上到下，由左到右遍历原输入图像中的每个可能的小块位置，每次截取一个patch_size大小的小块
    for y in range(h - patch_size + 1):
        for x in range(w - patch_size + 1):
            example_patch = example_image[y:y + patch_size, x:x + patch_size]
            score = np.sum((example_patch - target_neighborhood) ** 2)  # 均方误差
            if score < best_score:
                best_score = score
                best_patch = example_patch
    return best_patch
```

图 2 (a) 搜索最匹配的像素块（均方误差）

```python
from scipy.spatial.distance import cosine

def match_patch(target_image, example_image, patch_size, target_pos):
    """在示例图像中匹配目标图像当前位置的邻域（使用余弦相似度）"""
    t_y, t_x = target_pos
    target_neighborhood = target_image[t_y:t_y + patch_size, t_x:t_x + patch_size]
    # 扁平化邻域为向量
    target_vector = target_neighborhood.flatten()
    best_score = float('inf')  # 初始余弦相似度为无穷大（cosine返回的是距离）
    best_patch = None

    h, w = example_image.shape[:2]
    # 由上到下，由左到右遍历示例图像中的小块
    for y in range(h - patch_size + 1):
        for x in range(w - patch_size + 1):
            example_patch = example_image[y:y + patch_size, x:x + patch_size]
            # 扁平化示例图像中的小块为向量
            example_vector = example_patch.flatten()

            # 计算余弦相似度（实际上是计算余弦距离）
            score = cosine(target_vector, example_vector)

            if score < best_score:  # 选择相似度最高的小块（即距离最小）
                best_score = score
                best_patch = example_patch

    return best_patch
```

图 2 (b)搜索最匹配的像素块（余弦相似度）

最后，将上述两个函数集成到整个流程中（函数 texture_synthesis），在函数中，首先进行初始化输出图像为全零矩阵，调用 get_patch 函数从输入图像中选取小块，随后通过收缩最匹配的像素块从而不断填充输出图像，最后得到填充完毕的图像。

```python
def texture_synthesis(example_image, target_shape, patch_size):
    """基于纹理合成的算法，生成目标图像"""
    target_image = np.zeros(target_shape, dtype=np.uint8)  # 初始化目标图像

    # 随机选择一个小方块并放置到目标图像的左上角
    patch, _ = get_patch(example_image, patch_size)
    target_image[:patch_size, :patch_size] = patch

    # 按照边界逐步填充目标图像
    h, w = target_image.shape[:2]
    #遍历图像中每行
    for t_y in range(0, h - patch_size + 1, patch_size):
        #遍历每列
        for t_x in range(0, w - patch_size + 1, patch_size):
            # 如果当前块有未合成的位置
            if np.any(target_image[t_y:t_y + patch_size, t_x:t_x + patch_size] == 0):
                # 在目标图像的当前位置匹配示例图像中的块
                best_patch = match_patch(target_image, example_image, patch_size, (t_y, t_x))
                target_image[t_y:t_y + patch_size, t_x:t_x + patch_size] = best_patch

    return target_image
```

图 3 texture_synthesis 函数

效果分析：代码使用了一个 216*303*3 的图像作为初始输入图像，如图 4 所示，并对比了不同窗口大小、不同窗口形状以及匹配误差度量对最终结果的影响。

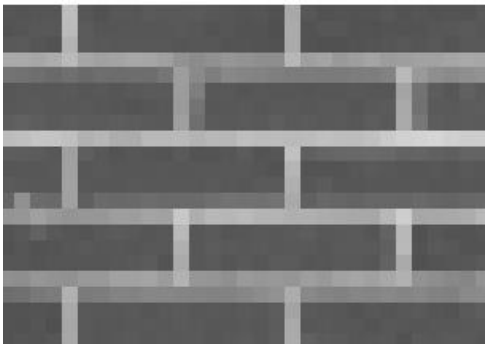


图 4 初始输入图像

在使用相同窗口形状（正方形）与相同误差度量方式时，改变其窗口大小，其生成结果如图 5 所示，图 5 (a)为在窗口大小为 16 时生成的图像，可以看出，生成的图像重复性较高；图 5 (b)为在窗口大小为 128 时生成的图像，此时窗口内包含更多的纹理信息，可以看出，生成的图像不再具有(a)中的高重复，但是仍有噪声。

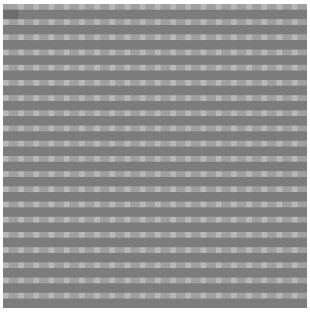


图 5 (a)

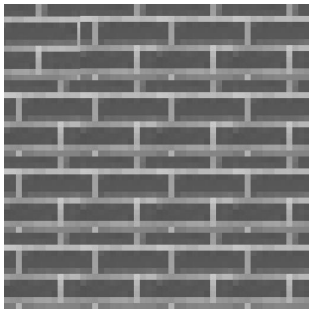


图 5 (b)

在使用不同窗口形状时（正方形和圆形）正方形窗口由于其规则的形状，能够简化匹配过程，并且计算效率较高。它适用于纹理结构较为均匀且无明显方向性或边界的图像。使用正方形窗口时，合成结果通常呈现出较为均匀的分布，但在处理具有复杂纹理或方向性的图像时，可能会出现明显的接缝或不自然的过渡。而圆形窗口更适合于处理具有一定方向性或结构化的纹理。圆形窗口能够更好地适应自然纹理的曲线和不规则性，避免了正方形窗口在合成过程中出现的边界效应。然而，圆形窗口的使用也有其挑战，尤其是在合成过程中对圆形区域的边缘

处理较为复杂，在目前测试中，合成圆形区域时，经常因为图像尺寸的问题导致生成的纹理图像具有圆形的外框，如图 6 所示。

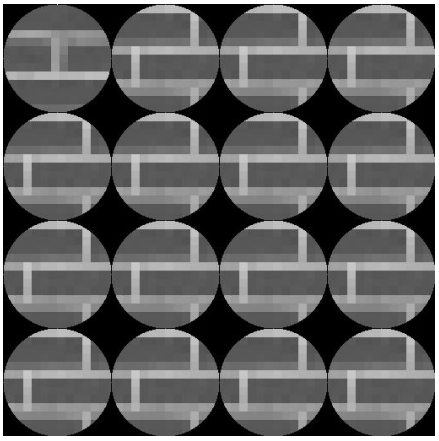


图 6 圆形窗口生成易出现的问题

在使用不同的误差度量方式时，这里采用了余弦相似度和均方误差的方式。均方误差的优点是计算简便，能够衡量像素之间的差异，特别适合用于处理平滑或者纹理变化较小的图像，其生成的图像效果能够较好地保留细节，减少较为明显的纹理断裂。然而，均方误差并不善于处理具有方向性或结构化的纹理，因此在一些复杂的纹理合成任务中可能出现不自然的边界。相反，余弦相似度更适合用来处理具有方向性或结构匹配要求的纹理合成任务。它强调像素之间的相对方向性，因此能够更好地捕捉到纹理的整体结构信息，从而生成更加平滑且无缝的纹理效果。通过对比结果，如图 7(a)(b)所示，可以看到使用余弦相似度的图像相比于均方误差生成的图像效果更佳，余弦相似度能够有效避免纹理之间的隔断和接缝，生成的图像更加自然且一致,但是在生成图像时间方面，余弦相似度所耗费的时间更长。

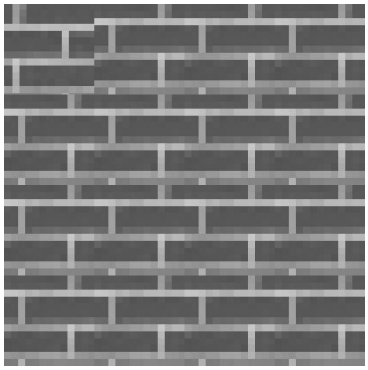


图 7（a）

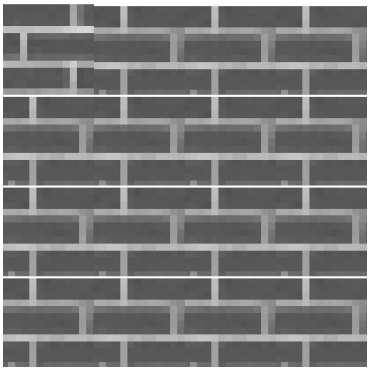


图 7（b）

2. 阅读论文：Texture Synthesis by Non-parametric Sampling，ICCV 1999. Image Quilting for Texture Synthesis and Transfer, SIGGRAPH 2001

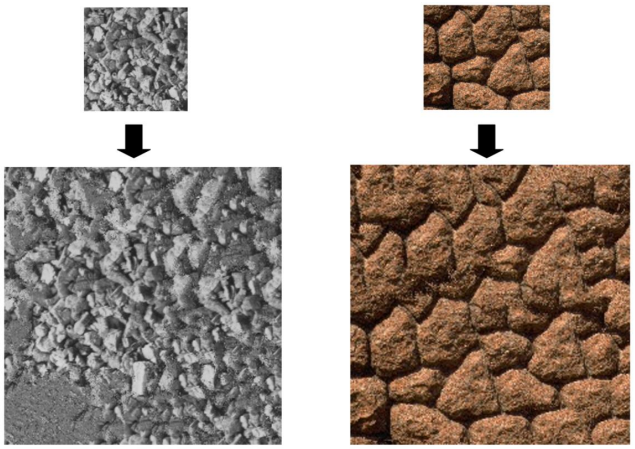
（1）Texture Synthesis by Non-parametric Sampling，ICCV 1999

该文章是由 Efros 和 Leung 在 1999 年发表在 ICCV 上的一篇论文，他们提出了一种基于非参数采样的方法来进行纹理合成，该方法的目标是生成与给定输入纹理相似的合成纹理图像，而不需要明确的统计建模或参数化过程。

该方法的核心思想是利用已有纹理中的信息进行局部区域的样本选择和拼接，通过不断从输入纹理图像中复制小的样本快，并根据相似度将它们拼接到生成图像中。这种方式避免了传统纹理合成方法中需要建模纹理的复杂统计特性（如 Gabor 滤波器或 Markov 随机场等）。其优点是简单且效果自然，能够较好地保留输入纹理的局部结构特征，但是其运行较慢。

作者将纹理建模为 MRF，像素的邻域被建模为该像素周围的方形窗口，并且在已知邻域像素时可以通过相似搜索和条件概率分布估计来合成图像的像素，但这种方法不能用于合成整个纹理，甚至不能用于填充洞（除非洞只是一个像素），因为对于任何像素，只有一些邻近像素的值是已知的。因此，作者提出了一种来自香农信息理论的启发式方法，该方法通过从样本图像中随机选择一个 3x3 的种子块并逐层扩展生成纹理，从而能够在优先已知信息的基础上生成合理的像素值。

但是这种方法也有不足，该算法目前只能处理正对平行的纹理，但可以通过形状从纹理技术来预处理图像，使其变为正对平行的姿态，然后在合成后进行后处理恢复原始形状。算法的一个问题是，当纹理样本中包含太多不同类型的纹理元素或相同的纹理元素在不同光照下变化时，可能会导致纹理合成“滑移”到错误的搜索空间，生成无意义的内容，或陷入样本图像的某个特定区域，导致合成结果是原始纹理的直接复制如下图。



（2）Image Quilting for Texture Synthesis and Transfer, SIGGRAPH 2001

该论文同样是由 Efros 发表，他们提出了一种简单的基于图像的方法来生成新的视觉外观，主要思想是通过将现有图像的小块拼接在一起从而合成新的图像同时也提出了一种用于纹理传输的推广方法，即将某个纹理应用到另一个图像上，使其虽然展示的颜色不同，但是纹理能够看出是前一个图像的纹理。

首先是基于块的纹理合成方法，合成的基本单元被定义为来自输入纹理图像中所有重叠块的集合中的一个用户指定大小的方块。为了合成新的纹理图像，首先，作者将图像填充为从中随机选取的块。这种初步的合成结果看起来相对合理，并且对于某些纹理，它的效果可能与许多复杂的算法相当。然而，这种方法的结果并不令人满意，因为无论如何平滑块之间的接缝，尤其对于结构化纹理，合成图像中的块之间的边界通常是显而易见的。

接下来的步骤是引入块的重叠放置。不同于随机选取块，作者提出通过某种度量标准在中寻找一个块，使得新放置的块与相邻块之间的匹配度最大化。通过这种方式，合成的纹理能够减少块之间明显的接缝或不匹配，从而使图像看起来更加连续和平滑，即最小误差边界切割法和图像拼接算法，其思想是在两个图像中找到一个最小代价路径（与第二章类似，主要是通过动态规划的方式进行求解），当最小代价路径找到后，将两个图像按照该路径进行切割并拼接到一起，从而实现最佳切割路径，从而确保了重叠区域的误差最小化，从而让拼接的图像的连接处的纹理更加平滑自然（和上课讲的方法相同，所以此处运用了上课的 ppt），这种思想如图 8 所示.

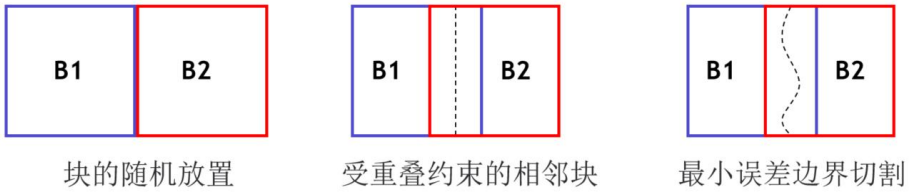


图 8 最佳切割路径

这种引入重叠和块选择策略的方法，能够更有效地处理具有结构化特征的纹理，使得合成图像在局部上下文中更加一致。与纯粹的随机拼接方法相比，这种基于块的合成方法能够更好地处理纹理之间的连续性和一致性，显著提高了纹理合成的质量。

随后在此基础上，作者提出了纹理传输的方法，这种方法下合成的图像不仅需要遵守源纹理的局部结构，还需要遵守图像之间的对应关系。为了实现这一点，作者修改了图像拼接算法中的误差项，将重叠匹配误差和源纹理块与目标图像位置之间的对应关系的误差加权求和。通过这种加权，能够在纹理合成与目标图像对应关系之间进行权衡。权重参数 决定了这两者之间的平衡。

由于增加了这个额外的约束，单次合成可能无法得到令人满意的结果。因此，作者提出了迭代的方法，在每次迭代中减少块的大小。与非迭代版本的唯一区别是，在满足局部纹理约束时，不仅要与重叠区域的相邻块匹配，还要与之前迭代中已合成的部分进行匹配。如图 8，使用这种方法，将人物的图像“映射”到另一个背景下，从而实现纹理传输。同时我们可以发现，最终的结果具有黑边，在实际应用的过程中，这种黑边现象通常是合成的图像与合成前预定好的图像框架不匹配造成的，即黑边的像素全是框架初始化时的“0”像素。

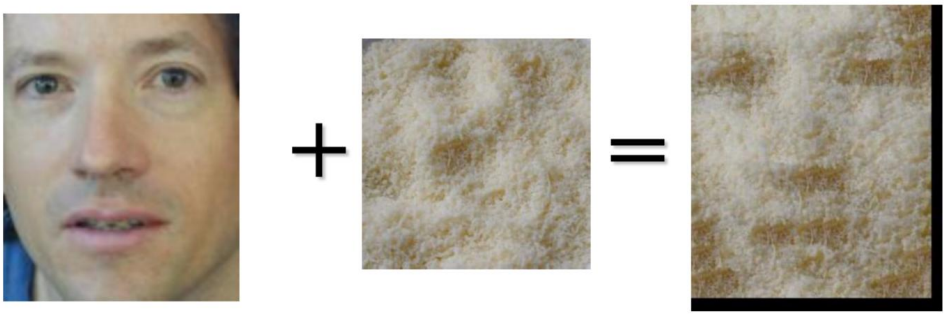


图 9 纹理传输

这种基于块的纹理合成和纹理传输技术通过最小化重叠区域的误差并采用最小成本路径切割（如动态规划），能够高效地生成无缝的纹理图像。这项技术能够应用于计算机图形学、游戏开发、图像修复、电影特效、虚拟现实等领域，尤其适用于需要处理复杂表面纹理的场景。通过优化块之间的接缝匹配和迭代合成还可用于艺术创作、风格迁移以及医学影像和文化遗产保护等领域，提供精确且自然的纹理合成与图像修复方案。