

专栏首页 未竟东方白 【图形学】形态抗锯齿MLAA详解与Python实现

【图形学】形态抗锯齿MLAA详解与Python实现

发布于2021-08-20 11:22:25 阅读 1.4K

这几天正好有需求复现了一下MLAA算法用来处理图像，复现完就写了这份笔记，内容不难度但是流程比较繁琐。本篇5k字，其中有些地方的实现可能不太标准，才疏学浅，错漏也在所难免。本文同步存于我的Github仓库，有错误会在那里更新。
(<https://github.com/ZFhuang/Study-Notes/blob/main/Content/%E5%9B%BE%E5%BD%A2%E5%AD%A6/%E5%BD%A2%E6%80%81%E6%8A%97%E9%94%AF%E9%BD%BFMLAA%E4%B8%8EPython%E5%AE%9E%E7%8E%B0/README.md>)

- 形态抗锯齿MLAA的详解与Python实现
 - 总览
 - 参考资料
 - 流程概览
 - 细节
 - 查找边缘
 - 模式分类
 - 重新矢量化
 - 计算权重
 - 混合颜色
 - 结果

总览

Morphological Antialiasing (MLAA), 中文一般翻译为形态抗锯齿(反走样), 是一种常见的抗锯齿算法。其于2009年由Intel的Alexander Reshetov提出, 启发了后续一批基于图像自身形态进行抗锯齿操作的算法例如FXAA和CMAA。相比传统的基于超采样的抗锯齿算法, MLAA是一种纯粹的后处理算法, 无须法线和深度等信息就可以直接对渲染器的帧缓冲进行抗锯齿处理, 因此这类方法由于即插即得的易用性而得到广泛的应用。

MLAA的思路基于人眼感知的一大特征: 对形状失真的敏感性远强于对颜色失真的敏感性。因此类似MSAA的想法, MLAA通过一定的策略插值将失真强烈的几何边缘进行模糊, 又保留平滑部分不进行处理, 一方面防止了纹理部分出现额外的失真, 另一方面大大减少了计算量。

而形态抗锯齿的核心是"形态"部分。MLAA先在图片中找到代表几何边缘的部分, 然后将这些边缘分为多种不同的形态模式(pattern), 根据模式实施不同的模糊策略, 这个过程本质上是对边缘重新矢量化和再光栅化的过程。经过MLAA处理的图片如下图边缘较为平滑, 而内部纹理保持原样, 有效减少了图片失真又不至于产生过多的模糊。

目录

总览

流程概览

作者介绍

细节

查找边缘

模式分类

重新矢量化

计算权重 ZifengHuang

混合颜色

结果

关注

专栏

文章	阅读量	获赞	作者排名
112	56.6K	259	2966

精选专题

腾讯云原生专题
云原生技术干货，业务实践落地。

活动推荐

云安全最佳实践-创作...
火热征文中，发布文章赢千元好礼！

立即查看

腾讯云自媒体分享计划
入驻腾讯云开发者社区，共享百万资源包。

立即入驻

运营活动





流程概览

MLAA分为下面五大步骤:

1. 查找图片中明显的像素不连续区域作为需要处理的边缘
2. 将这些边缘分类为不同的模式(pattern)
3. 重新矢量化图像的边缘
4. 按照矢量化后的边缘计算用于颜色混合的权重
5. 将像素与周围像素进行按照权重进行混合得到平滑后的结果

参考资料

这里参考的核心文章是Reshetov的原始论文"Morphological Antialiasing"和Jimenez一年后发表的"Practical Morphological Anti-Aliasing". 两者的区别在于Reshetov的MLAA是在CPU上实现的, 目的是优化光线追踪渲染的图像, 计算量比较大, 而Jimenez针对光栅化渲染, 以牺牲一部分效果为代价在GPU上以极低的计算量实现了MLAA, 将MLAA的实用性提升了一大截.

这里我的Python实现综合了上面两篇文章. 主体仍然是Reshetov的实现方式, 但使用Jimenez的实现中利用图像来储存临时数据的思路辅助. 此文章的代码仓库的路径如下. 文章为了简洁采用的是提炼的部分代码作为伪代码辅助介绍:

<https://github.com/ZFhuang/MLAA-python>

下面是一些可供查阅的辅助资料:

Intel的MLAA主页
<https://software.intel.com/content/www/cn/zh/develop/articles/morphological-antialiasing-mlaa-sample.html>

2009到2017形态抗锯齿系列算法的发展
<http://www.iryoku.com/research-impact-retrospective-mlaa-from-2009-to-2017>

Jimenez实现的MLAA的项目主页
<http://www.iryoku.com/mlaa/>

从零开始的游戏引擎编写之路: 形态学抗锯齿
<https://www.bilibili.com/read/cv2269091>

图形学基础 - 着色 - 空间抗锯齿技术
<https://zhuanlan.zhihu.com/p/363624370>

细节

查找边缘

抗锯齿技术处理的目标是图像中边缘部分的锯齿状走样. MLAA首先需要查找出图像中的边缘信息. 在MLAA中, 图像边缘信息的查找相对单个通道进行的, 因此对于彩色图像来说, 需要通过某个方法将其转为单通道形式. 常用的方法是逐通道计算和转为灰度图再计算, 由于常见的图像三个通道的信息可能有很大差异, 因此将彩色图像转为灰度图像后再进行边缘查找是比较合适的算法. 对于图形学渲染得到的图像则还可以采用场景的深度图配合法线图来计算边缘, Jimenez论文中提到使用转换的灰度图效果最好, 深度图执

目录

总览

流程概览

参考资料

细节

查找边缘

模式分类

重新矢量化

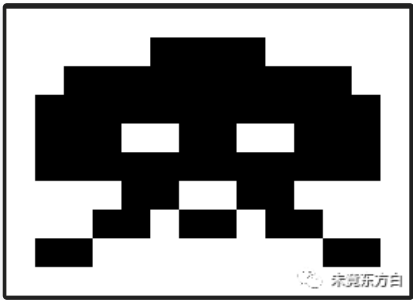
计算权重

混合颜色

结果

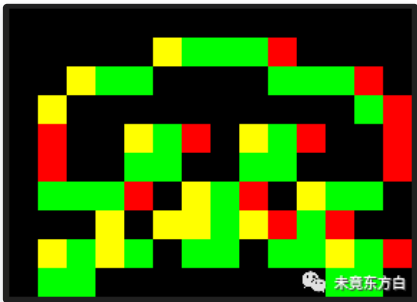
$$L = 0.2126 * R + 0.7152 * G + 0.0722 * B$$

下面是与Jimenez论文中的测试样例相同的太空侵略者的经典敌人，一张外星人单通道点阵图。本文后续的流程展示皆以此图为例。由于实现稍有不同所以后面展示的中间结果会有些许差别，但最终结果是一样的。



得到单通道图像后就是查找边缘的步骤了。MLAA将图像的边缘划分为两种：横向边缘和纵向边缘。遍历图像中每个像素，将当前像素与左边和上边相邻的像素做差对比，当差别大于某个阈值th时认为此像素覆盖边缘。Jimenez的论文中提到对于颜色域是[0,1]的图像来说，th=0.1是比较实用的选择。

对于查找边缘阶段，可以用一个初始全为0的三通道图片保存边缘信息。当出现差别的像素处于当前像素左侧时，我们认为边缘在两个像素相邻的那条边也就是左侧边，将图片的R通道设置为1；当出现差别的像素处于当前像素上方时，边缘处于当前像素上侧，将图片的G通道设置为1。一个像素可能同时存在两个边缘，完成边缘查找阶段后得到的边缘信息图会是由如下红绿黄三色构成的：



代码如下：

```
1 def _find_edges(img, th=0.1):
2     buffer = np.zeros((img.shape[0], img.shape[1], 3))
3     for y in range(1, img.shape[0]):
4         for x in range(0, img.shape[1]):
5             if abs(img[y, x]-img[y-1, x]) > th:
6                 buffer[y, x, 1] = 1
7     for y in range(0, img.shape[0]):
8         for x in range(1, img.shape[1]):
9             if abs(img[y, x]-img[y, x-1]) > th:
10                 buffer[y, x, 0] = 1
11     return buffer
```

模式分类

得到图片边缘之后，MLAA论文中将边缘视作走样并分为三个模式：L型，Z型，U型。下面是Reshetov对这三种模式给出的示意图：

目录

总览

流程概览

参考资料

细节

查找边缘

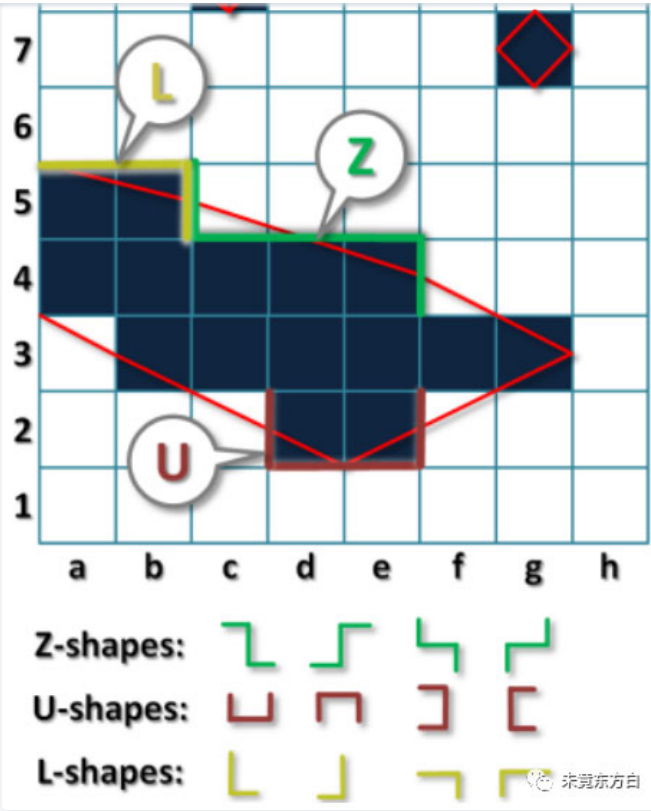
模式分类

重新矢量化

计算权重

混合颜色

结果



目录

总览

流程概览

参考资料

细节

[查找边缘](#)

模式分类

重新矢量化

计算权重

混合颜色

结果

但想要用程序直接寻找这三种模式是比较困难的, 所以这里我们对模式搜索算法进行优化, 将所有模式都转为长边与短边的组合. 注意到这些走样模式都是由长度为1的一到两条短边与长度未知的一条长边组成, 所有的模式都需要长边的存在, 因此我们将长边的出现视作模式搜索的起点, 当模式遇到短边或达到尽头时模式结束, 所以将短边或空像素视作模式搜索的终点, 从而将所有模式转换为两个子模式的组合.

然后首先将模式搜索分为基于X和基于Y两种搜索顺序, 以X优先搜索为例, 当遍历发现G通道的值为1时, 也就是当前像素上方有横边存在, 认为遇见了走样, 判断上面相邻像素和自身像素的R通道是否有1存在. 若上方相邻像素R通道为1, 此走样的前半段定为B型, 表示长边在短边下方, 若当前像素R通道为1, 此走样前半段定为T型, 表示长边在短边上, 若当前和上方像素R通道都为1, 定为H型, 表示长边的上下都有短边, 若R通道都为0, 此走样前半段定为L型, 表示形如原论文的L走样, 即一侧缺少短边.

完成了前半段的搜索后就开始后半段的搜索, 关注点在于计算出走样的长度和后半段的走样模式. 当遍历途中的像素或上方像素的R通道为1时, 表示这段走样来到了终点, 记录下走样所经过的像素数量就是走样的长度, 然后用和起点处相同的判断模式判断出终点处的走样属于TBHL四个模式中的某一个, 记录下来.

熟悉了这个流程后再看下面的两种典型走样情况, 第一个走样是原论文的Z型走样, 经过上面的拆解变为了TB型走样, 第二个走样是原论文的L型走样, 经过拆解变为了LB型走样. 图的下面是对应搜索的代码, 基于X优先遍历搜索完走样模式后, 再以类似的方法按Y搜索一次走样模式, 保存在一个列表里即可.

```
1 def _find_aliasings_x(img_edges):
2     list_aliasings = []
3     mask = np.zeros((img_edges.shape[0], img_edges.shape[1], 1))
4     for y in range(1, img_edges.shape[0]):
5         for x in range(0, img_edges.shape[1]):
6             if mask[y, x] == 0:
7                 if img_edges[y, x, 1] == 1:
8                     if img_edges[y, x, 0] == 1 and img_edges[y-1, x, 0] == 1:
9                         start_pattern = 'H'
10                    elif img_edges[y, x, 0] == 1:
11                        start_pattern = 'T'
12                    elif img_edges[y-1, x, 0] == 1:
13                        start_pattern = 'B'
14                    else:
15                        start_pattern = 'L'
```

```
18         [y, x, dis, start_pattern+end_pattern])
19     return list_aliases
20
21
22 def _cal_aliasing_info_x(img_edges, start_x, start_y, mask):
23     dis = 1
24     for x in range(start_x, img_edges.shape[1]):
25         if img_edges[start_y, x, 0] == 1 and img_edges[start_y-
26             pattern = 'H'
27             return dis, pattern, mask
28         if img_edges[start_y, x, 0] == 1:
29             pattern = 'T'
30             return dis, pattern, mask
31         if img_edges[start_y-1, x, 0] == 1:
32             pattern = 'B'
33             return dis, pattern, mask
34         if img_edges[start_y, x, 1] == 0:
35             break
36         mask[start_y, x] = 1
37         dis+=1
38     pattern = 'L'
39     return dis, pattern, mask
```

重新矢量化

完成对整张图走样模式的分类后, 我们就得到了两个列表分别保存了X方向的走样和Y方向的走样. 随后我们需要按照这些走样计算出每个像素用于与周围像素值混合的权重. 在计算权重之前, 开头的时候我们说到MLAA是在对图像的边缘进行重新矢量化以计算混合颜色的权重. 重新矢量化实际上就是在依据查找到的走样来估计真实的边缘.

对于每个走样, 我们取短边的1/2位置所为一侧的端点, 若两个短边属于同一侧, 则属于原论文的U型走样. 计算长边中点为分界点, 连接三点得到U型矢量, 若两个短边方向相反, 则属于原论文的Z型走样, 直接连接两个端点, 若短边长度为0, 则属于原论文的L型走样, 直接采用为0的一侧的顶点作为端点连接, 当出现两个短边也就是一侧是H型走样时, 为了图像边缘的平滑我们优先判定为Z型走样.

根据上面的步骤我们可以将X方向所有模式的走样重新矢量化, 下图是所有模式矢量化后的结果与对应的部分代码.

```
1 def _analyse_pattern(pattern):
2     if pattern[0] == 'H':
3         if pattern[1] == 'H':
4             start = 0
5             end = 0
6         elif pattern[1] == 'T':
7             start = -0.5
8             end = 0.5
9         elif pattern[1] == 'B':
10            start = 0.5
11            end = -0.5
12        elif pattern[1] == 'L':
13            start = 0
14            end = 0
15    elif pattern[0] == 'T':
16        ...
17    elif pattern[0] == 'B':
18        ...
19    elif pattern[0] == 'L':
20        ...
21    return start, end
```

计算权重

得到重新矢量化后的边缘后, 关键就是计算用于混合颜色的权重信息. 首先观察下面的Z型走样, 显然(2,1)处的像素被重建的边缘切分为两部分, 因此我们想到新的(2,1)像素的颜

目录

总览

- 流程概览
- 参考资料

细节

- 查找边缘
- 模式分类
- 重新矢量化
- 计算权重
- 混合颜色

结果



再观察像素(2,2), 我们可以看到此时像素的边缘正好被重建的边缘线经过, 因此两个像素处于相互混合的状态, 为了处理这种特殊的状态, 我们为每个像素都保存两个值, 一个储存指向上方的三角形面积, 代表了当前像素影响外部像素的溢出面积, 一个储存指向下方的三角形面积, 代表当前像素被外部像素影响的侵入面积. 对于这两个三角形面积的计算我采用了相似三角形的面积公式来处理, 面积比=相似比的平方, 整体计算比较繁琐且L型和Z型U型的计算差别较大, 下面的代码只展现了最通用的非偶数边长Z型和U型计算过程:

```
1 def _cal_area_list(dis, pattern):
2     start, end = _analyse_pattern(pattern)
3
4     if start == 0 and end == 0:
5         return None
6     elif end == 0:
7         h = start
8         tri_len = dis
9     elif start == 0:
10        h = end
11        tri_len = dis
12    else:
13        h = start
14        tri_len = dis/2.0
15
16    list_area = np.zeros((dis, 2))
17    tri_area = abs(h)*tri_len/2
18
19    if start==0:
20        ...
21    elif end==0:
22        ...
23    elif tri_len % 2 == 0:
24        ...
25    else:
26        for i in range(0, dis+1):
27            if abs(i-tri_len) <= 0.5:
28                if i < tri_len:
29                    area = (start*2)*(tri_area*(((tri_len-i)/tri_len)**2))
30                    if area > 0:
31                        list_area[i, 0] += area
32                    else:
33                        list_area[i, 1] -= area
34                else:
35                    area = (end*2)*(tri_area*(((i-tri_len)/tri_len)**2))
36                    if area > 0:
37                        list_area[i-1, 0] += area
38                    else:
39                        list_area[i-1, 1] -= area
40            elif i < tri_len:
41                area = (start*2)*(tri_area*(((tri_len-i)/tri_len)**2))
42                if area > 0:
43                    list_area[i, 0] = area
44                else:
45                    list_area[i, 1] = -area
46            elif i > tri_len:
47                area = (end*2)*(tri_area*(((i-tri_len)/tri_len)**2))
48                tri_area*(((i-tri_len-1)/tri_len)**2)
49                if area > 0:
50                    list_area[i-1, 0] = area
51                else:
52                    list_area[i-1, 1] = -area
53    return list_area
```

计算得到的面积权重可以分别用图像的一个通道来保存, 也就是保存为RGBA的四通道图像. 根据这个步骤计算后保存下来的权重图如下, 这张图和论文中的范例有些许差别可

目录

总览

流程概览

参考资料

细节

[查找边缘](#)

模式分类

重新矢量化

计算权重

混合颜色

结果

行限制的话, 就可以将每个模式下对应的面积权重提前计算出来并保存在一张四通道图片中, 这样在计算面积权重的时候就可以通过查表直接免去重复的面积计算, 这是Jimenez对MLAA的一大性能优化. 下面是将面积预计算后保存在一张图中的形式.

混合颜色

得到每个像素用于混合颜色的面积权重后, MLAA就只剩下最后一步了. 对于每个像素, 按照计算好的X方向权重和Y方向权重, 与周边的颜色进行对应的加权平均. 混合公式结果如下, 可以看到边缘原先十分锐利的像素得到了有效的模糊, 且模糊后的图像整体形状没有太大的改变. 后面是实际使用时的代码, 为了处理图像边缘时算法也能正常工作而做出了一些优化.

$$C_{newX} = (1 - W_{topout} - W_{oldin}) * C_{old} + W_{topout} * C_{top} + W_{oldin} * C_{down}$$
$$C_{newY} = (1 - W_{rightout} - W_{oldin}) * C_{old} + W_{rightout} * C_{right} + W_{oldin} * C_{left}$$
$$C_{new} = (C_{newX} + C_{newY})/2$$

```
1 def _blend_color(img_in, img_weight):
2     img_blended= np.zeros((img_in.shape[0],img_in.shape[1]))
3     for y in range(0, img_in.shape[0]):
4         for x in range(0, img_in.shape[1]):
5             img_blended[y, x]=(2-img_weight[y,x,0]-img_weight[
6                 if y!=0:
7                     img_blended[y, x]+=img_in[y-1,x]*img_weight[y,
8                 if y!=img_in.shape[0]-1:
9                     img_blended[y, x]+=(img_in[y+1,x]-img_in[y,x])*
10                if x!=0:
11                    img_blended[y, x]+=img_in[y,x-1]*img_weight[y,
12                if x!=img_in.shape[1]-1:
13                    img_blended[y, x]+=(img_in[y,x+1]-img_in[y,x])*
14                img_blended[y, x]/=2
15     return img_blended
```


结果

在Jimenez的实现中, MLAA中除了预计算的面积索引图外还借助显卡硬件加速的线性插值特性在边缘图中简化了模式分类的步骤, 具体的优化流程在这就不详细介绍了. 高度优化后MLAA的执行效率远高于效果相近倍率下的MSAA, 有很强的实用性.

需要注意到MLAA仅仅是形态抗锯齿系列算法的开创者, 其仍然存在非常多的缺点: 例如剧烈变换的场景下容易产生鬼影现象, 对走样边缘的判断只有一个像素也不够准确, 抗锯齿后文字容易模糊等. 后续人们在MLAA的基础上优化开发了例如SMAA, FXAA, CMAA等更实用的抗锯齿算法, 以后有机会再介绍.

下图是Jimenez论文中给出的结果, 可以看到MLAA能够达到和MSAA接近的抗锯齿效果.

文章分享自微信公众号:



未竟东方白

复制公众号名称

本文参与 腾讯云自媒体分享计划 , 欢迎热爱写作的你一起参与!

作者: 研究中

原始发表时间: 2021-08-11

如有侵权, 请联系 cloudcommunity@tencent.com 删除。

登录后参与评论

2 条评论

最新 高赞

- 用户5927379

2022-08-18

你好, 你的代码怎么调用测试呢

0 回复
- ZifengHuang 回复 用户5927379

2022-08-18

您好, 代码在仓库<https://github.com/ZFHuang/MLAA-python>中, 您有什么问题可以提issue

0 回复

目录

总览

流程概览

参考资料

细节

[查找边缘](#)

模式分类

重新矢量化

计算权重

混合颜色

结果

相关文章

深度详解 Python yield与实现

学Python最简单的方法是什么？推荐阅读：Python开发工程师成长魔法 Python yield与实现 yield的功能类似于return，...

小小科

3D 图形学基础 （上）

本文主要针对一些对3D有兴趣的同学，普及图形学知识，不涉及深入的技术探讨和样例介绍。对于不是从事相关开发...

serena

python实现与redis交互操作详解

本文实例讲述了python实现与redis交互操作。分享给大家供大家参考，具体如下：

砸漏

第5章-着色基础-5.4-锯齿和抗锯齿

想象一个大的黑色三角形在白色背景上缓慢移动。当一个屏幕网格单元被三角形覆盖时，代表这个单元的像素值应该...

charlee44

详解：Python代码实现强密码判断与生成

如今，用户在网络上越来越重视个人隐私和信息安全，抛开服务提供商的问题，我们用户端，设置一个好的用户名和...

Mintimate

本。

昱良

支持向量机（SVM）入门详解（续）与pytho...

接前文 支持向量机SVM入门详解：那些你需要消化的知识
让我再一次比较完整的重复一下我们要解决的问题：我们...

机器学习AI算法工程

排序算法 | 双调排序(Bitonic sort)详解与Python实现

上篇提到的珠排序（排序算法 | 珠排序(bead sort)详解与Python实现）是一种自然排序方法，本文介绍的双调排序则属于排序网络（sort net）的一...

昱良

详解 | 神经网络中BP算法的原理与Python实现

作者 | EdvardHua 最近这段时间系统性的学习了BP算法后写
下了这篇学习笔记。目录 什么是梯度下降和链式求导法则...

AI科技大本营

Python之Metaclass元类详解与实战:50行代...

由于工作需要，最近学习Python元编程方面的东西。本文介
绍了Metaclass的一个示例，是笔者在学习过程中编写的一...

用户5410317

[Python图像处理] 十.形态学之图像顶帽运算...

该系列文章是讲解Python OpenCV图像处理知识，前期主要
讲解图像入门、OpenCV基础用法，中期讲解图像处理的...

Eastmount

何恺明团队又出神作：将图像分割视作渲染问题，性能显著提升！

Facebook人工智能实验室Alexander Kirillov、吴育昕、何恺明、Ross Girshick等研究
人员近日发表新论文，提出一种高效、高质量的目...

新智元

Ross、何恺明等人提出PointRend：渲染思路做图像分割，显著...

实例分割是计算机视觉任务中一个重要的任务。传统的示例分割方法输入图像，并对图
像的每个像素点进行预测，推断像素点所属的实例标签，并区分属于不同实例的像素...

机器之心

（数据科学学习手札34）多层感知机原理详...

机器学习分为很多个领域，其中的连接主义指的就是以
神经元（neuron）为基本结构的各式各样的神经网络，规...

Feffery

目录

总览

流程概览

参考资料

细节

查找边缘

模式分类

重新矢量化

计算权重

混合颜色

结果

KNN (k-nearest neighbors, KNN) 作为机器学习算法中的一种非常基本的算法, 也正是因为其原理简单, 被广泛...

Feffery

[Python图像处理] 九.形态学之图像开运算、...

该系列文章是讲解Python OpenCV图像处理知识, 前期主要讲解图像入门、OpenCV基础用法, 中期讲解图像处理的...

Eastmount

(数据科学学习手札23) 决策树分类原理详...

作为机器学习中可解释性非常好的一种算法, 决策树 (Decision Tree)是在已知各种情况发生概率的基础上, 通...

Feffery

(数据科学学习手札24) 逻辑回归分类器原...

一、简介 逻辑回归 (Logistic Regression) , 与它的名字恰恰相反, 它是一个分类器而非回归方法, 在一些文献...

Feffery

[Python图像处理] 八.图像腐蚀与图像膨胀

该系列文章是讲解Python OpenCV图像处理知识, 前期主要讲解图像入门、OpenCV基础用法, 中期讲解图像处理的...

Eastmount

更多文章

目录

总览

流程概览

参考资料

细节

查找边缘

模式分类

重新矢量化

计算权重

混合颜色

结果

社区

活动

资源

关于

腾讯云开发者

专栏文章

阅读清单

互动问答

技术沙龙

技术视频

团队主页

腾讯云TI平台

自媒体分享计划

邀请作者入驻

自荐上首页

技术竞赛

技术周刊

社区标签

开发者手册

开发者实验室


视频介绍

社区规范

免责声明

联系我们

友情链接



扫码关注腾讯云开发者
领取腾讯云代金券

热门产品

域名注册

云服务器

区块链服务

消息队列

网络加速

云数据库

域名解析

云存储

视频直播

热门推荐

人脸识别

腾讯会议

企业云

CDN 加速

视频通话

图像分析

MySQL 数据库

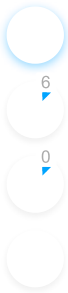
SSL 证书

语音识别



目录

Copyright © 2013 - 2022 Tencent Cloud. All Rights Reserved. 腾讯云 版权所有 京公网安备 11010802017518 粤B2-20090059-1



总览

流程概览

参考资料

细节

[查找边缘](#)

模式分类

重新矢量化

计算权重

混合颜色

结果