iAnime - 移动端简易线稿上色软件

ios大作业开题报告

林晟熠 201764683069 郑泽明 201730685141
郑炯豪 201730683444 杨泽宇 201730684175

I. 摘要

在这份开题报告中，我们将介绍开发、分析这样一款移动端简易线稿上色软件的动机，并且举出这款软件将要解决的现实生活问题，以及它在生活中的应用场景。除此之外，我们还会介绍软件的结构——包括ios移动端应用程序和后端服务程序在代码层面的结构和设计层面的架构。

II. 软件立项背景

考虑画师完成一幅作品的全过程，首先画师需要在纸上用各种铅笔勾勒出整幅作品的大致布局，例如大概在哪块位置会画一个人，哪些位置画风景，哪些位置画上一些点缀画面的元素等；接下来，画师需要回顾刚才勾勒出来的线条，修正一些起草时不太注意的冗余、混乱的线条，保留一些正确的、参与最终成稿的线条，并把它们加深；然后，画师对整幅画中需要上色的地方用简单的色块涂在对应的位置。由于不同的颜色可能会非常明显地改变作品的风格和协调性，确定色块组合的流程将会花上很长的时间，甚至不亚于一开始的起草和细化线条；等到确定了整幅画各个主上色区域的颜色、色块形状之后，画师最终还要非常细致的处理这些色块，添加必要的阴影、处理色块之间的颜色过渡，修补上一阶段因为上色而变得模糊的轮廓线条，最终才能完成一幅作品。

随着深度学习这一学科的不断发展，各种不同的神经网络被开发出来并加以利用，这些神经网络进行充分的训练之后可以帮助我们通过计算机完成很多以往必须人工手动完成的任务，其中就包括对线稿上色。前段时间，苏州大学、香港中文大学的研究团队发表了一篇《Two-stage Sketch Colorization》论文在ACM Transaction on Graphics上。该文章论述了如何使用卷积神经网络实现识别线稿边界、通过学习各种各样已经完成上色的线稿及其原稿，训练了一组可以根据线稿线条围成的形状自动推测当前区域（例如脸部、手脚，衣服边界）应该着何种颜色的神经网络，最后他们把算法以Python代码的形式实现了出来，放在了Github上。（Github仓库名: styles2paint)

我们的团队成员阅读了论文，并且下载了Github仓库代码，导入了一些线稿运行，发现他们的实现效果确实是可圈可点。同时，团队调查了身边一批擅长绘画，尤其是擅长绘画动漫人物的同学，他们表示在整个绘画作品的创作阶段中，最麻烦的就是确定这幅画的主色调，也就是这幅画主要应该上色的区域应该上何种颜色。有些画面效果并不是画好一小块就能完全体现出来，必须把相当大一块绘画区域完成上色之后才能从整体的角度看出这一块的颜色是不是协调。为此他们经常需要不断的涂涂改改，进行非常多次的尝试。他们也迫切的希望能够有一款软件能够按照他们的意愿，方便智能地为一些已标定的颜色区域进行初步上色，让他们能够在更短的时间内看到“最终效果”。

还有一些擅长绘画的同学则认为，有时候他们闲来无事会在草稿纸、本子的一些小角落上画上几笔，这些小块区域是比较难进行后续的上色、细化画面布局处理的。而对于有一些他们自己觉得可以接着画下去的作品，他们希望能够借助手机拍照或者是某种扫描技术把画在纸上的作品转换到电脑、手机上继续进行作画。

显然，styles2paint库能够非常好地满足他们上述需求。但是，团队成员在运行他们研究团队给出的代码时，发现他们为了演示算法的成果，做了一个简单的网页让用户可以按照他们的要求在线稿上打上一些“颜色锚点”和“颜色标记点“来控制他们的卷积神经网络更加精细地为线稿上色。由于这个工具主要的存在价值还是验证他们的学术成果，而并非方便普罗大众给他们自己的线稿上色，因此该工具的界面易用性不敢恭维，并且也没有一些类似于使用教程的文档帮助用户熟悉他们的工具。我们的团队成员当时花了一个上午进行各种各样的摸索才搞懂了他们提供的工具是如何使用的；不仅如此，工具内部的核心线稿处理逻辑也并没有为投入生产而进行更多的优化，性能较差且非常消耗硬件资源。

我们在一块具有4GB显存的GTX 1650显卡上运行后端线稿处理程序，程序一下子就吃了3.5GB的显存。最多只能支持两个用户同时处理线稿。我们成员在项目立项阶段有稍微深究性能问题的原因，是因为程序一下子把它们6个预训练的模型全部读进了显存。但是他们的代码实际上是可以判断当前传入的图片最适合用于处理的神经网络模型，因此对于显存紧张的机型，实际上是可以先把预训练网络读到内存，在真正需要的时候才把网络读进显存。除此之外，对于这些需要高算力的应用，我们可以借鉴分布式系统的架构，以及边缘计算的一些核心思想，将这些核心处理节点部署到多台机器上，再用一个负载均衡网关配合消息队列的架构实现和分布式节点之间的通信，采取异步的思想来提升性能，改善用户体验。有关我们对当前后端服务架构的改进将会在本报告的后面部分阐述。

综上所述，我们团队成员从当前已达成的技术储备出发，看到了这款软件的应用前景：软件能够辅助广大画师朋友完成他们原本耗时的主色调确定工作，减轻他们的脑力负担，帮助他们更好地产出；同时对于富有创意灵感的手绘者，能够更快地验证他们自己的创意和灵感，进一步激发他们创作的激情。有鉴于此，我们决定开发此软件，并借此大作业的机会分析、反省、提高我们这款软件各方面的设计架构，力图在做好各项软件项目架构指标的前提下实现创新、推广。

III. 软件功能介绍

接下来分点介绍此软件的主要功能，同时着重讲解线稿上色模块以及软件内置的社交模块。（由于时间仓促和初创规模有限，软件初版的社交功能可能达不到目前全球知名绘画交流社区pixiv的水平。但是我们后期会不断提高完善，争取赶上甚至超越pixiv，例如结合机器学习分析画风，通过画风推荐匹配用户，探索出一条新型绘画社交的路线。）

软件主要覆盖的移动平台:iOS

软件主要功能:

- 用户账户功能
    - 使用手机号注册/登陆/找回密码
    - 用户个人资料查看与编辑（昵称，性别，出生年月，个性签名，主页背景，绘画能力*）
    - 关注/取消关注某用户、查看自己关注的用户和关注自己的用户
    - 接收关注用户的动态推送提醒（如发表新作）
- 作品相关功能
    - 发布作品
        - 设置当前作品的标签（作者账号在后续可以在作品详情页面修改，下同）
        - 设置当前作品的标题
        - 设置当前作品的描述
        - 设置是否允许二次创作（相当于Fork到另外一个人的作品库中），是否允许保存到本地
        - 设置作品的可见性（仅自己可见、仅关注自己的用户可见、全部用户可见）
    - 喜欢/取消喜欢作品
    - Fork允许二次创作的作品

- 本地保存作者允许下载的作品
- 评论/回复某个作品
- 编辑作品
    - 从相机拍摄一张图片导入
    - 从相册选取一张照片导入
    - 从空白页面开始创作
- 进入编辑模式在画板上绘制笔迹
    - 在支持3D Touch及Apple Pencil的iOS设备上，编辑模式支持压力感应，支持RGB全色域改变线条颜色，支持调整笔迹粗细
    - 支持无限步撤销/恢复
    - 编辑模式双指缩放画布，自由移动画布
    - 在全画板区域标记颜色锚点**
    - 在全画板区域标记颜色标记点***
    - 使用橡皮擦擦除颜色锚点和颜色标记点
    - 编辑过程中随时可以提交“草稿”给服务器上色，并能在服务器运算完毕后立即看到结果，无需真正发布作品。
- 发现页功能
    - 每日为当前用户推荐20张其他用户作品
    - 每日更新”最受欢迎的上色”，“最受欢迎的线稿”，排序依据是当日净增的“喜欢”数目
    - 可以按照作品名、作品ID、用户和作品标签来智能搜索相关信息。要求服务器配合客户端返回结果。客户端解析后布局

注:

1. 绘画能力划分为: 初级触手，中级触手，高级触手，大触。
2. 颜色锚点是一种可以帮助神经网络确定图片整体颜色风格的标记点。例如要画一个黑头发，白衣服，蓝色裤子的人，可以放置一个黑色锚点点在头发处，白色锚点在衣服处，以此类推。
3. 颜色标记点是一种在颜色锚点的基础上更进一步精确描述画面局部颜色的标记点。还是以2中所述的人物为例，假设此人头上有一个红色的发圈，可以放置一个红色的颜色标记点在发圈处去提示神经网络，不要一味的按照颜色锚点的指示去染色，同时也需要考虑红色区域。

IV. 软件架构介绍

本节专门用于介绍iOS, 后台业务层, 后台线稿上色程序的相关架构。有一些架构是已经设计好并投入实现了，但是还有一些架构仍处于设计阶段。这些还在设计阶段的架构会特别注明。

iOS客户端:

iOS客户端在遵从苹果公司的MVC设计模式的基础上，对此设计模式进行了一些结构上的创新。在文件结构方面，首先根目录层仍然是放置主要的Views, Storyboards, Models, ViewControllers, Utils文件夹用于归类不同层的代码文件，以及项目本身的资源文件夹Assets.xasset，Info.plist等。

对于ViewControllers，在此文件夹下按照主要的功能模块进行进一步的划分。如下表所示：

模块名称	模块作用
Drafting	在编辑模式下响应用户各种交互，作出反应的控制器。
PaintingWork	作品详情页面控制器，负责拉取作品详细信息，传至View层展示；对于作品的作者，提供修改各种作品属性的功能。
Person	用户个人主页控制器。展示用户的个人主页，关注/取关用户，按照时间线顺序展示用户动态。
Welcome	App启动界面、欢迎界面控制器、控制登陆，注册，忘记密码，访客登陆等功能。（欢迎界面做了一个不断向下滚动的图片瀑布流，实现较为复杂，需要单独的控制器控制）
Index	我的作品页面控制器。
Home	用户个人中心控制器。展示用户账号相关信息和软件相关设置的入口。
Discovery	发现页控制器，包含拉取推荐作品，最受欢迎的各种作品以及搜索聚合功能。
Setting	软件相关设置控制器，提供对软件行为及展示相关的设置以及持久化的功能。
VCExtensions	视图控制器辅助类，用于增强系统内置的一些视图控制器的功能，实现一些更为复杂的效果。

同时，在Views和Models下，均已经把对应的代码文件按照上面功能模块的划分进一步细分文件夹来进行归类。因此，不难看出这是一种非常经典的MVC分层架构，在开发过程中只要关注正在开发的功能分别在MVC中对应的模块文件夹即可。

不过与传统MVC模式不同的是，我们将Models应该完成的数据获取、更新视图、持久化的工作移动到了ViewController中，Models内的类全部定义为贫血模型。这是我们开发团队再三深思、做了若干考察后作出的改变。

因为我们发现移动应用程序开发中与视图紧密连接的不再是Model，而是Controller（这也解释了为什么苹果公司把Controller起名ViewController）。传统MVC架构的网页应用开发（例如ASP.NET）是可以在View中通过ViewBag机制很轻松地拿到Model的数据、并完成Model和View中对应标签进行数据绑定的，但是在手机App开发中与View直接通信的是Controller，View与Model之间形成了一层天然的隔离。并且受限于手机性能，从服务器请求数据、进行IO操作、进行CPU密集型计算都最好切到其他线程上完成。而在Model层下切换线程的操作没有在ViewController里来的方便；最重要一点是只有ViewController能够以最自然的方式持有View实例，以当前ViewController携带的根View为树根向下遍历子树，捕获用户与View交互的若干事件并作出相应。

有鉴于此，我们最终考虑对MVC架构的Model层作出分离数据操作逻辑的修改。在目前的架构下，ViewController的作用显得更加重要：它负责更新View层，按照Model中对数据类型的规约封装数据传到服务层处理、解析从服务层返回的数据；Model层完全退化成贫血模型，几乎只是完成对基本数据类型的组合、封装，形成更加具有语义性、上下文相关的数据模型；除了一些模型强相关的计算逻辑以外（如我们在控制绘制笔迹时所需要用到的Vector2数据类型，它需要附带一些向量相关的数学运算逻辑，这些逻辑以extension的方式attach到Model本身）不再拥有任何逻辑。View层则与传统MVC架构中对View的定义没有太大差别。

在实际开发过程中团队成员也感觉作出以上修改后减少了许多View和Model之间进行通信的代码，能够使得开发人员更加关注App的实现逻辑，避免处理许多无谓的繁枝末节。

后端:

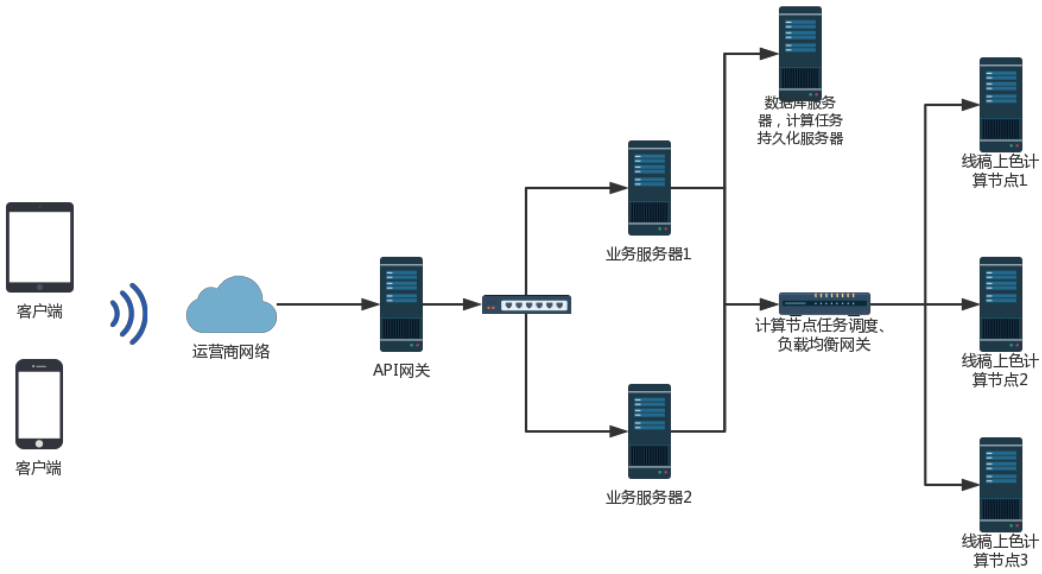
目前style2paints的服务端架构：

由于style2paints项目本质上还是一个验证学术成果的程序，因此它的架构非常简单，甚至可以说是没有架构可言。服务端后台只由4个核心文件组成：ai.py, config.py, server.py, tricks.py。其中config,py作用类似于配置文件，配置一些诸如GPU设备路径、是否打开调试信息输出、是否多线程等设置；server.py主要是通过bottle库起一个http server服务器用于运行他们的演示客户端，ai.py是一些加载预训练神经网络，以及调用神经网络的入口函数，tricks.py则是一些对线稿进行风格化的逻辑。除此之外没有更多。

显然，这样的架构确实是DEMO级别，无法支撑多用户，大并发，不适合分布式部署，一旦计算节点发生故障，当前计算节点的任务就永久丢失了，没有任何恢复机制。我们团队在较低配硬件上运行时不断遇到崩溃的问题，只能作为演示用。因此，为了使得该服务端更加健壮，提高可用性，我们将对其进行更进一步的修改。

项目预计实现的服务端架构：

如下图所示:



首先，客户端通过运行商网络连接后台服务。我们在后台服务的入口处放置了一台API网关。考虑到日后业务层硬件的横向扩展，业务层的不断迭代时进行热更新，以及面对突发情况对API做限流，熔断，负载均衡，健康监测等功能，网关在充当请求过滤、转发工作的同时还能完成上述提高服务可用性的额外工作。接下来API网关通过万兆内网连接业务服务器。业务服务器上就真正地运行实现上述“软件主要功能”的业务逻辑代码。

业务层考虑横向扩展做成分布式之后，再在业务层服务器上存储数据就显得不太合适了。因此考虑将数据库和静态文件持久化在专门的数据服务器上。在业务服务器出现问题的时候仍然能一定程度上保证数据的安全。

接下来，对于客户端发过来的“线稿上色”请求，需要后端具有高算力的计算节点完成计算任务。因此我们引入了一个计算任务调度的网关。这个网关能够接受从业务服务器处发过来的线稿上色请求，选择一个当前任务负担最轻的计算节点发送计算任务；同时，网关监测计算节点的在线状态、任务处理状态。一旦网关发现某个计算任务超过5分钟仍未完成，或者是某个计算节点离线了，网关可以自动地下线这台服务器，把它尚未完成的任务重新分配给其他仍在线的服务器，从而最大程度上保证用户的计算请求都能被顺利执行。

由于计算任务调度网关相对于业务服务器而言就是一个Consumer，并且我们的业务服务器默认其是分布式结构，因此不可能在业务服务器和调度网关间建立长连接获取任务完成情况；最终我们考虑调度网关提供一个接口供业务服务器查询任务完成情况，查询请求由客户端轮询发出，这样，轮询请求可以得到API网关的有效过滤，不会对调度网关造成过重的压力，也算是一种折中。

团队预计采用了以上的后端服务架构后，能够显著计算节点提高处理上色请求的能力，支持更多用户同时在线上色线稿。在技术选型阶段，团队成员尝试了简单地把RabbitMQ配置成均衡网关，在没有其他更多调度设置，一切默认Round-Robin的情况下已经能做到3台家用笔记本电脑支撑大约10人同时在线上色。我们认为结合调优的RabbitMQ和对style2paints节点的计算代码进行一些状态上报，性能优化的改造之后能够实现更高的任务承载量。

V. 总结

本报告介绍了移动端线稿上色软件iAnime的软件设计、前后端代码结构，后端部署架构。接下来我们将根据以上设计要点，对项目进行开发、优化。