Графика, Python и ИИ



Тимур Агаев

Тимур Агаев

Графика, Python и ИИ

https://litres.ru/74734442
SelfPub; 2026

Аннотация

Эта книга — не сухой учебник, а живой roadmap для тех, кто хочет автоматизировать работу с графикой, научиться создавать интерфейсы на Python и многое другое, не увязая в бесконечных мануалах. Автор, ленивый по собственному признанию, но увлечённый практик, делится личным опытом: как за двадцать строк кода превратить обычное фото травы в мультяшную текстуру для игры или в пиксельный блок в стиле Minecraft.

Первая часть посвящена библиотеке Pillow, вторая часть — это погружение в Tkinter и работу с холстами (Canvas), третья часть — справочник по фильтрам Pillow, а завершает книгу практический кейс по созданию мудборда: от концепции и ручного коллажа до работы с нейросетью, генерации промптов и ручной доводки в редакторе. Автор честно рассказывает о граблях: почему ИИ «делает мыло», как слои в редакторе спасают цветовую гамму, и даже остаётся место для небольших философских размышлений.

Содержание

Pillow вместо Photoshop: как за 20 строк кода делать мультяшные и Minecraft-текстуры 4
Конец ознакомительного фрагмента. 15

Тимур Агаев
Графика, Python и ИИ

Pillow вместо Photoshop: как за 20 строк кода делать мультяшные и Minecraft-текстуры

Автоматизация

Изучая Adobe Photoshop (в частности любимый мною CS6 – так как запросто устанавливается на любой чайник), программы по монтажу (VSDC Video Editor, Wondershare Filmora, Movavi и Capcut), и другого типа редакторы мне не могла не прийти мысль об автоматизации. Как минимум потому, что я ленивый, а в общем и целом – для автоматического создания материалов и текстур. Создавая игры любой разработчик уткнётся носом в одну простую вещь: у него есть готовый и расписанный лор, персонажи и их дизайны, может даже саундтрек и примеры кода, но нет второстепенных текстур. Под второстепенными текстурами я имею в виду то, что не влияет на лор, но влияет на игровой опыт (к примеру, блок земли в Minecraft не поможет нам убить дра-

кона, но данная текстура жизненно необходима игре).

Следовательно, нужно было научиться автоматизировать процессы обработки графических материалов. Последовательная обработка подразумевает код, а автоматизация – наличие функций. Желательно было всё это оформить на Python. Идея понятна: хочу обрабатывать визуал таким образом, чтобы целый ряд фильтров и сопутствующих преобразований накладывались по одному клику, без упорного затирания UI Photoshop до дыр. И что делать? Здесь на помощь приходит один известный форк PIL (Python Imaging Library) под названием Python Pillow.

Как оказалось, сообщество Python уже давно задумалось над тем, как можно через код работать с графикой. PIL – древняя библиотека для работы с растровой графикой, поддержка которой давно прекращена. Потому на её основе возник форк под названием Pillow, о котором сейчас и поговорим. Стоит отметить, что на мой субъективный взгляд Pillow лучше всего работает именно с форматом PNG, потому рассматривать работу библиотеки с файлами иных расширений мы не станем, по крайней мере в этой статье. Также отмечу, что это не совсем образовательная статья, а скорее Roadmap, потому что доступ к каким угодно учебным материалам сегодня есть из любой точки земного шара, и гораздо важнее поговорить именно про последовательность действий при освоении данной технологии – какой она должна быть, чтобы иметь достаточное понимание для выполнения опреде-

лённого рода задач.

Базовые функции

Всё ПО работает под одинаковому принципу: мы хотим что-то поставить на вход таким образом, чтобы получить желаемый результат на выходе. Когда мы обрабатываем фото в Photoshop, мы на вход выгружаем картинку из имеющихся файлов, в программе описываем необходимые изменения и экспортируем (получаем выходной файл). Точно такой же процесс нам нужно научиться реализовывать в Python, а если проще – загружать картинку и удалять её из реализуемых процессов (что-то вроде запуска и завершения функции?).

Загрузка происходит банально: в переменную записывается имя файла под функцией «open» (открыть). За это и люблю данный ЯП – читается как английский язык. Итак, мы получаем запись типа «original = PIL.Image.open("chto-to.png")». Удаление из реализуемых процессов – честно, не увидел в примерах это в виде необходимого параметра, но в конце всегда используются функции «save(название)» и «show», сохранить и отобразить соответственно. Также, сохраняя, можно указать формат: image.save("chto-to.png", format="PNG"), хоть это и необязательно.

Попробуем создать иконку? Вот допустим есть у меня PNG, и срочно нужна миниатюра 128x128 во имя добавления её в виде иконки Godot-проекта. Что делать? Задаём

size, при желании/необходимости в отдельной переменной в виде «size = (128, 128)», а потом задаём её для картинки: «original.thumbnail(size)». Thumbnail, кстати, так и переводится – миниатюра. После этого сохраняем результат и открываем.

Оно в целом маловажно, но если нужна информация о картинке, то можно узнать её «format» (например, PNG) и «mode» (например, RGB).

Что такое RGB? Давайте расскажу, тут интересно. Бытует мнение, что в картинках на каждый конкретный пиксель указан конкретный цвет. Господа и дамы, будь это так, никаких сложных программ для редактирования визуала бы не появилось – люди столько не живут, сколько потребовала бы реализация такого инструментария при подобном исходе. Разумеется всё сделано хитрее.

Картинки хранят по каналам: красный, зелёный и синий. В каждом канале определённое количество пикселей, а в каждом пикселе записан не готовый цвет, а числовое значение, указывающее, сколько в нём того или иного от конкретного канала. Столько-то красного, столько-то зелёного, столько-то синего. Таким образом каждый пиксель начинает иметь свой цвет, и вырисовывается красивая картинка. RGB — это аддитивная цветовая модель, в которой цвета создаются путем сложения красного, зелёного и синего (red, green and blue).

Pillow может разделить картинку на каналы с помощью

метода «split», а собрать – с помощью «merge». Можно раскрасить чёрно-белую картинку (просто каналам чёрного и белого вы задаёте иные цвета) через «colorize(image, black ="red", white ="yellow")». Накладывать картинки друг на друга можно через «blend». Картинку можно изменять так, как душе угодно: «image.convert("указать мод")» переводит image в указанный мод.

А теперь к творчеству. Инструмент есть, но нам нужна автоматизация. Логика понятна: делаем что-то вроде конвейера.

Картунизация. Края

Прежде всего мне захотелось сделать картунизацию. Зачем? Это лучше всего подходит для решения исходной задачи: мы из простого материала (фото древесины, сделанное в момент прогулки) делаем текстуру («мультяшная» текстура дерева для игры). В один клик, без заморочек. И... у меня это получилось!

Сейчас будем рассматривать не полноценный код, а отдельные функции, которые были задействованы – я буду описывать, что да как. А потом покажу результат: было/стало. Визуальное понимание, как правило, вместо тысячи слов.

Для картунизации нам нужно две: создание контуров и работа с цветами. И то и другое может показаться сложным, но по сути всё легко и даже по-своему банально: просто по-

очерёдно накладываешь фильтры и примерно в голове представляешь, как будет выглядеть результат. Допустим, размытие по Гауссу известно любому, кто хоть раз работал в Photoshop – этот метод используют, на моей памяти, чаще всего.

Следует сказать и о том, что работа будет вестись с форматом PNG в моде RGB – это просто, понятно, а что ещё нужно для хорошей творческой работы, верно? Одно расширение, три канала, и куча разнообразных функций. Только нам стоило бы добавить их:

```
from PIL import Image, ImageFilter, ImageOps, ImageEnhance, UnidentifiedImageError
```

Теперь можно открыть картинку и принудительно конвертировать (преобразовать) её в RGB. Вдруг исходник в другом формате? Не пойдёт.

```
img = Image.open(image_path).convert("RGB")
```

Отлично! Теперь подумаем, как нам сделать контур. Хочу наложить размытие, выделить границы по яркости, инвертировать (чтобы границы стали тёмными на белом фоне), превращаем в конкретную чёрно-белую маску контуров и утолщаем углы. В результате получатся чёрные, чёткие жирные контуры – как в мультиках.

Приступим.

```
blurred = img.filter(ImageFilter.GaussianBlur(blur_radius))
```

Для начала мы накладываем фильтр – блюр по Гауссу (размытие по Гауссу), задаём радиус и записываем получив-

шееся в отдельную переменную.

edges = blurred.convert("L").filter(ImageFilter.FIND_EDGES)

Теперь нужен поиск краёв. Переводим с помощью «convert("L")» картинку в чёрно-белый формат (нам нужно только темнее/светлее), накладываем фильтр FIND_EDGES (по сути переводится как: НАЙТИ КРАЯ). Стоит уточнить, что «L» – это не совсем «чистый чёрно-белый», а градации серого.

Но вы помните, что нам нужно именно инвертированное изображение? Так как удобнее, когда края чёрные, а не белые. Давайте зададим и это:

edges = ImageOps.invert(edges)

Функция «invert» меняет в чёрно-белой картинке чёрный на белый, а белый – на чёрный. Итак, теперь у нас есть чёрные края на белом фоне, чудесно!

Но теперь готовьтесь... Будет тяжело, одна строчка прямо-таки ух:

edges = edges.point(lambda x: 0 if x < edge_threshold else 255)

Стоп, давайте поясню, что всё это значит! Это называется пороговая бинаризация. Нам нужно убрать лишний шум и оставить только чёткие края, верно? Значит так, эта функция буквально говорит: всё, что меньше edge_threshold (задаётся отдельно как переменная) – будет 0, то есть чёрным, а где больше – будет 255 – то есть белым. Таким образом

мы убираем шум: теперь у нас чёткое разделение на чёрные края и белый фон.

То есть «point» оставляет только сильные края, а всё остальное превращает в фон. Понимать его можно как «оставить только уверенные границы, остальное отбрось».

```
if edge_thickness > 0:
edges = edges.filter(ImageFilter.MinFilter(3))
```

И утолщаем получившиеся контуры. Запомните, что мы записали их в «edges».

Теперь нам нужны цвета. А потом мы всё «склеим»!

Картунизация. Цвета

Границы выделили. Возвращаемся к исходному изображению и работаем теперь с цветами. Снова применяем блюр по Гауссу, нам необходимо сгладить шум и мелкие детали.

```
color = img.filter(ImageFilter.GaussianBlur(blur_radius))
```

Теперь нам нужны насыщенность и контраст:

Управляем насыщенностью:

```
color = ImageEnhance.Color(color).enhance(saturation)
```

Управляем контрастом:

```
color = ImageEnhance.Contrast(color).enhance(contrast)
```

Тут наверное вам стало интересно, какие лично я задавал параметры для картунизации? Давайте поделюсь своими переменными, но вы, дорогие мои, можете поиграться и сами: num_colors=8, saturation=1.6, contrast=1.2,

edge_threshold=80, blur_radius=3, edge_thickness=1. «threshold» и «thickness» переводятся как порог и толщина соответственно, последнее в коде является флагом.

Ура, теперь можем перейти к самому вкусному – а именно, к самой картунизации!

Это те функции, ради которых всё и затевалось изначально:

color = color.quantize(colors=num_colors, method=Image.MEDIANCUT).convert("RGB")

color = color.filter(ImageFilter.MedianFilter(size=5))

Запомните, что мы получили «color».

Самое шикарное здесь – это метод «quantize» – дословно переводится как «квантовать». Он

выбирает цвет и метод: цвет – до которого нужно сократить (у нас это «num_colors»), метод – которым следует пользоваться для достижения задачи. У нас это MEDIANCUT: алгоритм строит палитру из указанного количества (N кол-во) наиболее представительных цветов и заменяет каждый пиксель ближайшим цветом из палитры. В результате получается «постеризованный» вид.

После снова переводим к RGB для дальнейшей работы.

Но после квантования часто следует использовать MedianFilter (медианный фильтр), чтобы убрать лишние шумы, артефакты, сделать картинку более гладкой, как бы чистой, без «крошки» на границах между цветами. Он как бы заменяет пиксели медианой соседних пикселей, тем самым

выравнивая.

Теперь нам нужно границы и получившуюся пальтру сложить между собой! У нас есть переменные «edges» (контуры, границы) и «color» (палитра).

Собираем в композицию:

result = Image.composite(color, Image.new("RGB", img.size, "black"), edges)

Проще: результат равен композиции(палитра цветов, чёрный фон того же размера на всякий случай, и контуров). Получается рисунок в стиле комикса/мультфильма.

Пикселизация

А что если мне нужна буквально текстура в стиле Minecraft? Как это сделать?

В сей раз без мук обойдёмся. Нам нужно сделать пикселизацию под сетку (всё-таки в Minecraft текстура – это сетка из клеток, каждая из которых имеет свой цвет). Был удивлён, что для этого потребовалось всего несколько строк! Настолько это просто.

Да, я из тех, кто сначала показывает решение сложной задачи – а потом даёт простую, чтобы на констрасте вы, дорогие читатели, почувствовали себя специалистами (хотя если разобрались с первым примером, вы действительно поняли Python Pillow).

Переменные: pixel_size: int = 16, num_colors: int = 24,

saturation: float = 1.3, contrast: float = 1.2.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, купив полную легальную версию на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.