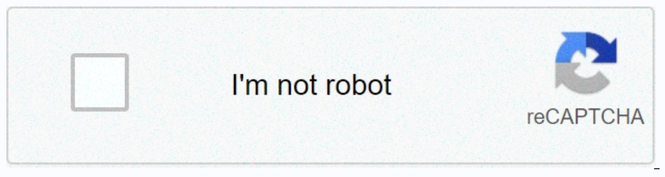

Continue

Cartoon photo editor android github

Add a description, photo, and links to the cartoon subject page so developers can learn about it more easily. Curate this topic To associate the repository with the comic book topic, go to the repo's landing page and select Manage topics. Learn more You can't perform this action these days. You signed in with a different tab or window. Reload to update the session. You signed out in another tab or window. Reload to update the session. We use optional third-party analytics cookies to understand how you use GitHub.com so that we can build better products. get more information. We use optional third-party analytics cookies to understand how you use GitHub.com so that we can build better products. You can always update your selection by clicking Cookie Settings at the bottom of the page. For more information, please refer to our Privacy Policy. We use important cookies to perform important website features, such as being used to log you in. Learn more Always Active We use analytics cookies to understand how you use our websites so that we can make them better, such as being used to collect information about the pages you visit and how many clicks you need to perform a task. Learn more Page 2 3D modeling uses specialized software to create a digital model of a physical object. It is an aspect of 3D computer graphics, used for video games, 3D printing and VR, among other applications. You can't perform that action at the moment. You signed in with a different tab or window. Reload to update the session. You signed out in another tab or window. Reload to update the session. We use optional third-party analytics cookies to understand how you use GitHub.com so that we can build better products. get more information. We use optional third-party analytics cookies to understand how you use GitHub.com so that we can build better products. You can always update your selection by clicking Cookie Settings at the bottom of the page. For more information, please refer to our Privacy Policy. We use important cookies to perform important website features, such as being used to log you in. Learn more Always Active We use analytics cookies to understand how you use our websites so that we can make them better, such as being used to collect information about the pages you visit and how many clicks you need to perform a task. Learn more class MyActivity : Activity() { ... val PESDK_RESULT = 2 funny openEditor(inputPhoto: Uri) { PhotoEditorBuilder(this) .setSettingsList(PhotoEditorSettingsList() .configure<LoadSettings> { it.source = inputPhoto } .configure<UiConfigText> { } .configure<UiConfigFrame> { it.setFrameList(FramePackBasic.getFramePack()) } .configure<UiConfigFilter> { it.setFilterList(FilterPackBasic.getFilterPack()) } .configure<UiConfigOverlay> { it.setOverlayList(OverlayPackBasic.getOverlayPack()) } .configure<UiConfigSticker> { it.setStickerLists(StickerPackEmoticons.getStickerCategory()) }) .startActivityForResult(dette, PESDK_RESULT) } ... } var app = { initialisere:</UiConfigSticker> </UiConfigOverlay> </UiConfigFilter> </UiConfigFrame> </UiConfigText> </LoadSettings> </LoadSettings> () { document.addEventListener(deviceready, this.onDeviceReady.bind(this), fake); }, onDeviceReady: function() { PESDK.openEditor((result) => { console.log(result); // success callback }, (error) => { console.log(error); // failure callback }, PESDK.loadResource(www/img/example.jpg)); }, }; app.initialize(); import {Component} from @angular/core; import {Platform} from '@ionic/angular'; declare was PESDK; @Component({ selects: 'app-home', templateUrl: 'home.page.html', styleUrls: ['home.page.scss'], }) export class Home page { constructor (platform: Platform) { platform.ready().then() => { PESDK.openEditor((result) => { console.log(result); // success callback (error) => { console.log(error); // failure callback }, PESDK.loadResource('www/assets/example.jpg')); }); } MagicPhotos lets you quickly and easily give photos a dramatic look by applying different effects while keeping your chosen details untouched. This draws the viewer's attention to the individual image details, creating striking images. With its intuitive and responsive user interface, MagicPhotos is super easy and fun to use. You only use your finger as a brush to apply effects to certain areas of the image or bring back original image. Photo editor supports multiple effects (Decolorize (Color Splash), Recolor, Sketch, Cartoon and so on) and Photo Retouching mode that allows you to remove small errors and unwanted content (watermarks, people, text, power lines as well as other unwanted objects) from your photos, as well as apply controlled blur to selected details of the image you want to hide (license plates, human faces and so on). The program is easy to use and provides full control over the editing process. Visit the project site for more information. Photo by Jean-Philippe Delberghe at UnsplashOpenCV is a powerful tool to do intensive operations in real time, even for resource-restricted mobile devices. Through a couple of tutorials, we're going to create an Android app that applies different effects to photos. In part 1 of the series we discussed horizontal and vertical image stitching: This tutorial will explore creating a cartoon image effect by reducing the number of colors representing the image using lookup tables (LUTs). The sections covered in this tutorial are as follows:What is LUT? OpenCV Built-in LUTs.Color Reduction using LUT for gray images. Color reduction using LUT for color images. Using a Single LUT for each Color Image Channel.Adaptive Thresholding.Painted Using Median Filter.Cartoon Effect.Building Android App.The source code of this tutorial is available in this GitHub project. The following image will be used in the experiments. Photo from Pixabay by Hulki Okan TabakLets getting started. When talking about images, a lookup table (or LUT for short) is a mapping from one set of colors to another set of A use of LUTs in image processing is to change the colors to be in a given image. Let's explain it by example. Let's say the following 3x3 array represents an image. If each pixel is represented as 3 bits, the pixel values are from 0 to 7.If for a given reason we should change the colors that the previous image appears after, how can we make this change? One method is to use an LUT as the one shown below. It works by assigning each old pixel value in the previous image to its new value. According to under LUT, each pixel in the old image with value 0 will be replaced by the new value 2. Each pixel with value 4 is replaced by value 6.In order for LUT to work with all pixel values in the image, lut must have a variety of entries equal to the number of colors contained in the image. For the image above it is represented as 3 and a total of 8 colors. Thus, LUT must contain 8 entries, one for each color. If it's a gray image represented by 8 bits — that is, it's 256 colors — LUT must have 256 entries. By using LUT for all image pixels, the new image is given below. Just look at the table to replace each pixel with the new value. That's why it's called a lookup table. In OpenCV, there are a number of LUT's to change the colors used to display an image. The Imgproc.applyColorMap() method is used to use LUT over an image. Note that a LUT can also be called a swatch. This method accepts 3 arguments:Food src: Source Food to change their colors. Food dst: Destination mat that receives result.int swatches: Integer that represents the swatch used to change the colors. There are different swatches in OpenCV such as autumn, summer, winter, pink, rainbow and more. The following code is used to use the summer color map of the image used in the tutorial. Note that the summer color map has a value of 6 for the third integer argument. But we can use Imgproc.COLORMAP_SUMMER, which is a constant with value 6.The result is given in the next figure. The next figure represents the range of colors for the summer color. The color of the left side is green, which means that the dark pixels of the image will be replaced by green colors. As the color glows, the color will move from green to dark yellow. The sky in the image used in this tutorial has bright colors, thus taking the new image from the yellow color more than the green color. Let's discuss a different swatch. The following example uses the pink swatch. The result appears in the next shape. To explain why the image appears this way, let's look at the color scale of the pink swatch. The third argument is set to Imgproc.COLORMAP_PINK, which is a constant with value 10.The scope of colors for this swatch is shown below. On the left side there is a black, which means that the dark pixels of the original image will be replaced by a dark color. As the color lights up, the swatch moves to a lighter color until it turns white. To understand how the swatches or LUTs are Let's create our own swatch. Before we start working with color images, let's start with the simplest case, where a LUT is created for a gray image. A gray image is usually represented as 8 bits. As a result, there are 256 color levels from 0 to 255. To make a LUT, it must have 256 elements. The first element of lut defines the new color for the first level of the gray image (0), the second LUT element is associated with level 1, and so on. So to create a LUT, just think about creating a Food array with 256 items. The following code bar creates a math with zeros. The size of this mat is 1x256 because the gray image has 256 colors. The type of this mat is set to 8UC1, which means that the value in each item is an unsigned integer represented in 8 bits. Thus, the acceptable values for each item are the full figures from 0 to 255. If a value is below 0, it will be cut at 0. The same happens for values above 255, as they are returned back to 255.Note that the above mat can be used as lye without any problem, but the resulting image will be all zeros because all 256 colors in the gray image will be associated with the value 0 in LUT. We should add different values to lut. This can only be done with for loops that assign a variety of different values to LUT. The code is given below. The number of different colors in lut is defined in the numColors variable. The first loop assigns the unique values for the zero mat that was created earlier. It uses a step similar to 256/numColors.This guarantees that the outer loop creates a variety of colors similar to that specified in the numColors variable. When numColors is set to 5, the step is 256/5=51.2. Thus, the values for the loop variable x will be 51.2, 102.4, 153.4, 204,8, and 256,2.Because the type of lookupTable Food is 8UC1, all these float numbers are converted to integer by removing the decimal point. That is, 51.2 will be 51 and 204.8 will be 204. For the value 256.2, which exceeds the maximum acceptable value for type 8UC1, it will be clipped at 255. Thus, the unique values in the Mat 51, 102, 154, 204 and 255.The indices in which these items are added are equal to their values. That is, the value of the Food element in index 51 is 51, the value of index 102 is 102, and so on. The outer loop adds 5 elements to these values. All other elements of LUT are zeros as initialized at the time of creating lookupTable Food. The inner loop works by filling these zeros. The items from index 0 to the index of the first unique item (51) are all set to 51. The items from index 51 to 102 are set to 102, the items from index 102 to 154 are set to 154, and so on. The result is a mat where there are only 5 unique values representing all 256 elements of lut. To reuse the above code, a method called createLUT() is created and shown below. It accepts the number of colors. It checks that the colors are between 0 and 256 (inclusive). It returns a mat that represents LUT. Once you have ready lut, the next step is to use it above the image. This is done using the LUT() method that is available in the Kernel class. It accepts 3 arguments that are:1. Food src: Source Mat.2. Mat lute: LUT3. Mat dst: Destination to save the result. After preparing LUT and understanding how to use it in the photo, we need to apply it to a gray image. The following code does the following: Reads an image. Converts the image from BGR to gray. Saves the original image file. Creates LUT as discussed earlier. Uses lut over the image using the LUT() method. Saves the result of using LUT. The image is converted from BGR to gray using the cvtColor() method available in the Imgproc class. It accepts 3 arguments, which is the source, destination, and an integer representing the conversion code. The code for the conversion from BGR to gray is 6. We can also use Imgproc.COLOR_BGR2GRAY, which is a constant with value 6.We can also create a method called reduceColorsGray() to combine the above code, as shown below. It accepts the image that LUT will be used in addition to the number of colors. This method returns the mat after applying LUT. By simply calling the method reduceColorsGray() is created and used LUT on the image, and the result is returned. Here's an example:The original image after it's converted to a gray image is shown below. The result image from using a LUT with 5 colors above the above gray image is shown below. To convert the image to binary, just use 2 colors only in LUT. The result is given below. Note that if you set the number of colors to 0 or 1, a black image returns. By increasing the number of colors, the image caused by using lut approaches the input gray image. The next figure shows the result of using 20 colors. Here is the result after using 50 colors: It is expected that making a LUT with 256 colors will return the original gray image. This is shown below. There is no difference between the image before and after applying LUT. At this point, we have applied a LUT to a gray image by specifying the number of unique colors. The next section discusses applying a LUT to a color image. Applying LUT to a color image is not so different from applying it to just a gray image because a color image actually consists of a series of gray images. To use LUT over color images, there are 2 techniques: Using a single LUT for all color image channels. Use a single LUT for each color channel. Let's discuss these 2 approaches. The first approach is to just prepare a single LUT and apply it above the color image. In this case, single LUT will be applied to all image channels. Compared to the previous code to use LUT over a gray image, the only change is to remove the line that converts the image to gray and leave a color image as it is. The code is Below. The previous code uses only 5 colors in LUT. Please note that such a LUT is identical to the one created earlier. That is, the 5 unique values are 51, 102, 154, 204 and 255. A LUT with these 5 values is applied to the 3 channels (red, green and blue). As such, each of the 3 channels will have 5 unique gray levels. If each channel has 5 gray levels, the total number of colors produced by such levels is 5x5x5 = 125. Thus, the color image produced using LUT will have 125 unique colors. The resulting image from the above code is shown below. As the number of colors increases, the image tends to be more similar to the original color image. The next figure shows the result after using 15 colors that build LUT. When the number of colors is set to 256, the original image is rendered. If you set the number of colors to 0 or 1, a black image is returned. After applying a single LUT to all color image channels, the next section creates a LUT for each channel. For example, if there are three channels, there will be 3 LUTs – one for each channel. Using a single LUT for each color channelBefore we want to create a LUT for each channel, LUT is created for the 3 channels as shown below. The number of colors for all channels is set to 5.After that, we need to split the image into its 3 channels. This is done by creating a Java list that accepts 3 Food matrices. Using the Core.split() method, the image is divided into 3 channels, and these channels are saved in the list. You can access each item in the list by using the get() method after you pass the index of the item. For example, BGR.get(0) returns the first channel (the blue one). The code below applies 3 LUTs to the 3 channels. Now the BGR list has the 3 channels after using its associated LUT. The remaining step is to re merge these 3 channels to create a BGR image using the Core.merge() method, according to the following line: To collect the above code, a method named reduceColors() is created. It accepts

the Food image to change the colors as well as the number of colors for each channel. The line below provides an example to call the method above:Because the number of colors for all channels is equal, the result will be identical to using a single LUT on all these channels. The result (shown below) is identical to the result achieved earlier using a single LUT for all channels with 5 colors. The difference is displayed when you use a different number of colors for the different channels. For example, the following line calls the method reduceColors() and passes 50, 0, and 0 respectively for the number of colors for the red, green, and blue channels. The result is shown below, where the red color dominates the other colors:The same effect occurs when you enlarge the value of 1 channel more than the other 2 channels. For example, the result of performing the next line will be green, as shown in the next figure:I according to the next line, the number of for the red and green channels is 50, but it is 0 for blue. The result will be yellow as given in the next figure. Note that yellow is the color as a result of combining red and green: Let's summarize what we have discussed so far. First we discussed and defined LUT. Then we looked at some built-in LUTs in OpenCV and used them using the applyColorMap() method. After that, we created our own custom LUT, which is just a mat with 256 elements. Using the LUT() method, we applied the custom LUT to a gray image. After that, we applied a single LUT to a color image. Finally, we applied 3 LUTs to a color image created, with one for each channel. To create a cartoon image, there are two additional steps that we will discuss in the following sections. One step that helps achieve the cartoon effect is to convert the image to binary by thresholding it. When the image is converted to gray, the simplest threshold uses a single value to classify the pixels as black or white. The pixels above the threshold are classified as white, and the pixels below it are black. The type of threshold klering used in this tutorial is adaptive threshold because it changes the threshold dynamically when applied to different types of images. Before using the adaptive threshold, the image from BGR is converted to GRAY according to the line below, given that img is the original color image. Then the adaptive threshold can be used using the adaptiveThreshold() method, as shown below. For information about the arguments, please visit the official documentation here. The result of the adaptive threshold is shown below. Please note that there is a lot of noise in the result. To reduce the noise, it is preferred to filter the gray image before it is converted to binary. A type of filter is the median filter. The median filter is a type of smoothing filter supported in OpenCV using the Imgproc.medianBlur() method. It accepts 3 arguments:src: Source Mat.dst: Destination Mat where the output will be saved.ksize: core size. Here's an example of calling this method over a gray image. The result appears in the next shape. The result is smooth and the image is less exposed to noise pixels. After applying the median filter, the image is less noisy, and thus the result of applying the adaptive threshold above the filtered image will be much better (shown below). To create the cartoon image, the result of using LUT is combined over the color image and the result of the adaptive threshold together. Now we have completed the 3 steps required to create the cartoon image, which is as follows: Reduce image colorN removal using median filterAdaptive thresholdingThe next step is to combine them. The complete code for creating the cartoon image is included below. At the end of the code, the image is combined as a result of the method reduceColors() with the image as a result of the adaptiveThreshold() method using the Core.bitwise_and() method. Please note that of adaptiveThreshold() method is a gray image with only 1 channel, but the result of the reduceColors() method is a color image with 3 channels. Before Core.bitwise_and() method, we need to make sure that the 2 images have the same size. This is why the result of the adaptiveThreshold() method is converted back to a color image before using the Core.bitwise_and() method. We can create a method that contains the above code: Here is an example to use the above method:The result of running the code above is shown below:The result is changed by changing the parameters sent to the method reduceColors(). For example, if you use reduceColors(img1, 80, 15, 0) returns the result below. Note that the pipeline for producing such cartoon images is an art more than an imaging science. So if the current results do not match your needs, you can add new steps or change existing steps using different parameters. After completing the effect of this tutorial, our next step is to edit the Android app created in the previous tutorial to add this effect. Build Android AppActivity XML setup is listed below. A new button is added for each of the effects discussed in this tutorial. The Android app interface appears below. The Java code activity is listed below, which includes all features discussed previously and managing button-click events. After clicking the Cartoon Image button, the resource image is converted to a cartoon image as shown in the next character. This tutorial, which is part 2 of the image effects using OpenCV for Android series, discussed a new image effect to create cartoon images from color images. The main idea is to reduce the number of colors used to represent the image. The reduced color image is then combined with a binary image created using adaptive thresholding after being filtered using the median filter, which is used to remove noise. In the next tutorial in the series, we look at image transparency in OpenCV – add an alpha channel to the color space and manipulate it to merge different images. Editor's note: Heartbeat is a contributor-driven online publication and community dedicated to exploring the new intersection of mobile app development and machine learning. We are committed to supporting and inspiring developers and engineers from all walks of life. Editorially independent, Heartbeat is sponsored and published by Fritz AI, the machine learning platform that helps developers learn devices to see, hear, feel and think. We pay our contributors, and we don't sell ads. If you would like to contribute, proceed to our call to contributors. You can also sign up to receive our weekly newsletters (Deep Learning Weekly and Fritz AI Newsletter), join us at Slack, and follow Fritz AI on Twitter for all the latest mobile machine learning. Learning.