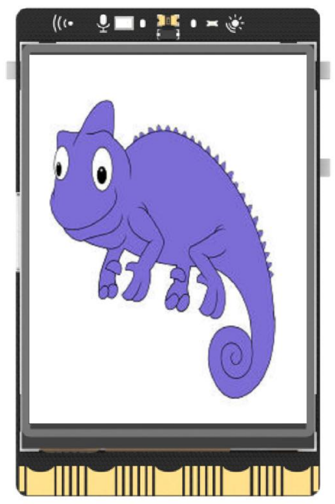
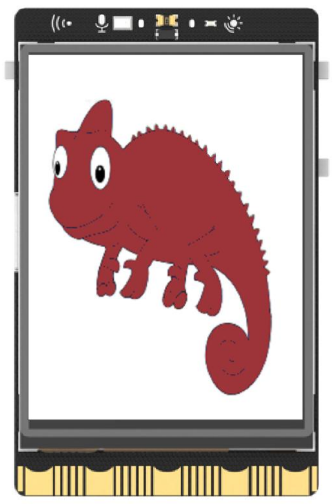
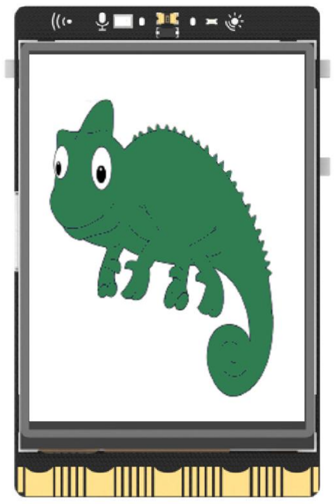
Lesson 11 Chameleon Screens

Chameleons are highly adaptable arboreal reptiles. Over millions of years of evolution, they have developed a biological instinct to change their body coloration, often unintentionally, to escape predators and approach their prey. This incredible ability allows them to blend seamlessly into their surrounding environment.

However, this fascinating skill is often only heard of in stories. So, how can we experience it firsthand? In this lesson, let's simulate the effect of a chameleon using a transparent screen and a color-detecting tool.

Task Objectives

Using a color sensor to detect the ambient color, we can display the chameleon character on the screen in the detected color.

		
The original image	After color change 1	After color change 2

Knowledge points

1. Get to know the color sensor

2. Understand image recognition and the basic properties of images

3. Learn about some types of image processing

4. Familiarize yourself with the opencv library

5. Learn how to use the opencv library for image processing

6. Learn how to use the Pinpong library to detect colors

Material List

Hardware List:

UNIHIKER x 1	Type-C & Micro Dual-Use USB Cablex1
I2C 颜色识别传感器 - TCS34725 x1	两头 PH2.0-**4P**(无倒扣)白色硅胶绞线

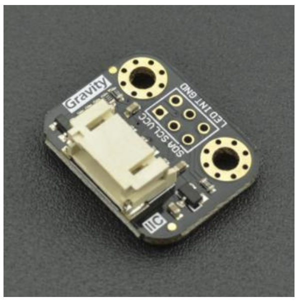
Software Preparation: Mind+ Programming Softwarex1

Knowledge background

1. What is a color sensor?

A color sensor is a sensing device that can identify the color of an object. The sensor emits light onto the surface of the object, calculates the color components (red, green, blue) based on the reflected light, and then outputs the RGB values. The sensor is equipped with four high-brightness LEDs around it, which allows it to function properly even in low-light environments.

Tips: When in use, connect the color sensor to the IIC interface. When selecting colors, be sure to place the color sensor 3-10mm above the object.



2. What is image recognition?

(1) Concept of image recognition

Image recognition refers to the technology that uses computers to process, analyze, and understand digital images in order to identify various patterns of targets and objects.

(2) Analysis of image recognition cases

This project is one of the common cases of image recognition. Let's take a look at how it is implemented below.

Purpose: Change the color of the "dragon" in the original image (blue) to the specified background color (e.g. RGB value 238 130 238).

Drawing implementation steps:

Step 1: Identify the blue main part in the original image.

Step 2: Use a brush to draw the blue main part in the desired color.

Computer image processing implementation steps:

Step 1: Detect the blue part in the original image. The method is to obtain a blue mask image using the HSV color space (image 1).

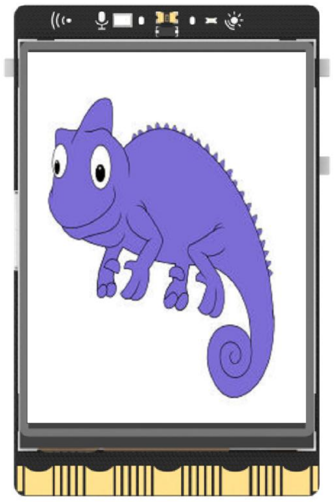
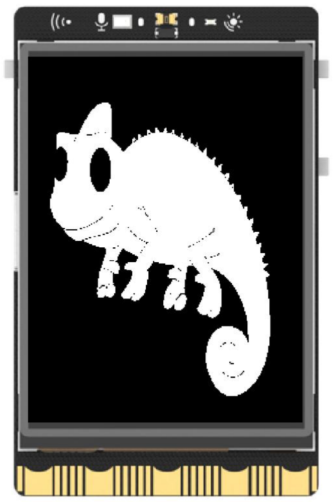
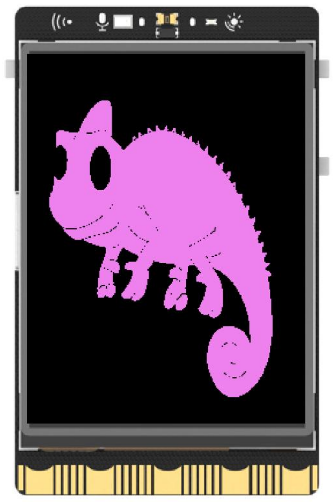
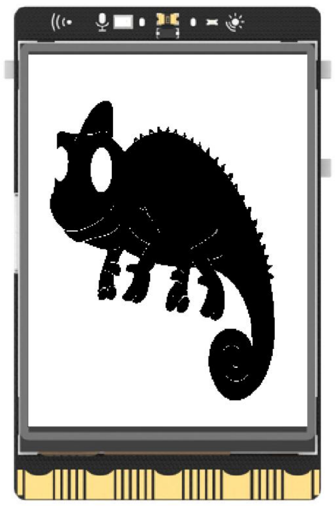
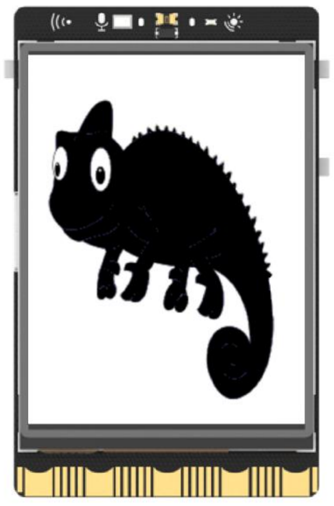
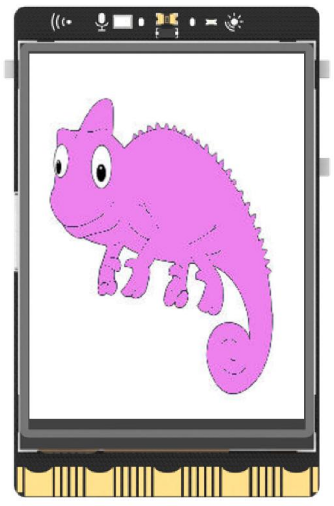
Step 2: Extract the image to be displayed from the background image. The method is to perform a mask operation between image 1 and the background image to obtain image 2.

Step 3: Invert Figure 1 to obtain image 3.

Step 4: Perform a mask operation between the original image and image 3 to obtain image 4.

Step 5: Perform a bitwise OR operation between image 2 and image 4 to obtain image 5 (the final effect).

Tips: The specific operation method will be introduced later.

 original image	 image1	 image2
 image3	 image4	 image5

(3) Basic Properties of Images

Images have some basic properties, such as pixels, dimensions, and colors. By reading the data, computers can obtain the basic properties of the images.

A. Pixels

Pixels, also called pixel points or elements, are the most basic composition elements. As shown in the figure below, if the image is enlarged enough, you can see small colored blocks like mosaics, which are pixel points. Each pixel point represents a color, and multiple pixel points together form a rich and colorful image.

B. Size

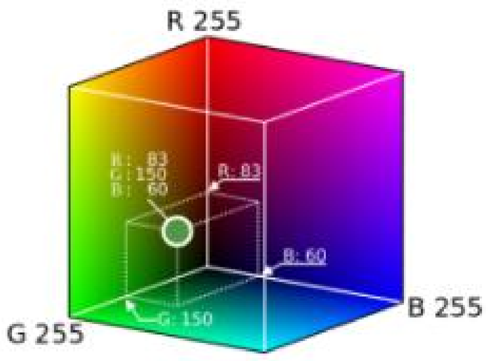
Image size refers to the number of pixels in the horizontal and vertical directions of an image. For example, if an image is composed of 100x100 pixels, then its size is 100x100.

C. Color

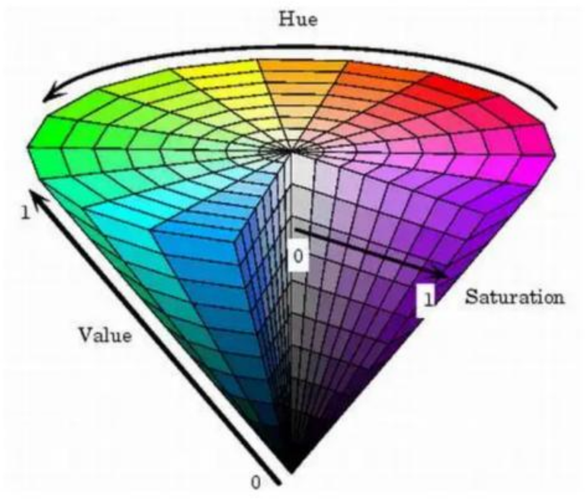
Images are displayed in color, and in computers, color spaces (also known as color models) are used to represent the colors in an image, such as the commonly used RGB and HSV color spaces. Color spaces represent colors through a set of numbers, and there are different types of color space definitions for different applications.

For example, RGB color space is used in monitors, which is based on the emission of light (RGB corresponds to the three primary colors of light: Red, Green, Blue); CMY color space is commonly used in industrial printing, which is based on the reflection of light (CMY corresponds to the three primary colors in painting: Cyan, Magenta, Yellow); and HSV color space is created based on the intuitive characteristics of colors (HSV corresponds to Hue, Saturation, Value).

The RGB color space is shown in the following figure, which is a three-dimensional space containing Red, Green, and Blue. In image recognition, we can use a set of vectors to represent colors, such as using (0,0,0) to represent black and (255,255,255) to represent white. Here, 0-255 represents the color space being quantized into 256 values, with the lowest brightness value being 0 and the highest brightness value being 255. In this color space, there are 256*256*256 colors.



The HSV color space, as shown in the figure below, is a truncated cone model that includes hue, saturation, and value. The human visual system is more sensitive to brightness than color values, and the HSV color space is created based on the physiological characteristics of the human eye. The hue H has a value range of 0° to 360°, starting from red and counting counterclockwise, with red at 0°, green at 120°, and blue at 240°. The saturation S has a value range of 0.0 to 1.0, extending outward from the center of the circle. The value V has a value range of 0.0 (black) to 1.0 (white), extending from the top to the bottom of the truncated cone.



Tip: The RGB color space and the HSV color space can be converted to each other.

3. Image processing

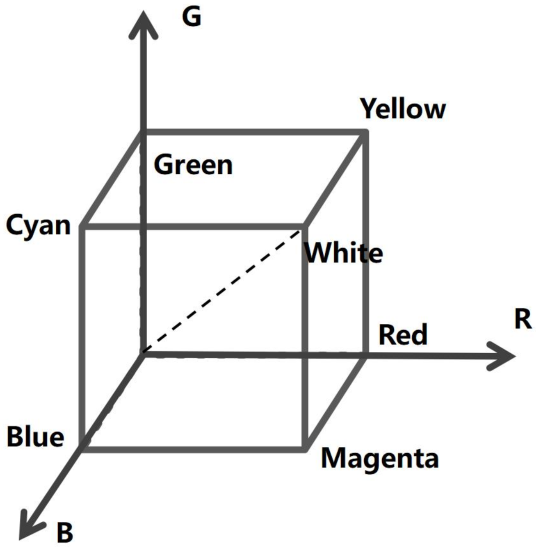
Image processing is an important part of image recognition. An image may contain a massive amount of ambiguous information, and the purpose of image processing is to eliminate irrelevant information, restore useful and real information, enhance the detectability of effective information, and simplify the data to the greatest extent possible. For example, in the image below, by enhancing the brightness of the image, the image becomes clearer.

There are many ways to process images, and here we will only briefly introduce some common ones.

(1) Image Grayscale

Image grayscale refers to displaying an image using different shades of black as the base color, usually ranging from 0% (white) to 100% (black) brightness values. Grayscale conversion can convert a color image into a grayscale image.

In the RGB model, when R=G=B, it represents a grayscale color, and the value of R=G=B is called the grayscale value. In the following figure, the dashed line in the cube represents the grayscale color when R=G=B. When R=G=B=0, the grayscale value is 0, and the color is black; when R=G=B=255, the grayscale value is 255, and the color is white.



(2) Image Binarization

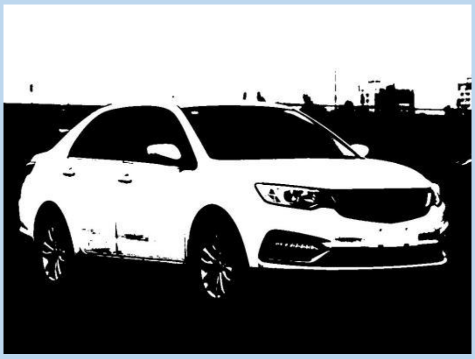
Image binarization is a special type of grayscale conversion, which sets the grayscale value to either 0 or 255, that is, the entire image is either black or white. The purpose of image binarization is to remove interference information as much as possible and obtain target information. Generally, the image is first grayscale converted and then binarized.

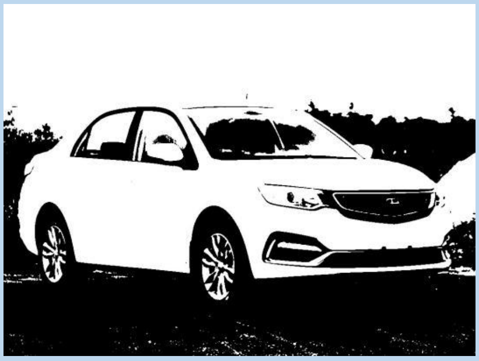
The most commonly used method for image binarization is to set a global threshold value T, which divides the image into two groups of pixels: those greater than T and those less than T, and then sets each group of pixels to either white or black. This method is called fixed thresholding.

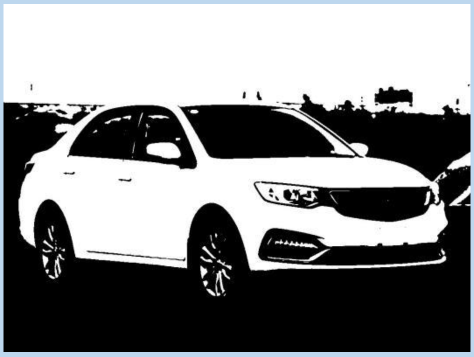
Different threshold values have a significant impact on the result of binarization, as shown in the following figure, where the image details are clearly different with different threshold values.

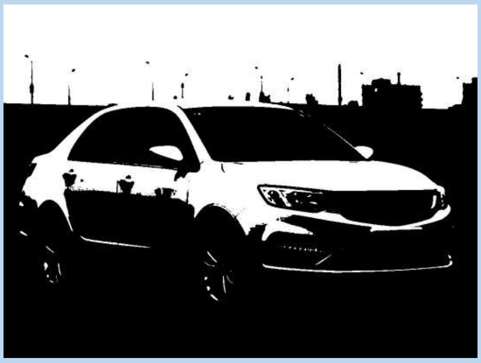
Normal image

grayscale image


binary threshold (127)


binary threshold (50)


binary threshold (100)


binary threshold (178)

(3) Bitwise Logical Operations on Images

Bitwise logical operations on images refer to performing bitwise logical operations on each pixel of two images, also known as bitwise operations.

There are four common bitwise operations: bitwise AND, bitwise OR, bitwise XOR, and bitwise NOT.

A. Bitwise AND Operation

Analogous to circuits, the AND operation means that the result is true only when both logical values are true. Bitwise AND operation is represented by the keyword "and", where 1 represents true and 0 represents false.

Operand 1	Operand 2	Result	Rule
0	0	0	and(0,0)=0
0	1	0	and(0,1)=0
1	0	0	and(1,0)=0
1	1	1	and(1,1)=1

Here is a table illustrating the bitwise AND operation, which involves converting numerical values into binary values and performing the AND operation at corresponding positions:

Numerical value	decimal	binary
Operand 1	198	1100 0110
Operand 2	219	1101 1011
Bitwise AND result	194	1100 0010

Characteristics of Bitwise AND operation:

Performing a bitwise AND operation between any numerical value N (ranging from 0-255) and the numerical value 0 (binary 0000 0000) will always result in the numerical value 0.

Performing a bitwise AND operation between any numerical value N (ranging from 0-255) and the numerical value 255 (binary 1111 1111) will always result in the original numerical value.

B. Bitwise OR operation

Analogous to a circuit, the OR operation states that if either of two logical values is true, the result is true. The bitwise OR operation is represented by the operator "or".

Operand 1	Operand 2	Result	Rule
0	0	0	or(0,0)=0
0	1	0	or(0,1)=1
1	0	0	or(1,0)=1
1	1	1	or(1,1)=1

Bitwise OR operation involves converting numerical values to their binary representation and performing OR operation on corresponding bits. Here is a table to illustrate the operation:

Numerical value	decimal	binary
Operand 1	198	1100 0110
Operand 2	219	1101 1011
Bitwise OR result	223	1101 1111

C. Bitwise NOT operation

The NOT operation involves the negation of a value. Analogous to a circuit, the result is true if the input is false, and vice versa. The bitwise NOT operation is represented by the operator "not".

Operand	Result	Rule
0	1	not(0)=1
1	0	not(1)=0

Bitwise NOT operation involves converting numerical values to their binary representation and performing NOT operation on corresponding bits. Here is a table to illustrate the operation:

Numerical value	decimal	binary
Operand	198	1100 0110
Bitwise NOT result	57	0011 1001

D. Bitwise XOR operation

The XOR operation, also known as half-adder, involves the comparison of two values. Analogous to a circuit, the result is true only if the two inputs are different. The bitwise XOR operation is represented by the operator "xor".

Operand 1	Operand 2	Result	Rule
0	0	0	xor(0,0)=0
0	1	0	xor(0,1)=1
1	0	0	xor(1,0)=1
1	1	1	xor(1,1)=0

Bitwise XOR operation involves converting numerical values to their binary representation and performing XOR operation on corresponding bits. Here is a table to illustrate the operation:

Numerical value	decimal	binary
Operand 1	198	1100 0110
Operand 2	219	1101 1011
Bitwise OR result	29	0001 1101

Color images and binary images can both undergo bitwise operations, which involves performing the operation on each pixel of two images to obtain the result for each pixel, and then displaying the final image. In a color image, each pixel is represented by an (R, G, B) value, which is first converted to binary values before performing the bitwise operation. For example, when performing bitwise XOR operation on pixel 1 (0, 198, 219) and pixel 2 (198, 219, 1), all values are first converted to binary values, as shown in the table below:

Pixel	R (decimal)	G (decimal)	B (decimal)	R (binary)	G (binary)	B (binary)
1	0	198	219	00000000	11000110	11011011
2	198	219	1	11000110	11011011	00000001

After the conversion, the bitwise XOR operation is performed on each corresponding bit, resulting in the following binary value:

Numerical value	R (decimal)	G (decimal)	B (decimal)
Operand 1	00000000	11000110	11011011
Operand 2	11000110	11011011	00000001
Bitwise OR result	11000110	00011101	11011010

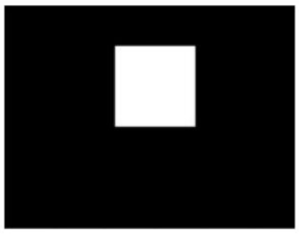
The result is converted back to decimal form, resulting in the pixel value (198, 29, 219), which represents the XOR result of pixel 1 (0, 198, 219) and pixel 2 (198, 219, 1). When performing bitwise operations on two color images, each pixel undergoes the operation to obtain the result for each pixel, and then the final image is displayed.

The same principle applies to grayscale images and binary images. It is important to note that only images with the same size can undergo bitwise operations. If two images have different sizes, they can be cropped to the same size before performing the operation.

(4) Image Masking

In simple terms, an image mask is a binary image used to locally mask another image. The binary image is generally referred to as the mask image.

For example, in the image below, the binary image on the left is used to mask the color image in the middle, resulting in the masked image shown on the right.


Mask image

Original image

Mask result

(5) Color Space Conversion of Images

In the basic properties of images, we introduced the RGB color space and the HSV color space. Some color spaces can be converted to each other, and there are two commonly used conversions: BGR ↔ Gray and BGR ↔ HSV. When segmenting images by color, the HSV color space is usually used.

For example, in the image below, by specifying green in the HSV color space, we can obtain the contour of the green pattern in Image 1 and process other parts of the image as black. We use Image 2 as a mask image and perform a masking operation with Image 1 to obtain Image 3, which segments the green part of Image 1.

Image 1

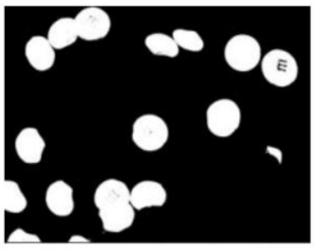


Image 2

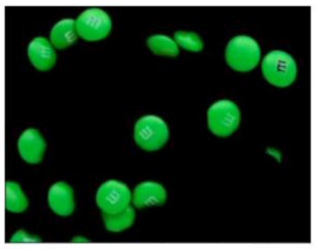
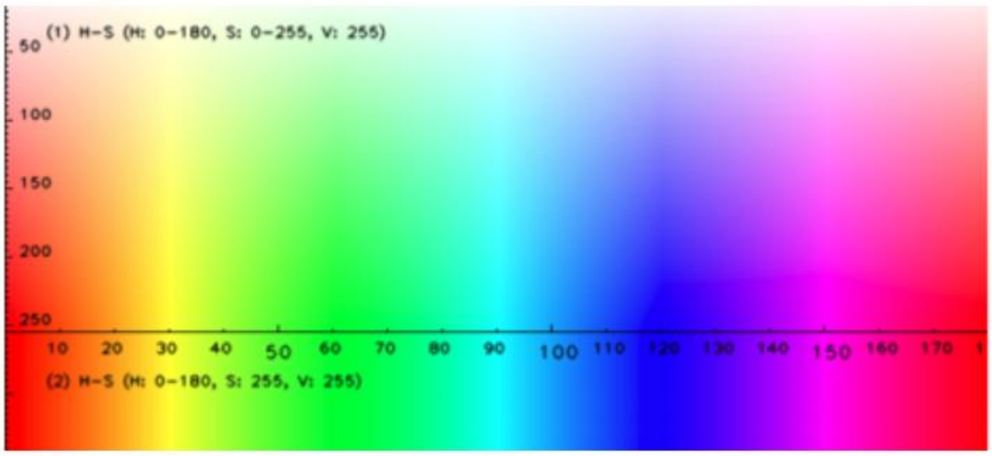


Image 3

The image below can be used for quick reference of HSV colors. The horizontal axis represents the hue H, with a value range of 0-180. Generally, the value of hue H can determine a range of colors, for example, H between 50-80 is green. By specifying the range of hue H, the mask method can be used to segment the specified color of the image.



4. What is the opencv Library?

opencv stands for Open Source Computer Vision Library. It is an open-source cross-platform computer vision library commonly used in the fields of image processing and computer vision. It can run on Linux, Windows, Android, and Mac OS.

opencv consists of a series of C functions and a small number of C++ classes. Therefore, it is lightweight and efficient. It also provides interfaces for languages such as Python, Ruby, and MATLAB. When using opencv in Python, we need to import it using "import cv2". "cv2" is the C++ namespace name of opencv, which is used to indicate that the interface being called is developed using C++.

5. Common Functions in the opencv Library

There are many functions in the opencv library, but we only use a small subset of them.

(1) The "imread()" Function for Reading Specified Images

The opencv library provides the "imread()" function, which can be used to read a specified image. To use this function, you need to first import the opencv library.

```
import cv2 # Import the OpenCV library

sample = cv2.imread("img/sample.png") # Read the image file "sample.png" from

the "img" folder, and name it "sample"
```

In this case, "img/sample.png" refers to the sample.png image located in the img folder.

Tips: The image obtained by using the "imread()" function is in the RGB color space and is arranged in the BGR order.

(2) The "namedWindow()" Function for Creating Windows

The opencv library provides the "namedWindow()" function, which can be used to create an image window, and name the window and set default properties.

```
cv2.namedWindow('winname',cv2.WND_PROP_FULLSCREEN) # Create a window n

amed "winname", with the default property of being able to display in full screen.
```

In this case, "winname" refers to the name of the image window, while "cv2.WND_PROP_FULLSCREEN" refers to the property that allows the window to be displayed in full screen.

(3) The "setWindowProperty()" Function for Setting Image Windows

The opencv library provides the "setWindowProperty()" function, which can be used to set certain properties for a specified image window.

```
cv2.setWindowProperty('winname',cv2.WND_PROP_FULLSCREEN, cv2.WINDOW_F
ULLSCREEN) # Set the "winname" window to be full screen.
```

In this case, "winname" refers to the name of the image window, "cv2.WND_PROP_FULLSCREEN" refers to the property that allows the window to be displayed in full screen, and "cv2.WINDOW_FULLSCREEN" refers to the setting for making the window full screen.

(4) The "imshow()" Function for Displaying Images

The opencv library provides the "imshow()" function, which can be used to display images on a specified window.

```
sample = cv2.imread("img/sample.png") # Read the "sample.png" image from the
"img" folder and assign it to the variable "sample"

cv2.namedWindow('winname', sample) # Display the image "sample" on the
window named "winname"
```

In this case, "winname" refers to the name of the window where the image will be displayed, and "sample" refers to the specific image that will be displayed.

(5) The "waitKey()" Function for Waiting for Keyboard Input

The opencv library provides the "waitKey()" function, which waits for keyboard input and can be used to refresh images. The function takes an argument in parentheses that represents the frequency time in milliseconds for waiting for a key press. If no argument is provided, the function defaults to waiting indefinitely for a key press, while displaying the current image.

```
cv2.waitKey()# Refreshes the image (waits for a user key event; if no argument is p
rovided, the function will wait indefinitely and display the original image). This co
mment explains the function.
```

(6) The "destroyAllWindows()" Function for Closing All Image Windows

The opencv library provides the "destroyAllWindows()" function, which can be used to close all image windows and is typically used when ending a program.

```
cv2.destroyAllWindows() # Close all windows.
```

(7) The "cvtColor()" Function for Converting Image Color Spaces

When using the "imread()" function to read an image in opencv, the image is in RGB color space and is arranged in BGR order. Therefore, in order to segment the image by color, we need to convert it to the HSV color space. This can be achieved using the "cvtColor()" function in the opencv library.

```
sample = cv2.imread("img/sample.png") # Read the image file "sample.png" from the "img" folder, and name it "sample"

hsv = cv2.cvtColor(sample,cv2.COLOR_BGR2HSV) # Convert the image from the BGR color space to the HSV color space, and name the new image "hsv".
```

In this case, "sample" refers to the original image read with the "imread()" function in RGB color space, "cv2.COLOR_BGR2HSV" refers to the method of converting the color space from BGR to HSV, and "hsv" is a variable used to store the image converted to the HSV color space.

(8) The "inRange()" Function for Color Thresholding

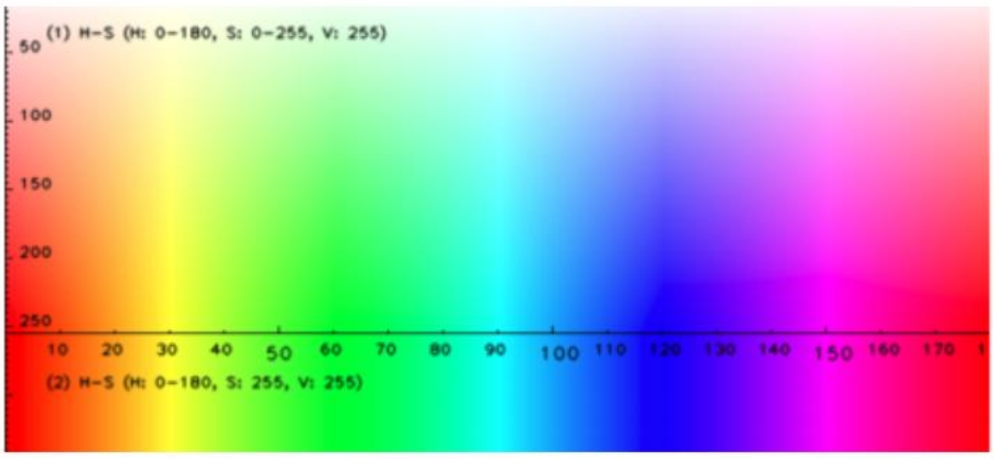
Color thresholding refers to finding the contours of a specified color in an image. To segment the color in an image, we need to first determine the color range.

The "inRange()" function in the opencv library can be used to perform color thresholding on images in the HSV color space.

```
l = 50

u = 80

sample = cv2.imread("img/sample.png")# Read the image "sample.png" from the "img" folder and name it "sample".

hsv = cv2.cvtColor(sample,cv2.COLOR_BGR2HSV) # Convert the color space of the image "sample" to HSV.
```

```
lower = np.array([l,90,90]) # Set the threshold lower limit.

upper = np.array([u,255,255]) # Set the threshold upper limit.

mask = cv2.inRange(hsv, lower, upper) # Remove the background by setting the i
mage values in "hsv" that are lower than "lower" or higher than "upper" to 0, and
setting the values between "lower" and "upper" to 255.
```

In this case, the value of "l" (50) and "u" (80) correspond to the range of hue "H". "lower" and "upper" are the two color values that are used to define the color range, as shown in the green area in the following image. "hsv" is the image in the HSV color space, and "mask" is a variable used to store the image after color thresholding.



(9) Bitwise Logical Operations on Images

The four common bitwise logical operations on images can be implemented in opencv using the following functions: "cv2.bitwise_and()", "cv2.bitwise_or()", "cv2.bitwise_xor()", and "cv2.bitwise_not()".

basic meaning	function name
Bitwise AND	cv2.bitwise_and()
Bitwise OR	cv2.bitwise_or()
Bitwise XOR	cv2.bitwise_xor()
Bitwise NOT	cv2.bitwise_not()

A. The "cv2.bitwise_and()" function performs bitwise AND operation.

```
color = np.zeros((320, 240, 3), dtype=np.uint8) # Create a three-dimensional matri
x of zeros with a data type of uint8.
color[:,:,0] = 12 # B # Traverse all rows and columns, and assign the value of the bl
ue channel to 12.
color[:,:,1] = 12 # G # Traverse all rows and columns, and assign the value of the gr
een channel to 12.
color[:,:,2] = 12 # R # Traverse all rows and columns, and assign the value of the re
d channel to 12.
mask = cv2.inRange(hsv, lower, upper) # Threshold the image based on color.
B2 = cv2.bitwise_and(color, color, mask=mask) # Perform a bitwise AND operation
.
```

Here, "color" refers to the original image, "mask" is another image, and "B2" is the result of performing bitwise AND operation between the two original images, followed by another bitwise AND operation between the resulting image and the mask image, to obtain the final image.

B. The "cv2.bitwise_or()" function performs bitwise OR operation.

```
color = np.zeros((320, 240, 3), dtype=np.uint8) # Create a three-dimensional matri
x of zeros with a data type of uint8.
color[:,:,0] = 12 # B # Traverse all rows and columns, and assign the value of the bl
ue channel to 12.
color[:,:,1] = 12 # G # Traverse all rows and columns, and assign the value of the gr
```

```
een channel to 12.

color[:,:,2] = 12 # R # Traverse all rows and columns, and assign the value of the re

d channel to 12.

mask = cv2.inRange(hsv, lower, upper) # Threshold the image based on color.

B3 = cv2.bitwise_or(color, mask) # Perform a bitwise OR operation.
```

Here, "color" refers to the original image, "mask" is another image, and "B3" is the result of performing bitwise OR operation between the two images.

C. The "cv2.bitwise_not()" function is used to perform bitwise NOT operation.

```
color = np.zeros((320, 240, 3), dtype=np.uint8) # Create a three-dimensional matri

x of zeros with a data type of uint8.

color[:,:,0] = 12 # B # Traverse all rows and columns, and assign the value of the bl

ue channel to 12.

color[:,:,1] = 12 # G # Traverse all rows and columns, and assign the value of the gr

een channel to 12.

color[:,:,2] = 12 # R # Traverse all rows and columns, and assign the value of the re

d channel to 12.

mask = cv2.inRange(hsv, lower, upper) # Threshold the image based on color.

B4 = cv2.bitwise_not(object_mask) # Perform a bitwise NOT operation.
```

In this context, "color" refers to the original image, "mask" is another image, and "B4" is the resulting image obtained by performing bitwise NOT operation on the two images.

6. Creating a Zero Matrix with the numpy "zeros()" Function

numpy is a scientific computing library in Python. We have already learned about the "array()" function, which can be used to create an array (a two-dimensional matrix). In addition to this, there are many other functions available in numpy.

(1) The "zeros()" function can be used to create a multi-dimensional zero matrix in numpy. To use this function, we need to first import the library.

```
import numpy as np

color = np.zeros((320, 240, 3), dtype=np.uint8) # Create a three-dimensional matri

x of zeros with a data type of uint8.
```

In this context, "320" refers to the first dimension, "240" refers to the number of rows, and "3" refers to the number of columns. "uint8" is an 8-bit unsigned integer data type that represents integers in the range of [0, 255]. It is specifically used for storing matrices of various images. To perform various operations on the current matrix as an image type, the data type must be defined as "uint8".

7. Obtaining Color Values with the Pinpong Library

The TCS34725 module in the pinpong.libs.dfrobot_tcs34725 package is a Python library specifically designed for the TCS34725 color sensor. It can be used to obtain color values. To use it, the module needs to be imported and the TCS34725 class needs to be instantiated.

(1) The "begin()" method is used to confirm whether the initialization connection is successful in the TCS34725 class. If the connection is successful, it will return True, otherwise it will return False.

```
from pinpong.libs.dfrobot_tcs34725 import TCS34725 # Import the TCS34725 mo

dule from the pinpong.libs.dfrobot_tcs34725 package.

tcs = TCS34725() # Instantiate the TCS34725 class to create a tcs object (color sen

sor).

if tcs.begin(): # Initialize the color sensor and return True if it is detected.

    print("Sensor detected") # Print "Sensor detected" if the sensor is detected.
```

(2) Detecting Color Values with the "get_rgbc()" Method

The "get_rgbc()" method in the TCS34725 class can be used to detect the RGBC color values and obtain the data. It returns the value of red (R) first, followed by green (G), blue (B), and finally clear (C). However, the R, G, and B values detected here need to be converted to the actual color values using a data conversion method.

```
r1, g1, b1, c1 = tcs.get_rgbc() # Get the RGBC data.

# Data conversion

if c1:

    r1 /= c1

    g1 /= c1

    b1 /= c1

    r1 *= 256

    g1 *= 256

    b1 *= 256
```

Here, r1, g1, and b1 are variables used to store the R, G, and B values, respectively, while c1 represents the clear value, which refers to the raw light that has been filtered of infrared light. The R, G, and B values are calculated as a proportion of the clear value, and then mapped to a value between 0 and 256 based on this proportion to obtain the corresponding color value.

Hands-on practice

Task Description 1: Controlling Color Changes with Terminal Input

Manually input the RGB values, and display the corresponding color on the character in the image, achieving a chameleon-like color-changing effect.

1. Hardware setup

Connect the UNIHIKER to the computer via a USB cable.

2. program coding

STEP1: Creating and Saving Project Files

Launch Mind+ and save the project as "011 Chameleon Screens".

STEP2: Creating and Saving Python Files

Create a Python program file named "main1.py" and double-click to open it.

STEP3: Import image folder

Import the "img" folder containing the background images into the project folder.

STEP 4: Programming

(1) Importing Required Libraries

For this task, we need to use the opencv library to process images, the numpy library to handle data, and the time library to set delays. Therefore, we need to import the corresponding libraries first.

```
import cv2 # Import the OpenCV library

import numpy as np # Import the NumPy library

import time
```

(2) Reading the Image, Creating an Image Window, and Setting it to Full Screen

The chameleon effect is based on image processing, so here we need to read the image, create a window for displaying the image, and set it to full screen mode so that the chameleon can be displayed on the screen.

```
sample = cv2.imread("img/sample.png") # Read the image file "sample.png" from

the "img" folder, and name it "sample"

cv2.namedWindow('winname',cv2.WND_PROP_FULLSCREEN) # Create a window n

amed "winname", with the default property of being able to display in full screen.

cv2.setWindowProperty('winname',cv2.WND_PROP_FULLSCREEN, cv2.WINDOW_F

ULLSCREEN) # Set the "winname" window to be full screen.
```

(3) Setting the Upper and Lower Limits for Hue (H)

To enable the chameleon character in the image to change color freely, we need to first identify the "chameleon" part in the image. In this image, the background is white and the chameleon is blue. Therefore, we can circle the chameleon by specifying the range of hue (H).

```
'''Use the HSV color space to create an image mask'''

# Define a blue mask, specifying the range of hue values from l to u. (In the origin
al image, the part of the chameleon that changes color is blue.)

l = 100

u = 140
```

(4) Defining a Function to Obtain the Mask Image

Next, we need to obtain a mask image. Here, we can organize the steps for obtaining the mask image into a function for convenient later use.

```
# Define a function to obtain the mask image. This function takes an image, l, and
u as input parameters.

def get_hsv_mask(img, l, u):

    hsv = cv2.cvtColor(img,cv2.COLOR_BGR2HSV) # Convert the image from the BG
R color space to the HSV color space, and name the new image "hsv".

    lower = np.array([l,90,90]) # Define the lower threshold values.

    upper = np.array([u,255,255]) # Define the upper threshold values.

    mask = cv2.inRange(hsv, lower, upper) # Remove the background, setting the pi
xel values in the "hsv" image that are below "lower" or above "upper" to 0, and set
ting the pixel values between "lower" and "upper" to 255.
```

```
    return mask  # Return the mask image. This image is in black and white, with a
black background and a white chameleon.
```

(5) Defining a Function to Display the Chameleon Image with a Specified Color

Afterwards, we define a function to display the "chameleon" part of the chameleon image with a specified color. To achieve this, we will perform multiple steps. First, we create a 3-dimensional zero matrix to represent a new image and assign the specified RGB color value to it, forming a color image. Next, we obtain a mask image of the original image through color thresholding. Then, we perform a bitwise AND operation between the custom color image and itself, and then perform an AND operation between the result and the mask image. After that, we invert the original mask image bitwise to obtain a new mask image. Then, we perform a bitwise AND operation between the original image and itself, and then perform an AND operation between the result and the new mask image. Afterwards, we perform a bitwise OR operation between the resulting image and the previously defined custom color image to obtain the chameleon image with the desired color. Finally, we display the image in a window.

```
# Define a function to obtain the mask image. This function takes an image, l, and
u as input parameters.

def get_hsv_mask(img, l, u):

    hsv = cv2.cvtColor(img,cv2.COLOR_BGR2HSV) # Convert the image from the BG
R color space to the HSV color space, and name the new image "hsv".

    lower = np.array([l,90,90]) # Define the lower threshold values.

    upper = np.array([u,255,255]) # Define the upper threshold values.

    mask = cv2.inRange(hsv, lower, upper) # Remove the background, setting the pi
xel values in the "hsv" image that are below "lower" or above "upper" to 0, and set
ting the pixel values between "lower" and "upper" to 255.

    return mask  # Return the mask image. This image is in black and white, with a
```

```
black background and a white chameleon.



# Define a function to display the chameleon image with the specified color.

def brg(r,g,b):

    # RGB background

    color =  np.zeros((320, 240, 3),dtype=np.uint8) # Create a three-dimensional zer
o matrix with a data type of uint8.

    color[:,:,0] = b # B # Set the blue channel values of all pixels to "b".

    color[:,:,1] = g # G # Set the green channel values of all pixels to "g".

    color[:,:,2] = r # R # Set the red channel values of all pixels to "r".


    '''Use color thresholding to obtain the outline of the chameleon pattern---obtai
n the mask image (black background, white chameleon)'''

    # Obtain the mask image (black background, white chameleon) (Image 1).

    object_mask = get_hsv_mask(sample,l,u)


    '''Black is (0,0,0), and the result of performing a logical AND operation with any
value is still 0. White is (255,255,255), and the result of performing a logical AND o
peration with any value is still the value itself.'''
```

```
    # Perform a logical AND operation between the "color" image and itself, and th
en perform a logical AND operation between the result (which is still the "color" i
mage) and the mask image (Image 1), to obtain the black background color cham
eleon image (Image 2).
    background_masked = cv2.bitwise_and(color,color, mask=object_mask)

    # Perform a bitwise NOT operation on the original mask image (Image 1), to ob
tain a white background black chameleon image, which is a new image (Image 3).
    object_mask_not = cv2.bitwise_not(object_mask)

    # Perform a logical AND operation between the original image and itself, and th
en perform a logical AND operation between the result (which is still the original i
mage, with a white background and a blue chameleon) and Image 3 (white backgr
ound and black chameleon) (mask operation), to obtain a white background black
 chameleon image (Image 4).
    object_masked = cv2.bitwise_and(sample,sample, mask=object_mask_not)

    # Perform a bitwise OR operation between Image 2 (black background color ch
ameleon) and Image 4 (white background black chameleon), to obtain the white b
ackground color chameleon image (Image 5--the final result image), and name it "
final_output".
```

```
    final_output = cv2.bitwise_or(background_masked,object_masked)


    # Display the image and keep it on the screen.

    cv2.imshow('winname', final_output) # Display the "final_output" image on the "
winname" window.

    cv2.waitKey() # Refresh the image (wait for the user to press a key to trigger the
 refresh. If no argument is specified, the default value of 0 is used, which means th
at the image will be displayed indefinitely, showing the initial image).
```

(6) Keyboard Input for Color Selection

Finally, we set up a keyboard input in the terminal to select the desired color value, and then call the above function to achieve the chameleon color-changing effect.

```
'''Allow the user to manually input color values in the terminal, and map them to t
he chameleon's body.'''

r,g,b=map(int,input('Enter r,g,b separated by spaces:').split()) # Accept multiple us
er input values and convert them to integers. The values are separated by spaces
when entered.



brg(r,g,b) # Call the "brg" function to display the chameleon image with the specif
ied color. The specified color values are entered via the keyboard.



cv2.destroyAllWindows() # Close all windows.
```

Tips: The complete example program is as follows:

```
import cv2 # Import the OpenCV library
import numpy as np # Import the NumPy library
import time

sample = cv2.imread("img/sample.png") # Read the image file "sample.png" fro
m the "img" folder, and name it "sample"
cv2.namedWindow('winname',cv2.WND_PROP_FULLSCREEN) # Create a window
named "winname", with the default property of being able to display in full scree
n.
cv2.setWindowProperty('winname',cv2.WND_PROP_FULLSCREEN, cv2.WINDOW_
FULLSCREEN) # Set the "winname" window to be full screen.

'''Use the HSV color space to create an image mask'''
# Define a blue mask, specifying the range of hue values from l to u. (In the origi
nal image, the part of the chameleon that changes color is blue.)
l = 100
u = 140

# Define a function to obtain the mask image. This function takes an image, l, an
d u as input parameters.
def get_hsv_mask(img, l, u):
    hsv = cv2.cvtColor(img,cv2.COLOR_BGR2HSV) # Convert the image from the B
```

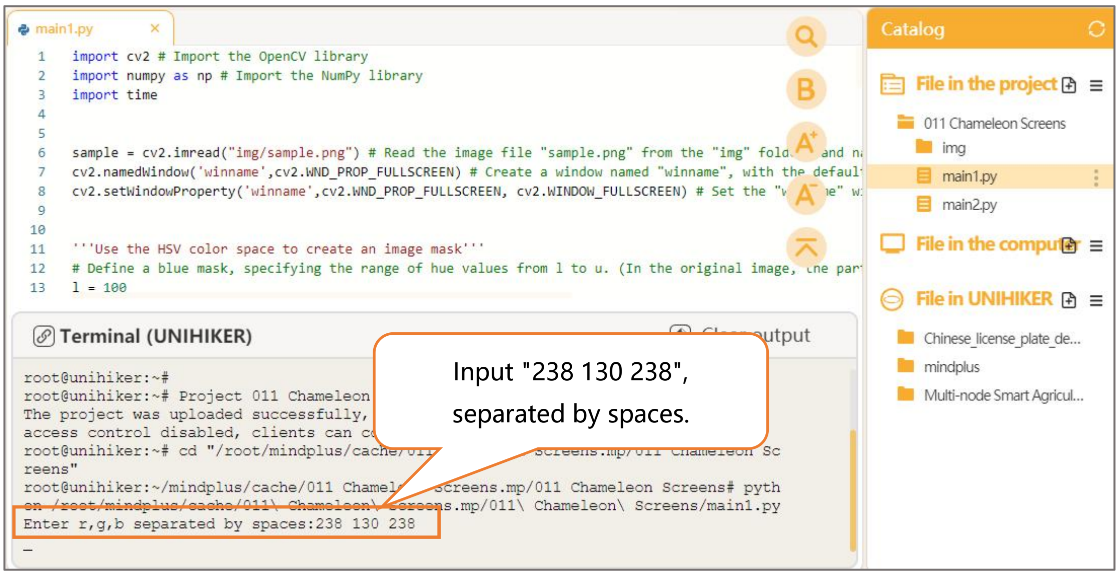
```
GR color space to the HSV color space, and name the new image "hsv".
    lower = np.array([l,90,90]) # Define the lower threshold values.
    upper = np.array([u,255,255]) # Define the upper threshold values.
    mask = cv2.inRange(hsv, lower, upper) # Remove the background, setting the
pixel values in the "hsv" image that are below "lower" or above "upper" to 0, and
setting the pixel values between "lower" and "upper" to 255.
    return mask  # Return the mask image. This image is in black and white, with a
 black background and a white chameleon.

# Define a function to display the chameleon image with the specified color.
def brg(r,g,b):
    # RGB background
    color =  np.zeros((320, 240, 3),dtype=np.uint8) # Create a three-dimensional z
ero matrix with a data type of uint8.
    color[:,:,0] = b # B # Set the blue channel values of all pixels to "b".
    color[:,:,1] = g # G # Set the green channel values of all pixels to "g".
    color[:,:,2] = r # R # Set the red channel values of all pixels to "r".

    '''Use color thresholding to obtain the outline of the chameleon pattern---obta
in the mask image (black background, white chameleon)'''
    # Obtain the mask image (black background, white chameleon) (Image 1).
    object_mask = get_hsv_mask(sample,l,u)
    '''Black is (0,0,0), and the result of performing a logical AND operation with any
 value is still 0. White is (255,255,255), and the result of performing a logical AND
 operation with any value is still the value itself.'''
    # Perform a logical AND operation between the "color" image and itself, and t
hen perform a logical AND operation between the result (which is still the "color"
 image) and the mask image (Image 1), to obtain the black background color cha
meleon image (Image 2).
    background_masked = cv2.bitwise_and(color,color, mask=object_mask)

    # Perform a bitwise NOT operation on the original mask image (Image 1), to o
btain a white background black chameleon image, which is a new image (Image
3).
    object_mask_not = cv2.bitwise_not(object_mask)

    # Perform a logical AND operation between the original image and itself, and t
hen perform a logical AND operation between the result (which is still the origina
l image, with a white background and a blue chameleon) and Image 3 (white bac
kground and black chameleon) (mask operation), to obtain a white background b
lack chameleon image (Image 4).
    object_masked = cv2.bitwise_and(sample,sample, mask=object_mask_not)

    # Perform a bitwise OR operation between Image 2 (black background color c
hameleon) and Image 4 (white background black chameleon), to obtain the whit
```

```
e background color chameleon image (Image 5--the final result image), and nam
e it "final_output".
    final_output = cv2.bitwise_or(background_masked,object_masked)

    # Display the image and keep it on the screen.
    cv2.imshow('winname', final_output) # Display the "final_output" image on the
"winname" window.
    cv2.waitKey() # Refresh the image (wait for the user to press a key to trigger th
e refresh. If no argument is specified, the default value of 0 is used, which means
that the image will be displayed indefinitely, showing the initial image).

'''Allow the user to manually input color values in the terminal, and map them to
the chameleon's body.'''
r,g,b=map(int,input('Enter r,g,b separated by spaces:').split()) # Accept multiple u
ser input values and convert them to integers. The values are separated by space
s when entered.


brg(r,g,b) # Call the "brg" function to display the chameleon image with the speci
fied color. The specified color values are entered via the keyboard.

cv2.destroyAllWindows() # Close all windows.
```

3. Running the Program

STEP1： Remote Connect to the UNIHIKER, Run the Program, and Observe the Results

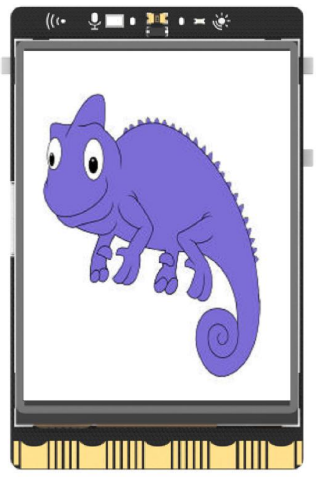
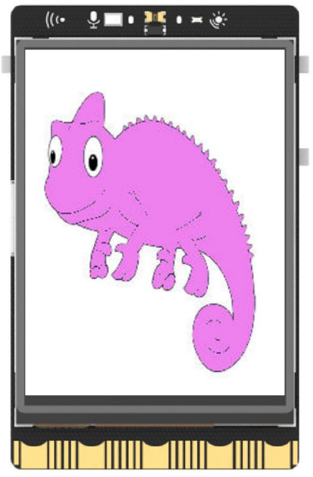
By observing the Mind+ terminal, we can see a prompt for inputting color RGB values. Here, we can input any color value, such as "238 130 238", and then press the Enter key.



```python
import cv2 # Import the OpenCV library
import numpy as np # Import the NumPy library
import time


sample = cv2.imread("img/sample.png") # Read the image file "sample.png" from the "img" fold   and na
cv2.namedWindow('winname',cv2.WND_PROP_FULLSCREEN) # Create a window named "winname", with the defaul
cv2.setWindowProperty('winname',cv2.WND_PROP_FULLSCREEN, cv2.WINDOW_FULLSCREEN) # Set the "w   ne" wi


'''Use the HSV color space to create an image mask'''
# Define a blue mask, specifying the range of hue values from l to u. (In the original image, the par
l = 100
```

```
root@unihiker:~#
root@unihiker:~# Project 011 Chameleon
The project was uploaded successfully,
access control disabled, clients can co
root@unihiker:~# cd "/root/mindplus/cache/011                  Screens.mp/011 Chameleon Sc
reens"
root@unihiker:~/mindplus/cache/011 Chamele    Screens.mp/011 Chameleon Screens# pyth
on /root/mindplus/cache/011\ Chameleon\ Screens.mp/011\ Chameleon\ Screens/main1.py
Enter r,g,b separated by spaces:238 130 238
_
```

After observing the UNIHIKER, we can see that the main body of the "chameleon" has changed to the corresponding color we inputted.

Original image	image after color modification

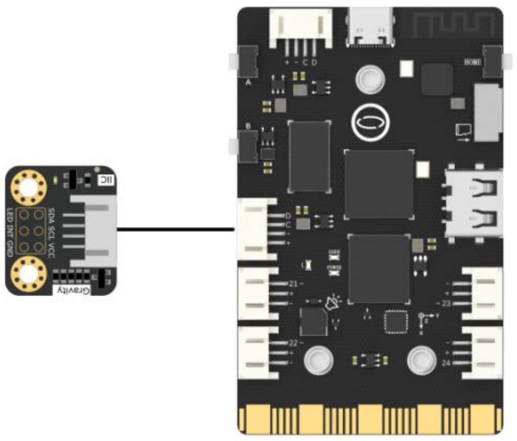
Tips: If you want to input a different color value to experience a different effect, you need to rerun the program.

Task Description 2: Using Sensors to Control Color Change Effect

In the previous task, we controlled the color change of the "chameleon" by inputting a specified RGB color value and displaying it on the screen. Now, we will use sensors to collect color values and reflect them on the "chameleon".

1. Hardware setup

STEP1: Connect the color sensor to the I2C pins on the UNIHIKER.



2. program coding

STEP1: Creating and Saving Python Files

Create a Python program file named "main2.py" and double-click to open it.

STEP 2: Programming

In this task, we will replace the keyboard input of color values with real-time detection using a color sensor. Therefore, we only need to make some adjustments in the programming.

(1) Import the Pinpong library color sensor module.

```
from pinpong.board import Board, Pin # Import the Board and Pin modules from t
he pinpong.board package
```

```
from pinpong.extension.unihiker import * # Import all modules from the pinpong.

extension.unihiker package

from pinpong.libs.dfrobot_tcs34725 import TCS34725 # Import the TCS34725 mo

dule from the pinpong.libs.dfrobot_tcs34725 package
```

(2) Add code to initialize the UNIHIKER board and create a color sensor object.

```
Board().begin() # Initialize the board

tcs = TCS34725() # Instantiate the TCS34725 class to create a color sensor object n

amed "tcs"
```

(3) Add code to initialize the color sensor and confirm if it is connected successfully.

```
# Confirm that the color sensor is successfully connected by initializing it

while True:

    if tcs.begin(): # If the color sensor is successfully initialized and read, return True

        print("Found sensor") # Print "Found sensor"

        break # Exit the loop

    else: # Otherwise

        print("No TCS34725 found ... check your connections") # Print "No TCS34725

found ... check your connections"

        time.sleep(1)
```

(4) Add code to define a function that retrieves RGB values from the color sensor.

```
# Define a function to get RGB color values from the sensor

def get_RGB(tcs):

    r1, g1, b1, c1 = tcs.get_rgbc() # Get the rgbc data

    # Convert the data

    if c1:

        r1 /= c1

        g1 /= c1

        b1 /= c1

        r1 *= 256

        g1 *= 256

        b1 *= 256

    return r1, g1, b1 # Return the values of r, g, and b
```

(5) Real-time Color Detection and Color Change

Finally, we call the color sensor detection function and the color change display function in a loop to obtain real-time color values and display the color change effect on the image window. We also set a key press event to stop the program by pressing the "b" key.

```
while True:

    r1, g1, b1 = get_RGB(tcs) # Call the get_RGB() function to obtain the RGB values
 of the color through the sensor detection

    brg(r1, g1, b1) # Call the brg function to display the chameleon image with the
```

```
specified color, and the specified color value is obtained from the sensor above

    print((b1, g1, r1)) # Print the color value



    # Press the 'b' key to stop the program

    if cv2.waitKey(10) & 0xFF == ord('b'):

        break


cv2.destroyAllWindows() # Close all windows
```

Tips: The complete example program is as follows:

```
import cv2 # Import the OpenCV library

import numpy as np # Import the NumPy library

import time

from pinpong.board import Board, Pin # Import the Board and Pin modules from

 the pinpong.board package

from pinpong.extension.unihiker import * # Import all modules from the pinpong

.extension.unihiker package

from pinpong.libs.dfrobot_tcs34725 import TCS34725 # Import the TCS34725 mo

dule from the pinpong.libs.dfrobot_tcs34725 package
```

```python
sample = cv2.imread("img/sample.png") # Read the "sample.png" image from the "img" folder and assign it to the variable "sample"

cv2.namedWindow('winname', cv2.WND_PROP_FULLSCREEN) # Create a window named "winname" with the default property of being able to display in full screen

cv2.setWindowProperty('winname', cv2.WND_PROP_FULLSCREEN, cv2.WINDOW_FULLSCREEN) # Set the "winname" window to display in full screen


Board().begin() # Initialize the board

tcs = TCS34725() # Instantiate the TCS34725 class to create a color sensor object named "tcs"


# Confirm that the color sensor is successfully connected by initializing it

while True:

    if tcs.begin(): # If the color sensor is successfully initialized and read, return True

        print("Found sensor") # Print "Found sensor"

        break # Exit the loop

    else: # Otherwise

        print("No TCS34725 found ... check your connections") # Print "No TCS3472
```

```
5 found ... check your connections"

        time.sleep(1)



# Define a function to get RGB color values from the sensor

def get_RGB(tcs):

    r1, g1, b1, c1 = tcs.get_rgbc() # Get the rgbc data

    # Convert the data

    if c1:

        r1 /= c1

        g1 /= c1

        b1 /= c1

        r1 *= 256

        g1 *= 256

        b1 *= 256

    return r1, g1, b1 # Return the values of r, g, and b



'''Implementing Image Masking using HSV Color Space'''

# Define a blue mask by specifying the range of hue values (l as the lower limit a
nd u as the upper limit)

# (The part of the chameleon in the original image is blue)

l = 100

u = 140


# Organize the steps for obtaining the masked image into a function for easier ca

lling in the future

# Define a function to get the masked image

def get_hsv_mask(img, l, u): # Define the function get_hsv_mask and pass in para

meters img, l, u

    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV) # Convert the BGR color space

of the image img to the HSV color space and name the new image hsv

    lower = np.array([l, 90, 90]) # Set the threshold, lower limit

    upper = np.array([u, 255, 255]) # Set the threshold, upper limit

    mask = cv2.inRange(hsv, lower, upper)  # Remove the background, set the ima

ge values in hsv that are lower than lower or higher than upper to 0, and set the

values between lower and upper to 255

    return mask # Return the masked image---black and white image---black back

ground and white chameleon
```

```
# Define a function to display a chameleon image with the specified color
def brg(r, g, b):
    # RGB background
    color = np.zeros((320, 240, 3), dtype=np.uint8) # Create a three-dimensional z
ero matrix with a type of uint8
    color[:, :, 0] = b # B # Traverse all rows and columns, assign the value of the bl
ue channel as b
    color[:, :, 1] = g # G # Traverse all rows and columns, assign the value of the gr
een channel as g
    color[:, :, 2] = r # R # Traverse all rows and columns, assign the value of the red
 channel as r

    '''Obtaining the Chameleon Pattern Contour through Color Thresholding---Get
ting the Masked Image (Black Background, White Chameleon)'''
    # Get the masked image---black background, white chameleon (Image 1)
    object_mask = get_hsv_mask(sample, l, u)

    '''Black is (0,0,0), and an operation with any value and 0 will still result in 0. Whi
te is (255,255,255), and an operation with any value and it will still result in the va
```

```
lue itself.'''
    # Perform a bitwise AND operation on the color image with itself first, then pe
rform the operation on the resulting image (which is still the color image) and th
e masked image (Image 1) to obtain the black background color chameleon (Ima
ge 2)
    background_masked = cv2.bitwise_and(color, color, mask=object_mask)

    # Take the bitwise NOT of the original mask image (white background, black c
hameleon) to obtain the black background, white chameleon image, which is the
new image (Image 3)
    object_mask_not = cv2.bitwise_not(object_mask)

    # Perform a bitwise AND operation on the original image with itself first, then
perform the operation on the resulting image (which is still the original image, w
hite background, blue chameleon) and Image 3 (white background, black chamel
eon) to obtain the white background, black chameleon image (Image 4) through
mask operation
    object_masked = cv2.bitwise_and(sample, sample, mask=object_mask_not)

    # Perform a bitwise OR operation on Image 2 (black background, color chamel
eon) and Image 4 (white background, black chameleon) to obtain the white back
```

```
ground, color chameleon image (Image 5---final result image) and name it final_
output
    final_output = cv2.bitwise_or(background_masked, object_masked)


# Display the image
    cv2.imshow('winname', final_output) # Display the final_output image on the w
inname window
#cv2.waitKey(5)  # Refresh the image every 5ms


while True:
    r1, g1, b1 = get_RGB(tcs) # Call the get_RGB() function to obtain the RGB value
s of the color through the sensor detection
    brg(r1, g1, b1) # Call the brg function to display the chameleon image with the
 specified color, and the specified color value is obtained from the sensor above
    print((b1, g1, r1)) # Print the color value


    # Press the 'b' key to stop the program
    if cv2.waitKey(10) & 0xFF == ord('b'):
        break
```

```
cv2.destroyAllWindows() # Close all windows
```

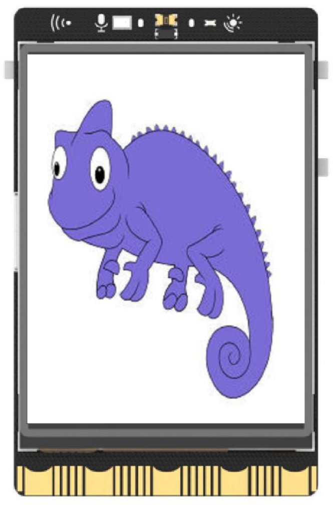
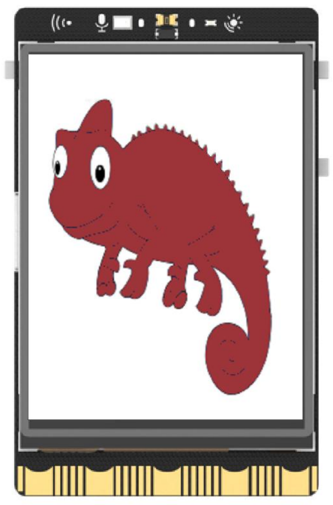
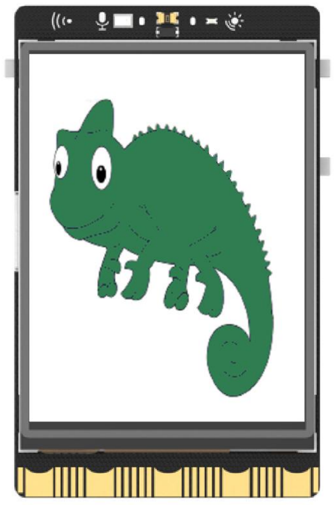
3. Running the Program

STEP 1: Connect to the UNIHIKER remotely and run the program.

STEP 2: Color Detection Using the Sensor

By bringing the color sensor close to objects of different colors, you can observe that the "dragon" on the board quickly changes to match the color of the object.

Tips: Place the sensor in front of the object being measured, with a distance between 3-10mm.

		
Original image	image after color modification 1	image after color modification 2

Challenge Yourself

In Task 1, after displaying the color change effect once, if we want to experience the effect of other color values, we need to run the program again and input new RGB values. Can we optimize the program by adding a button control, so that we can restart the program every time we press the on-board "A" button?