Drawing and importing sprites

Sprites

Overview

In this tutorial, you'll learn how to draw your own icons and images, convert them to the Ringo compatible format and use them in your games and apps.

Since most of the things on Ringo are really small in pixel size (**the screen itself is 160x128**), pretty much everything you draw will be low-res and there will be no way to make something uber-complex. However, lack of pixels can sometimes cause real troubles when you're trying to draw shapes that are not that simple.

Bored of the default icons? No worries, you'll soon know how to draw and import new ones!

Luckily for us, the Internet is full of these things and there are plenty of pixel-like drawings that are free to use and available to everyone. You can pretty much find everything you want - from the icons all the way to the monsters in games.

Most of this tutorial will be based on how to make your own sprites. To do that, we'll be using **GNU Image Manipulation Program** or simply **GIMP.**

GIMP is one of the world's most versatile and popular graphics editors and the best thing about it is - it's free to use and open-source!

The installation of this software is pretty straightforward and we'll not go through it here, so make sure you download it from **here** and then come back.
Of course, you can use other editors, such as **Photoshop** or even **Aesprite,** which is probably the best pixel-art tool on the market. These programs are not free to use, so we'll focus on the one that is the most accessible to everyone.

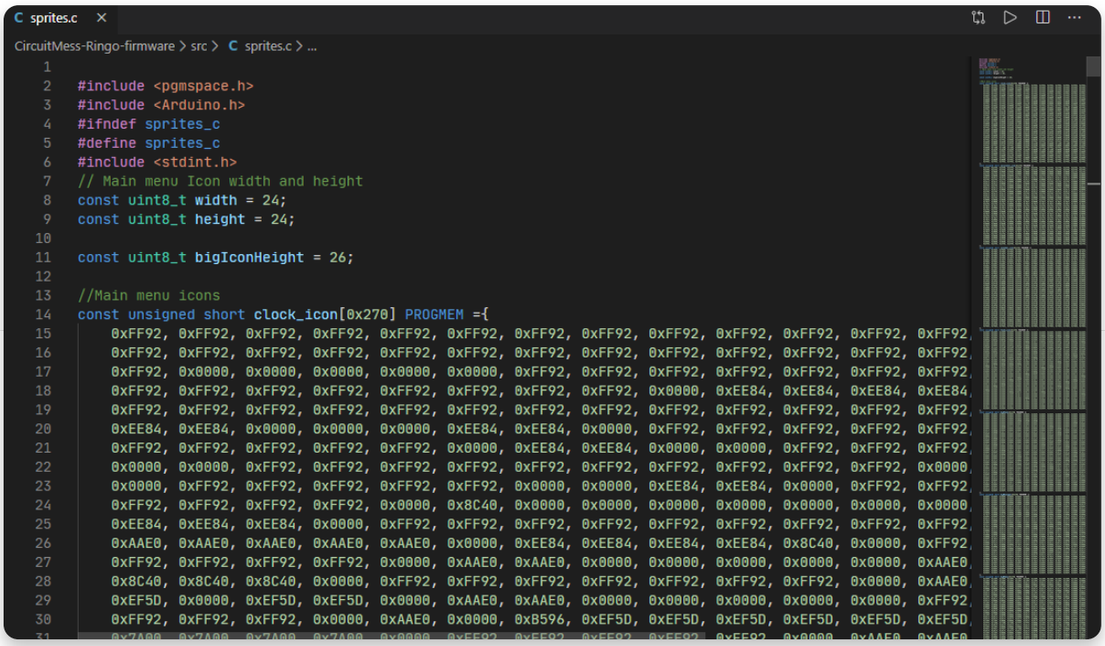
Let's go!

How do sprites work?

You may be wondering how do icons and indicators get loaded from Ringo. Are they just imported from the SD card? In which format are they?

Well actually, pretty much everything you see on Ringo is clear code. Backgrounds, simple shapes, and individual pixels are being drawn directly to the screen and everything else that is a little bit more complex is located in one very specific file called **sprites.c.**

Every icon and image is actually converted in pure code that corresponds with **RGB565 bitmap**, but more on that a little bit later.

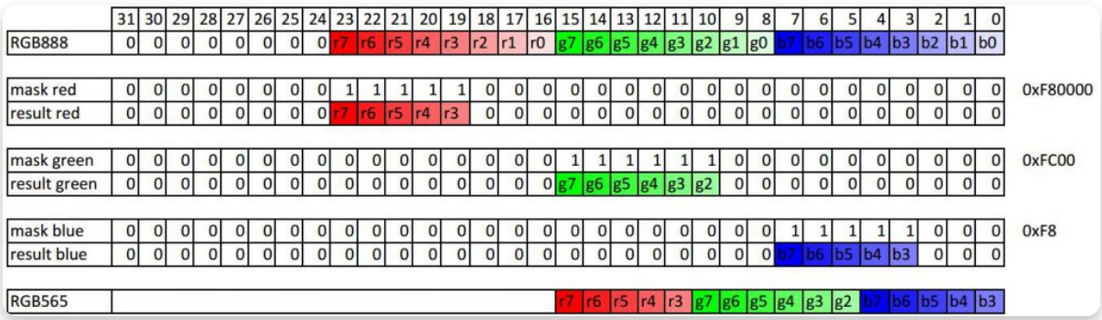


This is how sprites.c looks like in VSCode editor

At first, it seems that these are just some random hexadecimal values, but actually they are **color codes**.

> **RGB565**
>
> There any many different ways the colors are defined digitally. Probably the most popular encoding is RGB, which actually means RED, GREEN, and

BLUE. Every color is represented by the mix of those three colors. RGB565 uses a total of 16 bits to code one specific color. 5 bits are used for both red and blue values, while 6 bits are used for the green value. All in all, those 16 bits are represented with a 4-digit hexadecimal number since one hex digit is 4 binary digits. For example, code 0xF800 is translated to RGB(128,0,0) which would translate to clean red. This way of coding is much more memory efficient than RGB888, which uses 8 bits of information for each color, and it still allows for a huge array of different colors.

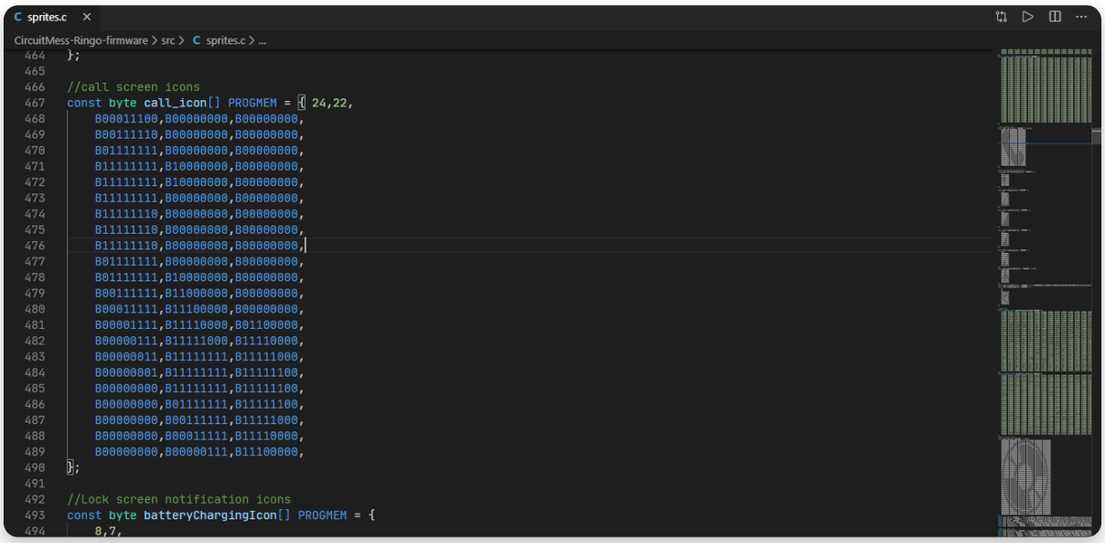


	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
RGB888	0	0	0	0	0	0	0	0	r7	r6	r5	r4	r3	r2	r1	r0	g7	g6	g5	g4	g3	g2	g1	g0	b7	b6	b5	b4	b3	b2	b1	b0	
mask red	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0xF80000
result red	0	0	0	0	0	0	0	0	r7	r6	r5	r4	r3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
mask green	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0xFC00
result green	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	g7	g6	g5	g4	g3	g2	0	0	0	0	0	0	0	0	0	0	
mask blue	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0xF8
result blue	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	b7	b6	b5	b4	b3	0	0	0	
RGB565																	r7	r6	r5	r4	r3	g7	g6	g5	g4	g3	g2	b7	b6	b5	b4	b3		

Conversion from RGB888 to RGB565

If you want to learn how to do this conversion manually, **click here**.

Additionally, there are some simple **black** icons, that are just represented by **0s and 1s**. If the pixel should be black it's set to 1 and if not it's set to 0. These types of sprites are **much simpler** and take up **much less memory,** so don't use colored ones unless you have to.



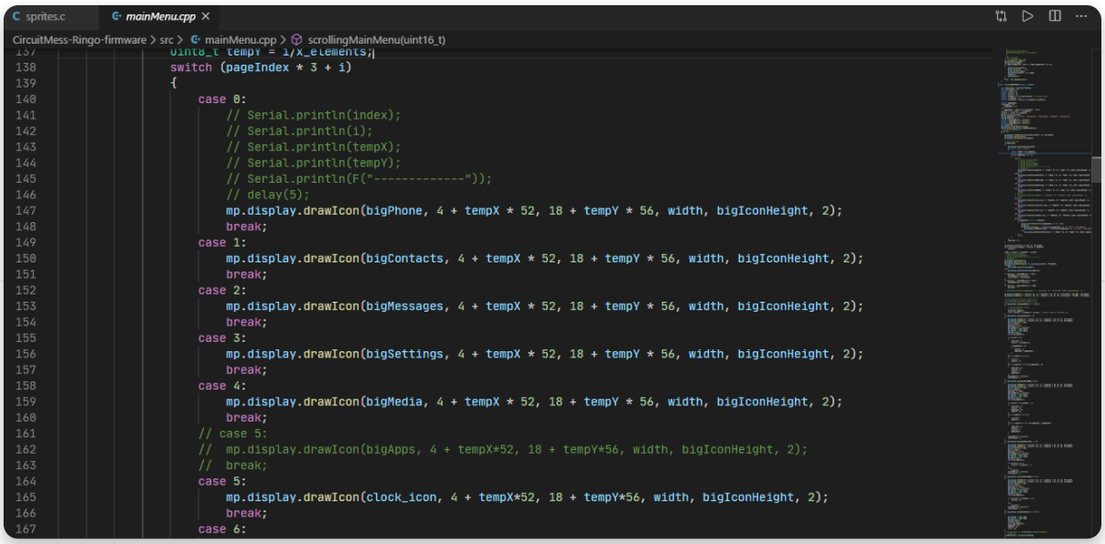
Simple phone icon being written in 0s and 0s

All of these sprites that are located in sprites.c are saved in **PROGMEM** which is internal phone memory. SD card is not necessary for these to work so you can have a clean phone experience even without the SD card. However, **internal memory is very limited**, so you always have to make sure not to overfill it with unnecessary things.

So how to get all the way from drawing our first pixel to displaying the bitmap on the phone?
Well, these are the necessary steps:

1. **Draw the bitmap inside of a graphic editor**

2. **Convert the bitmap to RGB565 code**
3. **Save that code to sprites.c (or one of the files)**
4. **Draw the bitmap on the display**



Icon drawing on the main menu - icons' RGB565 codes are being loaded from sprites.c

Alternatively, you can also **import direct bitmaps** and display them without converting them to the code. However, due to the limited internal memory, this can only be done from the SD card. **It is the technique that is used to import icons for games and apps that are added additionally**.

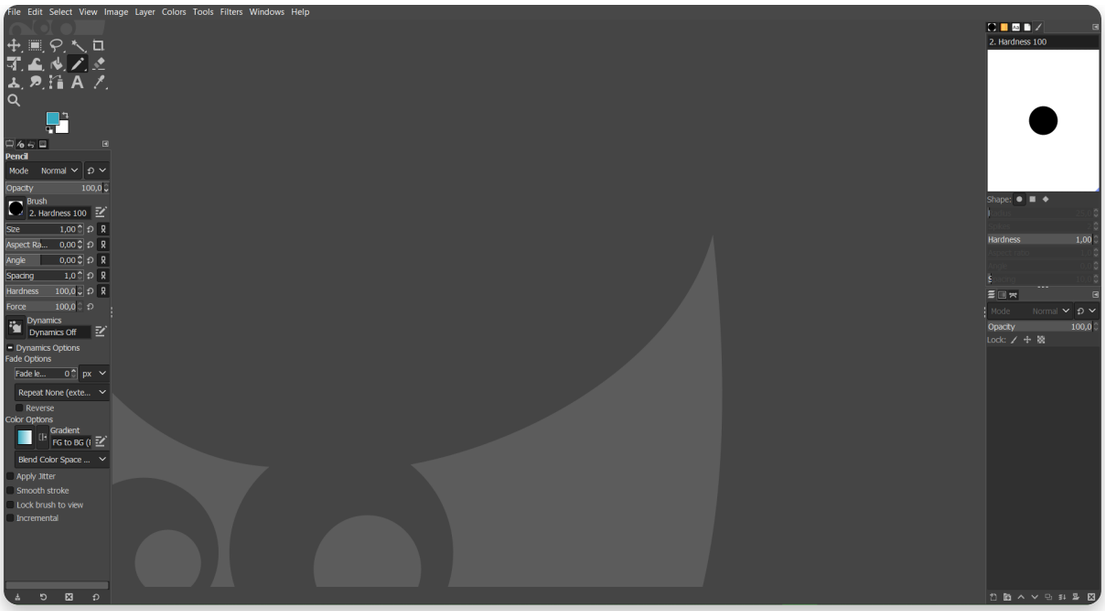
Drawing bitmaps directly onto the screen - without conversion

This is pretty much all you need to know - **now let's get to drawing**!

Drawing

Using GIMP

Now, let's open up GIMP and start drawing!



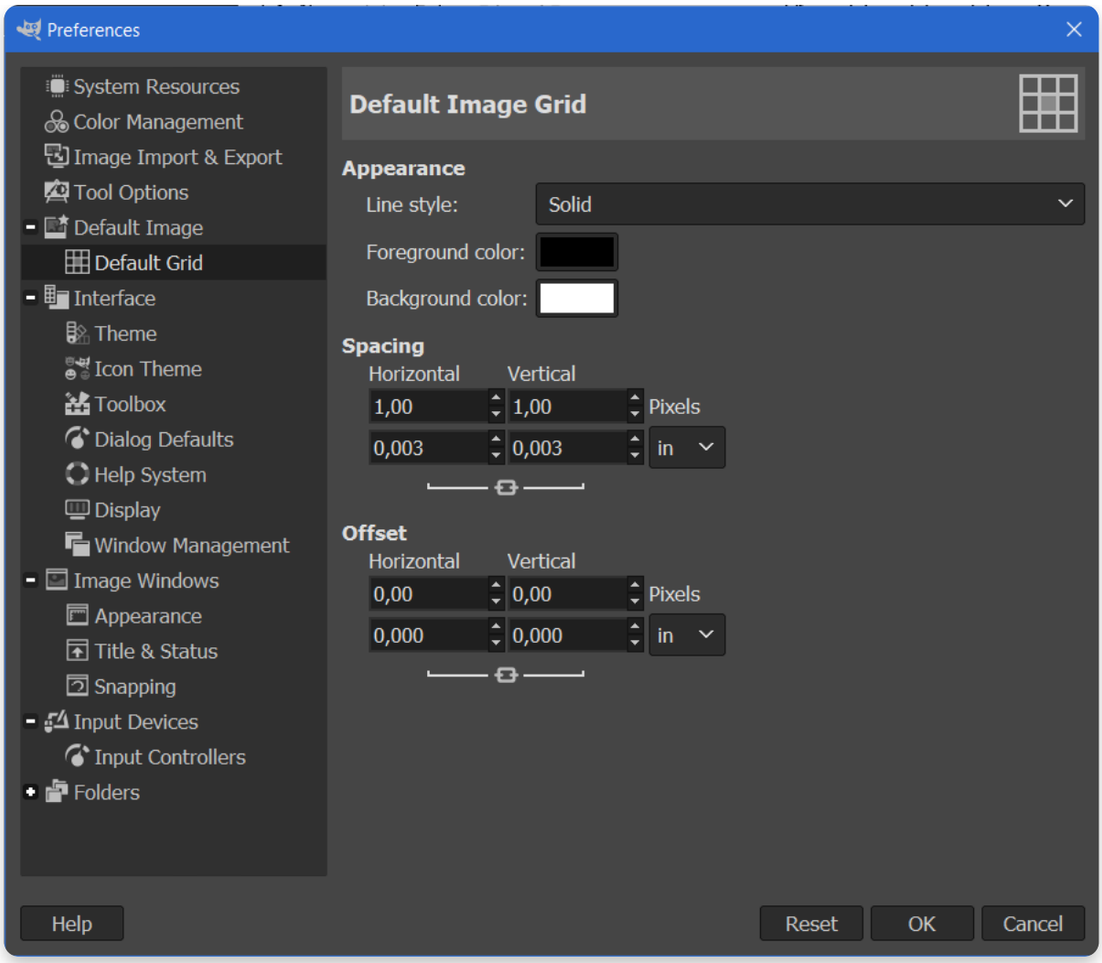
The main screen in GIMP

Now, this is how the main screen looks in GIMP. Just like in most advanced editors, there are a lot of options, most of which we are not going to use, so don't worry if you're a complete beginner at this.

Sprites that we're going to draw are coming in two sizes: **24x26 pixels and 24x24 pixels**. The bigger ones are dimensions of the ones in the main menu, while the smaller ones are located in various apps throughout the phone. There are many more different sprite sizes in Ringo firmware, but these are most common, so we're going to draw those.

Starting a new drawing

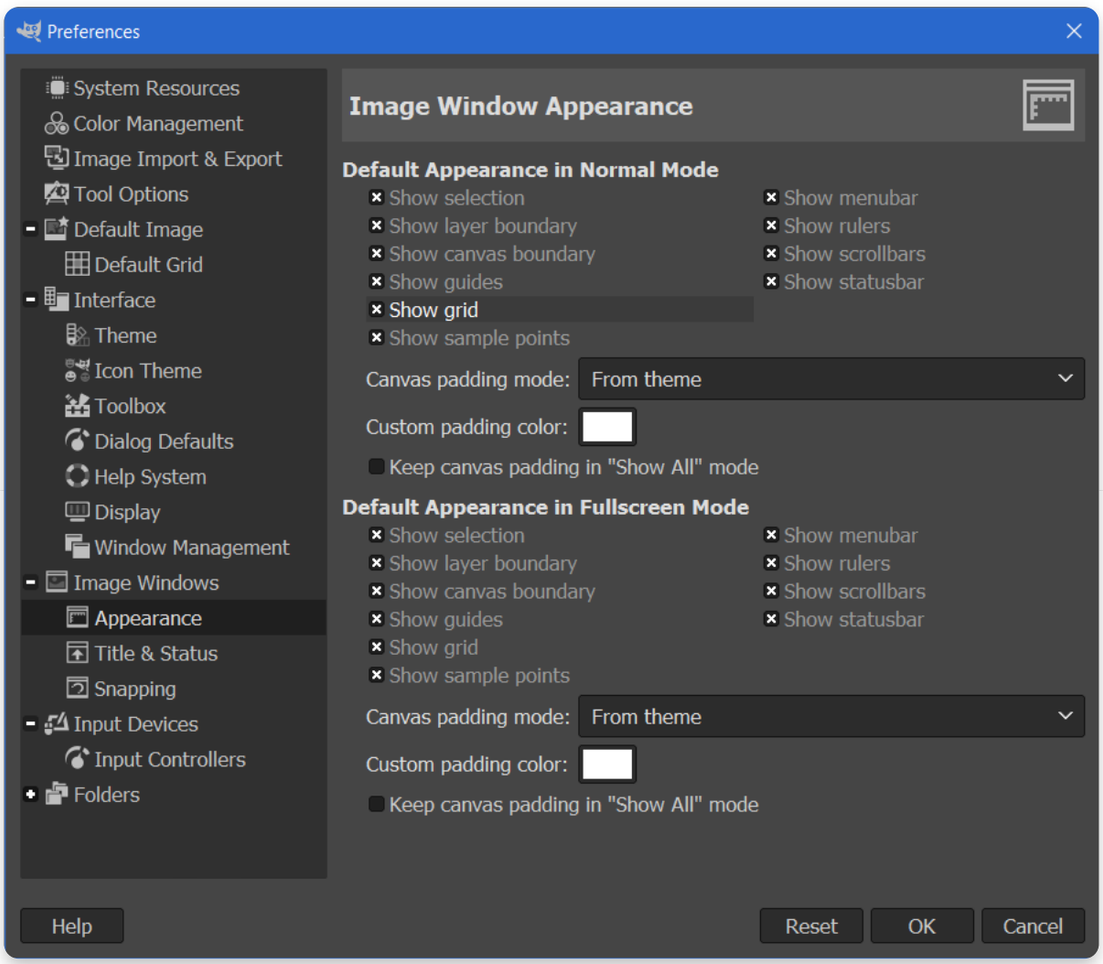
Before starting a new project, go to **Edit -> Preferences**. Since we're going to draw pixel-perfect images, we're going to set up a **grid**. This step is not necessary but it will help you a lot with precision drawing.



Setting grid spacing

Select '**Default grid**' and set both **horizontal and vertical spacing to 1,00 pixels**.

Then go to '**Appearance**' and **check both 'Show grid' boxes**.

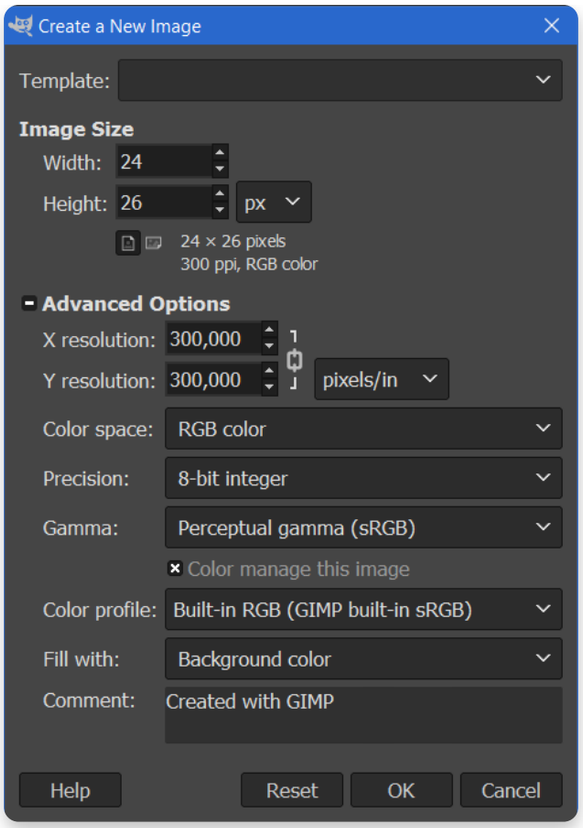


Setting default window appearance

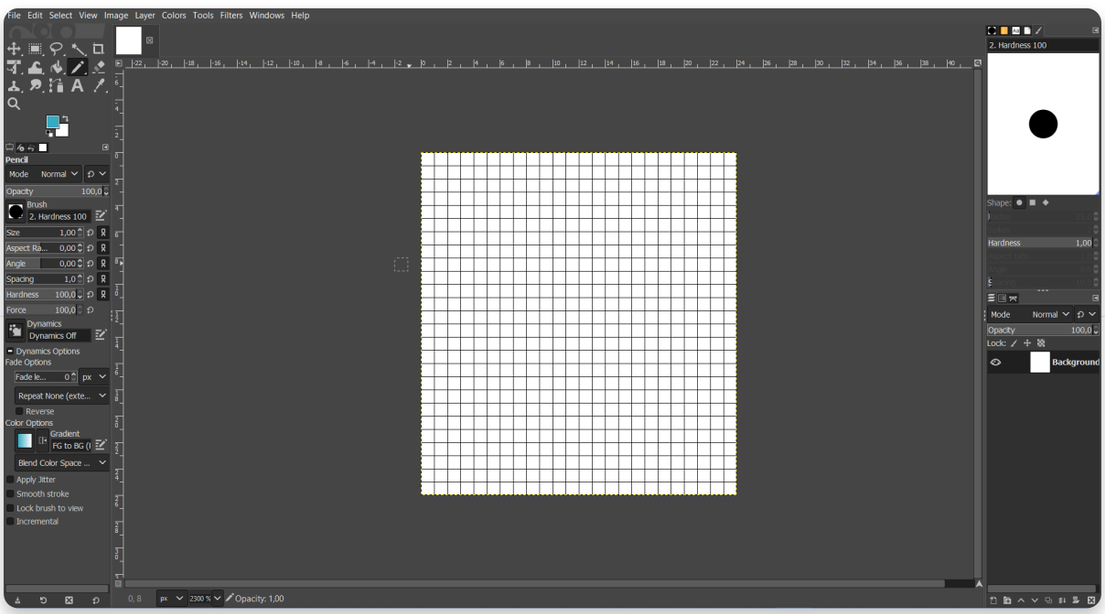
Click 'OK' and start a new drawing by going to **File -> New...**

The only thing we need to change here is setting our desired size. For this first example, set the size to **24x26 pixels.**

Click 'OK' and start the drawing.



Creating a new drawing



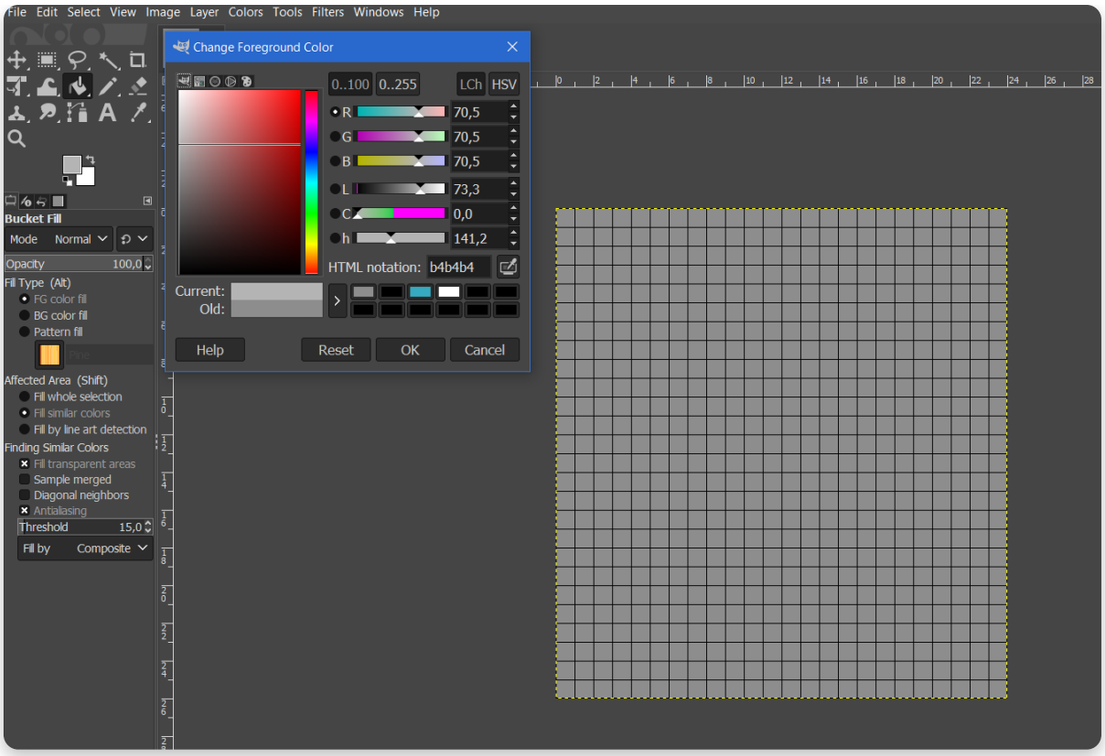
Empty canvas with the grid on

Drawing

Now there are two tools that we are going to use the most. You can find all of the common tools in the **upper left corner, just below the toolbar**.

Bucket fill tool and **pencil tool** are located in the **middle of the second row**. Bucket tool will help us fill bigger parts of the canvas with one color, while the pencil will color only one pixel at the time.

Color picker is located right below the tools and that's where you'll be able to pick your colors. Pretty simple, pretty easy.

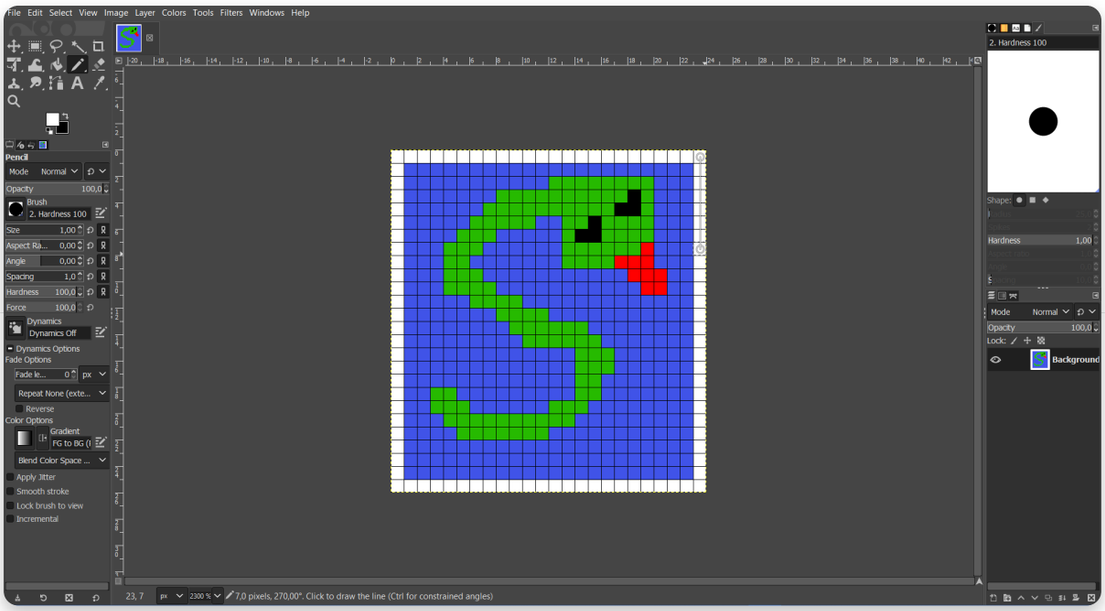


Color picker

These are pretty much all of the tools you are going to use - so go and make a cool picture!

What we've made in this little tutorial is a replacement icon for the **Snake app**.

It is not a masterpiece by any means, but it will serve its purpose.



New Snake app image

Exporting the image to Ringo

Now that our picture is drawn, it's time to export it.

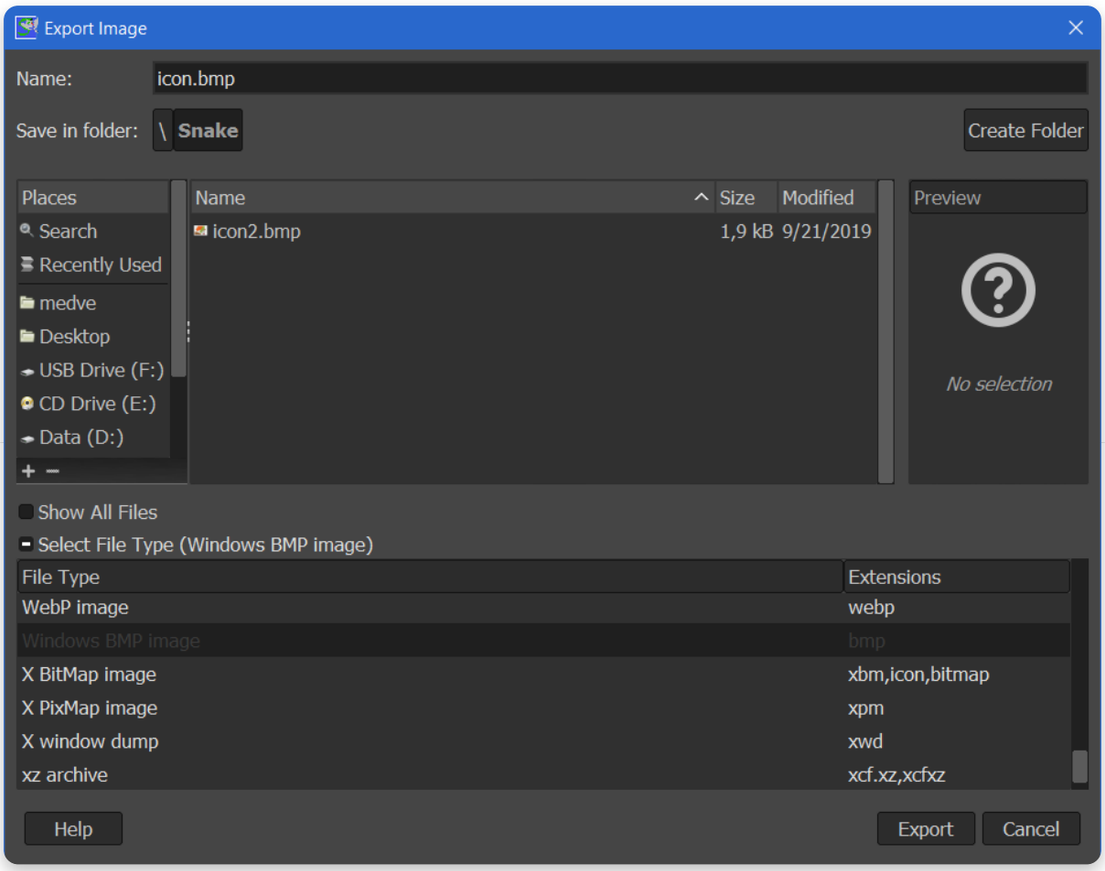
For this step, it's time to insert **the SD card from your Ringo into your computer.**

Before saving this icon, it would be good to locate your Snake app folder on the card and **rename the 'icon.bmp' to 'icon2.bmp'.**

Now go to **File -> Export** and locate your SD card in the explorer.

Go to the **'Snake' folder and save this file as 'icon.bmp'.**

It's also important to set the image to 'Windows BMP image' so that it has a .bmp extension.



Exporting your new image

When you finish the export, eject the SD card from your computer and put it back in your phone.

Restart the phone and that's it - your Snake app just got a new icon!

In the previous chapter, there was a snippet of code where you could see how are these app icons being drawn.

Here it is once again.

```
default:
    if(pageIndex * 3 + i < elements)
    {
        Serial.println(directories[pageIndex * 3 + i - 9]);
        delay(5);
        if(SD.exists(String("/" + directories[pageIndex * 3 + i - 9] + "/icon.bmp")))
            mp.display.drawBmp(String("/" + directories[pageIndex * 3 + i - 9] + "/icon.bmp"), 4 + tempX * 52, 18 + tempY * 56, 2);
        else
            mp.display.drawIcon(defaultIcon, 4 + tempX * 52, 18 + tempY * 56, width, bigIconHeight, 2);
    }
    break;
```

We see that we've drawn the bitmap called '**icon.bmp**' that is located in the app folder. That process is being repeated for every app that is located on the SD card, which is for every game. Also, these icons are actually being shown in a doubled resolution.

Index 2 at the end of the 182nd row means that the size of the icon is doubled. That's why the icon is taking up 48x52 pixels on your Ringo screen.

Now, let's see how to convert these images to code.

Converting bitmap to code

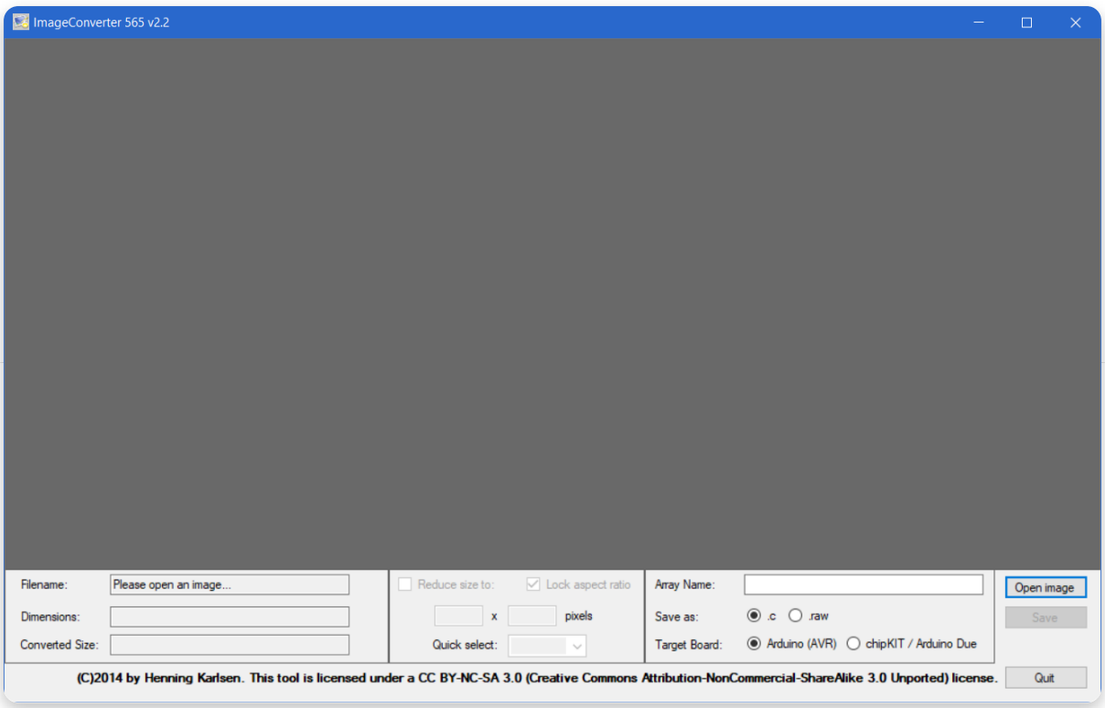
Conversion can be done in multiple ways, but we'll focus just on some of them.

These types of conversion are usually done by software, so we're going to do the same.

Software conversion

One of the programs that are rather simple, yet capable of doing this, is **Image Converter 565**.

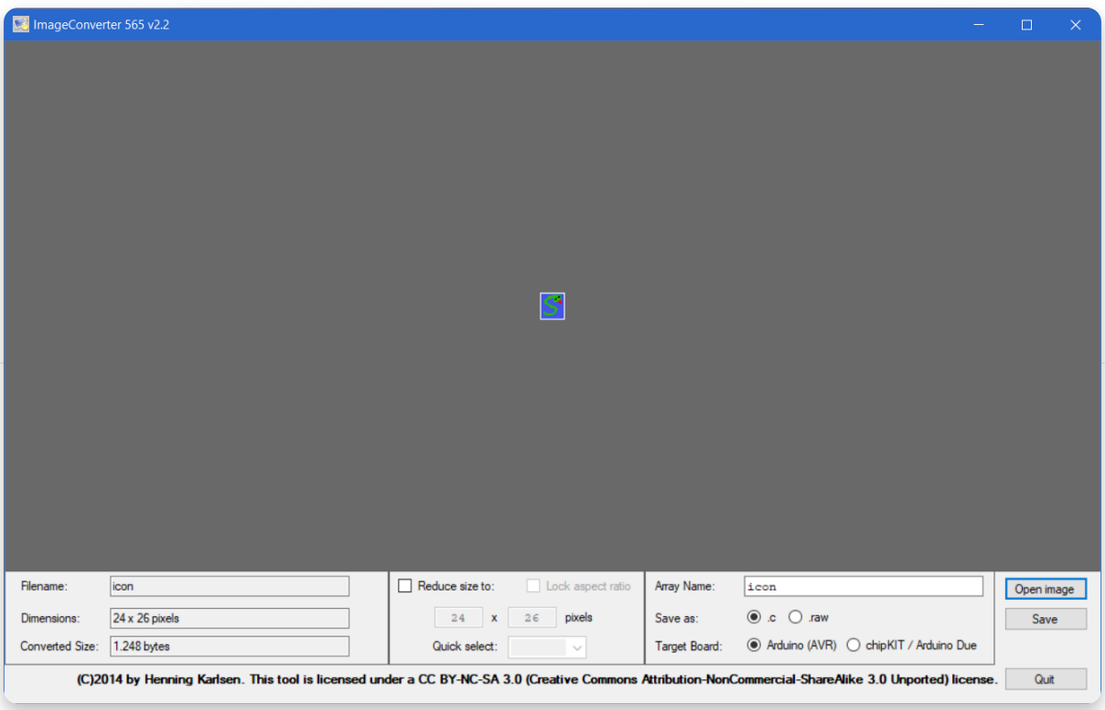
You can download it from our website and when you open it, you'll get a screen like this.



ImageConverter565 interface

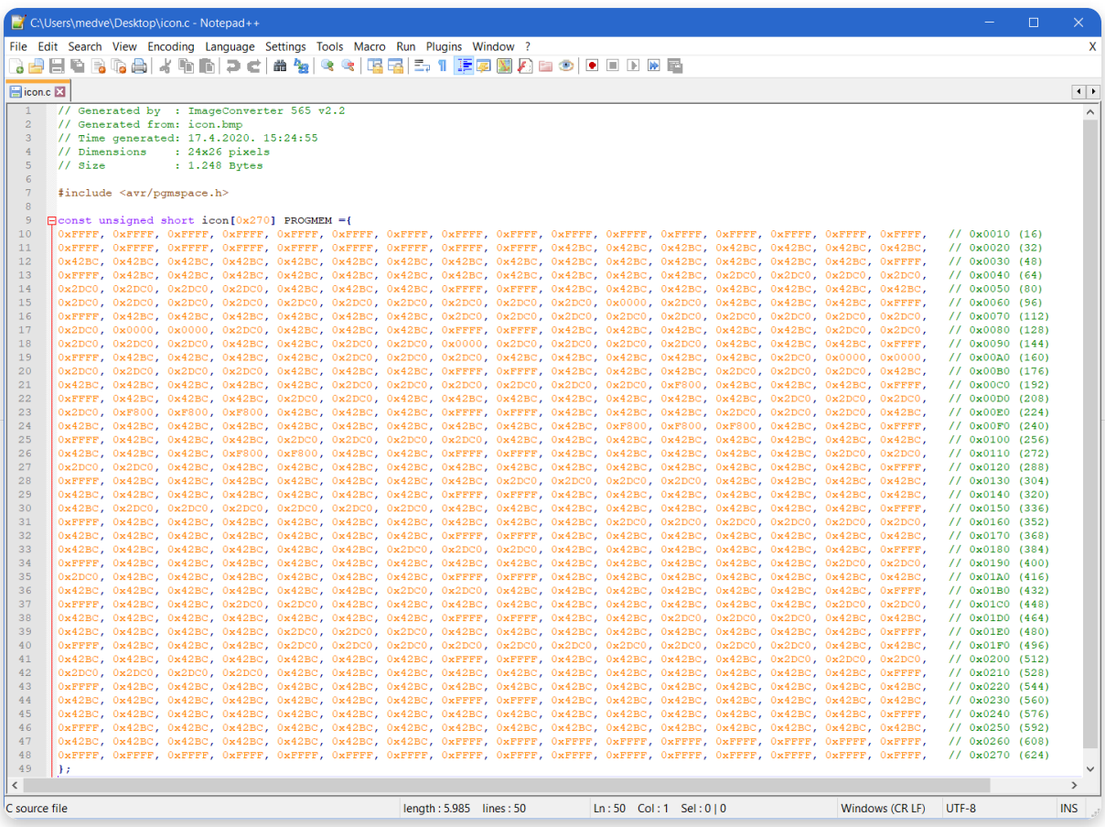
The program itself is pretty simple - you upload the bitmap, convert it, and export the code!

You can also change the desired size if you want so.



An image has been imported

The end product will look something like this.



Full bitmap code in RGB656 (Notepad++ editor)

All you need from this code is this constant called '**icon**'. Just copy it inside your program and use it!

Another one of those programs is **LCD image converter** which does things pretty similarly and is equally easy to use.

Alternatively, if you're working with smaller images, you can just **copy the color codes pixel-by-pixel** inside of an array just like this one. Just remember to convert the color to RGB565 (since the colors are usually in RGB888).

Displaying the code

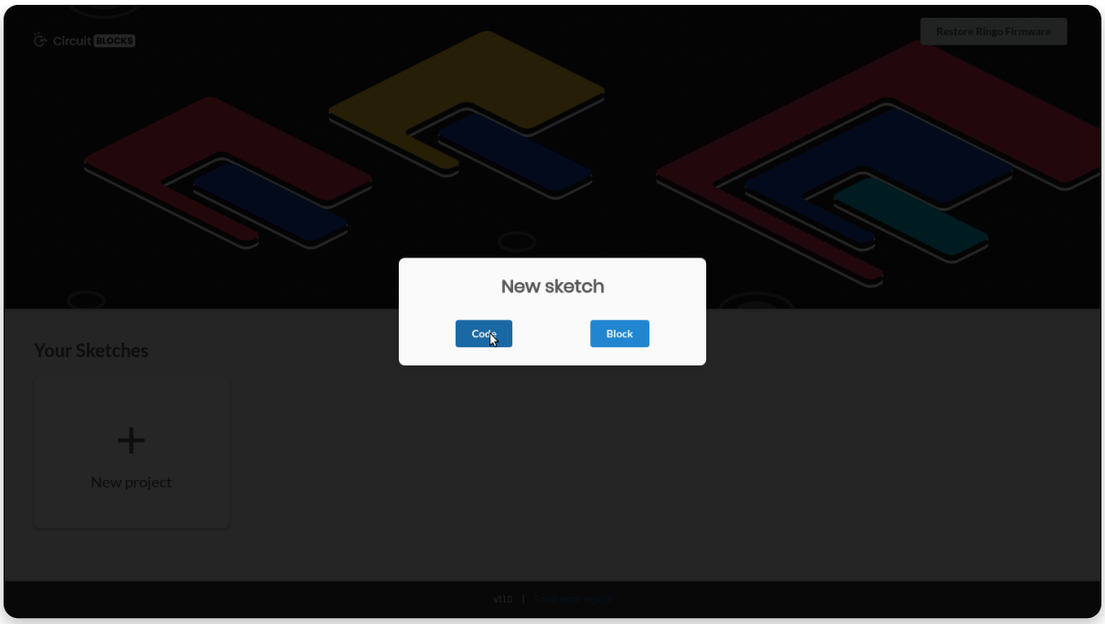
Now that we've got the needed code, it's time to use it in a program.

We've already shown that '**sprites.c**' file that holds various icons' codes. Since all of those icons are declared as global constants, we're going to do the same thing here.

To do this next step, you have to open either **CircuitBlocks** or **Arduino IDE**. If you have setup Ringo library in any other text editor, that will do the trick as well.

This tutorial is going to be done inside CircuitBlocks, but the process itself is the same for Arduino.

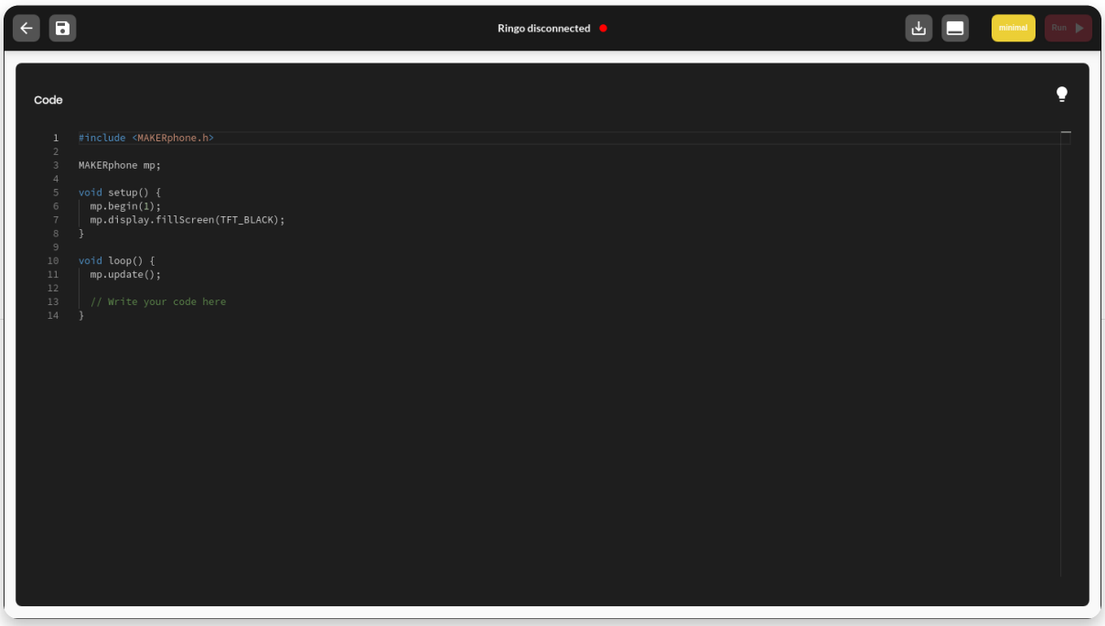
First, open CircuitBlocks and **start a new project**.



When selecting a new sketch type press 'Code'

You need to select the **'Code'** button since we're going to have to type a little bit here.

This process can't be done in a **'Block'** project. What you can do is take your block project's code, copy it to the code project, and then add the finishing touches there.



Empty new 'Code' sketch

Now, let's copy our bitmap snippet as the global constant.

For that all we need to do is write the following line:

C/C++

```
const unsigned short snake_icon[SIZE] PROGMEM = { ... }
```

With our colors' hex codes being between the brackets.

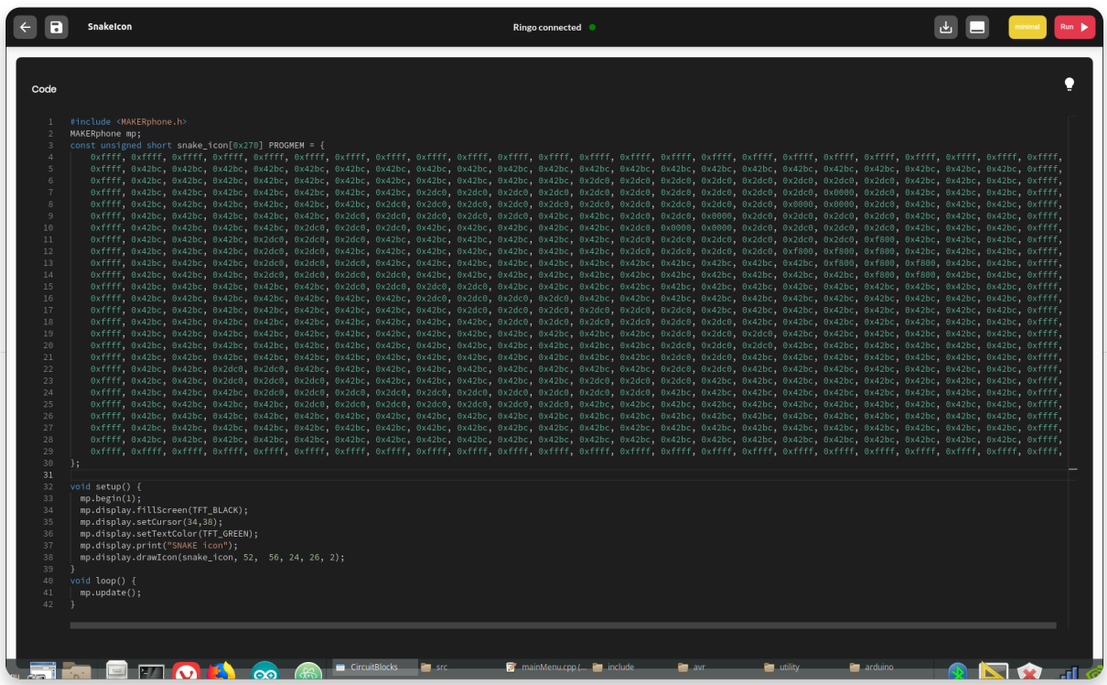
That means that it would all look like this

```
const unsigned short snake_icon[0x270] PROGMEM = {
 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc,  0x42bc
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x2dc0, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x42bc, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x2dc0, 0x2dc0, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0,  0x2dc0
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
};
```

The **SIZE** variable actually determines how many pixels are there. Since we have an icon that is **24x26**, that would be a total of **624 pixels**. If you convert that to hexadecimal (notice the '**0x**' before the number) it would translate to **270**, which is the number we've got here.

The rest of our variable are just color codes for each specific pixel. If you look closely, you'll notice that the **white color** (**0xffff**) that colors our icon's outline, is on the "edges" of the code.

Now, all we need to do is print that little icon out.



Printing out our new snake icon

We've also added a little text so that the icon is not lonely on the screen!

Function '**mp.display.drawIcon()**' has variables in this order: (**imageFile, locationX, locationY, width, height, scale**).

Pretty self-explanatory as it is, just like in the previous example.

Here is the full code:

ARDUINO

```
#include <MAKERphone.h>
MAKERphone mp;
const unsigned short snake_icon[0x270] PROGMEM = {
 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc,  0x42bc
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x2dc0, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x42bc, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0, 0x2dc0, 0x2dc0, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x2dc0, 0x2dc0, 0x2dc0,  0x2dc0
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc,
 0xffff, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x42bc, 0x
    0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
};

void setup() {
  mp.begin(1);
  mp.display.fillScreen(TFT_BLACK);
  mp.display.setCursor(34,38);
  mp.display.setTextColor(TFT_GREEN);
  mp.display.print("SNAKE icon");
  mp.display.drawIcon(snake_icon, 52,  56, 24, 26, 2);
}
void loop() {
  mp.update();
}
```

See, it's not that hard!

Transparent bitmaps

What if we want to have an icon that is not a rectangle, but rather a circle or some other shape?

Well, since all icons are rectangles, that means that some pixels are going to have to be **transparent**.

The problem with bitmaps - **they don't have a transparent channel**. Luckily, you

can just pick a color that will serve as a transparent one, so that each pixel with that color will not be shown.

In the function **drawIcon()**, you can also add a color parameter.

```
1 void TFT_eSPI::drawIcon(const unsigned short* icon, int16_t x, int16_t y, ui
```

So for example, if we wanted to draw our snake icon without white color outlines, our function would look something like this:

```
1 mp.display.drawIcon(snake_icon, 0, 0, 24, 26, 1, 0xffff);
```

On our screen now we have **24x26 snake icon located in the upper left corner of the screen**, and of course, **without white borders**.

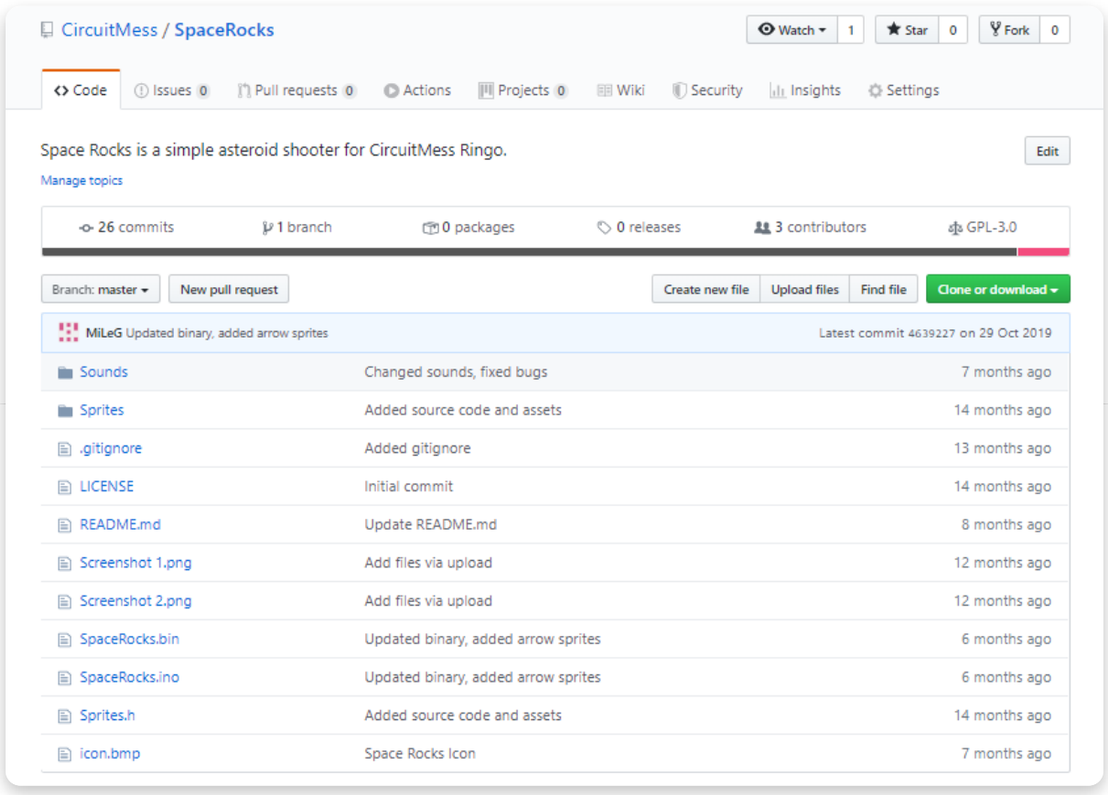
Now you can get to drawing and importing some more complex images to your new apps!

Using external bitmaps and editing firmware

When creating a more complex app, it's usually better to have **bitmaps in a separate folder** and just to load them whenever you need them.

That way you can fix your images on the go and don't have to waste a lot of time converting them to code back and forth.

A good example of that one is our **Space Rocks** game for Ringo. You can check its repository here.

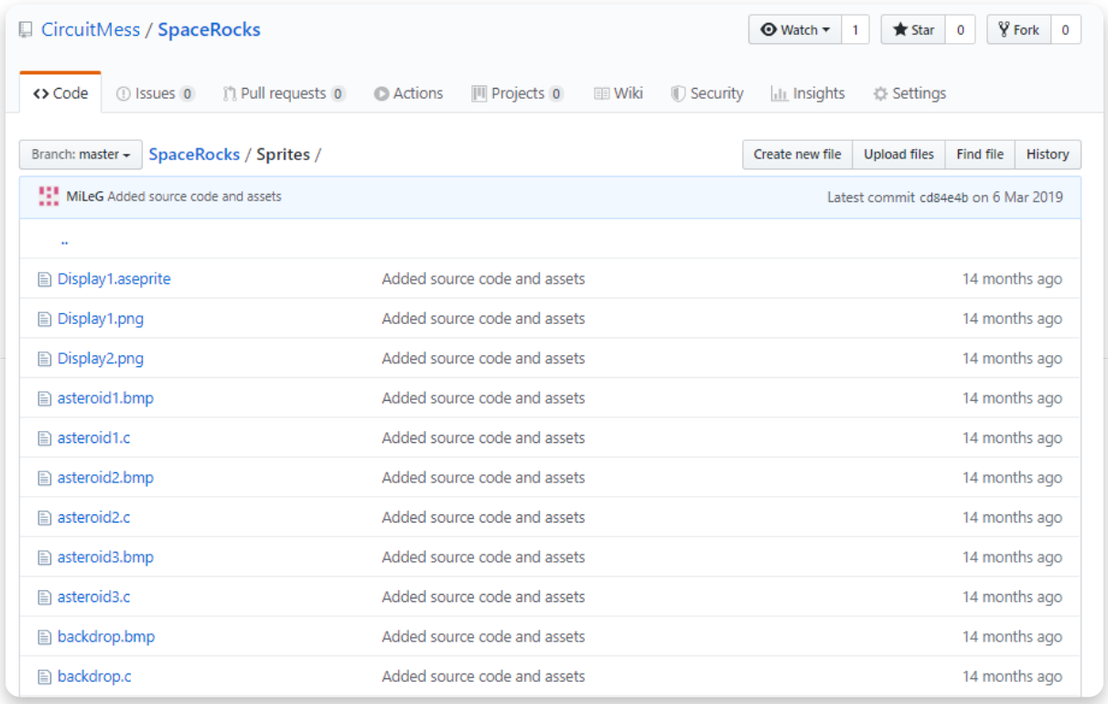


Space Rocks repository on GitHub

You'll notice that besides the main files, Space Rocks has both **Sounds** and **Sprites** folders.

Both of those folders have files that are being used in-game.

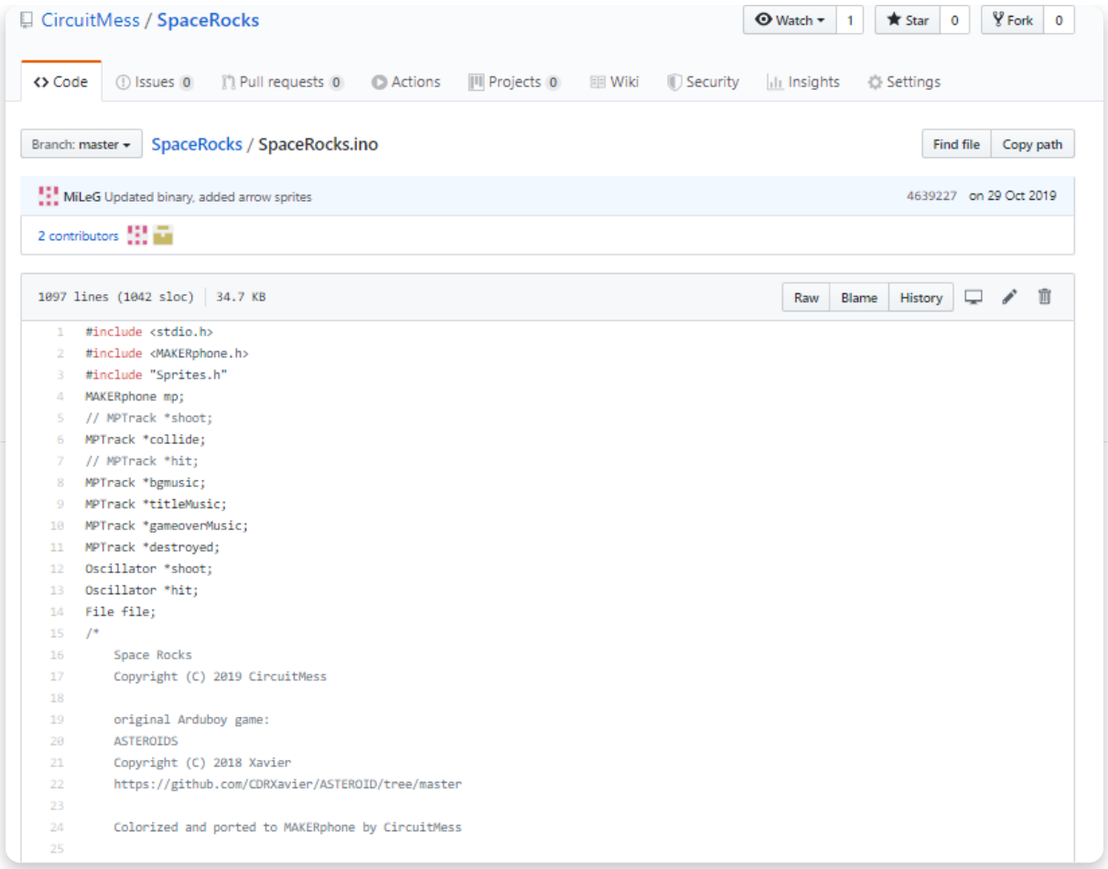
Here is how the **Sprites** folder looks on the inside.



Sprites folder from the Space Rocks repository

You'll notice some **bitmaps** and **.c** files. That way you can use both functions that we've learned in the previous lesson.

Checking out the **SpaceRocks.ino** file will help you understand this whole process much better, so make sure to check it out in detail if you wish to learn more.



First few lines of the SpaceRocks.ino file

Editing firmware (advanced)

If you're feeling reaaaally good and want to change some of the things in our original phone firmware, you can do so!

We always encourage new creations and personalizations, as long as you share them on our **forum** so that everyone else can use them too. :)

If you're unfamiliar with the process of **building the firmware form source files**, our most active forum members along with our staff have created a **cool topic** where you can find all the details, and fixes to potential problems, on how to download, build and upload your own version of Ringo firmware.

If you have any experience in doing these things, it will not take you more than **15 minutes to set everything up.**

However, if you've never encountered these things, you might have some more issues. We still encourage you to do so since you'll learn so much in the process.

Once you've done so, the file on which you'll be focusing the most is **sprites.c**. We've already shown you how the file looks and how things work, so there's no doubt that you get the grip of things and personalize your phone in a matter of minutes!

If you get stuck, you can always hit us up on our community forum, where the whole community and our staff are eager to help you and explore further parts of this project.

Now - start making!