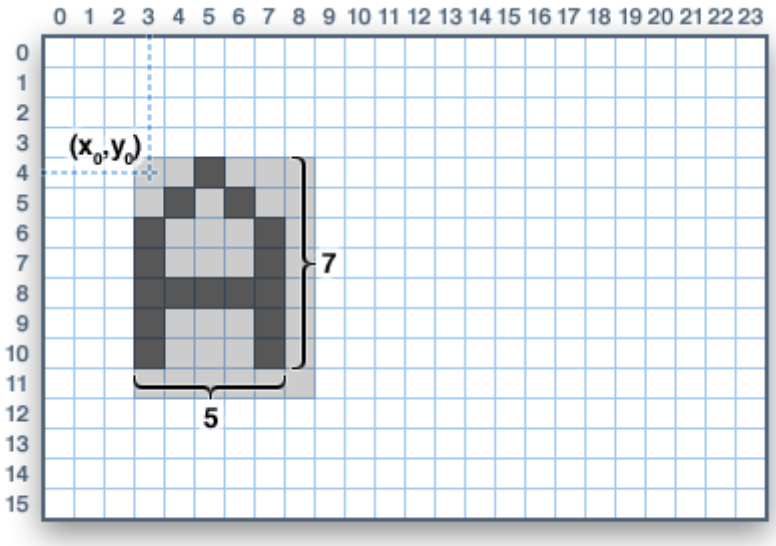
# Adafruit GFX Graphics Library

Created by Phillip Burgess

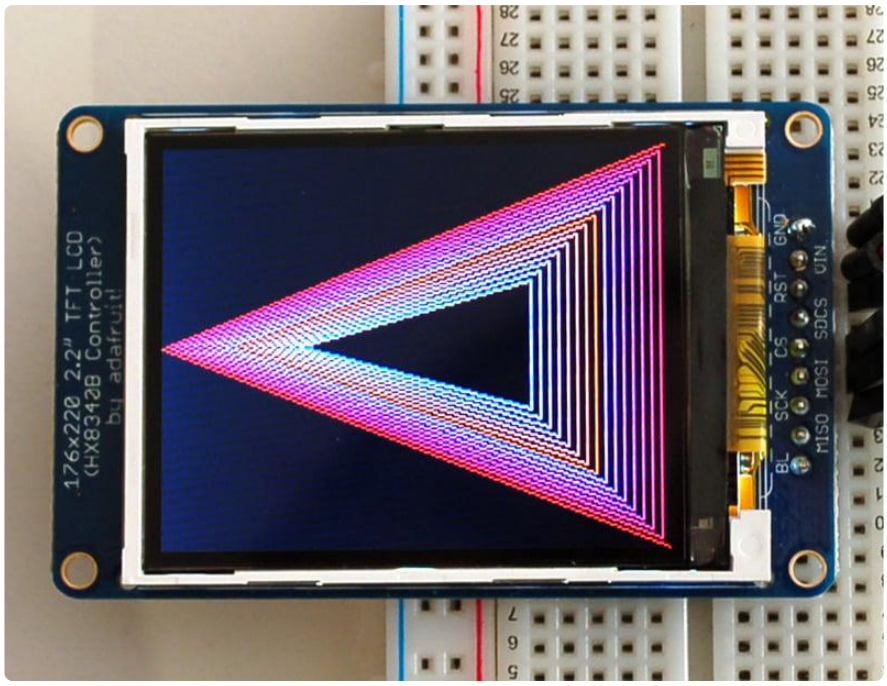



https://learn.adafruit.com/adafruit-gfx-graphics-library

Last updated on 2022-12-01 04:15:05 PM EST

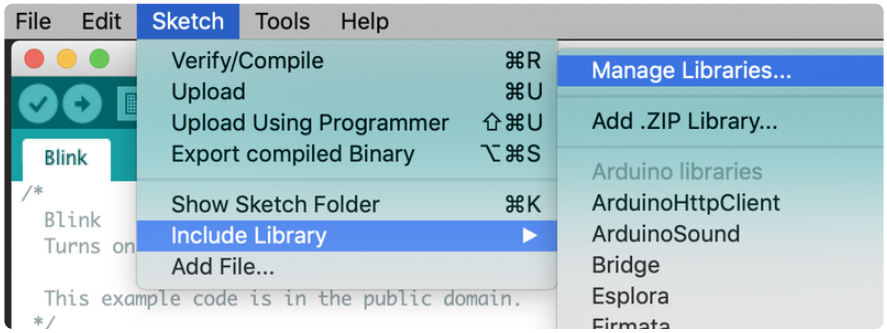
# Table of Contents

# Overview

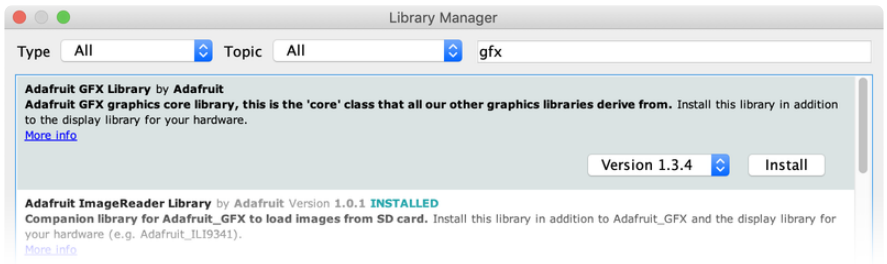



The Adafruit_GFX library for Arduino provides a common syntax and set of graphics functions for all of our LCD and OLED displays and LED matrices. This allows Arduino sketches to easily be adapted between display types with minimal fuss...and any new features, performance improvements and bug fixes will immediately apply across our complete offering of color displays.

Adafruit_GFX always works together with an additional library unique to each specific display type. These can be installed using the Arduino Library Manager. From the Arduino “Sketch” menu, select “Include Library,” then “Manage Libraries...”




In the Arduino Library Manager window, search for a display’s driver type (e.g. “SSD1325”) and the appropriate Adafruit library can be found in the results. Required companion libraries (“dependencies,” like Adafruit_GFX or Adafruit_BusIO) now get installed automatically. If using an older version of the Arduino IDE, you’ll have to search for and install those additional libraries manually.




Some of the libraries that operate alongside Adafruit_GFX include:

- RGBmatrixPanel (), for our 16x32 (http://adafru.it/420) and 32x32 (http://adafru.it/607) RGB LED matrix panels.
- Adafruit_TFTLCD (), for our 2.8" TFT LCD touchscreen breakout (http://adafru.it/335) and TFT Touch Shield for Arduino (http://adafru.it/376).
- Adafruit_HX8340B (), for our 2.2" TFT Display with microSD (http://adafru.it/797).
- Adafruit_ST7735 (), for our 1.8" TFT Display with microSD (http://adafru.it/358).
- Adafruit_PCD8544 (), for the Nokia 5110/3310 monochrome LCD (http://adafru.it/338).
- Adafruit-Graphic-VFD-Display-Library (), for our 128x64 Graphic VFD (http://adafru.it/773).
- Adafruit-SSD1331-OLED-Driver-Library-for-Arduino () for the 0.96" 16-bit Color OLED w/microSD Holder (http://adafru.it/684).
- Adafruit_SSD1306 () for the Monochrome 128x64 (http://adafru.it/326) and 128x32 (http://adafru.it/661) OLEDs.

And many others, except for some very early “retired” products. Remember, just search for the display driver type in the Arduino Library manager, install, and the rest is automatic now.

The libraries are written in C++ for Arduino but could easily be ported to any microcontroller by rewriting the low-level pin access functions.

# The Old Way

Much older versions of the Arduino IDE software require installing libraries manually; the Arduino Library Manager did not yet exist. If using an early version of the Arduino software, this might be a good time to upgrade. Otherwise, this tutorial explains how to install and use Arduino libraries (). Here are links to download the GFX and BusIO libraries directly (use the links above to get the corresponding display-specific libraries):

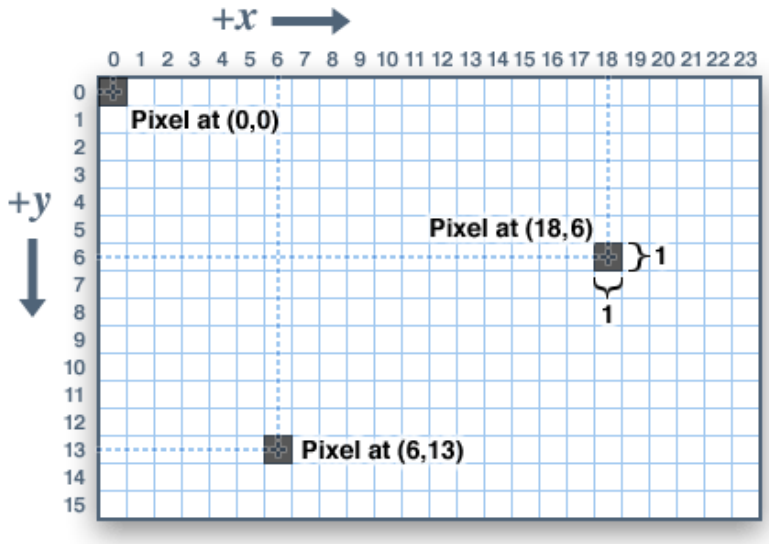
Download Adafruit_GFX Library

Download Adafruit_BusIO Library

# Coordinate System and Units

Pixels — picture elements, the blocks comprising a digital image — are addressed by their horizontal (X) and vertical (Y) coordinates. The coordinate system places the origin (0,0) at the top left corner, with positive X increasing to the right and positive Y increasing downward. This is upside-down relative to the standard Cartesian coordinate system of mathematics, but is established practice in many computer graphics systems (a throwback to the days of raster-scan CRT graphics, which worked top-to-bottom). To use a tall “portrait” layout rather than wide “landscape” format, or if physical constraints dictate the orientation of a display in an enclosure, one of four rotation settings can also be applied, indicating which corner of the display represents the top left.

Also unlike the mathematical Cartesian coordinate system, points here have dimension — they are always one full integer pixel wide and tall.

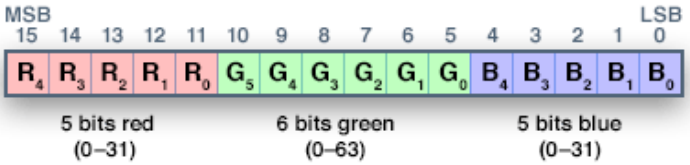



Coordinates are always expressed in pixel units; there is no implicit scale to a real-world measure like millimeters or inches, and the size of a displayed graphic will be a

function of that specific display's dot pitch or pixel density. If you're aiming for a real-world dimension, you'll need to scale your coordinates to suit. Dot pitch can often be found in the device datasheet, or by measuring the screen width and dividing the number of pixels across by this measurement.

The library will safely "clip" any graphics drawn off the edges of the screen. In fact this is done on purpose sometimes, as with scrolling text displays.

For color-capable displays, colors are represented as unsigned 16-bit values. Some displays may physically be capable of more or fewer bits than this, but the library operates with 16-bit values...these are easy for the Arduino to work with while also providing a consistent data type across all the different displays. The primary color components — red, green and blue — are all "packed" into a single 16-bit variable, with the most significant 5 bits conveying red, middle 6 bits conveying green, and least significant 5 bits conveying blue. That extra bit is assigned to green because our eyes are most sensitive to green light. Science!




For the most common primary and secondary colors, we have this handy cheat-sheet that you can include in your own code. Of course, you can pick any of 65,536 different colors, but this basic list may be easiest when starting out:

```
// Color definitions
#define BLACK    0x0000
#define BLUE     0x001F
#define RED      0xF800
#define GREEN    0x07E0
#define CYAN     0x07FF
#define MAGENTA  0xF81F
#define YELLOW   0xFFE0
#define WHITE    0xFFFF
```

For monochrome (single-color) displays, colors are always specified as simply 1 (set) or 0 (clear). The semantics of set/clear are specific to the type of display: with something like a luminous OLED display, a "set" pixel is lighted, whereas with a

reflective LCD display, a “set” pixel is typically dark. There may be exceptions, but generally you can count on 0 (clear) representing the default background state for a freshly-initialized display, whatever that works out to be.

---

# Graphics Primitives

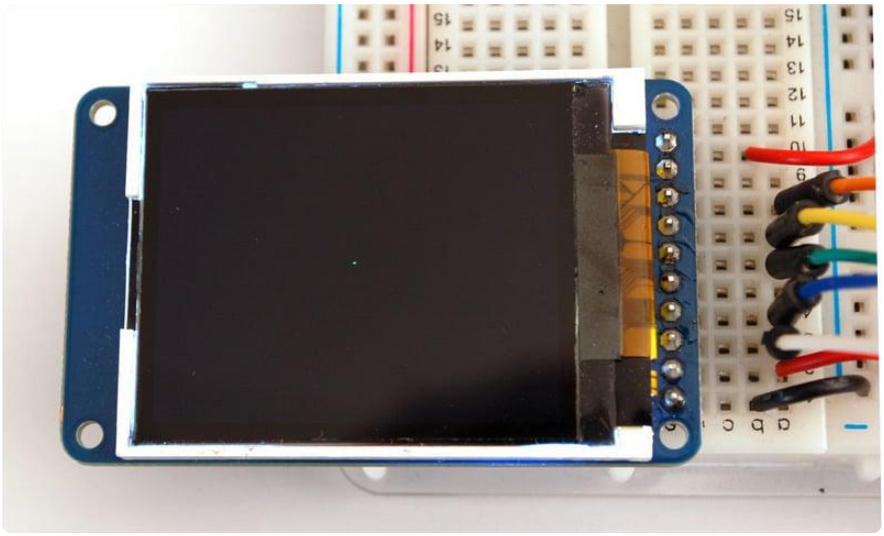
Each device-specific display library will have its own constructors and initialization functions. These are documented in the individual tutorials for each display type, or oftentimes are evident in the specific library header file. The remainder of this tutorial covers the common graphics functions that work the same regardless of the display type.

The function descriptions below are merely prototypes — there’s an assumption that a display object is declared and initialized as needed by the device-specific library. Look at the example code with each library to see it in actual use. For example, where we show print(1234.56), your actual code would place the object name before this, e.g. it might read screen.print(1234.56) (if you have declared your display object with the name screen).

## Drawing pixels (points)

First up is the most basic pixel pusher. You can call this with X, Y coordinates and a color and it will make a single dot:

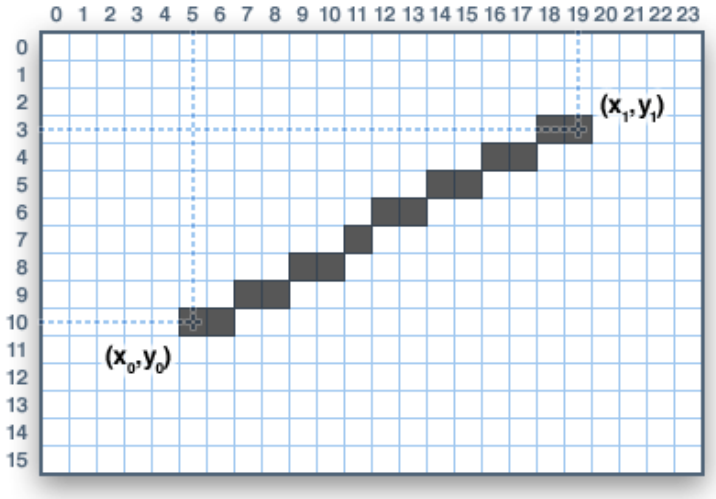
```
void drawPixel(uint16_t x, uint16_t y, uint16_t color);
```

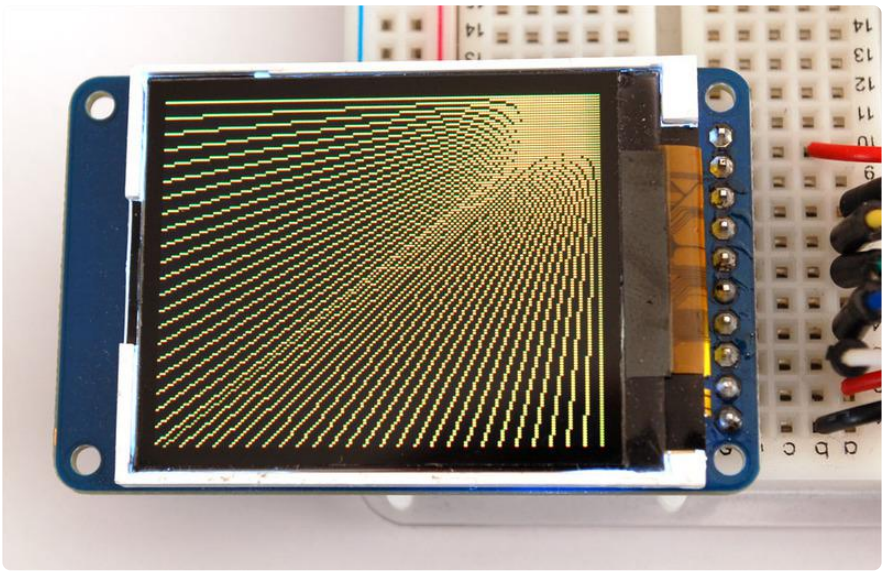


## Drawing lines

You can also draw lines, with a starting and end point and color:

```
void drawLine(uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1, uint16_t color);
```



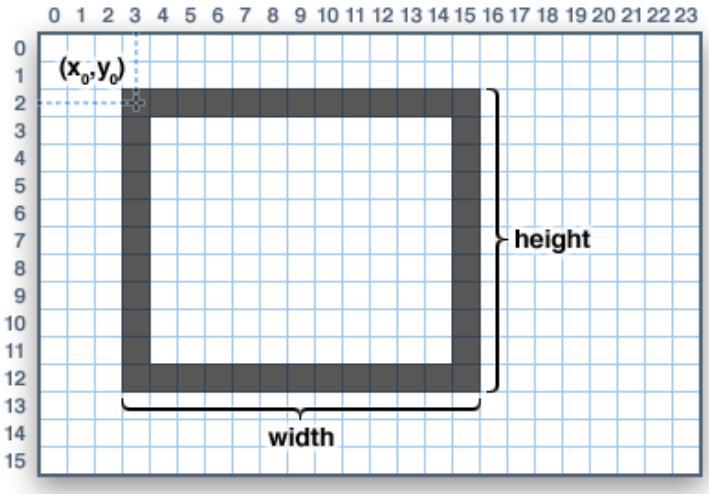
For horizontal or vertical lines, there are optimized line-drawing functions that avoid the angular calculations:

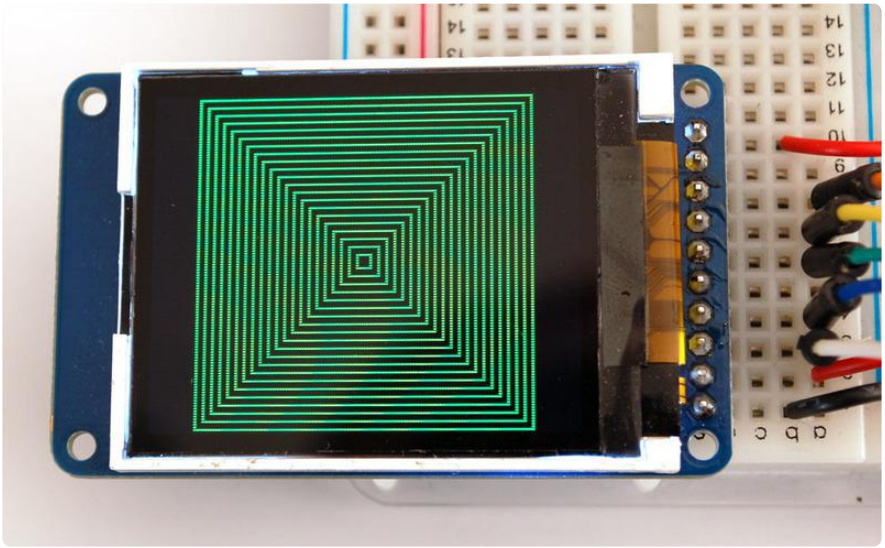
```
void drawFastVLine(uint16_t x0, uint16_t y0, uint16_t length, uint16_t color);
void drawFastHLine(uint8_t x0, uint8_t y0, uint8_t length, uint16_t color);
```

## Rectangles

Next up, rectangles and squares can be drawn and filled using the following procedures. Each accepts an X, Y pair for the top-left corner of the rectangle, a width and height (in pixels), and a color. drawRect() renders just the frame (outline) of the rectangle — the interior is unaffected — while fillRect() fills the entire area with a given color:

```
void drawRect(uint16_t x0, uint16_t y0, uint16_t w, uint16_t h, uint16_t color);
void fillRect(uint16_t x0, uint16_t y0, uint16_t w, uint16_t h, uint16_t color);
```



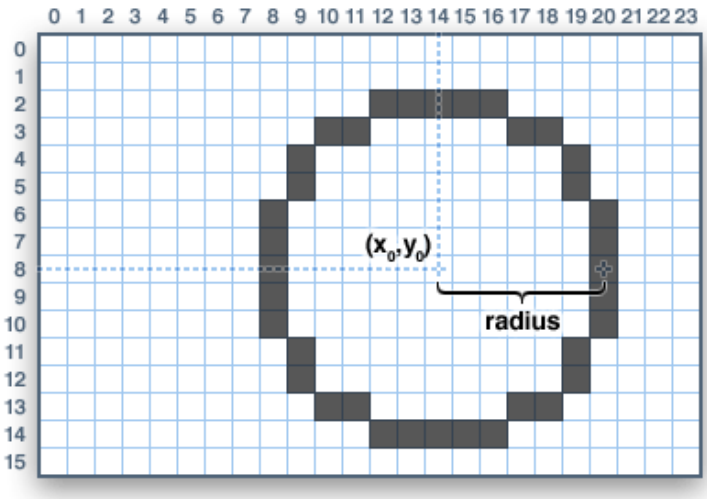



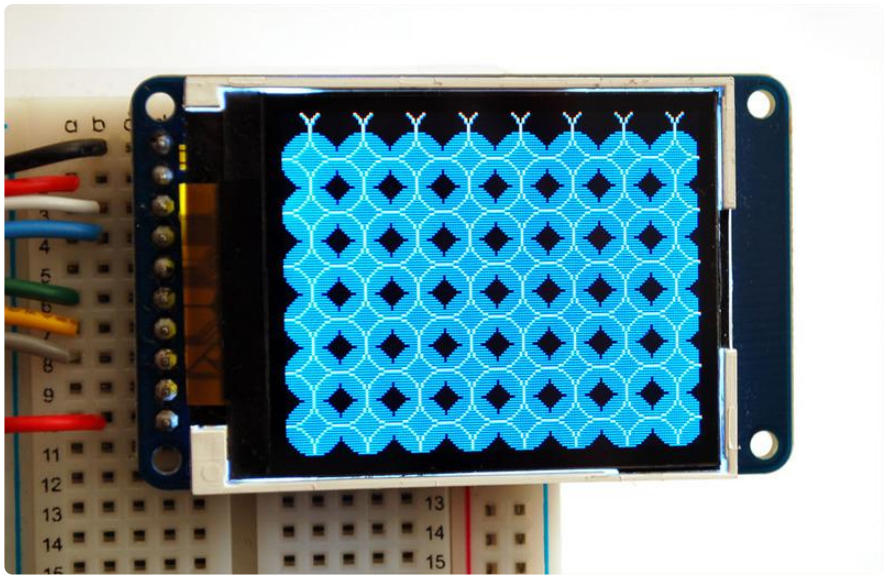
To create a solid rectangle with a contrasting outline, use fillRect() first, then drawRect() over it.

## Circles

Likewise, for circles, you can draw and fill. Each function accepts an X, Y pair for the center point, a radius in pixels, and a color:

```
void drawCircle(uint16_t x0, uint16_t y0, uint16_t r, uint16_t color);
void fillCircle(uint16_t x0, uint16_t y0, uint16_t r, uint16_t color);
```

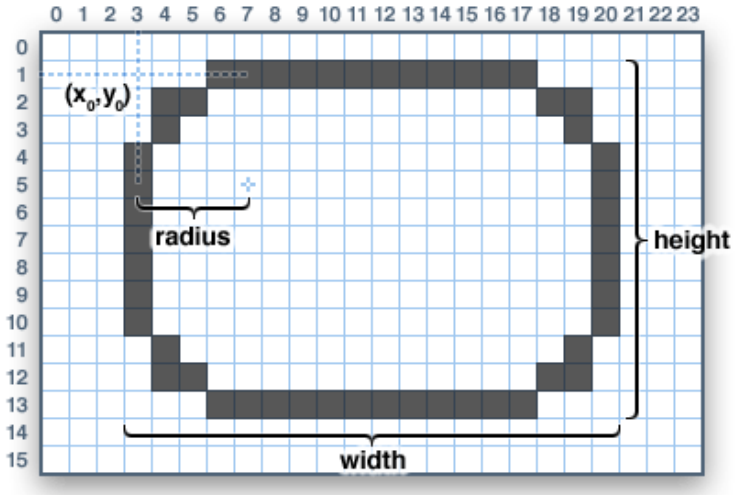


## Rounded rectangles

For rectangles with rounded corners, both draw and fill functions are again available. Each begins with an X, Y, width and height (just like normal rectangles), then there's a corner radius (in pixels) and finally the color value:

```
void drawRoundRect(uint16_t x0, uint16_t y0, uint16_t w, uint16_t h, uint16_t
radius, uint16_t color);
void fillRoundRect(uint16_t x0, uint16_t y0, uint16_t w, uint16_t h, uint16_t
radius, uint16_t color);
```

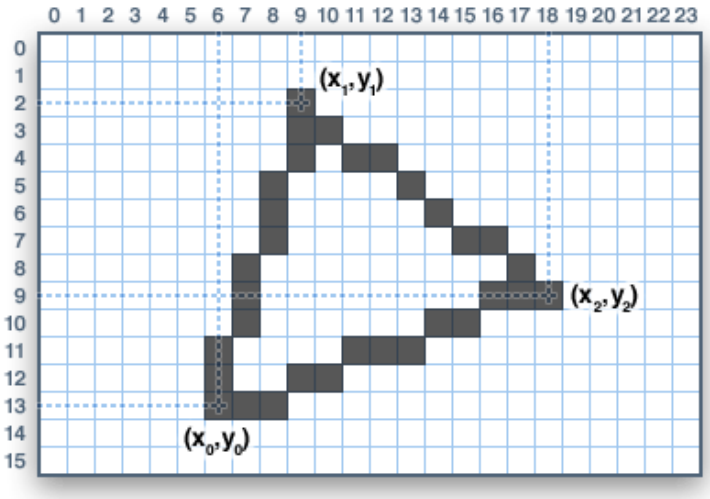



Here's an added bonus trick: because the circle functions are always drawn relative to a center pixel, the resulting circle diameter will always be an odd number of pixels. If an even-sized circle is required (which would place the center point between pixels) , this can be achieved using one of the rounded rectangle functions: pass an identical width and height that are even values, and a corner radius that's exactly half this value.

## Triangles

With triangles, once again there are the draw and fill functions. Each requires a full seven parameters: the X, Y coordinates for three corner points defining the triangle, followed by a color:

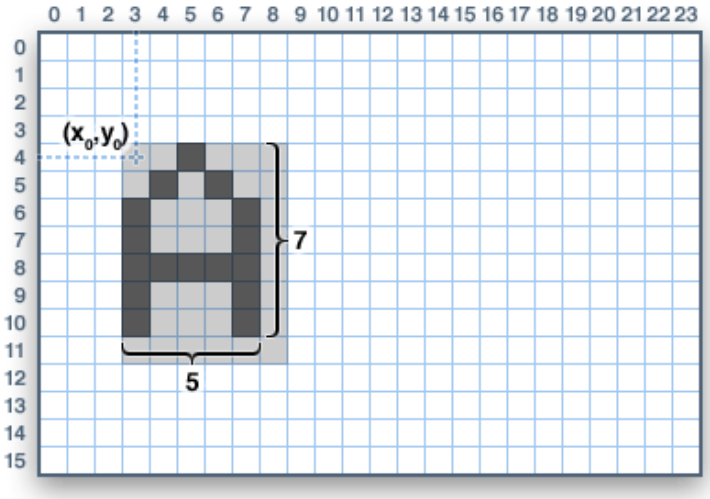
```
void drawTriangle(uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1, uint16_t x2,
uint16_t y2, uint16_t color);
void fillTriangle(uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1, uint16_t x2,
uint16_t y2, uint16_t color);
```




## Characters and text

There are two basic string drawing procedures for adding text. The first is just for a single character. You can place this character at any location and with any color. An optional size parameter can be passed which scales the font by this factor (e.g. size=2 will render the default font at 10x16 pixels per character). It's a little blocky that way but having just a single font helps keep the program size down.

```
void drawChar(uint16_t x, uint16_t y, char c, uint16_t color, uint16_t bg, uint8_t
size);
```



Text is very flexible but operates a bit differently. Instead of one procedure, the text size, color and position are set up in separate functions and then the print() function is used — this makes it easy and provides all of the same string and number formatting capabilities of Arduino's familiar Serial.print() and println() functions ()! But you precede these with the display object instead of Serial.

```
void setCursor(int16_t x0, int16_t y0);
void setTextColor(uint16_t color);
void setTextColor(uint16_t color, uint16_t backgroundcolor);
void setTextSize(uint8_t size);
void setTextWrap(boolean w);
```

Begin with setCursor(x, y), which will place the top left corner of the text wherever you please. Initially this is set to (0,0) (the top-left corner of the screen). Then set the text color with setTextColor(color) — by default this is white. Text is normally drawn "clear" — the open parts of each character show the original background contents, but if you want the text to block out what's underneath, a background color can be specified as an optional second parameter to setTextColor(). Finally, setTextSize(size) will multiply the scale of the text by a given integer factor. Below you can see scales of 1 (the default), 2 and 3. It appears blocky at larger sizes because we only ship the library with a single simple font, to save space.

Text background color is not supported for custom fonts (explained on "Using Fonts" page). For these, you will need to determine the text extents and explicitly draw a filled rectangle before drawing the text. This is on purpose and by design.

After setting everything up, you can use print() or println() — just like you do with Serial printing ()! For example, to print a string, use print("Hello world") - that's the first line of the image above. You can also use print() for numbers and variables — the second line above is the output of print(1234.56) and the third line is print(0xDEADBEEF, HEX).

By default, long lines of text are set to automatically "wrap" back to the leftmost column. To override this behavior (so text will run off the right side of the display — useful for scrolling marquee effects), use setTextWrap(false). The normal wrapping behavior is restored with setTextWrap(true).

## Extended Characters, CP437 and a Lurking Bug

The standard built-in font includes a number of symbols and accented characters outside the normal letters and numbers you'd use in print() strings. These can be accessed with drawChar(), passing an 8-bit value (0–255, though commonly expressed in hexadecimal, 0x00–0xFF) for the third argument.

The built-in font is based on the original IBM PC character set, known as Code Page 437 (CP437 for short) (). Many embedded systems still use this as it's compact and well established.

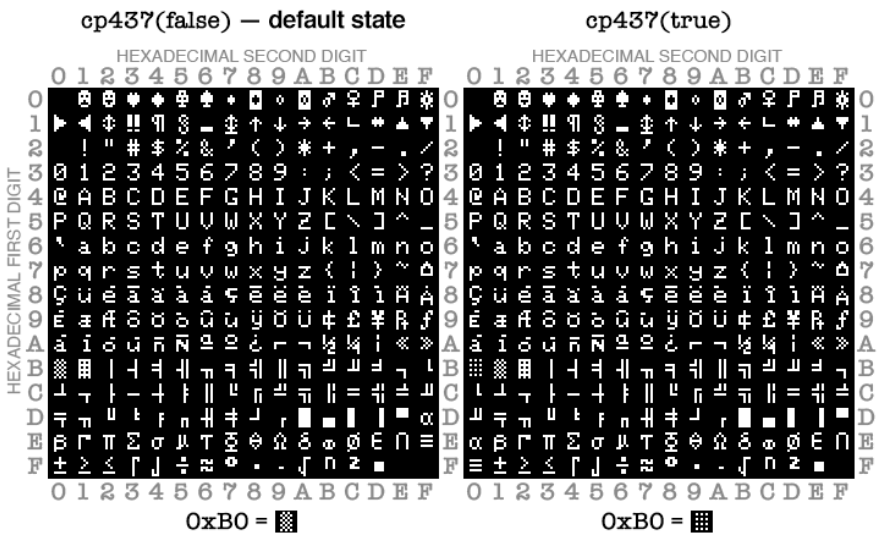
Years ago, when originally transcribing CP437 into the GFX library, one symbol was accidentally omitted. Nothing fatal, code runs fine, but every subsequent symbol was then off by one compared to the "real" CP437 character set. By the time this was discovered, so much code had been written — projects shared online but also in fixed media like books and magazines — that fixing the bug would break every existing project that relied on those extended characters!

So the error has been left in place, on purpose, but this creates a different issue if one is adapting code from elsewhere that relies on the correct CP437 symbol values.

A compromise solution is a function that enables or disables the "real" CP437 sequence. By default this is off, the off-by-one order is used, so that all the old GFX projects in books work without modification. The correct order can be enabled with:

```
display.cp437(true);
```

Here's a map of the built-in character set, both the standard erroneous version, and the corrected version used when one calls cp437(true). Notice this only affects the last five rows of symbols; everything prior to character 0xB0 is unaffected:




The presence of the extended Code Page 437 symbols is only guaranteed in the built-in font. Custom fonts (explained elsewhere) rarely include these.

See the "Using Fonts ()" page for additional text features in the latest GFX library.

## Bitmaps

You can draw small monochrome (single color) bitmaps, good for sprites and other mini-animations or icons:

```
void drawBitmap(int16_t x, int16_t y, uint8_t *bitmap, int16_t w, int16_t h,
uint16_t color);
```

This issues a contiguous block of bits to the display, where each '1' bit sets the corresponding pixel to 'color,' while each '0' bit is skipped. x, y is the top-left corner where the bitmap is drawn, w, h are the width and height in pixels.

The bitmap data must be located in program memory using the PROGMEM directive. This is a somewhat advanced function and beginners are best advised to come back to this later. For an introduction, see the Arduino tutorial on PROGMEM usage ().

Here's a handy webtool for generating bitmap -> memorymaps ()

### Clearing or filling the screen

The fillScreen() function will set the entire display to a given color, erasing any existing content:

```
void fillScreen(uint16_t color);
```

## Hardware-Specific functions

Some displays may have unique features like screen invert or hardware-based scrolling. Documentation for those functions can be found in the corresponding display-specific guide. Since these are not common features across all GFX-compatible displays, they are not described here.

---

# Rotating the Display

You can also rotate your drawing. Note that this will not rotate what you already drew, but it will change the coordinate system for any new drawing. This can be really handy if you had to turn your board or display sideways or upside down to fit in a particular enclosure. In most cases this only needs to be done once, inside setup().

We can only rotate 0, 90, 180 or 270 degrees - anything else is not possible in hardware and is too taxing for an Arduino to calculate in software


```
void setRotation(uint8_t rotation);
```

The rotation parameter can be 0, 1, 2 or 3. For displays that are part of an Arduino shield, rotation value 0 sets the display to a portrait (tall) mode, with the USB jack at the top right. Rotation value 2 is also a portrait mode, with the USB jack at the bottom left. Rotation 1 is landscape (wide) mode, with the USB jack at the bottom right, while rotation 3 is also landscape, but with the USB jack at the top left.

For other displays, please try all 4 rotations to figure out how they end up rotating as the alignment will vary depending on each display, in general the rotations move counter-clockwise

When rotating, the origin point (0,0) changes — the idea is that it should be arranged at the top-left of the display for the other graphics functions to make consistent sense (and match all the function descriptions above).

If you need to reference the size of the screen (which will change between portrait and landscape modes), use width() and height().

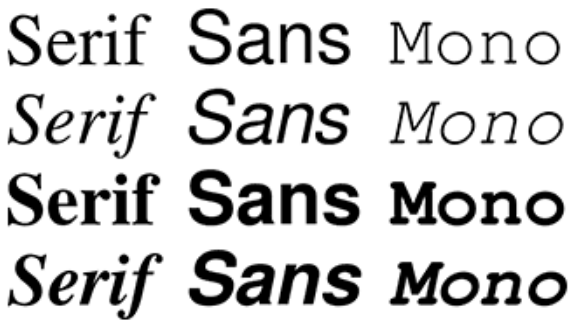
```
uint16_t width();
uint16_t height();
```

Each returns the dimension (in pixels) of the corresponding axis, adjusted for the display's current rotation setting.

---

# Using Fonts

More recent versions of the Adafruit GFX library offer the ability to use alternate fonts besides the one standard fixed-size and -spaced face that's built in. Several alternate fonts are included, plus there's the ability to add new ones.




The included fonts are derived from the GNU FreeFont () project. There are three faces: “Serif” (reminiscent of Times New Roman), “Sans” (reminiscent of Helvetica or Arial) and “Mono” (reminiscent of Courier). Each is available in a few styles (bold, italic, etc.) and sizes. The included fonts are in a bitmap format, not scalable vectors, as it needs to work within the limitations of a small microcontroller.

Located inside the “Fonts” folder inside Adafruit_GFX, the included files (as of this writing) are:

```
FreeMono12pt7b.h     FreeSansBoldOblique12pt7b.h
FreeMono18pt7b.h     FreeSansBoldOblique18pt7b.h
FreeMono24pt7b.h     FreeSansBoldOblique24pt7b.h
FreeMono9pt7b.h          FreeSansBoldOblique9pt7b.h
FreeMonoBold12pt7b.h     FreeSansOblique12pt7b.h
FreeMonoBold18pt7b.h     FreeSansOblique18pt7b.h
FreeMonoBold24pt7b.h     FreeSansOblique24pt7b.h
FreeMonoBold9pt7b.h      FreeSansOblique9pt7b.h
FreeMonoBoldOblique12pt7b.h FreeSerif12pt7b.h
FreeMonoBoldOblique18pt7b.h FreeSerif18pt7b.h
FreeMonoBoldOblique24pt7b.h FreeSerif24pt7b.h
FreeMonoBoldOblique9pt7b.h  FreeSerif9pt7b.h
FreeMonoOblique12pt7b.h      FreeSerifBold12pt7b.h
FreeMonoOblique18pt7b.h      FreeSerifBold18pt7b.h
FreeMonoOblique24pt7b.h      FreeSerifBold24pt7b.h
FreeMonoOblique9pt7b.h       FreeSerifBold9pt7b.h
FreeSans12pt7b.h     FreeSerifBoldItalic12pt7b.h
FreeSans18pt7b.h     FreeSerifBoldItalic18pt7b.h
FreeSans24pt7b.h     FreeSerifBoldItalic24pt7b.h
FreeSans9pt7b.h          FreeSerifBoldItalic9pt7b.h
FreeSansBold12pt7b.h     FreeSerifItalic12pt7b.h
FreeSansBold18pt7b.h     FreeSerifItalic18pt7b.h
FreeSansBold24pt7b.h     FreeSerifItalic24pt7b.h
FreeSansBold9pt7b.h      FreeSerifItalic9pt7b.h
```

Each filename starts with the face name (“FreeMono”, “FreeSerif”, etc.) followed by the style (“Bold”, “Oblique”, none, etc.), font size in points (currently 9, 12, 18 and 24 point sizes are provided) and “7b” to indicate that these contain 7-bit characters (ASCII codes “ ” through “~”); 8-bit fonts (supporting symbols and/or international characters) are not yet provided but may come later.

# Using GFX Fonts in Arduino Sketches

After #including the Adafruit_GFX and display-specific libraries, include the font file(s) you plan to use in your sketch. For example:

```
#include &lt;Adafruit_GFX.h&gt;    // Core graphics library
#include &lt;Adafruit_TFTLCD.h&gt; // Hardware-specific library
#include &lt;Fonts/FreeMonoBoldOblique12pt7b.h&gt;
#include &lt;Fonts/FreeSerif9pt7b.h&gt;
```

Each font takes up a bit of program space; larger fonts typically require more room. This is a finite resource (about 32K max on an Arduino Uno for font data and all of your sketch code), so choose carefully. Too big and the code will refuse to compile (or in some edge cases, may compile but then won't upload to the board). If this happens, use fewer or smaller fonts, or use the standard built-in font.

Inside these .h files are several data structures, including one main font structure which will usually have the same name as the font file (minus the .h). To select a font for subsequent graphics operations, use the setFont() function, passing the address of this structure, such as:

```
tft.setFont(&amp;FreeMonoBoldOblique12pt7b);
```

Subsequent calls to tft.print() will now use this font. Most other attributes that previously worked with the built-in font (color, size, etc.) work similarly here.

To return to the standard fixed-size font, call setFont(), passing either NULL or no arguments:

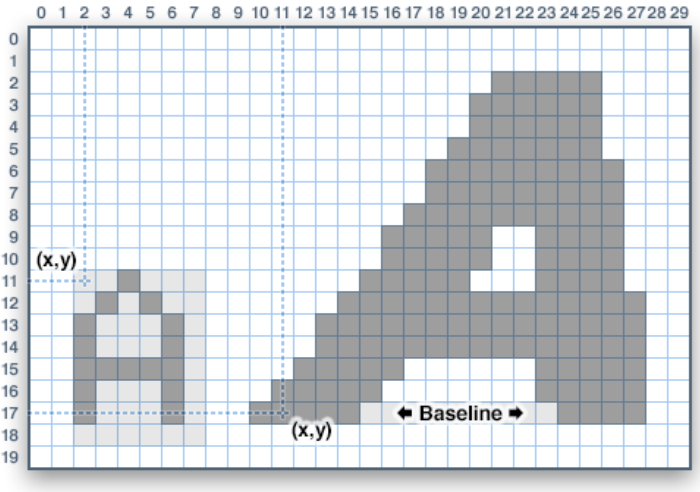
```
tft.setFont();
```

You can see a complete example of custom fonts in action in the MagTag Quotes Example () source code. It's really just a few extra lines compared to a "normal" GFX text program.

Some text attributes behave a little differently with these new fonts. Not wanting to break compatibility with existing code, the "classic" font continues to behave as before.

For example, whereas the cursor position when printing with the classic font identified the top-left corner of the character cell, with new fonts the cursor position indicates the baseline — the bottom-most row — of subsequent text. Characters may vary in size and width, and don't necessarily begin at the exact cursor column (as in below, this character starts one pixel left of the cursor, but others may be on or to the right of it).

When switching between built-in and custom fonts, the library will automatically shift the cursor position up or down 6 pixels as needed to continue along the same baseline.




One “gotcha” to be aware of with new fonts: there is no “background” color option… you can set this value but it will be ignored.

## This is on purpose and by design.

The background color feature is sometimes used with the “classic” font to overwrite old screen contents with new data. This only works because those characters are a uniform size; that won’t work with proportionally-spaced fonts, where the bounds of a string can vary, and an indeterminate number of characters may overlap the same region.

To replace previously-drawn text when using a custom font, either:

- Use getTextBounds() to determine the smallest rectangle encompassing a string, erase the area using fillRect(), then draw new text:

```
int16_t  x1, y1;
uint16_t w, h;

tft.getTextBounds(string, x, y, &amp;x1, &amp;y1, &amp;w, &amp;h);
```

getTextBounds expects a string, a starting cursor X&Y position (the current cursor position will not be altered), and addresses of two signed and two unsigned 16-bit integers. These last four values will then contain the upper-left corner and the width &

height of the area covered by this text — these can then be passed directly as arguments to fillRect().

This will unfortunately “blink” the text when erasing and redrawing, but is unavoidable. The old scheme of drawing background pixels in the same pass only creates a new set of problems.

or:

- Create a GFXcanvas1 object (an offscreen bitmap) for a fixed-size area, draw custom text in there and copy to the screen using drawBitmap().

```
// In global declarations:
GFXcanvas1 canvas(128, 32); // 128x32 pixel canvas

// In code later:
canvas.println("I like cake");
tft.drawBitmap(x, y, canvas.getBuffer(), 128, 32, foreground, background); // Copy
to screen
```

This is illustrative of syntax, not a complete program — change `x`, `y`, `foreground` and `background` to the desired coordinates and color values suited to the display. Some displays also require an explicit display() or show() call to refresh the screen contents.

This will be flicker-free but requires more RAM (about 512 bytes for the 128x32 pixel canvas shown above), so it’s not always practical on AVR boards with only 2K. Arduino Mega or any 32-bit board should manage fine.

See the “Minimizing Redraw Flicker” page for more info on using canvases.

# Adding New Fonts

If you want to create new font sizes not included with the library, or adapt entirely new fonts, we have a command-line tool (in the “fontconvert” folder) for this. It should work on many Linux- or UNIX-like systems (Raspberry Pi, Mac OS X, maybe Cygwin for Windows, among others).

Building this tool requires the gcc compiler and FreeType () library. Most Linux distributions include both by default. For others, you may need to install developer tools and download and build FreeType from the source (). Then edit the Makefile to match your setup before invoking “make”.

fontconvert expects at least two arguments: a font filename (such as a scalable TrueType vector font) and a size, in points (72 points = 1 inch; the code presumes a screen resolution similar to the Adafruit 2.8" TFT displays). The output should be redirected to a .h file...you can call this whatever you like but I try to be somewhat descriptive:

```
./fontconvert myfont.ttf 12 &gt; myfont12pt7b.h
```

The GNU FreeFont files are not included in the library repository but are easily downloaded (). Or you can convert most any font you like.

The name assigned to the font structure within this file is based on the input filename and font size, not the output. This is why I recommend using descriptive filenames incorporating the font base name, size, and "7b". Then the .h filename and font structure name can match.

The resulting .h file can be copied to the Adafruit_GFX/Fonts folder, or you can import the file as a new tab in your Arduino sketch using the Sketch→Add File... command.

If in the Fonts folder, use this syntax when #including the file:

```
#include &lt;Fonts/myfont12pt7b.h&gt;
```

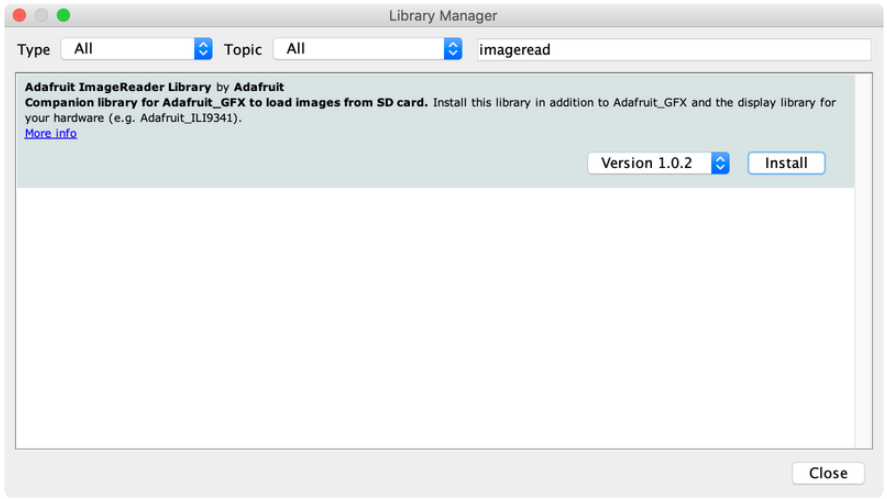
If a tab within your sketch, use this syntax:

```
#include "myfont12pt7b.h"
```

# Loading Images

Loading .BMP images from an SD card (or the flash memory chip on Adafruit "Express" boards) is an option for most of our color displays...though it's not built into Adafruit_GFX and must be separately installed.

The Adafruit_ImageReader library handles this task. It can be installed through the Arduino Library Manager (Sketch→Include Library→Manage Libraries...). Enter "imageread" in the search field and the library is easy to spot:




While you're there, also look for the Adafruit_SPIFlash library and install it similarly.

There's one more library required, but it can't be installed through the Library Manager. The Adafruit fork of the SdFat library needs to be downloaded as a .ZIP file, uncompressed and installed the old-school Arduino library way ().

Download SdFat (Adafruit fork) Arduino library

# Using the Adafruit_ImageReader Library

The syntax for using this library (and the separate installation above) are admittedly a bit peculiar...it's a side-effect of the way Arduino handles libraries. We purposefully did not roll this into Adafruit_GFX because any mere mention of an SD card library will incur all of that library's considerable memory requirements...even if one's sketch doesn't use an SD card at all! A majority of graphics projects are self-contained and don't reference files from a card...not everybody needs this functionality.

There are several example sketches in the Adafruit_ImageReader/examples folder. It's recommended that you dissect these for ideas how to use the library in your own projects.

They all start with several #includes...

```
#include &lt;Adafruit_GFX.h&gt;          // Core graphics library
#include &lt;Adafruit_ILI9341.h&gt;     // Hardware-specific library
#include &lt;SdFat.h&gt;                 // SD card &amp; FAT filesystem library
#include &lt;Adafruit_SPIFlash.h&gt;    // SPI / QSPI flash library
#include &lt;Adafruit_ImageReader.h&gt; // Image-reading functions
```

One of these lines may vary from one example to the next, depending which display hardware it's written to support. Above we see it being used with the Adafruit_ILI9341 display library required of certain shields, FeatherWings or breakout boards. Others examples reference Adafruit_HX8357, Adafruit_ST7735, or other color TFT or OLED display libraries...use the right one for the hardware you have.

Most of the examples can work from either an SD card, or the small flash storage drive that's on certain Adafruit "Express" boards. The code to initialize one or the other is a little different, and the examples check whether USE_SD_CARD is #defined to select one method vs. the other. If you know for a fact that your own project only needs to run on one type or the other, you really only need the corresponding initialization.

For SD card use, these two globals are declared:

```
SdFat                SD;         // SD card filesystem
Adafruit_ImageReader reader(SD); // Image-reader object, pass in SD filesys
```

For a flash filesystem, there are some special declarations made that help us locate the flash device on different Express boards, then declare three globals:

```
// SPI or QSPI flash filesystem (i.e. CIRCUITPY drive)
#if defined(__SAMD51__) || defined(NRF52840_XXAA)
  Adafruit_FlashTransport_QSPI flashTransport(PIN_QSPI_SCK, PIN_QSPI_CS,
    PIN_QSPI_IO0, PIN_QSPI_IO1, PIN_QSPI_IO2, PIN_QSPI_IO3);
#else
  #if (SPI_INTERFACES_COUNT == 1)
    Adafruit_FlashTransport_SPI flashTransport(SS, &amp;SPI);
  #else
    Adafruit_FlashTransport_SPI flashTransport(SS1, &amp;SPI1);
  #endif
#endif
Adafruit_SPIFlash    flash(&amp;flashTransport);
FatFileSystem        filesys;
Adafruit_ImageReader reader(filesys); // Image-reader, pass in flash filesys
```

The “reader” object will be used to access the image-loading functions later.

Then…we declare a display object (called “tft” in most of the examples) the usual way…for example, with the 2.8 inch TFT touch shield for Arduino, it’s:

```
#define SD_CS   4 // SD card select pin
#define TFT_CS 10 // TFT select pin
#define TFT_DC  9 // TFT display/command pin

Adafruit_ILI9341 tft = Adafruit_ILI9341(TFT_CS, TFT_DC);
```

That all takes place in the global variable section, even before the setup() function.

Now we need to do some work in setup(), and again it’s different for SD cards vs. flash filesystems…

For SD card use, it might look like this:

```
if(!SD.begin(SD_CS, SD_SCK_MHZ(25))) { // ESP32 requires 25 MHz limit
  Serial.println(F("SD begin() failed"));
  for(;;); // Fatal error, do not continue
}
```

This example is providing some very basic error handling…checking the return status of SD.begin() and printing a message to the Serial Monitor if there’s a problem.

Using a flash filesystem instead requires two steps:

```
if(!flash.begin()) {
  Serial.println(F("flash begin() failed"));
  for(;;);
}
```

```
  if(!filesys.begin(&amp;flash)) {
    Serial.println(F("filesys begin() failed"));
    for(;;);
  }
```

All other code is now the same regardless whether using an SD card or flash. That either/or setup required some extra steps but it's all smooth sailing now...

After the SD (or flash) and TFT's `begin()` functions have been called, you can then call `reader.drawBMP()` to load a BMP image from the card to the screen:

```
ImageReturnCode stat;
stat = reader.drawBMP("/purple.bmp", tft, 0, 0);
```

This accepts four arguments:

- A filename in "8.3" format (you shouldn't need to provide an absolute path (the leading "/"), but there are some issues with the SD library on some cutting-edge boards like the ESP32, so go ahead and include this for good measure).
- The display object where the image will be drawn (e.g. "tft"). This is the weird syntax previously mentioned...rather than tft.drawBMP(), it's reader.drawBMP(tft), because reasons.
- An X and Y coordinate where the top-left corner of the image is positioned (this doesn't need to be within screen bounds...the library will clip the image as it's loaded). 0, 0 will draw the image at the top-left corner...so if the image dimensions match the screen dimensions, it will fill the entire screen.

This function returns a value of type `ImageReturnCode`, which you can either ignore or use it to provide some diagnostic functionality. Possible values are:

- `IMAGE_SUCCESS` — Image loaded successfully (or was clipped fully off screen, still considered "successful" in that there was no error).
- `IMAGE_ERR_FILE_NOT_FOUND` — Could not open the requested file (check spelling, confirm file actually exists on the card, make sure it conforms to "8.3" file naming convention (e.g. "filename.bmp").
- `IMAGE_ERR_FORMAT` — Not a supported image format. Currently only uncompressed 24-bit color BMPs are supported (more will likely be added over time).
- `IMAGE_ERR_MALLOC` — Could not allocate memory for operation (drawBMP() won't generate this error, but other ImageReader functions might).

Rather than dealing with these values yourself, you can optionally call a function to display a basic diagnostic message to the Serial console:

```
reader.printStatus(stat);
```

If you need to know the size of a BMP image without actually loading it, there's the `bmpDimensions()` function:

```
int32_t width, height;
stat = reader.bmpDimensions("/parrot.bmp", &amp;width, &amp;height);
```

This accepts three arguments:

- A filename, same rules as the `drawBMP()` function.
- Pointers to two 32-bit integers. On successful completion, their contents will be set to the image width and height in pixels. On any error these values should be ignored (they're left uninitialized).

This function returns an `ImageReturnCode` as explained with the `drawBMP()` function above.

# Loading and Using Images in RAM

Depending on image size and other factors, loading an image from SD card to screen may take several seconds. Small images...those that can fit entirely in RAM...can be loaded once and used repeatedly. This can be handy for frequently-used icons or sprites, as it's usually much easier than converting and embedding an image as an array directly in one's code...a horrible process.

This introduces another ImageReader function plus a new object type, `Adafruit_Image`:

```
Adafruit_Image img;
stat = reader.loadBMP("/wales.bmp", img);
```

`loadBMP()` accepts two arguments:

- A filename, same rules as the previous functions.
- An `Adafruit_Image` object. This is a slightly more flexible type than the bitmaps used by a few drawing functions in the GFX library.

This returns an `ImageReturnCode` as previously described. If an image is too large to fit in available RAM, a value of `IMAGE_ERR_MALLOC` will be returned. Color images require two bytes per pixel…for example, a 100x25 pixel image would need 100*25*2 = 5,000 bytes RAM.

On success, the `img` object will contain the image in RAM.

The `loadBMP()` function is useful only on microcontrollers with considerable RAM, like the Adafruit "M0" and "M4" boards, or ESP32. Small devices like the Arduino Uno just can't cut it. It might be marginally useful on the Arduino Mega with very small images.

After loading, use the `img.draw()` function to display an image on the screen:

```
img.draw(tft, x, y);
```

This accepts three arguments:

- A display object (e.g. "tft" in most of the examples), similar to how `drawBMP()` worked.
- An X and Y coordinate for the upper-left corner of the image on the screen, again similar to `drawBMP()`.

We use `img.draw(tft,…)` rather than `tft.drawRGBBitmap(…)` (or other bitmap-drawing functions in the Adafruit_GFX library) because in the future we plan to add more flexibility with regard to image file formats and types. The `Adafruit_Image` object "understands" a bit about the image that's been loaded and will call the appropriate bitmap-rendering function automatically, you won't have to handle each separate case on your own.

If the image failed to load for any reason, `img.draw()` can still be called, it just won't do anything. But at least the sketch won't crash.

There is no BMP-to-flash function. This is on purpose and by design. We do something similar to that in the M4_Eyes () project and you're welcome to look through that code for insights, but generally speaking this is fraught with peril and not something we recommend. SD to screen or to RAM should cover most cases.

# Minimizing Redraw Flicker

A common need in microcontroller projects is to redraw all or part of a screen, such as when showing live readings from a sensor. The least-code approach to this usually is to erase all or part of the screen (using `fillScreen()` or `fillRect()`) and re-draw everything in the affected area. This does the job, but the off-and-on appearance can be distracting, especially if these redraws occur frequently and it becomes a steady flicker.

This isn't true of all GFX-compatible devices. Some displays (most LED matrices and some monochrome OLED screens) don't refresh until there's specifically a `show()`, `display()` or `update()` call in one's code (depending on the library), so this flicker is minimized or doesn't occur. Mostly it's an issue with color LCD or OLED screens, where graphics are rendered with every function call.

There are a couple of approaches one can use to minimize this effect. The first (and usually easiest) is suited to the standard fixed-size GFX font and is best for Arduino Uno and other memory-constrained microcontrollers. The other applies to custom fonts and any other graphics primitives, and is best for modern 32-bit microcontrollers with ample RAM (thought may still work on Uno for very small updates).

## Overwriting Text with the Built-In Font

This first method relies on the fact that the standard built-in font has uniformly-sized characters; it's sometimes referred to as the "5 by 7" pixel font (though really 6x8 pixels to allow at least 1 pixel between adjacent characters, and for descenders on some lowercase characters like "g" or "p"). Then...

The `setTextColor()` function, which normally accepts a single argument (a color to use for subsequent text printing), can optionally accept a second argument—a "background color" that applies to every pixel in the 6x8 box that's not part of the character shape. Normally each character box is transparent and only "foreground" pixels are set.

```
display.setTextColor(foreground, background);
```

Here's how that might be used in an Arduino sketch. Understand that this is not a complete program because every type of display has a distinct setup procedure. Com plete examples for PyPortal are given at the bottom of this page, providing a starting

point that can be adapted to other screen types. Look at the “graphicstest” example that accompanies most GFX-compatible libraries for insights.

```
// This is an incomplete Arduino example to minimally show
// the text overwrite approach. A real program would #include
// a display library header and declare a global 'display'.

void setup() {
  // Likewise, display initialization would take place here.

  // On color LCDs, this is white text on black background:
  display.setTextColor(0xFFFF, 0x0000);
  // On monochrome OLEDs, these might be 1 and 0 instead.
}

void loop() {
  display.setCursor(0, 0); // Position at top-left corner
  display.print("Hello");  // Print a message
  delay(1000);            // Pause 1 second
  display.setCursor(0, 0); // Back to top-left corner
  display.print("World");  // Print another message, same length
  delay(1000);            // Pause 1 second
}
```

The sketch alternately prints “Hello” and “World” at the top-left corner of the screen; each pass erases the text that came before, there’s no need to explicitly erase that area. (Try removing the second argument to `setTextColor()` and watch what happens.)

This works because both messages are the same 5-character length (30x7 pixels at the default text size, 60x14 at size 2 and so forth). If the messages are different lengths, it’s necessary to pad a string with extra spaces to overwrite the old text underneath.

One way to do this is by declaring a fixed-size character buffer and then using C’s for matted output via the `sprintf()` function. Let’s suppose a project will need up to 10 characters for each message. We begin by declaring a char array with 11 elements, because C strings require a trailing NUL (0) byte at the end:

```
char buf[11]; // 10 characters + NUL
```

Then we format a string into that buffer using `sprintf()` (string-print-formatted), some examples of which could include:

```
sprintf(buf, "%-10s", "Hello"); // Left-justified message
sprintf(buf, "%10s", "World");  // Right-justified message
sprintf(buf, "%10d", 42);       // Right-justified integer
```

And the buffer can then be passed to the normal `print()` or `println()` functions:

```
display.setCursor(x, y);
display.print(buf);
```

`sprintf()` has near infinite variety so we can't give every possible example here. Since it's a standard part of the C language, just searching around for "C formatted output" or just "sprintf" will turn up plenty of references. It's quite potent! Note however that the Arduino implementation is somewhat scaled back to fit on a microcontroller; formatting floating-point values this way is not supported, for example.

The counterpoint to using `sprintf()` is one of those great power, great responsibility lessons. String and memory handling in C (and thus C++, and thus the whole Arduino ecosystem) is simplistic, and there's nothing in place...other than your own self-discipline, you hope...to prevent exceeding the length of that `char` array, writing data willy-nilly into other RAM and leading to unexpected behavior or program crashes.

One approach to overwriting floating-point values is to use the normal Arduino `print()` function to the display, which accepts an optional argument specifying the number of digits after the decimal point, so the output is always the same size:

```
float value = 3.14159;
display.print(value, 5); // Will ALWAYS be extended to X.XXXXX, even if 0's
```

Another approach, if numbers or messages to print may vary in length, is just to follow up with enough spaces to cover up any change in the number of characters. But this relies on there not being any other stuff toward the right edge of the screen and isn't suited to every situation:

```
int value = 42;
display.setTextWrap(false); // Allow spaces to go off right edge
display.setCursor(0, 0);
display.print(value);
display.print("      ");    // Cover anything previously in this space
```

## Restoring Normal Text Drawing

To turn this off and draw normal "transparent" text, call `setTextColor()` with just the foreground color argument:

```
display.setTextColor(foreground);
```

## Overwriting Text or Graphics Using an Offscreen Canvas

The above method has some advantages in that it requires minimal modification to existing programs—something that prints once is easily adapted to print repeatedly—and that it fits well within modest microcontrollers like the Arduino Uno.

Where it doesn't work is with custom fonts, or with non-text elements like graphics or indicators. In fact, the optional second argument to `setTextColor()` (the background color) is simply ignored when using custom fonts. This is on purpose and by design! With proportionally-spaced fonts, strings will occupy different-sized regions, even if they contain the same number of characters...the overwrite technique simply can't be relied on.

The method explained here uses some extra RAM. Most 32-bit microcontrollers have ample capacity for this, but the classic Uno may struggle in all but the simplest cases.

The GFX library can provide an offscreen canvas. It works just like drawing to a screen...except there's no screen, just a grid of pixels in memory. The canvas can then be passed to another function (explained later), which does draw it to the screen.

Flicker-free redraw then works like this:

- Create a canvas object; usually done just once, at program startup
- Then, each time a screen update is needed:
    - Clear the canvas
    - Print text or draw shapes to the canvas
    - Copy the canvas to the screen

A canvas doesn't need to match the size of the screen; if you're just updating a rectangle, it only needs to be that size. That's important because every pixel takes a little RAM. Also a program can have more than one canvas if needed.

There are different canvas depths for 1, 8 and 16-bit color. We'll focus on just 1 and 16 here; the 8-bit case is seldom seen.

The 1-bit canvas type—`GFXcanvas1`—provides two colors; foreground and background, or foreground and transparent, much like working with the built-in font and `setTextColor()`. For most single-color things like text, this is what you'd use.

A canvas might be declared in the global part of one's sketch, before the `setup()` function, like so:

```
GFXcanvas1 canvas(width, height);
```

width and height should be the canvas dimensions, in pixels. Each pixel requires 1 bit of RAM...so for instance, 120x30 pixels = 3,600 bits = 450 bytes...plus a couple dozen bytes overhead for the `GFXcanvas1` structure itself. A single small canvas like that can usually work in the modest 1.5K of an Arduino Uno, but complex programs, larger or multiple canvases, or color (explained later) require more capable devices.

Canvases use all the same drawing functions as normally provided by the GFX library. So, where one might use `display.fillScreen(0)` before, one can use `canvas.fillScreen(0)` instead (though the canvas is not a screen, it's helpful to keep the names uniform across everything). This applies to all the pixel, shape and text-drawing functions. With a `GFXcanvas1` object, drawing colors must be 1 (foreground or "set" pixel) or 0 (background or "clear" pixel).

So the idea here is to just wipe and redraw the entire contents of the canvas each time a redraw is needed. Although GFX provides the `getTextBounds()` function, it just isn't necessary to go to such fuss to be "optimal"—canvases are already super quick to work with.

As before, this example is incomplete and just highlights the important ideas here. A full working example for PyPortal (and adaptable to other screens) is given at the bottom of the page.

```
// This is an incomplete Arduino example to minimally show
// the canvas drawing approach. A real program would #include
// a display library header and declare a global 'display',
// also including and enabling a custom font.

// Then, in ADDITION to all that, there's...
GFXcanvas1 canvas(120, 30); // 1-bit, 120x30 pixels

void setup() {
  // Display init and font select would take place here.
  // See later examples for that.

  // Text might exceed width of canvas, so disable wrapping:
  canvas.setTextWrap(false);
}

void loop() {
  canvas.fillScreen(0);    // Clear canvas (not display)
  canvas.setCursor(0, 24); // Pos. is BASE LINE when using fonts!
  canvas.print(millis());  // Print elapsed time in milliseconds
  // Copy canvas to screen at upper-left corner. As written here,
  // assumes a color LCD, hence the color values of 0xFFFF (white)
  // for foreground, 0x0000 (black) for background. Mono OLED can
  // use 1 and 0. BOTH colors must be specified to overwrite the
```

```
  // prior screen contents there.
  display.drawBitmap(0, 0, canvas.getBuffer(),
    canvas.width(), canvas.height(), 0xFFFF, 0x0000);
}
```

Notice how the fill, cursor and print operations are all performed on the `canvas` object, but the bitmap-drawing operation is done on the `display` object. It's easy to confuse these; if something like a custom font doesn't seem to be working, confirm you've set that for the canvas, not the display!

Because GFX "clips" graphics drawn to the canvas, this can be used for interesting effects like scrolling text within a rectangle in one section of a screen.

If you have multiple numbers or areas of the screen to update, and these are all the same dimensions, a single canvas can be re-used among them; it's not always necessary to allocate multiple distinct canvases, unless the size varies.

`drawBitmap()` works with all display types; the same function can be used with a `GFXcanvas1` regardless whether the screen is a 16-bit color TFT display or a black-and-white OLED.

## A Color Canvas

The 16-bit canvas type—`GFXcanvas16`—works much like a 16-bit LCD screen. Instead of foreground and background (or transparent) colors, one has the whole 64K gamut of colors to work with. If you're only planning to draw text, you probably don't need this, a `GFXcanvas1` will suffice, and you can specify any single color when copying to the display.

Like the 1-bit variety, this can be declared in the global part of one's sketch, before the `setup()` function:

```
GFXcanvas16 canvas(width, height);
```

Unlike the 1-bit variety, `GFXcanvas16` uses inordinate RAM; 2 bytes per pixel. That 120x30 pixel example from earlier now requires 7,200 bytes...way beyond the reach of the Arduino Uno's 1.5K RAM, but practical for more modern microcontrollers to handle.

There are some differences when copying a color canvas to the screen. First, one now uses the `drawRGBBitmap()` function, which accepts mostly the same arguments

but omits the foreground and background colors (since the canvas itself is now full color):

```
display.drawRGBBitmap(0, 0, canvas.getBuffer(), canvas.width(), canvas.height());
```

Second, `drawRGBBitmap()` only works on color screens, unlike `drawBitmap()` which works across all display types. Color reduction is a subjective process and would incur a lot of extra code, so this capability was omitted. Best to pair monochrome screens with `GFXcanvas1` instead.

## Examples

Here's the simple "text overwrite" example as written for PyPortal. This could be adapted to other screens by changing the display declaration and initialization; see the "graphicstest" example that accompanies most display libraries.

```
// Simple (text overwrite) flicker-free example for PyPortal

#include <Adafruit_GFX.h>
#include <Adafruit_ILI9341.h>

#define TFT_D0        34 // Data bit 0 pin (MUST be on PORT byte boundary)
#define TFT_WR        26 // Write-strobe pin (CCL-inverted timer output)
#define TFT_DC        10 // Data/command pin
#define TFT_CS        11 // Chip-select pin
#define TFT_RST       24 // Reset pin
#define TFT_RD         9 // Read-strobe pin
#define TFT_BACKLIGHT 25

// ILI9341 screen with 8-bit parallel interface:
Adafruit_ILI9341 display(tft8bitbus, TFT_D0, TFT_WR, TFT_DC, TFT_CS, TFT_RST,
TFT_RD);

void setup() {
  pinMode(TFT_BACKLIGHT, OUTPUT);       // PyPortal requires
  digitalWrite(TFT_BACKLIGHT, HIGH);    // turning on backlight

  display.begin();                      // Initialize and
  display.fillScreen(0x0000);           // clear display

  display.setTextColor(0xFFFF, 0x0000); // White text, black background
  display.setTextSize(2);               // 2X size text
}

void loop(void) {
  display.setCursor(0, 0); // Position at top-left corner
  display.print("Hello");  // Print a message
  delay(1000);             // Pause 1 second
  display.setCursor(0, 0); // Back to top-left corner
  display.print("World");  // Print another message, same length
  delay(1000);             // Pause 1 second
}
```

And here's a "1-bit canvas" example as written for PyPortal, using a large and friendly font. Again, this could be adapted to other screens by changing the display declaration and initialization; see the "graphicstest" example that accompanies most display libraries.

```
// Fancy (offscreen canvas) flicker-free example for PyPortal

#include <Adafruit_GFX.h>
#include <Adafruit_ILI9341.h>
#include <Fonts/FreeSerifBold18pt7b.h>

#define TFT_D0        34 // Data bit 0 pin (MUST be on PORT byte boundary)
#define TFT_WR        26 // Write-strobe pin (CCL-inverted timer output)
#define TFT_DC        10 // Data/command pin
#define TFT_CS        11 // Chip-select pin
#define TFT_RST       24 // Reset pin
#define TFT_RD         9 // Read-strobe pin
#define TFT_BACKLIGHT 25

// ILI9341 screen with 8-bit parallel interface:
Adafruit_ILI9341 display(tft8bitbus, TFT_D0, TFT_WR, TFT_DC, TFT_CS, TFT_RST,
TFT_RD);

GFXcanvas1 canvas(120, 30); // 1-bit, 120x30 pixels

void setup() {
  pinMode(TFT_BACKLIGHT, OUTPUT);       // PyPortal requires
  digitalWrite(TFT_BACKLIGHT, HIGH);    // turning on backlight

  display.begin();                      // Initialize and
  display.fillScreen(0x0000);           // clear display

  canvas.setFont(&FreeSerifBold18pt7b); // Use custom font and
  canvas.setTextWrap(false);            // clip text to canvas
}

void loop(void) {
  canvas.fillScreen(0);    // Clear canvas (not display)
  canvas.setCursor(0, 24); // Pos. is BASE LINE when using fonts!
  canvas.print(millis());  // Print elapsed time in milliseconds
  // Copy canvas to screen at upper-left corner. As written here,
  // assumes a color LCD, hence the color values of 0xFFFF (white)
  // for foreground, 0x0000 (black) for background. Mono OLED can
  // use 1 and 0. BOTH colors must be specified to overwrite the
  // prior screen contents there.
  display.drawBitmap(0, 0, canvas.getBuffer(),
    canvas.width(), canvas.height(), 0xFFFF, 0x0000);
}
```

Once more, using a 16-bit canvas instead. This example doesn't make good use of color in the canvas—it's still just white text on a black background—and is mostly just to show how the drawing syntax is a little different.

```
// Fancy (offscreen color canvas) flicker-free example for PyPortal

#include <Adafruit_GFX.h>
#include <Adafruit_ILI9341.h>
#include <Fonts/FreeSerifBold18pt7b.h>

#define TFT_D0        34 // Data bit 0 pin (MUST be on PORT byte boundary)
#define TFT_WR        26 // Write-strobe pin (CCL-inverted timer output)
```

```
#define TFT_DC        10 // Data/command pin
#define TFT_CS        11 // Chip-select pin
#define TFT_RST       24 // Reset pin
#define TFT_RD         9 // Read-strobe pin
#define TFT_BACKLIGHT 25

// ILI9341 screen with 8-bit parallel interface:
Adafruit_ILI9341 display(tft8bitbus, TFT_D0, TFT_WR, TFT_DC, TFT_CS, TFT_RST,
TFT_RD);

GFXcanvas16 canvas(120, 30); // 16-bit, 120x30 pixels

void setup() {
  pinMode(TFT_BACKLIGHT, OUTPUT);       // PyPortal requires
  digitalWrite(TFT_BACKLIGHT, HIGH);    // turning on backlight

  display.begin();                      // Initialize and
  display.fillScreen(0x0000);           // clear display

  canvas.setFont(&FreeSerifBold18pt7b); // Use custom font
  canvas.setTextWrap(false);            // Clip text within canvas
}

void loop(void) {
  canvas.fillScreen(0x0000); // Clear canvas (not display)
  canvas.setCursor(0, 24);   // Pos. is BASE LINE when using fonts!
  canvas.print(millis());    // Print elapsed time in milliseconds
  // Copy canvas to screen at upper-left corner.
  display.drawRGBBitmap(0, 0, canvas.getBuffer(), canvas.width(), canvas.height());
}
```