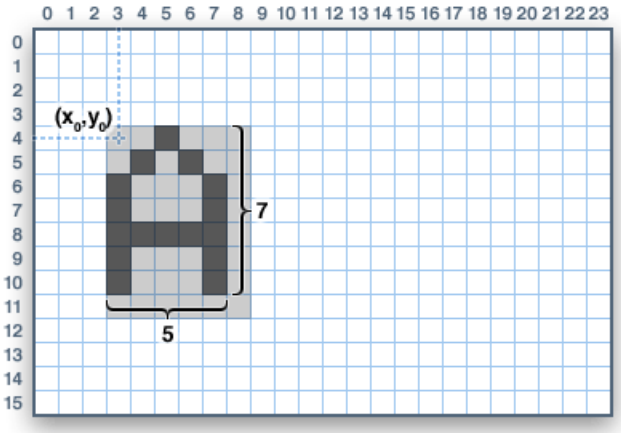
adafruit learning system

Adafruit GFX Graphics Library

Created by Phillip Burgess



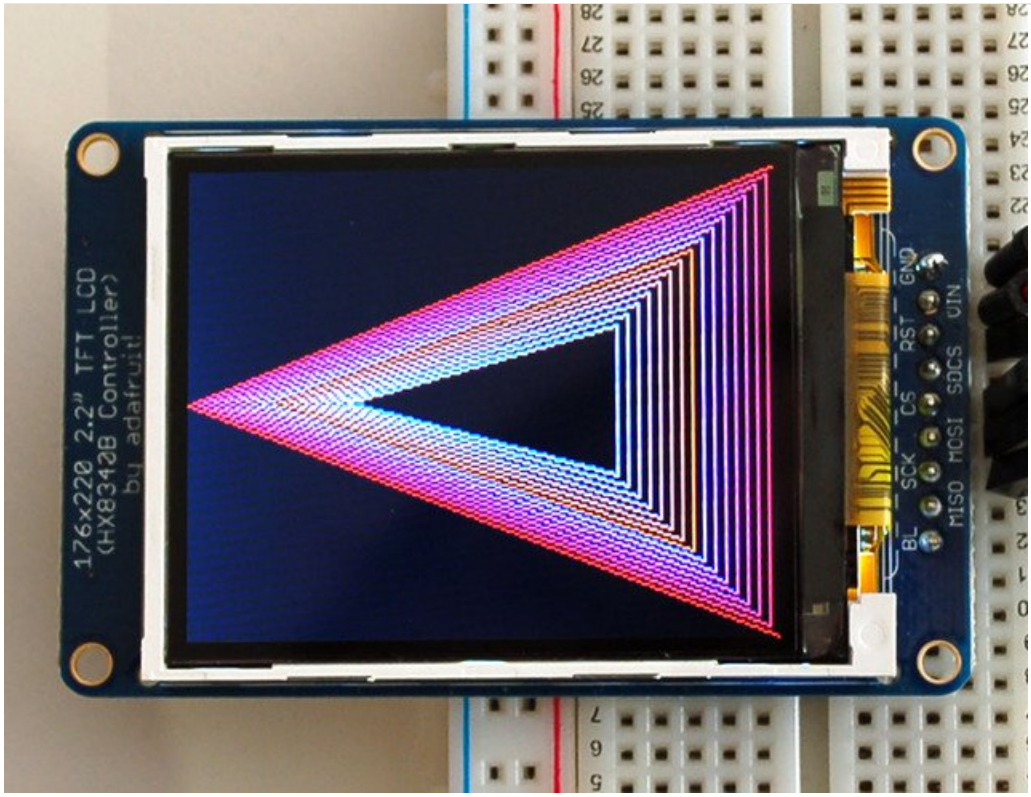
Last updated on 2018-08-22 03:31:08 PM UTC

Guide Contents

Guide Contents 2
Overview 3
Coordinate System and Units 5
Graphics Primitives 7
- Drawing pixels (points) 7
- Drawing lines 7
- Rectangles 8
- Circles 9
- Rounded rectangles 10
- Triangles 11
- Characters and text 11
- Bitmaps 13
- Clearing or filling the screen 13

Rotating the Display 15
Using Fonts 17
Using GFX Fonts in Arduino Sketches 18
Adding New Fonts 19

Overview



The Adafruit_GFX library for Arduino provides a common syntax and set of graphics functions for all of our LCD and OLED displays. 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.

https://adafru.it/cBB

https://adafru.it/cBB

The Adafruit_GFX library always works together with a second library provided for each specific display type — for example, the ST7735 1.8" color LCD requires installing both the Adafruit_GFX and Adafruit_ST7735 libraries. The following libraries now operate in this manner:

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

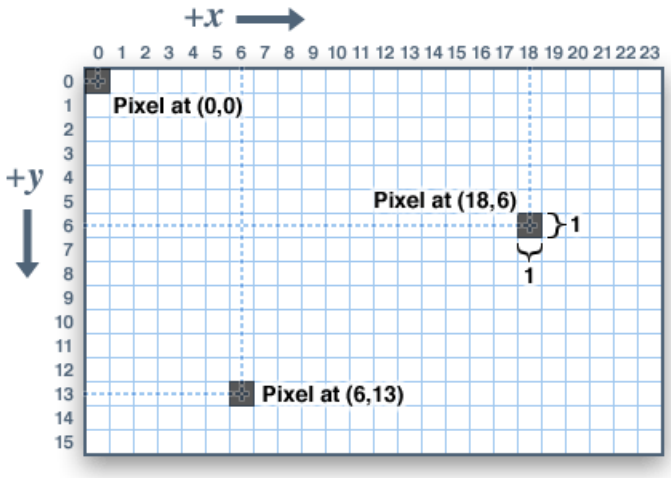
For information how to use and install libraries, see our tutorial (https://adafru.it/aYG)!

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

Coordinate System and Units

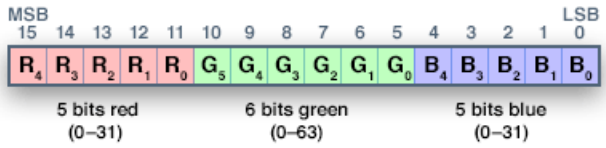
Pixels — picture elements, the blocks comprising a digital image — are addressed by their horizontal (X) and vertical (Y) coordinates. The coordinate system places the origin (0,0) at the top left corner, with positive X increasing to the right and positive Y increasing downward. This is upside-down 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.

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

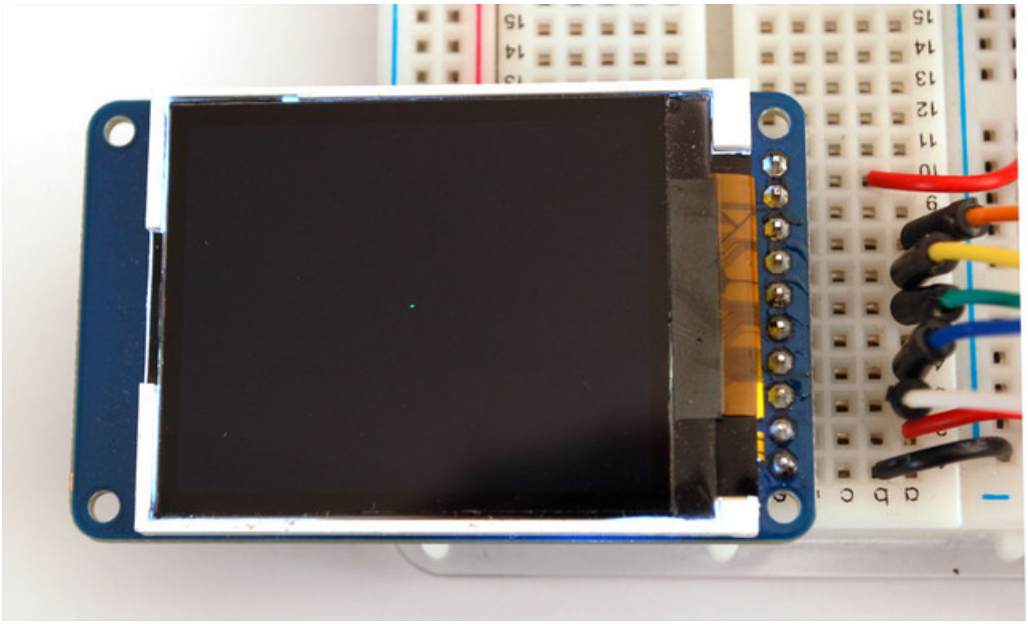
Each device-specific display library will have its own constructors and initialization functions. These are documented in the individual tutorials for each display type, or oftentimes are evident in the specific library header file. The remainder of this tutorial covers the common graphics functions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

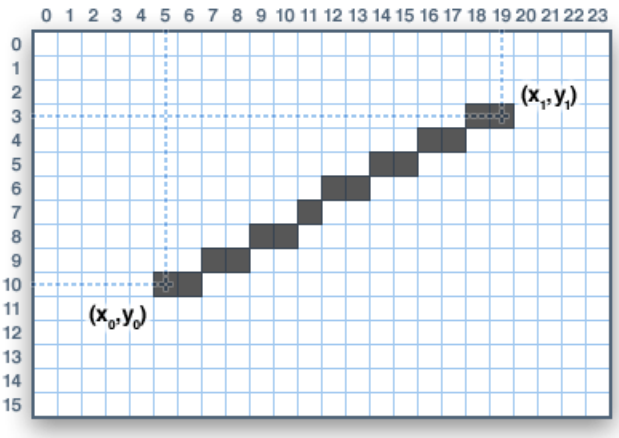
```
void drawPixel(uint16_t x, uint16_t y, uint16_t color);
```

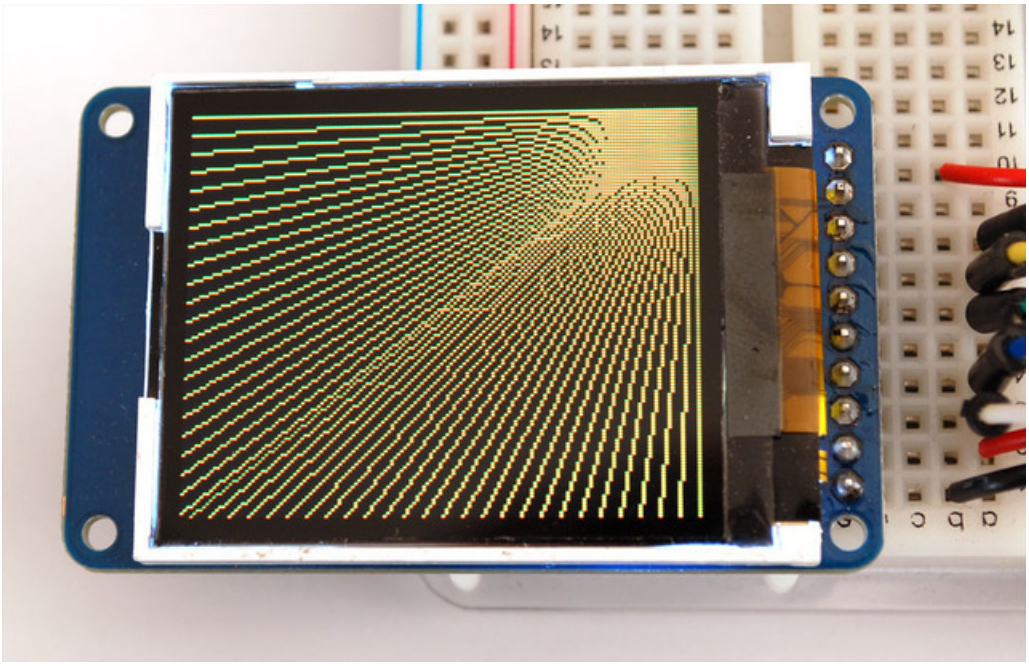


Drawing lines

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);
```





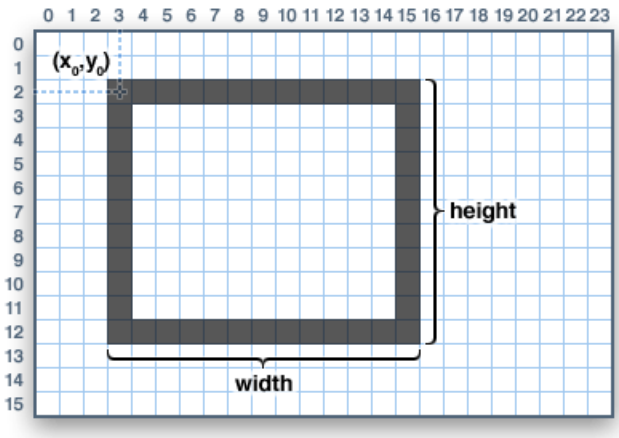
For horizontal or vertical lines, there are optimized line-drawing functions that avoid the angular calculations:

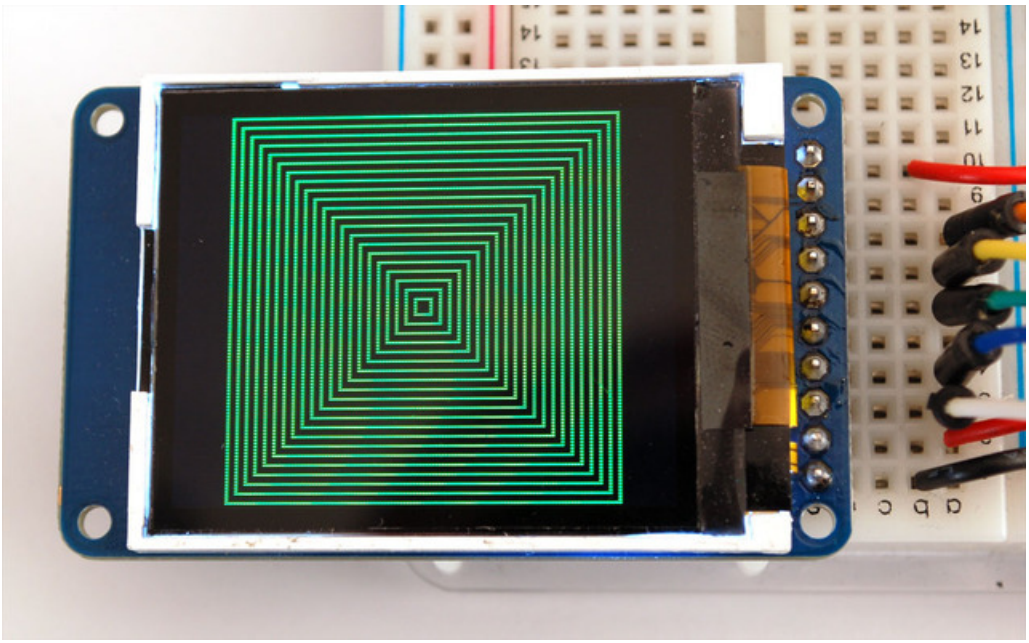
```
void drawFastVLine(uint16_t x0, uint16_t y0, uint16_t length, uint16_t color);
void drawFastHLine(uint8_t x0, uint8_t y0, uint8_t length, uint16_t color);
```

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);
```



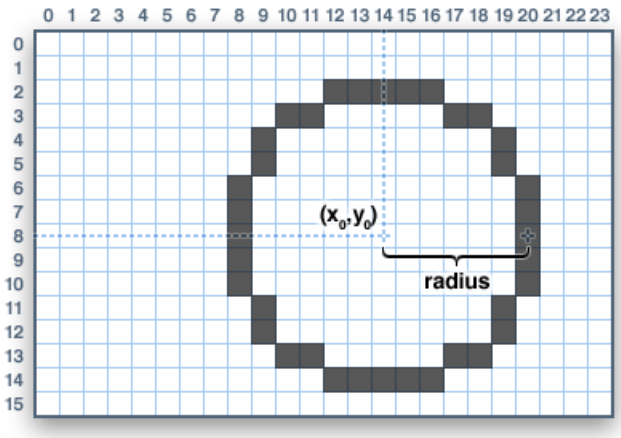


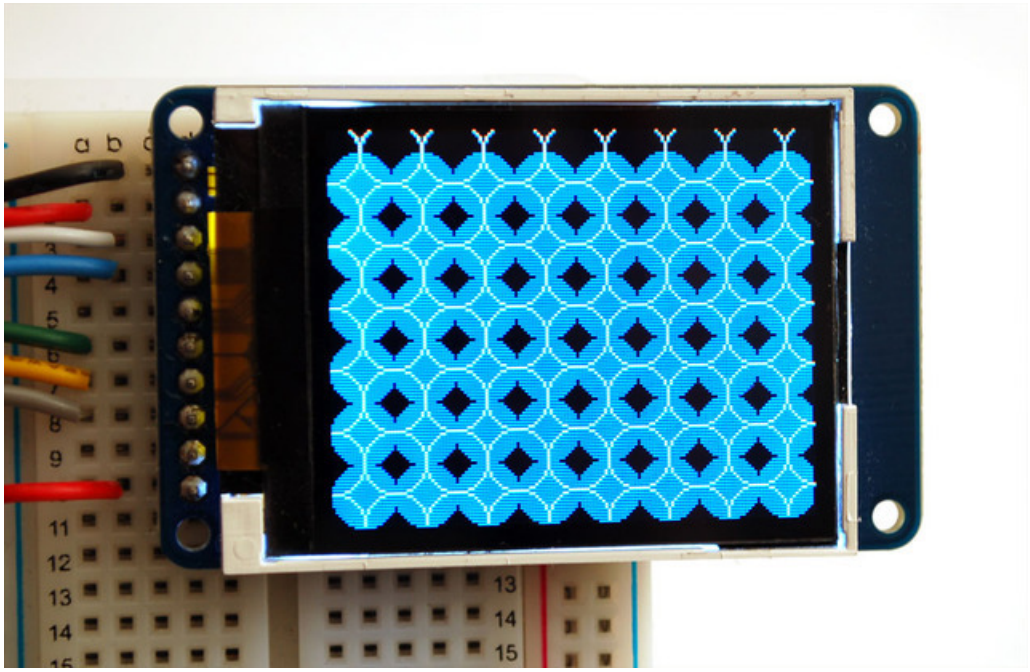
To create a solid rectangle with a contrasting outline, use fillRect() first, then drawRect() over it.

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);
```

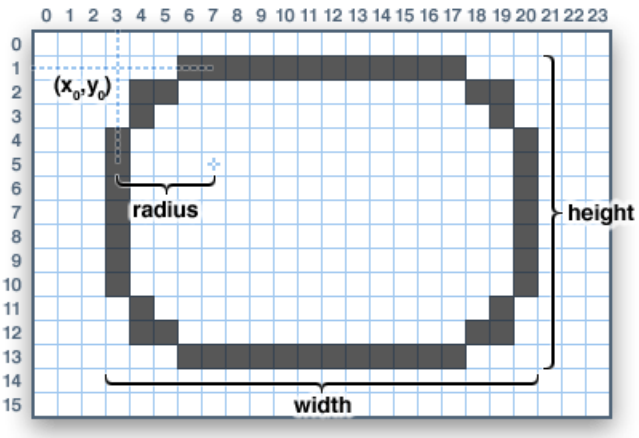




Rounded rectangles

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);
```

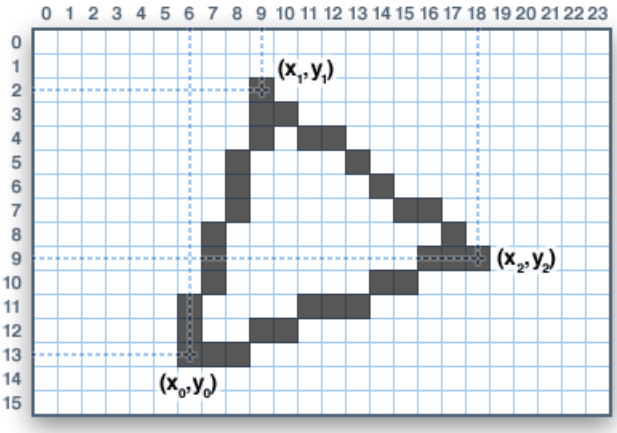


Here's an added bonus trick: because the circle functions are always drawn relative to a center pixel, the resulting circle diameter will always be an odd number of pixels. If an even-sized circle is required (which would place the center point *between* pixels), this can be achieved using one of the 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:

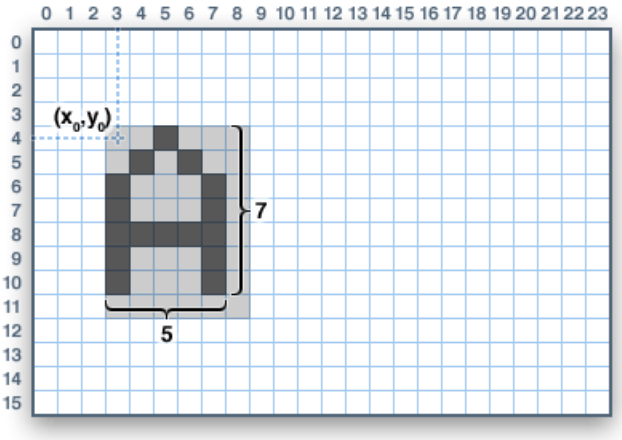
```
void drawTriangle(uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint16_t
void fillTriangle(uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint16_t
```



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. There's only one font (to save on space) and it's meant to be 5x8 pixels, but an optional size parameter can be passed which scales the font by this factor (e.g. size=2 will render the text at 10x16 pixels per character). It's a little blocky 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 the familiar Serial.print() function!

```
void setCursor(uint16_t x0, uint16_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 tosetTextColor(). 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.

Note that the text background color is not supported for custom fonts. For these, you will need to determine the text extents and explicitly draw a filled rectangle before drawing the text.

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 ofprint(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).

See the "Using Fonts (https://adafru.it/kAf)" 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 (https://adafru.it/aMw).

Here's a handy webtool for generating bitmap -> memorymaps (https://adafru.it/l3b)

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);
```

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 (https://adafru.it/kAg) 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 <Adafruit_GFX.h>    // Core graphics library
#include <Adafruit_TFTLCD.h> // Hardware-specific library
#include <Fonts/FreeMonoBoldOblique12pt7b.h>
#include <Fonts/FreeSerif9pt7b.h>
```

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(&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.

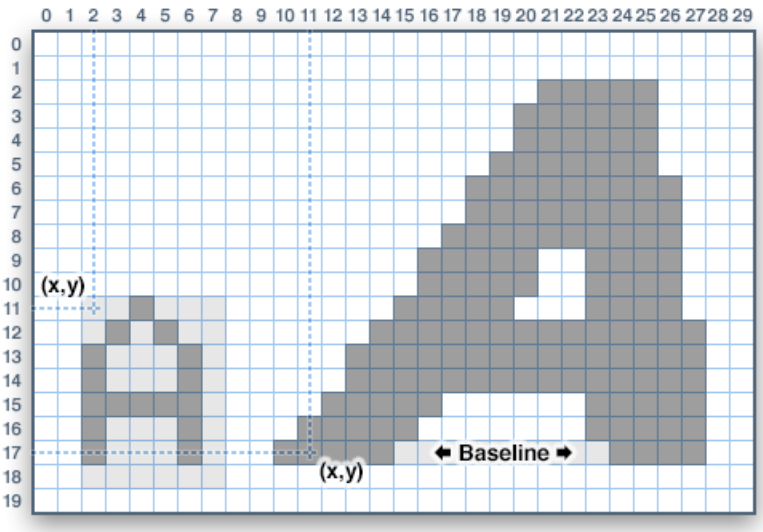
To return to the standard fixed-size font, call setFont(), passing either NULL or no arguments:

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

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 has typically been used with the “classic” font to overwrite old screen contents with new data. This only works because those characters are a uniform size; it’s not a sensible thing to do with proportionally-spaced fonts with characters of varying sizes (and which may overlap).

To replace previously-drawn test when using a custom font, use the getTextBounds() function 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, &x1, &y1, &w, &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.

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 (https://adafru.it/kAh) 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 (https://adafru.it/kAi). 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 > myfont12pt7b.h
```

The GNU FreeFont files are not included in the library repository but are easily downloaded (https://adafru.it/kAj). 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 "7p". 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 <Fonts/myfont12pt7b.h>
```

If a tab within your sketch, use this syntax:

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