



Les afficheurs graphiques

[Mise à jour le : 24/12/2021]



- **Ressource**

- [Adafruit GFX Graphics Library](#) pour **Arduino**.
- [BrainPad Drawing](#)

- **Lectures connexes**

- [Bibliothèque - Adafruit GFX Graphics Library](#)
- [0,96" 128x64 OLED 2864 Display module - SSD1306 \(I2C\)](#)
- [Adafruit 1,3" 128x64 OLED FeatherWing - SH1107 + 3 buttons \(I2C\)](#)
- [Adafruit 1.8" 128x160 Color TFT LCD display with MicroSD Card v2 - ST7735R \(SPI\)](#)



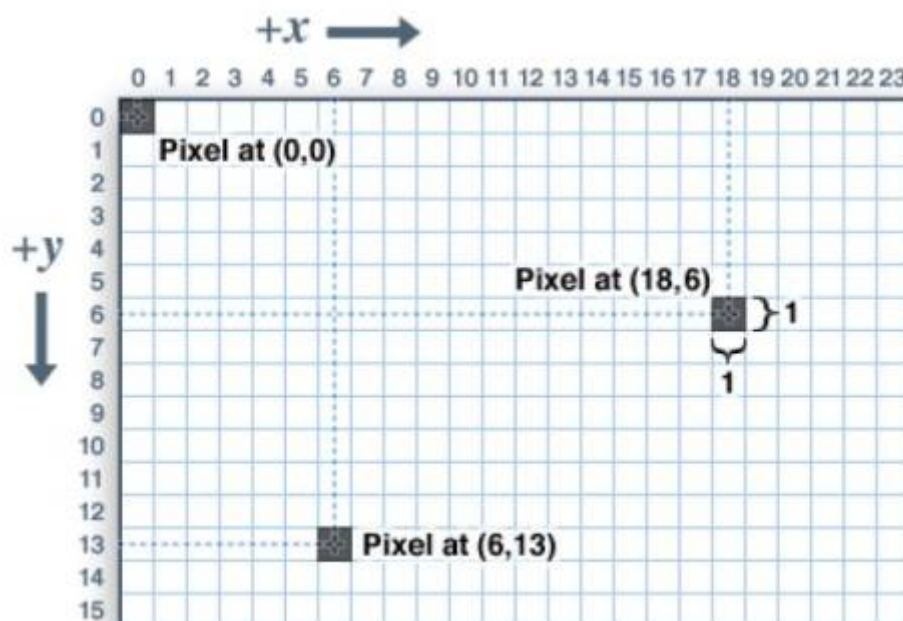
1. Introduction

Ce document décrit les principes implémentés dans les **bibliothèques graphiques** telles que [GFX Graphics Library](#) pour les cartes **Arduino** ou [Drawing](#) pour les cartes **BrainPad Pulse et Tick** de [GHI Electronics](#) en vue d'afficher du **texte**, des **dessins** ou des **images**.

2. Le système de coordonnées et les unités

Sur un écran, les **pixels** constituent une **image numérique**. On y accède par l'intermédiaire de leurs **coordonnées horizontales (X)** et **verticales (Y)**. Le système de coordonnées place l'origine (0,0) dans le coin supérieur gauche de l'écran, avec X positif croissant vers la droite et Y positif croissant vers le bas. L'axe Y est inversé par rapport au repère cartésien utilisé en mathématiques, mais c'est une pratique établie dans de nombreux systèmes graphiques informatiques. Si nécessaire, l'affichage

peut être pivoté.



Les **coordonnées** sont **toujours exprimées en pixels**; il n'y a pas d'échelle implicite au monde réel exprimé en millimètres ou en pouces, la taille d'un dessin sera fonction de la densité, en pixels, de l'afficheur. Si vous visez une représentation du monde réel, vous aurez besoin de mettre vos coordonnées à l'échelle. Le pas des points peut être trouvé dans la fiche technique de l'écran, ou en mesurant sa largeur et en divisant le nombre de pixels par cette mesure.



3. Les afficheurs monochromes

Pour les affichages monochromes (unicolores), les “couleurs” sont toujours spécifiées comme étant simplement **1 (afficher)** ou **0 (effacer)**. La sémantique **set/clear** est spécifique au type d'affichage : avec un affichage lumineux **OLED**, un pixel “set” est allumé, alors qu'avec un écran **LCD** réfléchissant, pour “set” le pixel est généralement sombre. Il peut y avoir des exceptions, mais généralement vous pouvez compter sur 0 (effacer) représentant l'état d'arrière-plan par défaut pour un affichage récemment initialisé.

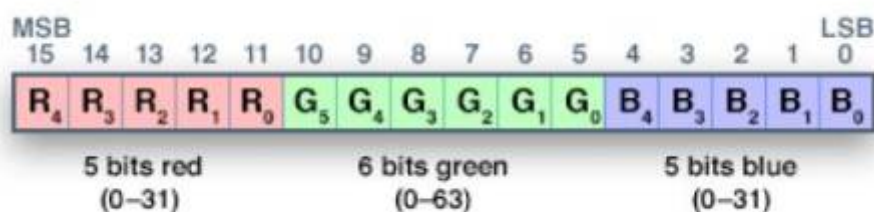


4. Les afficheurs couleur

Pour les afficheurs graphiques couleur, les couleurs sont représentées comme des valeurs sur **n bits non signées**.

Exemple

Codage **As565** d'une couleur sur 16 bits (65536 couleurs) dans la bibliothèque **Adafruit GFX Graphics Library**.



Ici, un bit supplémentaire est affecté au vert, car nos yeux sont plus sensibles à la lumière verte.

Codage des couleurs courantes

[couleur.cpp](#)

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

5. Les primitives graphiques

Ce document couvre les **fonctions graphiques courantes** fonctionnant de la même manière, quel que soit le type d'afficheur.

Chaque fonctionnalité graphique est proposée pour une carte **Arduino** (bibliothèque [GFX Graphics Library](#)) programmée en **C**, **C++** et pour une carte **BrainPad**, programmée en **Python**, disposant d'un afficheur monochrome à circuit SSD1306. Dans les exemples de code, les méthodes de la bibliothèque **GFX Graphics Library** sont appliquées à un objet **disp** supposé **déclaré** et **initialisé** grâce à la bibliothèque spécifique au périphérique. Les cartes BrainPad utilisent simplement des fonctions.

5.1 Bibliothèques et initialisation



- **Arduino**

Les bibliothèques à installer et la phase d'initialisation dépendent de l'afficheur utilisé. Le code ci-dessous permet une prise en main rapide des afficheurs nommés dans les onglets.

- [Ecran 0,96" à SSD1306](#)
- [Ecran 1,3" à SH1107](#)
- [Ecran 1,8" à ST7735](#)
- **Pour en savoir plus, voir :** [Adafruit 1.8" 128x160 Color TFT LCD display with MicroSD Card v2 - ST7735R \(SPI\)](#)



- **Déclaration** et **initialisation** de l'objet "écran graphique" **disp** nécessaire aux méthodes de la bibliothèque Adafruit GFX Graphics Library.

[init.cpp](#)

```
// Code testé sur un shield Adafruit 1,8" TFT avec 3 boutons, µSD et 5-Way Nav placé sur un Arduino Uno R3

// Bibliothèques
#include <SPI.h>
#include <Adafruit_GFX.h> // Bibliothèque de méthodes graphiques
```

```
#include <Adafruit_ST7735.h> // Bibliothèque spécifique au circuit de
l'afficheur
#include <Adafruit_seesaw.h> // Bibliothèque du convertisseur de bus
"Adafruit SeeSaw"
#include <Adafruit_TFTShield18.h> // Bibliothèque du Shield Adafruit
1,8" TFT avec 3 boutons, µSD et 5-Way Nav

Adafruit_TFTShield18 ss; // Objet shield pour contrôler la puce seesaw

// L'écran TFT et la carte SD partagent l'interface SPI.
// Pour la carte Arduino, le bus SPI est disponible sur
// pin 11 = MOSI, pin 12 = MISO, pin 13 = SCK.
#define SD_CS 4 // Sélection de la carte SD sur le Shield V2
#define TFT_CS 10 // Sélection de l'afficheur TFT sur le Shield V2
#define TFT_DC 8 // Lignes Données/Commandes de l'afficheur TFT sur
le Shield V2
#define TFT_RST -1 // Le reset de l'afficheur TFT est géré par seesaw !

// Déclaration d'un objet écran graphique
Adafruit_ST7735 disp = Adafruit_ST7735(TFT_CS, TFT_DC, TFT_RST);

void setup()
{
  Serial.begin(115200);
  while (!Serial);

  // Désactivation de l'afficheur et la carte SD
  pinMode(TFT_CS, OUTPUT);
  digitalWrite(TFT_CS, HIGH);
  pinMode(SD_CS, OUTPUT);
  digitalWrite(SD_CS, HIGH);

  // Démarrage de la puce Seesaw
  if (!ss.begin())
  {
    Serial.println("seesaw ne peut pas être initialisé!");
    while (1)
      ;
  }
  Serial.println("seesaw démarré");

  // Désactivation du rétroéclairage
  ss.setBacklight(TFTSHIELD_BACKLIGHT_OFF);
  // Reset de l'afficheur TFT
  ss.tftReset();

  // Initialisation de l'afficheur 1.8" TFT
  disp.initR(INITR_BLACKTAB); // Initialisation de la puce ST77355,
fond d'écran noir

  Serial.println("TFT OK!");
```

```
disp.fillScreen(ST77XX_BLACK); // fond d'écran noir

// Activation du rétroéclairage (100%)
// ss.setBacklight(TFTSHIELD_BACKLIGHT_ON);
// ou 30%
// ss.setBacklight(TFTSHIELD_BACKLIGHT_ON / 3);
// ou éclairage progressif
for (int32_t i = TFTSHIELD_BACKLIGHT_OFF; i < TFTSHIELD_BACKLIGHT_ON;
i += 100)
{
    ss.setBacklight(i);
    delay(1);
}
delay(100);

// Hello World !
// Remarque : pas de méthode display() pour cet afficheur
disp.setTextColor(ST77XX_WHITE); // texte blanc
disp.setCursor(0, 0);
disp.print("Hello World !");
}

void loop() {
    delay(10);
}
```

- **Pour en savoir plus, voir :** [Adafruit 1,3" 128x64 OLED FeatherWing - SH1107 + 3 buttons \(I2C\)](#)



- **Déclaration et initialisation** de l'objet "écran graphique" **disp** nécessaire aux méthodes de la bibliothèque Adafruit GFX Graphics Library.

init.cpp

```
// Code testé sur un ESP8266 sur lequel est placé un afficheur Adafruit
1,3" 128x64 OLED FeatherWing

// Bibliothèques
#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h> // Bibliothèque de méthodes graphiques
#include <Adafruit_SH110X.h> // Bibliothèque spécifique au circuit de
l'afficheur

// Déclaration d'un objet écran graphique
Adafruit_SH1107 disp = Adafruit_SH1107(64, 128, &Wire);

void setup(){
```

```
disp.begin(0x3C, true); // L'adresse de l'afficheur est 0x3C (par
défaut)

// Effacement du buffer d'affichage
disp.clearDisplay();
disp.display();

// Rotation de l'affichage : 90° horaire
disp.setRotation(1);

// Hello World !
disp.setTextColor(SH110X_WHITE);
disp.setCursor(0, 0);
disp.println("Hello World !");
disp.display();
}

void loop() {
  delay(10);
}
```

- **Pour en savoir plus, voir :** [Gravity 0,96" 128x64 OLED 2864 Display module - SSD1306 \(I2C\)](#)



- **Déclaration** et **initialisation** de l'objet "écran graphique" **disp** nécessaire aux méthodes de la bibliothèque Adafruit GFX Graphics Library.

[init.cpp](#)

```
// Code testé sur un Arduino Uno R3 auquel est connecté un afficheur
Gravity 0,96" 128x64 OLED

// Bibliothèques
#include "Wire.h"
#include "Adafruit_GFX.h" // Bibliothèque de méthodes graphiques
#include "OakOLED.h" // Bibliothèque spécifique au circuit de
l'afficheur

// Déclaration d'un objet écran graphique
OakOLED disp;

void setup() {
  // Initialisation de l'afficheur
  disp.begin();
}
```

```
// Hello World !
disp.setTextColor(1);
disp.setCursor(0, 0);
disp.println("Hello, World!");
disp.display();
}

void loop() {
  delay(10);
}
```



- **BrainPad**

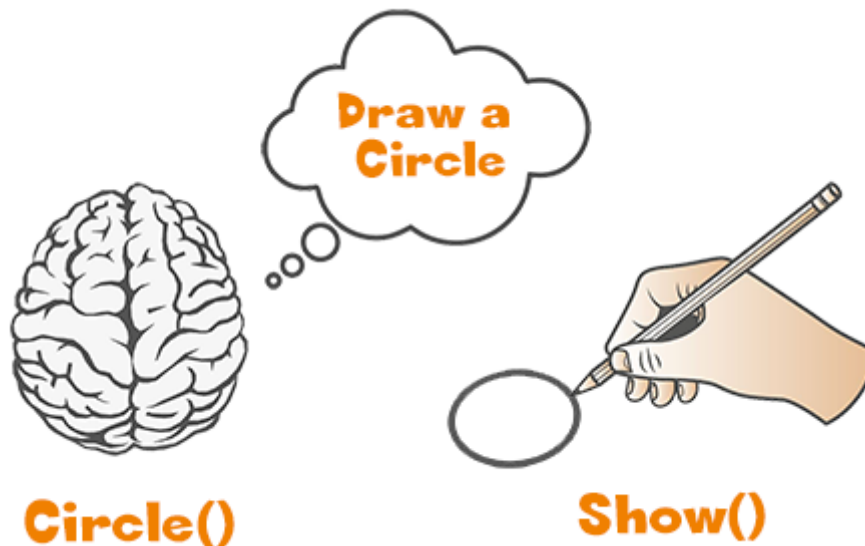
La bibliothèque graphique est intégrée. Il suffit d'ajouter la ligne suivante au début du code.

*.py

```
from BrainPad import *
```

5.2 Mémoire graphique, transférer l'image et effacer l'écran

Sur les systèmes informatiques, il est habituel de gérer les « dessins » en mémoire et non sur l'écran. On “dessine” en mémoire puis on transfère l'ensemble à l'afficheur. Cela ressemble à ce que nous faisons, nous pensons à ce que nous allons dessiner, puis nous le dessinons.



- [Adafruit GFX](#)
- [Brainpad](#)

• Syntaxe

La méthode **display()** transfère le contenu de la mémoire graphique à l'écran. Pour effacer l'écran, il faut **effacer** la mémoire graphique avec **ClearDisplay()** (sans effet sur l'affichage) et **transférer** son contenu (vide) avec **Display()**. Display() doit donc suivre un ClearDisplay().

```
void display();
void ClearDisplay();
```

• Exemple

*.cpp

```
// Exemple pour un afficheur Adafruit OLED FeatherWing
disp.setTextColor(SH110X_WHITE);
disp.setCursor(0,0);
disp.println("Hello World !");
disp.display();
```

• Syntaxe

La fonction **Show()** transfère le contenu de la mémoire graphique à l'écran. Pour effacer l'écran, il faut **effacer** la mémoire graphique avec **Clear()** (sans effet sur l'affichage) et **transférer** son contenu (vide) avec **Show()**. Show() doit donc suivre un Clear().

```
Clear()
Show()
```

- **Exemple**

*.py

```
# Code MicroPython
Clear()
Show()
```

Print() met à jour l'écran sans Show(). Cela est dû au code de la fonction Print() qui en facilite l'utilisation. Cependant, cela rend Print() plus lent que les autres fonctions de dessin. Une fois que vous avez décidé d'utiliser l'API d'affichage, vous ne devez pas utiliser Print().

5.3 Dessiner un point (pixel)

Le dessin d'un point (pixel) nécessite :

- les coordonnées x et y,
- la couleur.

- [Adafruit GFX](#)
- [Brainpad](#)

- **Syntaxe** pour dessiner un point aux coordonnées (x,y) dans la couleur *color*.

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

- **Exemple**

*.cpp

```
disp.drawPixel(10, 20, ST7735_GREEN);
```

- **Syntaxe**

Pour dessiner un point (un pixel) sur l'écran, la fonction **Point()** nécessite 3 arguments. Le premier est la coordonnée **x** sur l'écran, le second est la coordonnée **y**. Le dernier est la couleur. On notera que `Point()` est la seule fonction non affectée par `Color()`, et a plutôt besoin de couleur comme dernier argument.

Point(x,y,color)

- **Exemple**

*.py

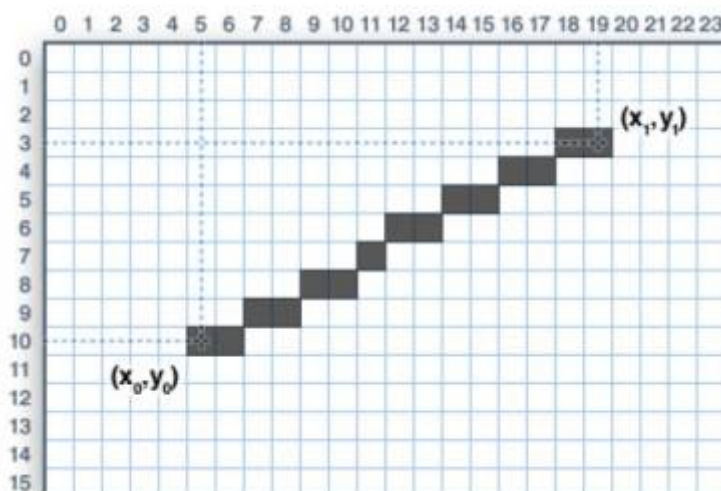
```
# Code MicroPython  
Point(64,32,White)
```

5.4 Dessiner une ligne

Le dessin d'une ligne nécessite :

- un point de départ (x0,y0),
- un point d'arrivée (x1,y1) et
- une couleur.

Pour les lignes horizontales ou verticales, il existe des fonctions optimisées.



- [Adafruit GFX](#)
- [Brainpad](#)

- **Syntaxe** pour dessiner une ligne entre le point de coordonnées (x0,y0) et le point de coordonnées (x1,y1) dans la couleur *color*.

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

Fonctions optimisées.

```
void drawFastVLine(uint16_t x0, uint16_t y0, uint16_t length, uint16_t color)
```

```
void drawFastHLine(uint16_t x0, uint16_t y0, uint16_t length, uint16_t color)
```

- **Exemple**

*.cpp

```
// Dessine une ligne
disp.drawLine(0, 0, 128, 160, ST7735_BLUE)

// Dessine une ligne verticale
disp.drawFastVLine(disp.width()/2, 0, disp.height(),
ST7735_WHITE);

//Dessine une ligne horizontale
disp.drawFastHLine(0, disp.height()/2, disp.width(),
ST7735_WHITE);
```

- **Syntaxe**

Pour dessiner une ligne, la fonction **Ligne()** nécessite 4 arguments : la position de départ **x1** et **y1** suivie de la position de fin **x2** et **y2**. Comme pour toutes les autres méthodes de dessin, il faut appeler **Show()** pour afficher la ligne à l'écran.

```
Ligne(x1, y1, x2, y2)
```

- **Exemple**

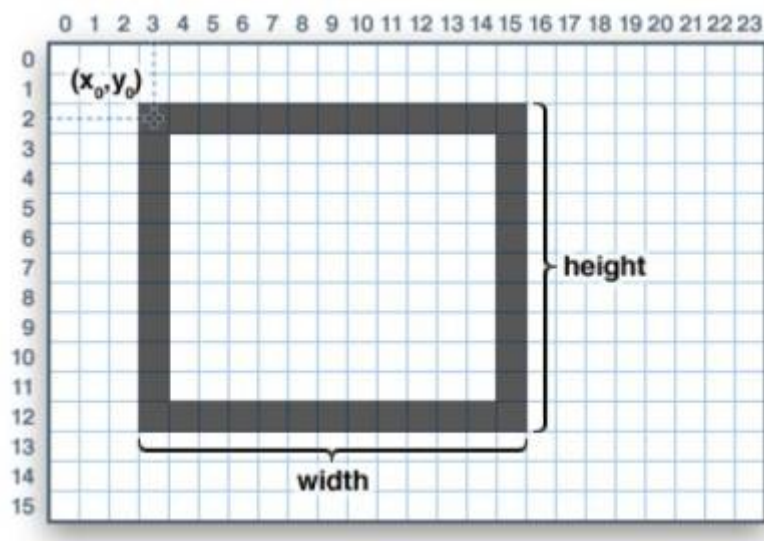
*.py

```
# Code MicroPython
Line(25,21,100,50)
```

5.5 Dessiner un rectangle

Pour dessiner ou remplir des rectangles et des carrés, il faut préciser :

- des coordonnées X, Y pour le coin supérieur gauche de la forme,
- une largeur et une hauteur (en pixels),
- une couleur.



- [Adafruit GFX](#)
- [Brainpad](#)

- **Syntaxe**

Les rectangles et les carrés peuvent être dessinés et remplis en utilisant les méthodes **drawRect()** et **fillRect()**. Chacune attend des coordonnées **x**, **y** pour le coin supérieur gauche du rectangle, une **largeur** et une **hauteur** (en pixels), et une **couleur**. **drawRect()** trace juste le cadre (contour) du rectangle, l'intérieur n'est pas affecté, tandis que **fillRect()** remplit toute la zone avec une couleur donnée.

```
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);
```

- **Exemple**

*.cpp

```
// Dessin du cadre (contour) d'un carré ou d'un rectangle  
disp.drawRect(40,85,20,30,ST7735_RED);  
  
// Remplissage d'un carré ou d'un rectangle
```

```
disp.fillRect(40,85,20,30,ST7735_RED);
```

Pour créer un rectangle uni avec un contour contrasté, utilisez d'abord `fillRect()`, puis `drawRect()` par-dessus.

• Syntaxe

La fonction **Rect()** dessine un rectangle à l'écran. `Rect()` nécessite 4 arguments : la coordonnée **x**, la coordonnée **y**, la largeur **w**, puis la hauteur **h**. La fonction **FillRect()** fonctionne comme `Rect()` mais remplit la forme.

```
Rect(x, y, w, h)  
FillRect(x, y, w, h)
```

• Exemple

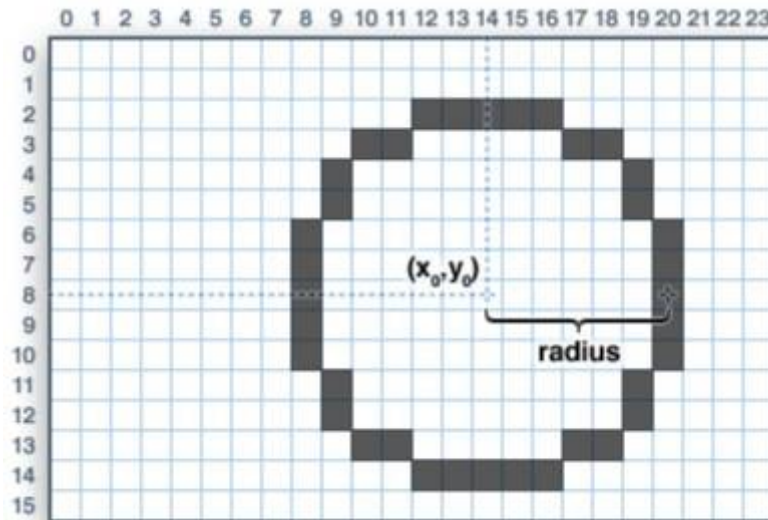
`*.py`

```
# Code MicroPython  
# Dessine un carré  
Rect(40,10,50,50)  
  
# Remplit le carré  
FillRect(40, 10, 50, 50)
```

5.6 Dessiner un cercle

Pour dessiner ou remplir un cercle, il faut préciser :

- des coordonnées X, Y pour le centre,
- un rayon en pixels,
- une couleur.



- [Adafruit GFX](#)
- [Brainpad](#)

• Syntaxe

Pour dessiner et remplir des cercles, on dispose des méthodes **drawCircle()** et **fillCircle()**. Chaque méthode accepte des coordonnées **x**, **y** pour le centre, un **rayon** en pixels et une **couleur**.

```
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);
```

• Exemple

*.cpp

```
// Dessine un cercle
disp.drawCircle(84,100,10,ST7735_GREEN);

// Remplit un cercle
disp.fillCircle(84,100,10,ST7735_GREEN);
```

• Syntaxe

La fonction **Circle()** génère un cercle en mémoire. Circle() nécessite 3 arguments : une position **x** et **y** sur l'afficheur et **r**, le rayon du cercle.

```
Circle(x,y,r)
```

- **Exemple**

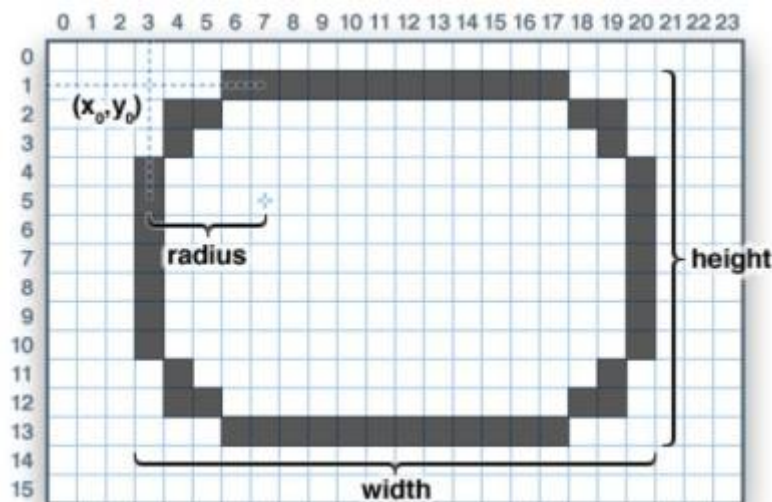
*.py

```
# Code MicroPython
Circle(60,32,25)
```

5.7 Dessiner un rectangle aux bords arrondis

Pour dessiner ou remplir des rectangles et des carrés aux bords arrondis, il faut préciser :

- des coordonnées X, Y pour le coin supérieur gauche de la forme,
- une largeur et une hauteur (en pixels),
- un rayon pour les coins (en pixels),
- une couleur.



- [Adafruit GFX](#)
- [Brainpad](#)

- **Syntaxe**

On dispose des méthodes **drawRoundRect()** et **fillRoundRect()**. Chaque méthode attend des coordonnées **x**, **y**, la **largeur** et la **hauteur** (tout comme les rectangles normaux), puis un **rayon** pour les coins (en pixels) et enfin la valeur de la **couleur**.

```
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);
```


• Exemple

*.cpp

```
// Dessin du cadre (contour) d'un carré ou d'un rectangle
disp.drawRoundRect(40,45,20,30,5,ST7735_YELLOW);

// Remplissage d'un carré ou d'un rectangle
disp.fillRoundRect(40,45,20,30,5,ST7735_YELLOW);
```

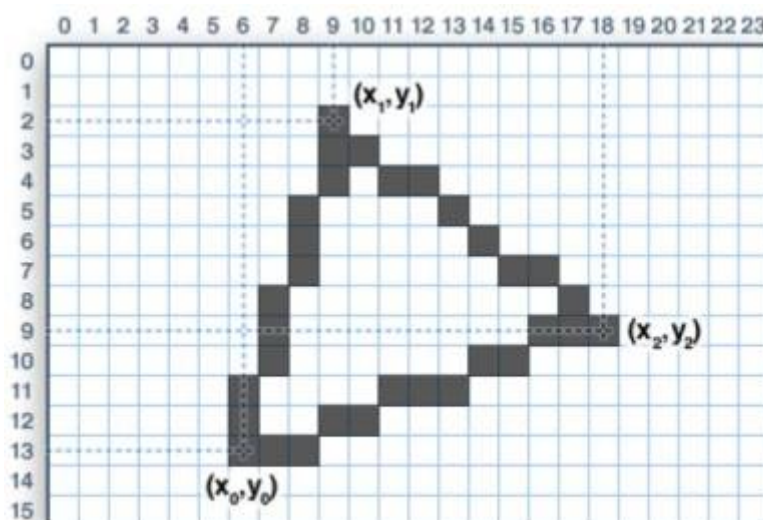
Non disponible sur Brainpad

Le diamètre d'un cercle comprend toujours un nombre impair de pixels (du fait de la présence d'un pixel au centre). Si pour réaliser un cercle, un diamètre comprenant un nombre pair de pixels est nécessaire (ce qui place le point central entre les pixels), cela peut être réalisé en dessinant un carré aux bords arrondis. En effet, il suffit de définir une largeur et une hauteur identiques et un rayon de coin correspondant exactement à la moitié de cette valeur.

5.8 Dessiner un triangle

Pour dessiner ou remplir un triangle, il faut préciser :

- les coordonnées X, Y pour les trois angles,
- la couleur.



- [Adafruit GFX](#)
- [Brainpad](#)

- **Syntaxe**

Les triangles disposent également des méthodes **draw** et **fill**. Chacune nécessite sept paramètres: les coordonnées **x**, **y** pour trois angles définissant le triangle, suivis de la **couleur**.

```
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);
```

- **Exemple**

[*.cpp](#)

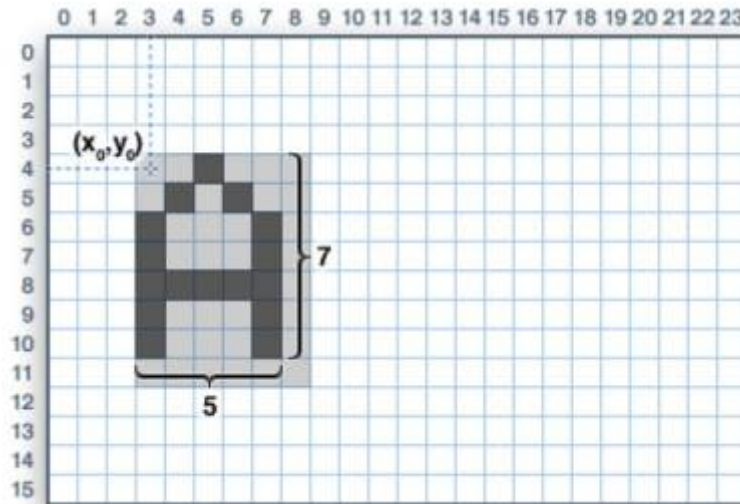
```
// Dessin d'un triangle  
disp.drawTriangle(100, 45, 70, 45, 70, 75, ST7735_CYAN);  
  
// Remplissage d'un triangle  
disp.fillTriangle(100, 45, 70, 45, 70, 75, ST7735_CYAN);
```

Non disponible sur Brainpad

5.9 Afficher du texte ou un caractère

Il est possible de placer du texte ou un caractère n'importe où sur l'écran dans une police particulière en lui donnant une couleur et une taille. Pour cela, il faut préciser :

- les coordonnées X, Y,
- le caractère ou le texte,
- la couleur,
- la taille.



- [Adafruit GFX](#)
- [Brainpad](#)

- **Caractère**

- **Syntaxe**

Un **caractère** peut être placé n'importe où sur l'écran dans n'importe quelle couleur. Il n'existe qu'une seule police (pour économiser sur la place mémoire). Ses dimensions sont **5x8 pixels** par défaut. Il est possible de passer un paramètre supplémentaire pour augmenter sa taille.

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

- **Exemples**

**.cpp*

```
// Dessin d'un caractère
disp.drawChar(110, 140, '@', ST7735_YELLOW, ST7735_BLACK, 2);
```

- **Text (chaîne de caractères)**

- **Syntaxe**

La taille du **texte (chaîne de caractères)**, la couleur et la position sont définies dans des fonctions séparées, puis la fonction **print()** ou **println()** peut être utilisée pour afficher le texte.

Positionnement du curseur en (x0, y0). (0,0) par défaut.

void **setCursor**(uint16_t x0, uint16_t y0);

Sélection de la couleur du texte. Blanc par défaut.

void **setTextColor**(uint16_t color);

ou Sélection de la couleur du texte avec le paramètre color et de l'arrière-plan

avec `backgroundColor`.

```
void setTextColor(uint16_t color, uint16_t backgroundcolor);
```

Sélection de la taille du texte. Un par défaut.

```
void setTextSize(uint8_t size);
```

Après avoir paramétré le texte, on utilise `print()` ou `println()`.

```
void print(string text);
```

```
void println(string text);
```

Permet de créer un effet de défilement `setTextWrap (false)`. Le texte revient à gauche par défaut `setTextWrap (true)`.

```
void setTextWrap(boolean w);
```

◦ Exemples

*.cpp

```
// Positionnement du curseur en (x0, y0)
disp.setCursor(5,5);
// Sélection de la couleur du texte
disp.setTextColor(ST7735_YELLOW);
// Sélection de la couleur du texte avec le paramètre color et
// de l'arrière-plan avec backgroundColor.
disp.setTextColor(ST7735_BLACK,ST7735_WHITE);
// Sélection de la taille du texte.
disp.setTextSize(2); // Taille du texte initiale x2
// Affiche une chaîne de caractère.
disp.print("Un texte sans retour à la ligne");
// Affiche une chaîne de caractère et retour à la ligne.
disp.println("Un texte avec retour à la ligne");
// setTextWrap (false) permet de créer un effet de défilement
// . Le texte revient à gauche par défaut setTextWrap (true).
tft.setTextWrap(true);
```

• Syntaxe

La fonction **Text()**, similaire à `Print()` permet de contrôler la position d'un texte à l'écran. `Text()` prend 3 arguments : le **texte** à afficher suivi de son emplacement **x** et **y** sur l'écran.

Contrairement à la fonction `Print()`, elle doit être suivie par **Show()**.

TextEx() fonctionne comme `Text()`, mais ajoute une **mise à l'échelle** afin de créer un texte plus gros, mince ou long. `TextEx()` prend 5 arguments. Les trois premiers sont exactement les mêmes que la fonction `Text()`. Les deux derniers définissent la largeur **scaleWidth** et la hauteur **scaleHeight** de l'échelle. L'échelle 1 est la taille standard, 2 est le double et ainsi de suite.

```
Text(texte, x, y)
```

```
TextEx(texte, x, y, scaleWidth, scaleHeight)
```

• Exemple

*.py

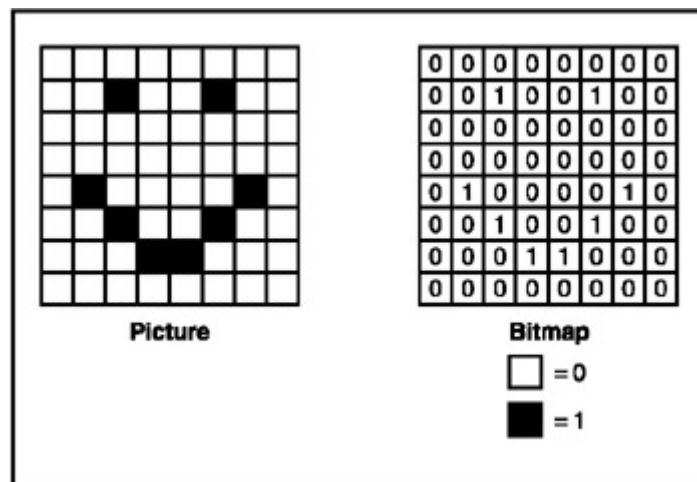
```
# Code MicroPython
Text("Hello Brain",30,30)

TextEx("Large Text",0,30,2,2)
```

5.10 Bitmaps

Il est possible d'afficher de petites images **bitmap** pour réaliser des **sprites** et autres mini animations ou des **icônes**. Pour cela, il faut préciser :

- les coordonnées X, Y,
- un bloc contigu de bits, où chaque bit à '1' définit le pixel à 'colorier', tandis qu'on ne tient pas compte de chaque bit à '0'.
- une largeur et une hauteur (en pixels),
- une couleur.



- [Adafruit GFX](#)
- [Brainpad](#)
- **Syntaxe** pour dessiner à la position (x,y), une image constituée du bloc de code adressé par le pointeur **bitmap** de largeur **w** et de hauteur **h** dans la couleur **color** (pixels à 1) et avec un fond **bg** (pixels à 0). Les données bitmap doivent être situées dans la mémoire du programme (**flash**) en utilisant la directive **PROGMEM** [lien](#).

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

• Exemple

*.cpp

```
/* Rectangle constitué de 4 lignes rouges, 2 lignes vertes, 4
lignes rouges
La forme est contenue dans un tableau placé en mémoire flash */
const PROGMEM byte sprite[] = {
0xff,0xff,0xff,0xff,0xff,
0xff,0xff,0xff,0x00,0x00,
0x00,0x00,0xff,0xff,0xff,
0xff,0xff,0xff,0xff,0xff
};
// Pixel à 1 en vert, pixel à 0 en rouge
disp.drawBitmap(20, 20, sprite, 16, 10, ST7735_GREEN, ST7735_RED);
```

Outil pour générer des bitmap : [image2cpp](#)

• Syntaxe

La fonction **CreateImage()** est utilisée pour créer une image utilisée par la suite avec la fonction **Image()** qui la positionne à l'écran. L'un des arguments requis pour la fonction **CreateImage()** peut prendre une **chaîne** ou un **tableau**. On utilise **0** pour le **noir** et **1** pour le **blanc**.

La fonction **CreateImage()** nécessite 6 arguments. Les deux premiers sont la taille de l'image **pixH** x **pixV** en nombre de pixels. Le 3ème est la variable **nomVar** contenant le tableau ou la chaîne. Les 4e et 5e arguments sont utilisés pour redimensionner l'image horizontalement **dimH** et verticalement **dimV**. Le dernier argument est la transformation de l'image **transf** résumé dans le tableau ci-dessous.

| Valeur | Transformation |
|--------|----------------------------------|
| 0 | Pas de transformation |
| 1 | Retourne l'image horizontalement |
| 2 | Retourne l'image verticalement |
| 3 | Fait pivoter l'image de 90° |
| 4 | Fait pivoter l'image de 180° |
| 5 | Fait pivoter l'image de 270° |

La fonction **Image()** nécessite 3 arguments : l'image **varImage** créée à l'aide de **CreateImage()** et la position **x** et **y** où elle apparaîtra à l'écran.

```
varImage = CreateImage(pixH, pixV, nomVar, dimH, dimV, transf)
Image(varImage, x, y)
```

• Exemple

*.py

```
# Code MicroPython
# Tableau
alien = [
    0, 0, 0, 1, 1, 0, 0, 0,
    0, 0, 1, 1, 1, 1, 0, 0,
    0, 1, 1, 1, 1, 1, 1, 0,
    1, 1, 0, 1, 1, 0, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1,
    0, 0, 1, 0, 0, 1, 0, 0,
    0, 1, 0, 1, 1, 0, 1, 0,
    1, 0, 1, 0, 0, 1, 0, 1]

# ou

# set
alien = (
    "  XX  " +
    " XXXX " +
    " XXXXXX " +
    "XX XX XX" +
    "XXXXXXXX" +
    "  X  X  " +
    " X XX X " +
    "X X  X X" )

monsterImage = CreateImage(8,8,alien,1,1,0)

Image(monsterImage,55,30)
```

5.11 Autres primitives

Il est généralement possible :

- de **remplir** l'écran avec une couleur,
- d'**effacer** l'écran ou une zone de l'écran,
- de faire **pivoter** l'affichage
- d'utiliser différentes **polices de caractères**
- d'ajouter de nouvelles polices de caractères

- [Adafruit GFX](#)
- [Brainpad](#)

- **Syntaxe**

Définit la couleur du fond d'écran après l'avoir effacé.

`void fillScreen(uint16_t color);`

rotation = 0(0°)→portrait, 1(90°)→paysage, 2(180°)→portrait ou 3(270°)→paysage dans le sens horaire.

`void setRotation(uint8_t rotation);`

Renvoie la largeur de l'écran en pixel.

`uint16_t width();`

Renvoie la hauteur de l'écran en pixel.

`uint16_t height();`

- **Exemple**

`*.cpp`

```
// Définit la couleur du fond d'écran après l'avoir effacé.
disp.fillScreen(ST7735_BLACK);
// rotation = 0(0°)→portrait, 1(90°)→paysage, 2(180°)→portrait ou
3(270°)→paysage dans le sens horaire.
disp.setRotation(1); // Rotation de 90°, affichage paysage
// Renvoie la largeur de l'écran en pixel.
uint16_t largeur = disp.width();
// Renvoie la hauteur de l'écran en pixel.
uint16_t hauteur = disp.height();
```

A faire

5.12 Polices de caractères

La bibliothèque Adafruit GFX inclut des polices dérivées du projet GNU [FreeFont](#). Elles proposent trois typographies : **serif** (avec empattement) comme la police Times New Roman, **sans serif** comme les polices Helvetica ou Arial et **mono** comme la police Courier. Elles se situent dans le sous-répertoire "Polices".

- **Utilisation des polices GFX dans les croquis Arduino** : inclure le(s) fichier(s) de police après les bibliothèques.

*.cpp

```
// Exemples
#include <Fonts//FreeMonoBoldOblique12pt7b.h>
#include <Fonts//FreeSerif9pt7b.h>

// On définit nomPolice comme police par défaut.
disp.setFont(&FreeMonoBoldOblique12pt7b);
```

From:

<https://webge.fr/dokuwiki/> - **WEBGE Wikis**

Permanent link:

<https://webge.fr/dokuwiki/doku.php?id=materiels:afficheurs:graphique&rev=1640337116>

Last update: **2021/12/24 10:11**

