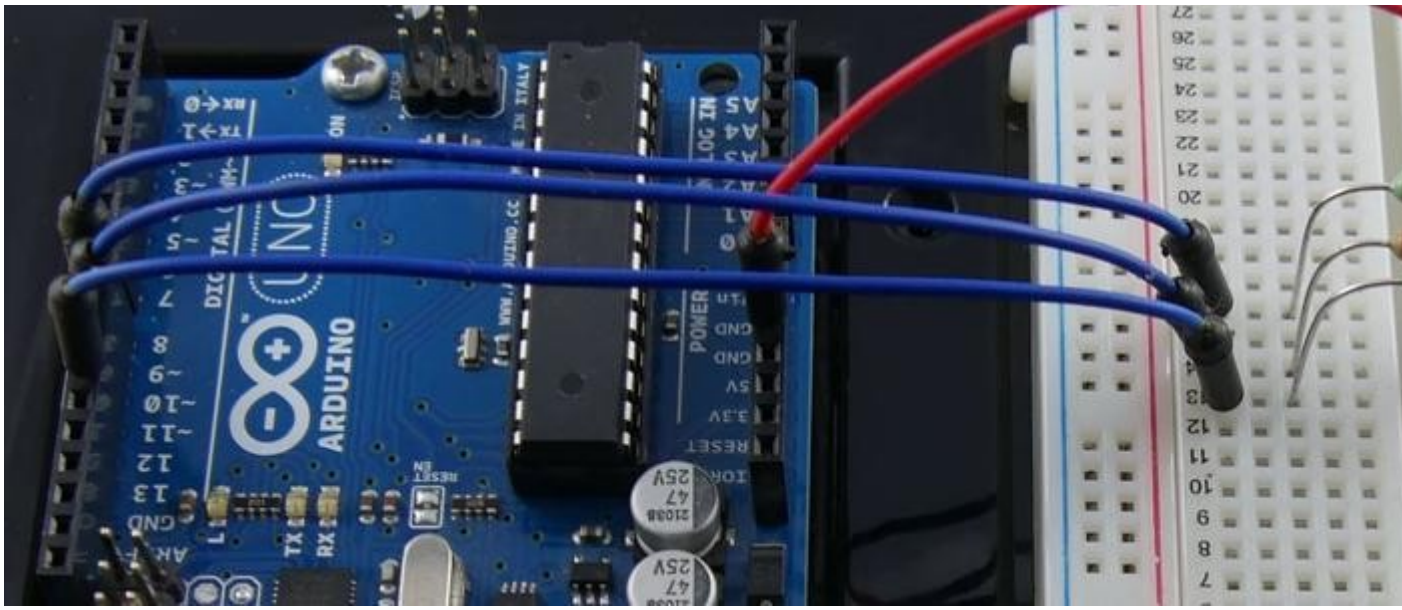
Utiliser des LEDs RGB avec une carte Arduino / Genuino

Les roses sont rouges, les violettes sont bleues, j'aime les LEDs, mais en RGB c'est mieux



par skywodd | avr. 23, 2016 | Licence (voir pied de page)

Catégories : Tutoriels Arduino | Mots clefs : Arduino Genuino LED DEL RGB RVB

Cet article n'a pas été mis à jour depuis un certain temps, son contenu n'est peut être plus d'actualité.

Dans ce tutoriel, nous allons apprendre ensemble à utiliser des LEDs RGB (trois couleurs). Nous verrons comment se présente une LED RGB et comment l'utiliser. En bonus, nous verrons comment corriger la non-linéarité de l'œil humain par rapport à la perception de la luminosité.

Sommaire

Bonjour à toutes et à tous !

Les LEDs sont de merveilleux petits composants électroniques. Il suffit de leur envoyer quelques volts de tension pour qu'elles illuminent (littéralement) votre journée.

On trouve des LEDs classiques monocouleur absolument partout. Mais il existe aussi des LEDs bicolores (on en parlera dans un autre article) et des LEDs à trois couleurs dites "RGB" (ou "RVB" en français, pour "Rouge Vert Bleu").

Ces LEDs RGB permettent de mettre de l'ambiance dans une pièce, ou à plus petite échelle, d'afficher une information en couleur. Dans cet article, nous allons apprendre à utiliser une LED RGB avec une carte Arduino / Genuino.

Les LEDs RGB

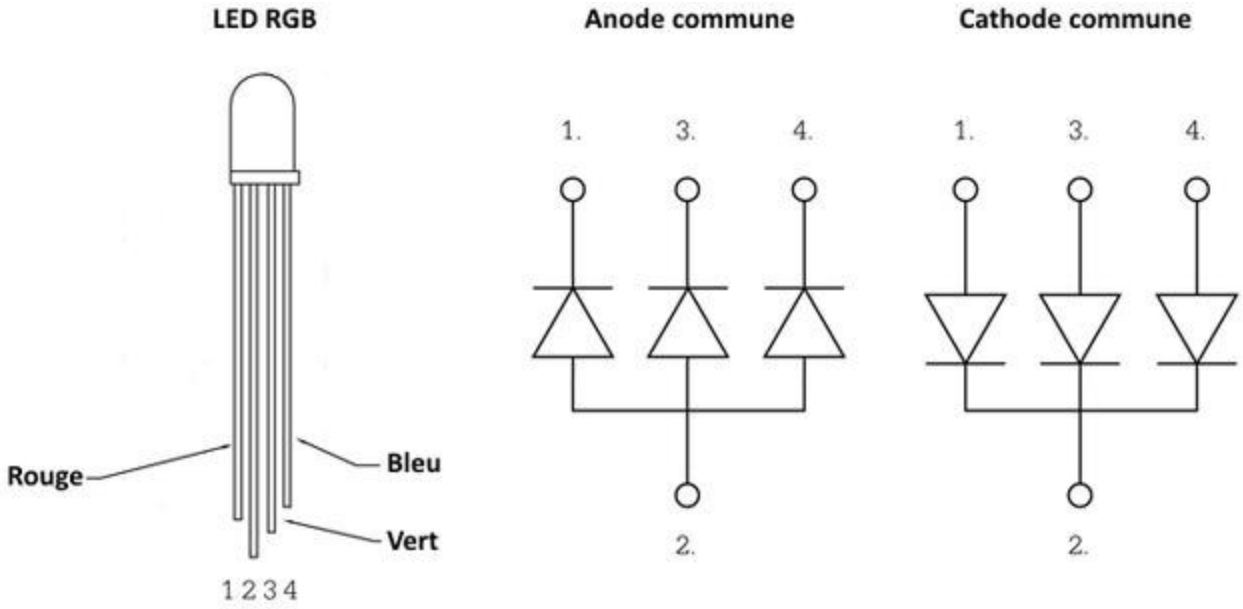
Photographie de plusieurs LEDs RGB

Les LEDs RGB sont en réalité composées de trois LEDs classiques Rouge, Verte et Bleu, emprisonnées ensemble pour l'éternité dans un même boitier plastique.

Le boitier des LEDs RGB existe en deux variantes : clair et diffusant. Le boitier clair (voir photo ci-dessus) permet de voir les trois couleurs séparément et donne un effet

assez sympathique. Le boitier diffusant, au contraire, ne permet pas de voir les différentes couleurs, mais une seule couleur correspondant au mélange des trois couleurs de base. Il est donc important de bien choisir le type de boitier en fonction de son projet

L'avantage d'une LED RGB par rapport à trois LEDs classiques est simple : il n'y a qu'un seul composant à câbler. Au lieu d'avoir trois composants à deux pattes, on a un unique composant à quatre pattes, ça demande moins de soudure et donc moins de temps à câbler.



Le brochage d'une LED RGB

Les LEDs RGB existent en deux versions : à anode commune ou à cathode commune.

Dans la version à anode commune, les trois anodes (le "+") des LEDs sont reliées ensemble. Cela signifie qu'il faut câbler la tension d'alimentation sur la broche commune et contrôler les LEDs via un signal à 0 volt pour les faire s'allumer.

Dans la version à cathode commune, les trois cathodes (le "-") des LEDs sont reliées ensemble. Cela signifie qu'il faut câbler la masse sur la broche commune et contrôler les LEDs via un signal à +5 volts (ou autre) pour les faires s'allumer.

Les versions à cathode commune sont plus simples à utiliser pour des débutants, car plus intuitives. Dans cette configuration, une tension de 5 volts allume la LED et une tension de 0 volt l'éteint. Pour un petit projet avec seulement quelques LEDs RGB, cela peut être intéressant.

Cependant, les versions à anode commune sont bien plus répandues. Elles sont moins simples à utiliser, car dans cette configuration une tension de 5 volts éteint la LED et une tension de 0 volt l'allume. C'est le monde à l'envers.

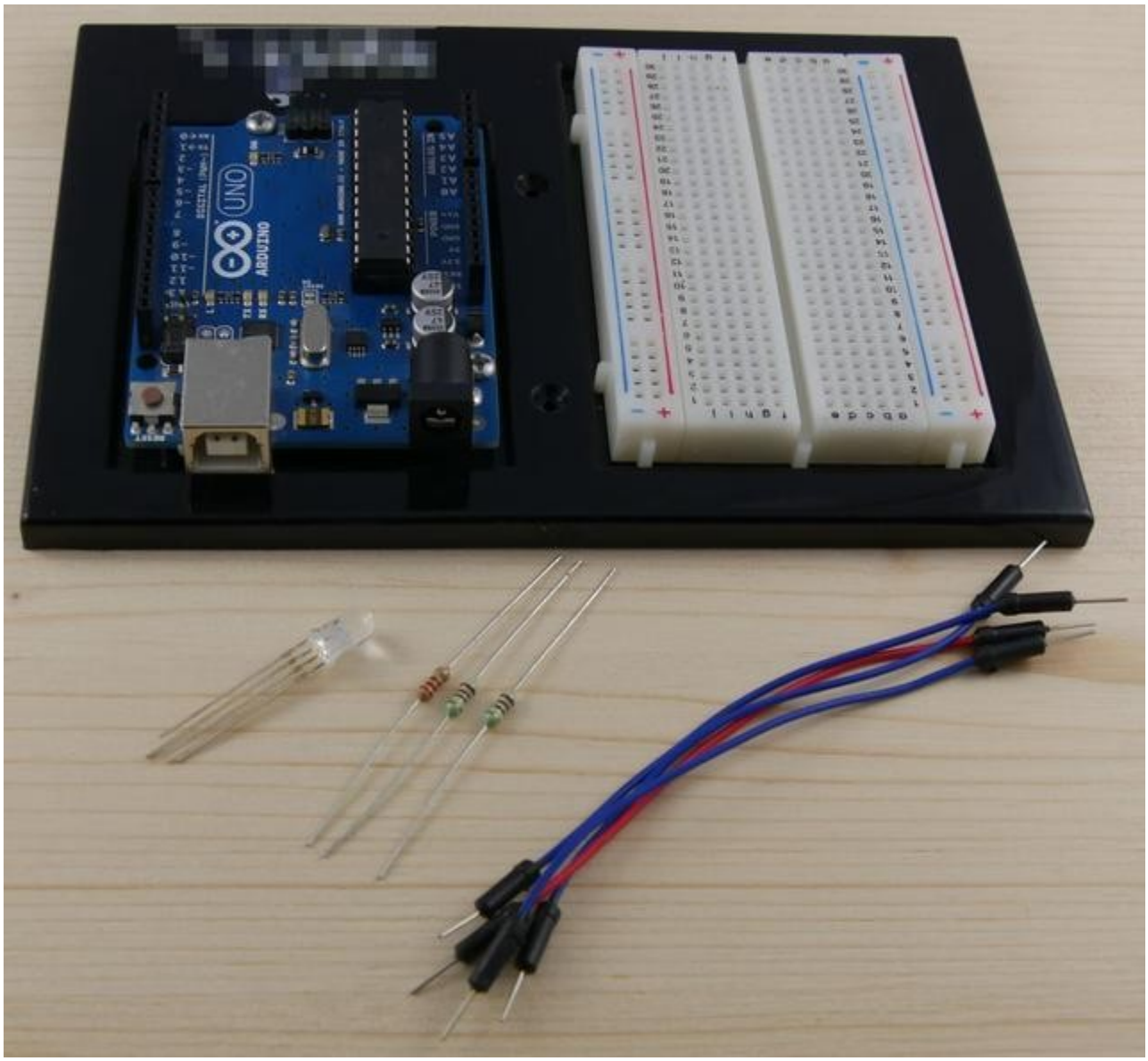
De façon générale, dans une vraie application, il est préférable d'utiliser des LEDs RGB à anode commune plutôt que des LEDs RGB à cathode commune. Elles sont certes moins pratiques à utiliser, car il faut inverser sa logique de pensées, mais ces versions ont l'immense avantage de pouvoir être contrôlées par des circuits intégrés

spécialisés comme le TLC5940 qui ne peuvent qu'absorber du courant et pas en générer, ce qui rend l'utilisation de LEDs RGB à cathode commune impossible dans ce cas.

Utiliser une LED RGB avec une carte Arduino / Genuino

Pour bien comprendre le fonctionnement d'une LED RGB, nous allons faire un montage très simple avec une LED RGB à anode commune et quelques résistances.

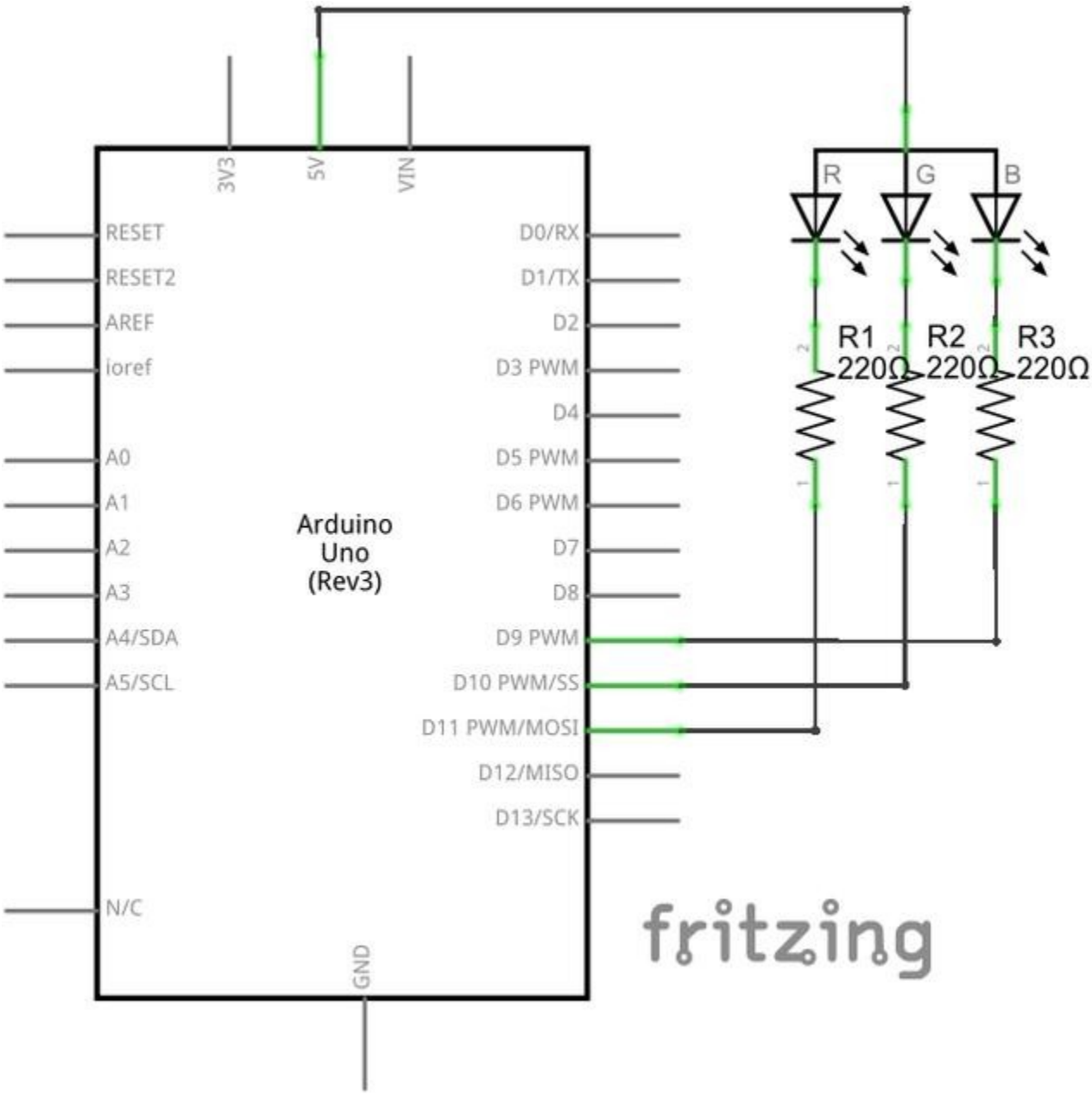
Le montage



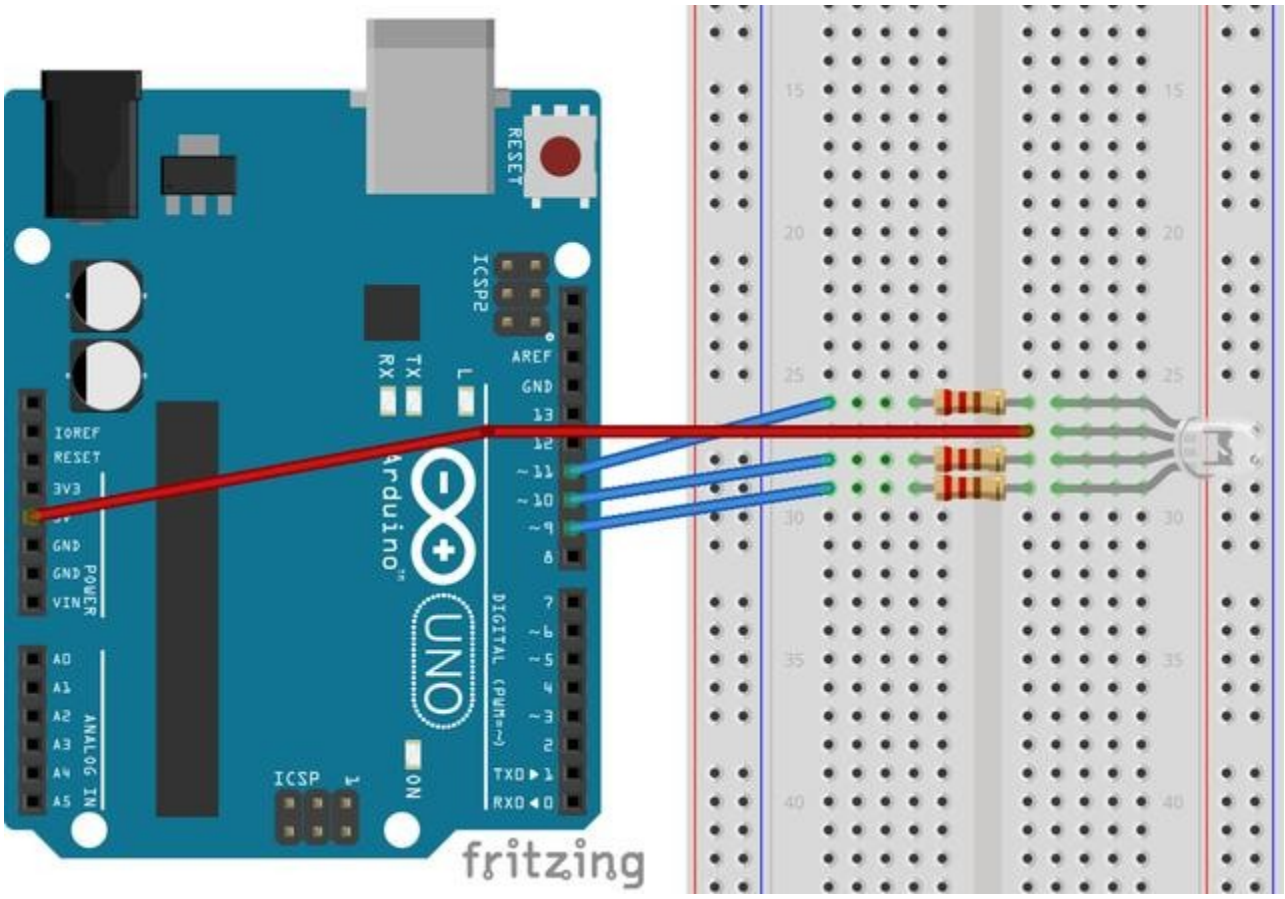
Matériel nécessaire

Pour réaliser ce montage, il va nous falloir :

- Une carte Arduino UNO (et son câble USB),
- Une LED RGB à anode commune (ou cathode commune, voir chapitre suivant dans ce cas),
- Trois résistances de 220 ohms (rouge / rouge / marron),
- Une plaque d'essai et des fils pour câbler notre montage.

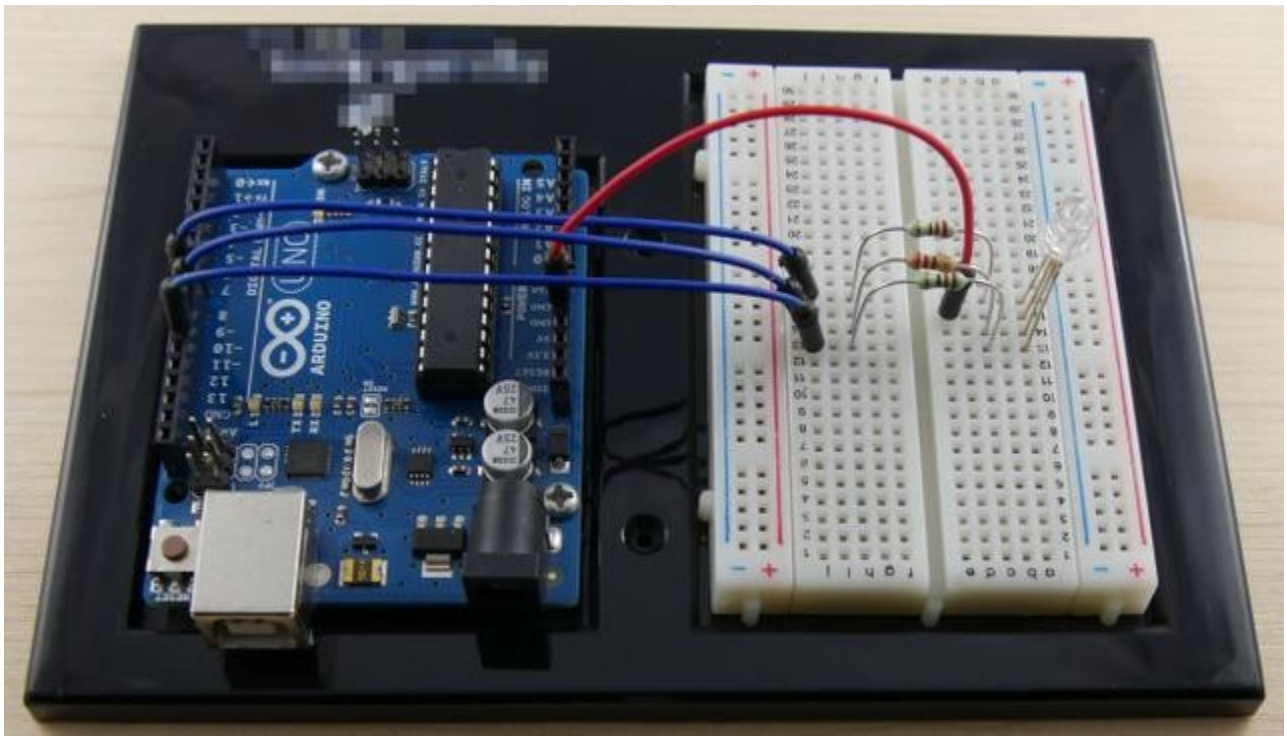


Vue schématique du montage



Vue prototypage du montage

Pour commencer notre montage, nous allons câbler la broche VCC de la carte Arduino à la broche commune de la LED RGB au moyen d'un fil. On relie ensuite chaque broche restante de la LED RGB à une résistance de 220 ohms.



Le montage fini

Pour finir, on câble chaque résistance respectivement aux broches `D9`, `D10` et `D11` de la carte Arduino.

N.B. Les broches utilisables avec la fonction `analogWrite()` (que l'on verra plus tard dans l'article) sont annotées avec un tilde (~) devant le nom de la broche.

Une LED, trois résistances

Pourquoi utiliser trois résistances ? Pourquoi ne pas utiliser une seule résistance sur la broche commune ? Ce serait plus simple !

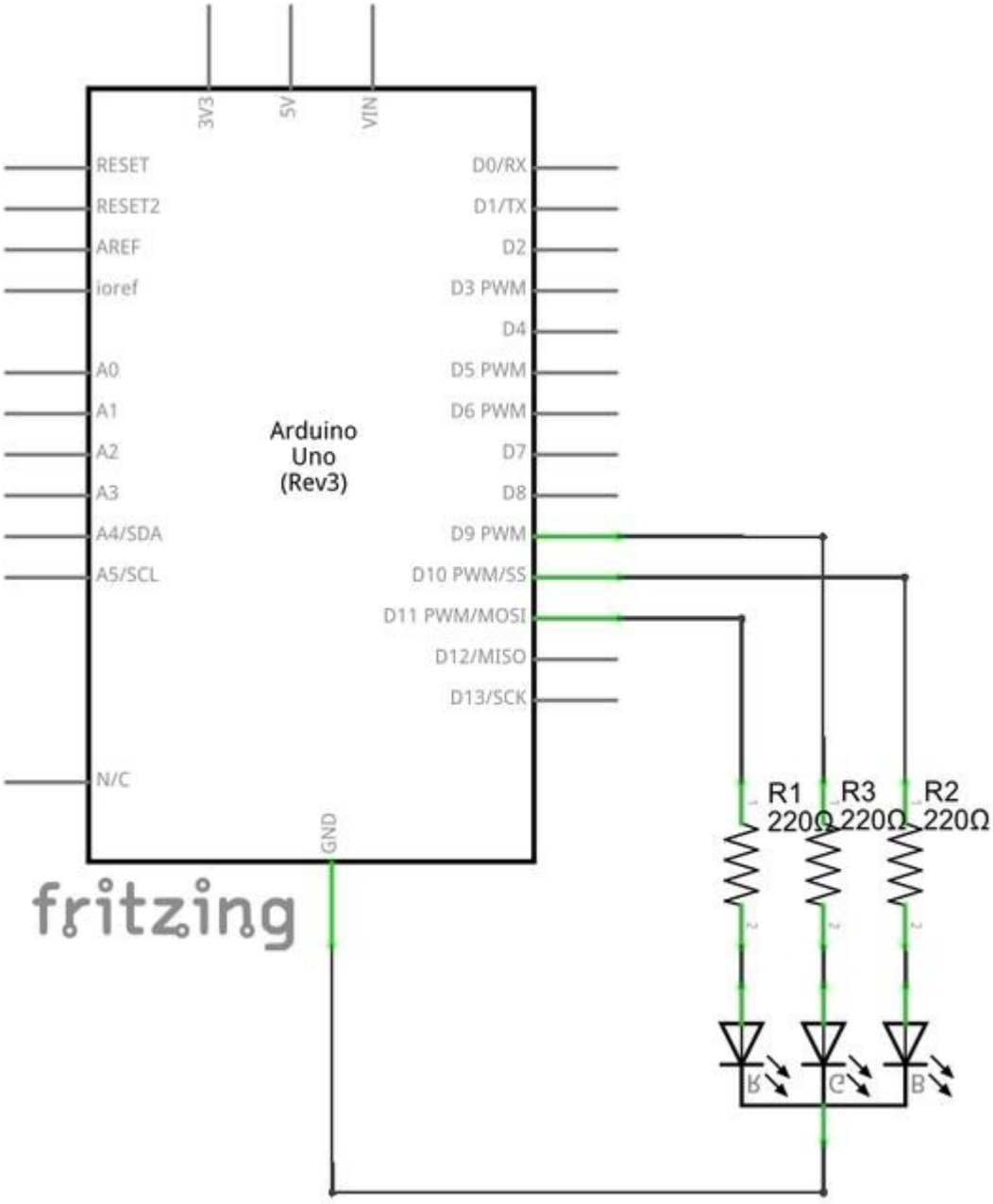
Oui, mais non, fausse bonne idée  On câble toujours une résistance de limitation de courant par LED.

On pourrait effectivement utiliser une seule résistance sur la broche commune, mais cela aurait trois conséquences :

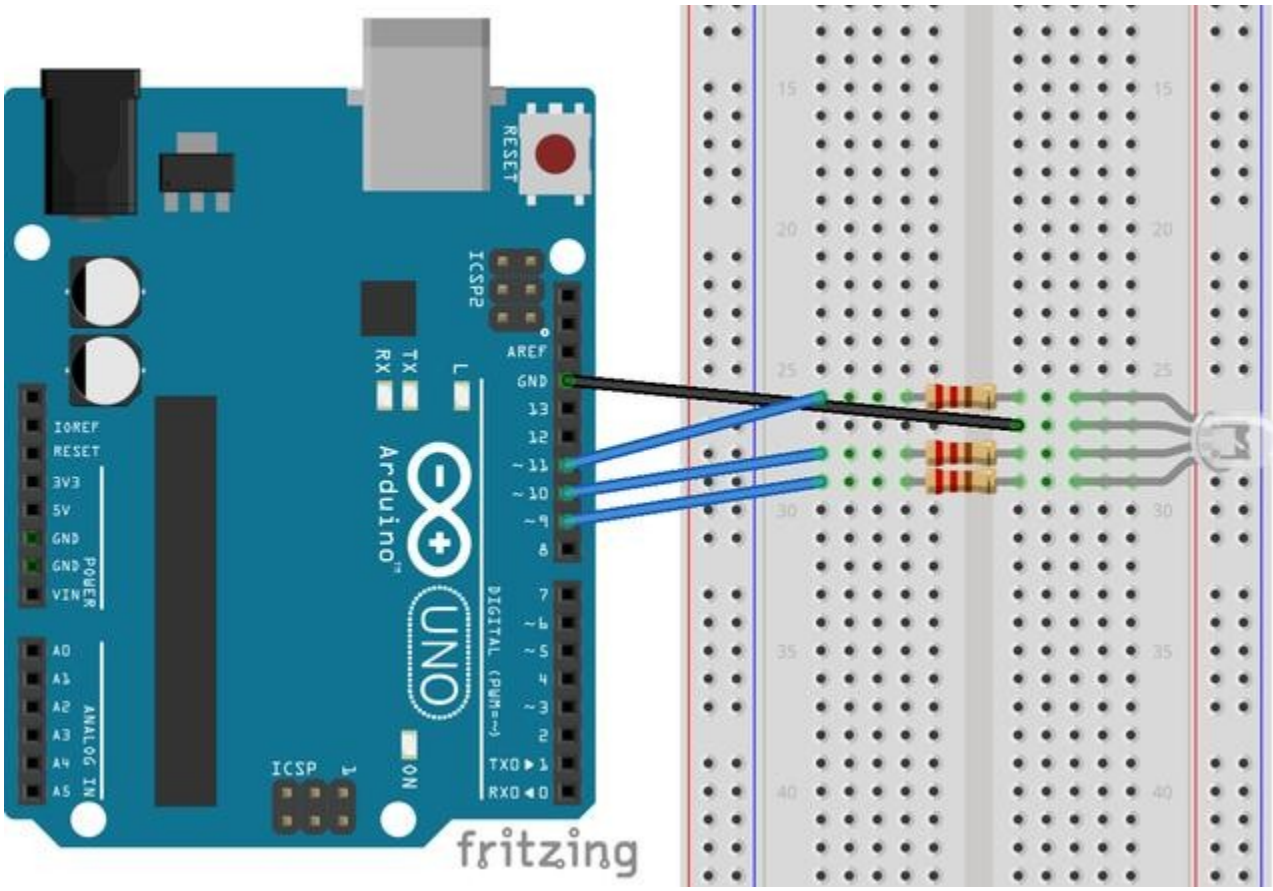
- allumer une seule LED lui ferait subir trop de courant,
- inversement, allumer plusieurs LED ferait diminuer la luminosité des LEDs,
- si une LED grille, toutes les autres suivront en cascade, car le courant restant deviendra plus fort à chaque LED défectueuse.

Une résistance ça ne coûte pas très cher, ne faites donc pas cette erreur de débutant 

Le montage (variante à cathode commune)



Vue schématique du montage (variante cathode commune)

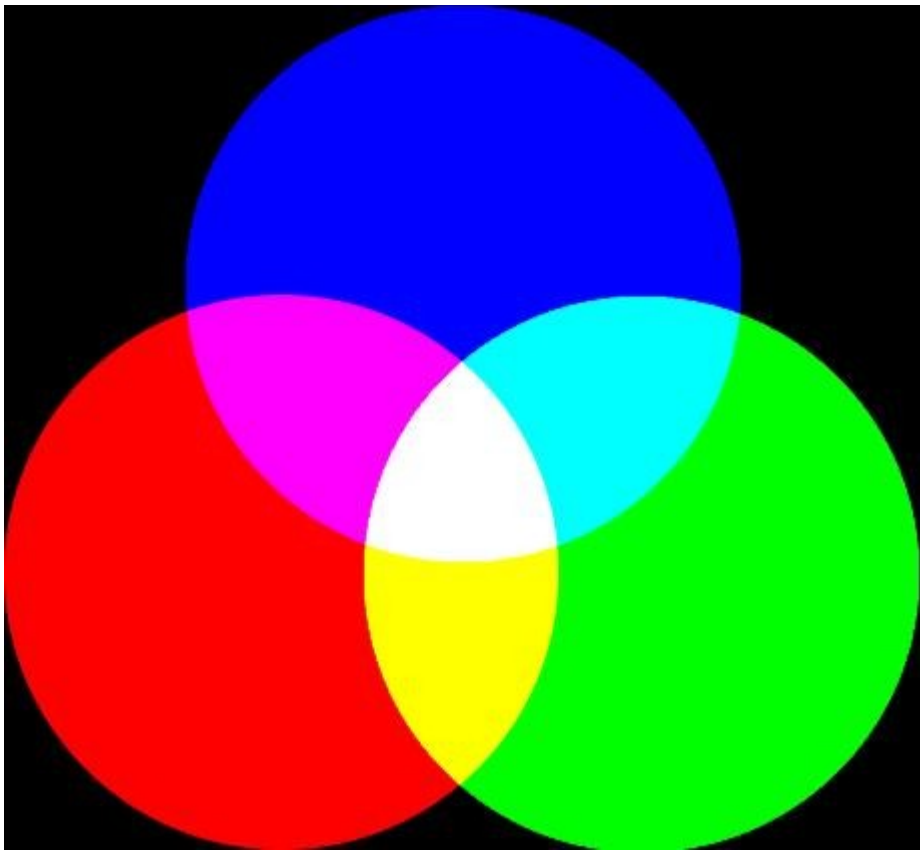


Vue prototypage du montage (variante cathode commune)

Le montage est identique à celui de la version à anode commune. La seule différence est que la broche commune de la LED RGB est reliée à la broche GND de la carte Arduino.

Le code V1

Commençons petit avec un code utilisant la fonction `digitalWrite()` pour contrôler chaque LED.



Couleurs primaires

Avec trois LED et deux états possibles par LED (éteinte et allumée), on obtient un total de 8 couleurs. Ça ne fait pas beaucoup de couleurs, mais c'est déjà un bon début.

```
/* Couleurs (format RGB) */
const byte COLOR_BLACK = 0b000;
const byte COLOR_RED = 0b100;
const byte COLOR_GREEN = 0b010;
const byte COLOR_BLUE = 0b001;
const byte COLOR_MAGENTA = 0b101;
const byte COLOR_CYAN = 0b011;
const byte COLOR_YELLOW = 0b110;
const byte COLOR_WHITE = 0b111;

```

```
/* Broches */
const byte PIN_LED_R = 9;
const byte PIN_LED_G = 10;
const byte PIN_LED_B = 11;
```

On commence le code très classiquement avec les déclarations des différentes constantes du programme.

Pour ce premier programme, j'ai décidé de coder les couleurs sur 3 bits, dans l'ordre R, G, B. Pour rendre cela plus lisible, j'ai utilisé le format obXXX qui est permis par le langage C/C++ pour déclarer des nombres en binaire.

J'ai ensuite déclaré trois constantes pour les trois broches de la LED RGB.

```
void setup() {

  // Initialise les broches
  pinMode(PIN_LED_R, OUTPUT);
  pinMode(PIN_LED_G, OUTPUT);
  pinMode(PIN_LED_B, OUTPUT);
  displayColor(COLOR_BLACK);
}
```

Il n'y a pas grand-chose de très passionnant à faire dans la fonction setup().

Il suffit de mettre les trois broches de la LED RGB en sortie et d'appeler notre fonction displayColor(), que l'on verra juste après, pour éteindre les LED au démarrage.

```
void displayColor(byte color) {

  // Assigne l'état des broches
  // Version cathode commune
  //digitalWrite(PIN_LED_R, bitRead(color, 2));
```

```
6
7   //digitalWrite(PIN_LED_G, bitRead(color, 1));
8   //digitalWrite(PIN_LED_B, bitRead(color, 0));
9
10  // Version anode commune
11  digitalWrite(PIN_LED_R, !bitRead(color, 2));
12  digitalWrite(PIN_LED_G, !bitRead(color, 1));
13  digitalWrite(PIN_LED_B, !bitRead(color, 0));
14 }
15
```

La fonction `displayColor()` va faire tout le boulot. Dans cette première version à 8 couleurs, elle se contente de faire des `digitalWrite()` sur les broches des LEDs rouge verte et bleu en fonction de la couleur passée en paramètre.

Pour "lire" l'état de chaque bit de la couleur, j'utilise la fonction bitRead() qui prend en paramètre un nombre et un numéro de bit (commençant à 0).

N.B. Vous remarquerez qu'il y a deux versions du code, une pour les LEDs RGB à anode commune et une pour les LEDs RGB à cathode commune. La différence réside simplement dans l'inversion de valeur (le point d'exclamation signifie "inverse de la valeur booléenne", exemple : `!0 == 1`). A vous de commenter, décommenter la bonne version du code en fonction de votre montage 😉

```
1 void loop() {
2
3   /* Code de démonstration */
4   displayColor(COLOR_RED);
5   delay(1000);
6
7   displayColor(COLOR_GREEN);
8   delay(1000);
9
10  displayColor(COLOR_BLUE);
11  delay(1000);
12
```

```
displayColor(COLOR_MAGENTA);
  delay(1000);

  displayColor(COLOR_CYAN);
  delay(1000);

  displayColor(COLOR_YELLOW);
  delay(1000);

  displayColor(COLOR_WHITE);
  delay(1000);

  displayColor(COLOR_BLACK);
  delay(1000);
}
```

La fonction `loop()` dans cet exemple se contente d'afficher chaque couleur en boucle avec un délai.

Le code complet avec commentaires :

```
/*
 * Code d'exemple pour une LED RGB (8 couleurs).
 */

/* Couleurs (format RGB) */
const byte COLOR_BLACK = 0b000;
const byte COLOR_RED = 0b100;
const byte COLOR_GREEN = 0b010;
const byte COLOR_BLUE = 0b001;
const byte COLOR_MAGENTA = 0b101;
const byte COLOR_CYAN = 0b011;
const byte COLOR_YELLOW = 0b110;
const byte COLOR_WHITE = 0b111;

/* Broches */
const byte PIN_LED_R = 9;
const byte PIN_LED_G = 10;
const byte PIN_LED_B = 11;

// Fonction setup(), appelée au démarrage de la carte Arduino
void setup() {

  // Initialise les broches
  pinMode(PIN_LED_R, OUTPUT);
  pinMode(PIN_LED_G, OUTPUT);
  pinMode(PIN_LED_B, OUTPUT);
  displayColor(COLOR_BLACK);
}

// Fonction loop(), appelée continuellement en boucle tant que la carte
```

```
Arduino est alimentée

void loop() {

  /* Code de démonstration */
  displayColor(COLOR_RED);
  delay(1000);

  displayColor(COLOR_GREEN);
  delay(1000);

  displayColor(COLOR_BLUE);
  delay(1000);

  displayColor(COLOR_MAGENTA);
  delay(1000);

  displayColor(COLOR_CYAN);
  delay(1000);

  displayColor(COLOR_YELLOW);
  delay(1000);

  displayColor(COLOR_WHITE);
  delay(1000);

  displayColor(COLOR_BLACK);
  delay(1000);
}

/** Affiche une couleur */
```

```
void displayColor(byte color) {


  // Assigne l'état des broches
  // Version cathode commune
  //digitalWrite(PIN_LED_R, bitRead(color, 2));
  //digitalWrite(PIN_LED_G, bitRead(color, 1));
  //digitalWrite(PIN_LED_B, bitRead(color, 0));
  // Version anode commune
  digitalWrite(PIN_LED_R, !bitRead(color, 2));
  digitalWrite(PIN_LED_G, !bitRead(color, 1));
  digitalWrite(PIN_LED_B, !bitRead(color, 0));
}
```

```
0

6
1

6
2

6
3

6
4

6
5

6
6

6
7

6
8

6
9

7
0

7
1
```

L'extrait de code ci-dessus est disponible en téléchargement sur cette page (le lien de téléchargement en .zip contient le projet Arduino prêt à l'emploi).

Le code V2

Huit couleurs c'est bien sympathique, mais c'est un peu léger. 16.7 millions de couleurs, ce serait déjà beaucoup mieux !

Pour atteindre ce nombre hallucinant de nuances de couleurs, nous allons faire un second code, utilisant la fonction analogWrite() cette fois-ci.

Cube RGB "True color" (RGB88)

La fonction `analogWrite()` fonctionne différemment de la fonction `digitalWrite()`, celle-ci permet d'avoir 254 niveaux intermédiaires entre `LOW` et `HIGH` (256 niveaux au total). Cette fonction fonctionne grâce au principe de PWM, mais on en parlera dans un article dédié 😉

La fonction `analogWrite()` n'est disponible que sur les broches compatibles PWM, annotées avec un tilde (~) devant leurs noms sur la carte Arduino.

```
1/* Broches */
2const byte PIN_LED_R = 9;
3const byte PIN_LED_G =
```

```
10;
const byte PIN_LED_B =
11;
```

Comme pour le code précédent, on commence par les déclarations de constantes. Dans cette version, il suffit de déclarer les trois broches de la LED RGB.

```
void setup() {

  // Initialise les broches
  pinMode(PIN_LED_R,
OUTPUT);
  pinMode(PIN_LED_G,
OUTPUT);
  pinMode(PIN_LED_B,
OUTPUT);
  displayColor(0, 0, 0);
}
```

La fonction `setup()` est quasiment identique à la version précédente, seul l'appel à la fonction `displayColor()` est différent.

```
void displayColor(byte r, byte g, byte b) {

  // Assigne l'état des broches
  // Version cathode commune
  //analogWrite(PIN_LED_R, r);
  //analogWrite(PIN_LED_G, g);
  //analogWrite(PIN_LED_B, b);

  // Version anode commune
  analogWrite(PIN_LED_R, ~r);
  analogWrite(PIN_LED_G, ~g);
  analogWrite(PIN_LED_B, ~b);
```

```
}
```

Premier gros changement : les couleurs sont données en paramètres séparément. Chaque composante de la couleur est représentée par un nombre sur 8 bits (un octet), cela fait au total 24 bits, soit 16 777 216 couleurs possibles.

Les appels à `digitalWrite()` sont remplacés par des appels à `analogWrite()`. La valeur de chaque composante de la couleur voulue est donnée directement en paramètre de `analogWrite()`.

Dans le cas des LEDs RGB à anode commune, il faut inverser la valeur. On pourrait faire `255 - r` par exemple, mais il existe une syntaxe plus courte pour cela : `~r`. L'opérateur tilde permet d'inverser l'état de chaque bit d'un nombre, par exemple : `~0b101 == 0b010`.

```
void loop() {

  /* Code de démonstration */
  displayColor(255, 0, 0);
  delay(1000);

  displayColor(0, 255, 0);
  delay(1000);

  displayColor(0, 0, 255);
  delay(1000);

  displayColor(255, 0, 255);
  delay(1000);

  displayColor(0, 255, 255);
  delay(1000);

  displayColor(255, 255, 0);
```

```
    delay(1000);

    displayColor(255, 255, 255);
    delay(1000);

    displayColor(0, 0, 0);
    delay(1000);
}
```

La fonction loop() se contente encore une fois d'afficher chaque couleur en boucle avec un délai.

PS Je vous laisse réfléchir comment mettre en place un effet de fondu ou de dégradé de couleur 

Le code complet avec commentaires :

```
/*
 * Code d'exemple pour une LED RGB (+16 millions de couleurs).
 */

/* Broches */
```

```
const byte PIN_LED_R = 9;
const byte PIN_LED_G = 10;
const byte PIN_LED_B = 11;

// Fonction setup(), appelée au démarrage de la carte Arduino
void setup() {

  // Initialise les broches
  pinMode(PIN_LED_R, OUTPUT);
  pinMode(PIN_LED_G, OUTPUT);
  pinMode(PIN_LED_B, OUTPUT);
  displayColor(0, 0, 0);
}

// Fonction loop(), appelée continuellement en boucle tant que la carte Arduino est alimentée
void loop() {

  /* Code de démonstration */
  displayColor(255, 0, 0);
  delay(1000);

  displayColor(0, 255, 0);
  delay(1000);

  displayColor(0, 0, 255);
  delay(1000);

  displayColor(255, 0, 255);
  delay(1000);
```

```
  displayColor(0, 255, 255);
  delay(1000);

  displayColor(255, 255, 0);
  delay(1000);

  displayColor(255, 255, 255);
  delay(1000);

  displayColor(0, 0, 0);
  delay(1000);
}

/** Affiche une couleur */
void displayColor(byte r, byte g, byte b) {

  // Assigne l'état des broches
  // Version cathode commune
  //analogWrite(PIN_LED_R, r);
  //analogWrite(PIN_LED_G, g);
  //analogWrite(PIN_LED_B, b);
  // Version anode commune
  analogWrite(PIN_LED_R, ~r);
  analogWrite(PIN_LED_G, ~g);
  analogWrite(PIN_LED_B, ~b);
}
```

```
4
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
```

L'extrait de code ci-dessus est disponible en téléchargement sur cette page (le lien de téléchargement en .zip contient le projet Arduino prêt à l'emploi).

Bonus : Correction Gamma

Si vous jouez un peu avec le code du chapitre précédent, vous remarquerez surement que la luminosité des LEDs est bizarre.

Au début la luminosité semble s'accroire rapidement, puis d'un coup, la luminosité ne change plus. Cela est dû à la façon dont l'oeil humain perçoit la lumière.

L'oeil humain est très fort quand il s'agit de détecter de toutes petites nuances de couleur ou de luminosité, mais seulement quand la luminosité est faible. Quand la luminosité dépasse un certain seuil, on ne remarque plus aucune nuance.

Cela est très embêtant, car les écrans d'ordinateur, les afficheurs, les LEDs, etc, sont capables d'afficher n'importe quelle nuance de couleur et/ou de luminosité de la même façon, qu'importe la luminosité demandée. Pour un écran d'ordinateur, il est aussi simple d'afficher une transition de couleurs RGB(0, 0, 0) -> RGB(1, 1, 1) ou RGB(254, 254, 254) -> RGB(255, 255, 255) par exemple.

Image de test avec et sans correction Gamma

Il a donc fallu trouver une solution à ce problème. Cette solution s'appelle la "correction gamma". Le principe est simple : compenser la non-linéarité de la

perception de la luminosité par l'oeil humain en application une correction à l'affichage.

L'image ci-dessus (qui au passage est une photo de test standard nommée "Lenna", du nom de l'actrice du magazine Playboy qui y est représentée – ce n'est pas une blague, on utilise une photo d'un magazine de fesse comme image de test standard depuis 1973) permet de bien voir l'effet de la correction gamma. Sans correction gamma, les couleurs paraissent délavées, fades.

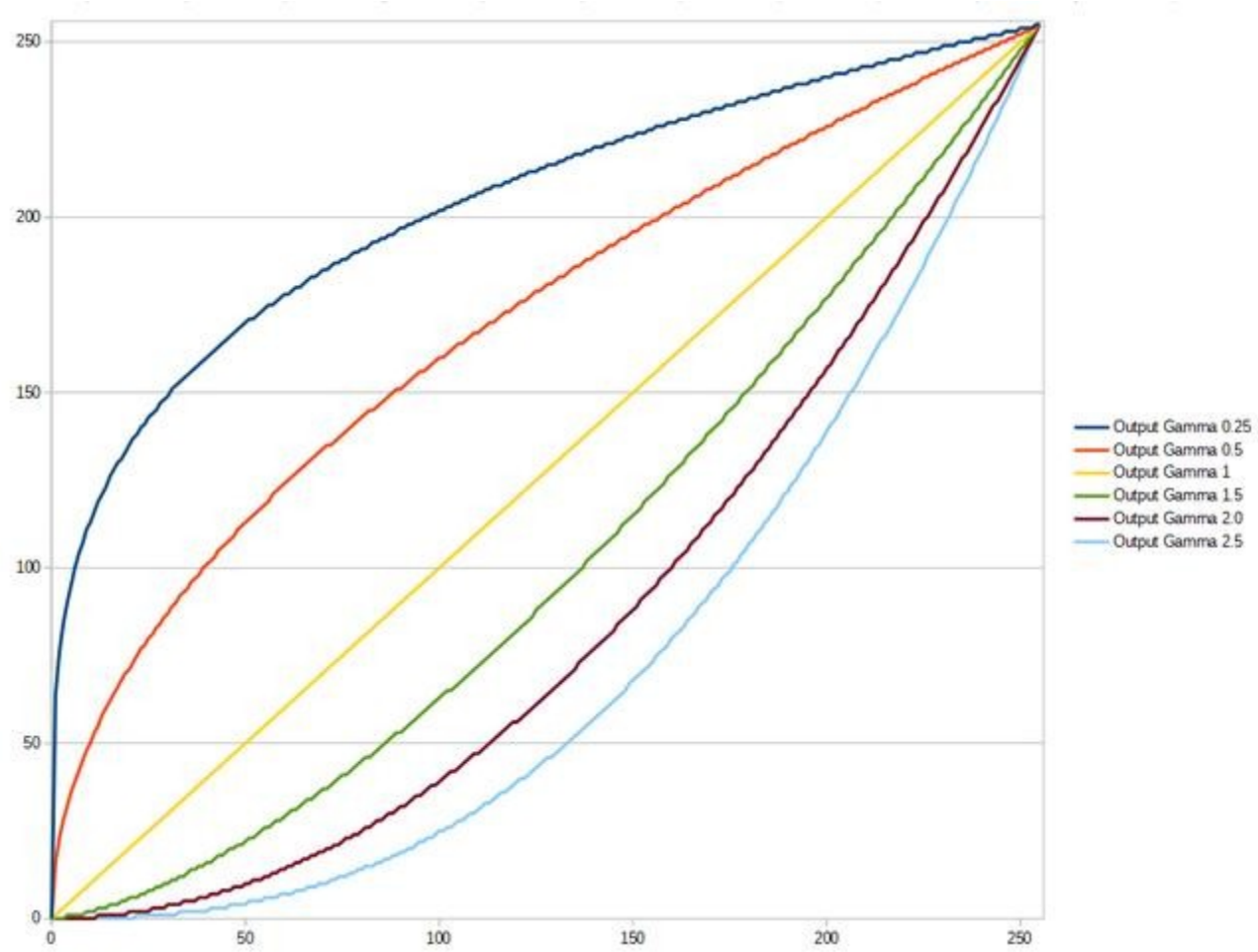


Illustration de différentes courbes de correction Gamma

Pour les amateurs de formules mathématiques, la fonction de courbe Gamma est : `sortie = valeur_sortie_max * entrée / valeur_entrée_max ^ 1 / coeff`. Le coefficient standard de correction est de 2.2, mais il existe d'autres standards comme 2.5. La courbe ci-dessus présente différentes valeurs de correction. Les valeurs supérieures à zéro corrigent le Gamma (plus sombre), les valeurs inférieures à zéro (dé)corrigent le Gamma (plus clair).

Afin de simplifier la correction Gamma en électronique, on utilise généralement une table de correction précalculée. Il suffit de donner une valeur en entrée de la table pour obtenir sa version corrigée en retour.

J'ai conçu un script Python 3 qui permet de générer un fichier C/C++ prêt à l'usage en fonction du coefficient de correction désiré :

```python
"""
Look-up table Gamma correction generator for Python 3.
"""

from math import pow


def gamma_lut(input_array_size, ouput_array_size=None,
              correction_factor=2.2, reverse_gamma=False):
    """
    Generate a Gamma correction LookUp Table from the given parameters.
    :param input_array_size: The input array size.
    :param ouput_array_size: The output array size (default to the input array size).
    :param correction_factor: The correction factor (between 0 and 3, included, default 2.2).
    :param reverse_gamma: Set to `True` for reverse Gamma correction (default `False`).
    :return: A generator expression.
    """
    assert input_array_size > 0, "Input array size must be positive"
    assert ouput_array_size is None or ouput_array_size > 0, "Output array must be positive"
    assert 0 < correction_factor <= 3, "Correction factor must be between 0 and 3 (included)"

    # Fix output array size
    ouput_array_size = ouput_array_size or input_array_size

    # fix correction factor
    if reverse_gamma:
        correction_factor = 1.0 / correction_factor
```

```
    # Gamma correction formula
    for index in range(input_array_size):
        yield round(pow(index / float(input_array_size - 1), correction_factor) * (ouput_array_size - 1))


def split_chunks(array, chunk_size):
    """ Split the given array into chunks of at most the given size. """
    return [array[x:x + chunk_size] for x in range(0, len(array), chunk_size)]


# Output array size - MUST be a power of two
INPUT_ARRAY_SIZE = 256
OUTPUT_ARRAY_SIZE = 256

# Gamma correction coefficient - standard: 2.2, Maxim standard: 2.5
GAMMA_CORRECTION_COEFF = 2.2

# Number of values per line
NUMBER_COUNT_PER_LINE = 16


# Output header
print("""/* ----- BEGIN OF AUTO-GENERATED CODE - DO NOT EDIT ----- */
#ifndef GAMMA_H_
#define GAMMA_H_

/* Dependency for PROGMEM */
#include <avr/pgmspace.h>


```

```
/** Gamma correction table in flash memory. */
static const uint8_t PROGMEM _gamma[] = {"""

# Print LUT data
gamma_table = gamma_lut(INPUT_ARRAY_SIZE, OUTPUT_ARRAY_SIZE, GAMMA_CORRECTION_COEFF)
gamma_table = [format(x, "3d") for x in gamma_table]
gamma_table_lines = split_chunks(gamma_table, NUMBER_COUNT_PER_LINE)
gamma_table_lines = ['  ' + ', '.join(line) for line in gamma_table_lines]
print(',\n'.join(gamma_table_lines))


# Output footer
print("""};


/**
 * Apply gamma correction to the given sample.
 *
 * @param x The input 8 bits sample value.
 * @return The output 8 bits sample value with gamma correction.
 */
static inline uint8_t gamma(uint8_t x) {
  return pgm_read_byte(&_gamma[x]);
}


#endif /* GAMMA_H_ */
/* ----- END OF AUTO-GENERATED CODE ----- */
""")
```

```
5
7
6
7
7
7
8
7
9
8
0
8
1
8
2
```

L'extrait de code ci-dessus est disponible en téléchargement sur cette page.

Voici un exemple de table de correction 8 bits (entrée) vers 8 bits (sortie) avec une coefficient standard de 2.2 :

```
1/* ----- BEGIN OF AUTO-GENERATED CODE - DO NOT EDIT ----- */
2#ifndef GAMMA_H_
3#define GAMMA_H_
4
5/* Dependency for PROGMEM */
6#include <avr/pgmspace.h>
7
8/** Gamma correction table in flash memory. */
9static const uint8_t PROGMEM _gamma[] = {
1    0,  21,  28,  34,  39,  43,  46,  50,  53,  56,  59,  61,  64,  66,  68,  70,
0    72,  74,  76,  78,  80,  82,  84,  85,  87,  89,  90,  92,  93,  95,  96,  98,
1    99, 101, 102, 103, 105, 106, 107, 109, 110, 111, 112, 114, 115, 116, 117,
1  118,
1  119, 120, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133,
2  134, 135,
1
```

```
  136, 137, 138, 139, 140, 141, 142, 143, 144, 144, 145, 146, 147, 148, 149, 150,
  151, 151, 152, 153, 154, 155, 156, 156, 157, 158, 159, 160, 160, 161, 162, 163,
  164, 164, 165, 166, 167, 167, 168, 169, 170, 170, 171, 172, 173, 173, 174, 175,
  175, 176, 177, 178, 178, 179, 180, 180, 181, 182, 182, 183, 184, 184, 185, 186,
  186, 187, 188, 188, 189, 190, 190, 191, 192, 192, 193, 194, 194, 195, 195, 196,
  197, 197, 198, 199, 199, 200, 200, 201, 202, 202, 203, 203, 204, 205, 205, 206,
  206, 207, 207, 208, 209, 209, 210, 210, 211, 212, 212, 213, 213, 214, 214, 215,
  215, 216, 217, 217, 218, 218, 219, 219, 220, 220, 221, 221, 222, 223, 223, 224,
  224, 225, 225, 226, 226, 227, 227, 228, 228, 229, 229, 230, 230, 231, 231, 232,
  232, 233, 233, 234, 234, 235, 235, 236, 236, 237, 237, 238, 238, 239, 239, 240,
  240, 241, 241, 242, 242, 243, 243, 244, 244, 245, 245, 246, 246, 247, 247, 248,
  248, 249, 249, 249, 250, 250, 251, 251, 252, 252, 253, 253, 254, 254, 255, 255
};


/**
 * Apply gamma correction to the given sample.
 *
 * @param x The input 8 bits sample value.
 * @return The output 8 bits sample value with gamma correction.
 */
static inline uint8_t gamma(uint8_t x) {
  return pgm_read_byte(&_gamma[x]);
}
```

```
#endif /* GAMMA_H_ */

/* ----- END OF AUTO-GENERATED CODE ----- */
```

L'extrait de code ci-dessus est disponible en téléchargement sur cette page.

Pour démontrer l'utilité de la correction Gamma, voici un code de démonstration qui allume progressivement la LED rouge avec puis sans correction Gamma :

```
/*
 * Code d'exemple pour une LED RGB (+16 millions couleurs) avec correction Gamma.
 */

/* Broches */
const byte PIN_LED_R = 9;
const byte PIN_LED_G = 10;
const byte PIN_LED_B = 11;

/* ----- BEGIN OF AUTO-GENERATED CODE - DO NOT EDIT ----- */
#ifndef GAMMA_H_
#define GAMMA_H_

```

```
/* Dependency for PROGMEM */
#include <avr/pgmspace.h>

/** Gamma correction table in flash memory. */
static const uint8_t PROGMEM _gamma[] = {
  0,  21,  28,  34,  39,  43,  46,  50,  53,  56,  59,  61,  64,  66,  68,  70,
  72,  74,  76,  78,  80,  82,  84,  85,  87,  89,  90,  92,  93,  95,  96,  98,
  99, 101, 102, 103, 105, 106, 107, 109, 110, 111, 112, 114, 115, 116, 117, 118,
 119, 120, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135,
 136, 137, 138, 139, 140, 141, 142, 143, 144, 144, 145, 146, 147, 148, 149, 150,
 151, 151, 152, 153, 154, 155, 156, 156, 157, 158, 159, 160, 160, 161, 162, 163,
 164, 164, 165, 166, 167, 167, 168, 169, 170, 170, 171, 172, 173, 173, 174, 175,
 175, 176, 177, 178, 178, 179, 180, 180, 181, 182, 182, 183, 184, 184, 185, 186,
 186, 187, 188, 188, 189, 190, 190, 191, 192, 192, 193, 194, 194, 195, 195, 196,
 197, 197, 198, 199, 199, 200, 200, 201, 202, 202, 203, 203, 204, 205, 205, 206,
 206, 207, 207, 208, 209, 209, 210, 210, 211, 212, 212, 213, 213, 214, 214, 215,
 215, 216, 217, 217, 218, 218, 219, 219, 220, 220, 221, 221, 222, 223, 223, 224,
 224, 225, 225, 226, 226, 227, 227, 228, 228, 229, 229, 230, 230, 231, 231, 232,
 232, 233, 233, 234, 234, 235, 235, 236, 236, 237, 237, 238, 238, 239, 239, 240,
 240, 241, 241, 242, 242, 243, 243, 244, 244, 245, 245, 246, 246, 247, 247, 248,
 248, 249, 249, 249, 250, 250, 251, 251, 252, 252, 253, 253, 254, 254, 255, 255
};
```

```
};


/**
 * Apply gamma correction to the given sample.
 *
 * @param x The input 8 bits sample value.
 * @return The output 8 bits sample value with gamma correction.
 */
static inline uint8_t gamma(uint8_t x) {
  return pgm_read_byte(&_gamma[x]);
}


#endif /* GAMMA_H_ */
/* ----- END OF AUTO-GENERATED CODE ----- */


// Fonction setup(), appelée au démarrage de la carte Arduino
void setup() {


  // Initialise les broches
  pinMode(PIN_LED_R, OUTPUT);
  pinMode(PIN_LED_G, OUTPUT);
  pinMode(PIN_LED_B, OUTPUT);
  displayColor(0, 0, 0);
}


// Fonction loop(), appelée continuellement en boucle tant que la carte Arduino est alimentée
void loop() {


  /* Code de démonstration */

```

```
  for (int i = 0; i < 256; i++) {
    displayColor(i & 255, 0, 0);
    delay(10);
  }
  delay(1000);

  for (int i = 0; i < 256; i++) {
    displayColor(gamma(i & 255), 0, 0);
    delay(10);
  }
  delay(1000);
}


/** Affiche une couleur */
void displayColor(byte r, byte g, byte b) {

  // Assigne l'état des broches
  // Version cathode commune
  //analogWrite(PIN_LED_R, r);
  //analogWrite(PIN_LED_G, g);
  //analogWrite(PIN_LED_B, b);
  // Version anode commune
  analogWrite(PIN_LED_R, ~r);
  analogWrite(PIN_LED_G, ~g);
  analogWrite(PIN_LED_B, ~b);
}
```

```
7

8
8

8
9
```

L'extrait de code ci-dessus est disponible en téléchargement sur cette page (le lien de téléchargement en .zip contient le projet Arduino prêt à l'emploi).

Conclusion

Ce tutoriel est désormais terminé.

Si ce tutoriel vous a plu, n'hésitez pas à le commenter sur le forum, à le diffuser sur les réseaux sociaux et à soutenir le site si cela vous fait plaisir.