

Lámpara led de fantasía animada con tira led WS2812B.

Se trata de crear una lámpara temática fantasía con leds que se animará a partir de sonidos ambientes o animación pre-programada con tira led WS2812B y leds RGB.

Tendrá un módulo KY-037 para poder interaccionar con el ruido ambiente.

Usará un zumbador para saber en qué modo está de forma sonora y cuando arranca o se apaga.

Usará un pulsador para cambiar los modos de forma manual.

Usaré un diseño mío ya hecho para otro proyecto, añadirle la tira de leds con arduino y diferentes sensores.

Se imprimirá todo en filamento PLA con impresora 3D y resina para impresora 3D.

Hardware empleado:

Arduino nano x1

Socket arduino nano x1

Tira Led WS2812B x1 (1,5 metros)

Led RGB X3

Fuente alimentación 5v x1

KY-037(Sensor de sonido) x1

Pulsador x1

Fuente Alimentación 5v 12A

(https://www.amazon.es/Adaptador-controlador-alimentaci%C3%B3n-Transformador-60W%EF%BC%89AC100-240V/dp/B08K4M7JSB/ref=sr_1_3?__mk_es_ES=%C3%85M%C3%85%C5%BD%C3%95%C3%91&dchild=1&ddw=wnm3CSkc8qlEE8F27exb_w%2C%2C&keywords=transformador%2B220%2Ba%2B5v%2B12a&qid=1635246560&qid=257-6758445-9957828&refinements=p_90%3A6820340031&nid=6820335031&s=electronics&sr=1-3&sres=B01G0Q3RWU%2CB07Q2WQ8DS%2CB08K4M7JSB%2CB01G0Q3RZ2%2CB00MWQD080%2CB08K4PQL3M%2CB07DNKMRP3%2CB07DCWJ4WQ%2CB07YXKL2R3%2CB00PTLSH9G%2CB08HRJVBLP%2CB07K4TB67%2CB07C53B1GZ%2CB07TS73K6F%2CB07MJO8K4P%2CB01N6MZ30A&th=1)

Material utilizado y herramientas:

Filamento PLA x1 (1Kg)

Cable 5v rojo x1 (1 metro)

Cable 5v negro x1 (1 metro)

Cable 5v verde x1 (1 metro)

Cable 5v azul x1 (1 metro)

Pintura Spray Montana Negro x1

Pintura Spray Montana Blanco x1

Pintura Spray Montana Dorado x1

Pintura Spray Montana Plata x1

Pinta uñas gel Translúcido x3

Polvo metalizado pinta uñas gel x4

Pegamento cianocrilato x2

Regleta empalmes x1

Barra Silicona x1

Regletas de conexión x1

Herramientas:

Impresora 3D Sapphire Plus
Impresora 3D Ender 3
Impresora 3D BQ Witbox one
Impresora 3D Resina Anycubic Photon
Pistola silicona caliente
Soldador
Estaño
Alicates de corte
Bisturí¹
Cutter
Destornillador
Calibre
Méetro
Multímetro
Taladro
Aerógrafo

Sofware utilizado y librerias:

Proteus
Fusion 360 (Modelado 3D)
Prusa Slicer (Laminador impresión 3D FDM)
Chitobox (Laminador impresión 3D en resina)
Blender
Soldador

The FastLED library (<https://github.com/FastLED/FastLED>)
Library NeoPixel of Adafruit (https://github.com/adafruit/Adafruit_NeoPixel)

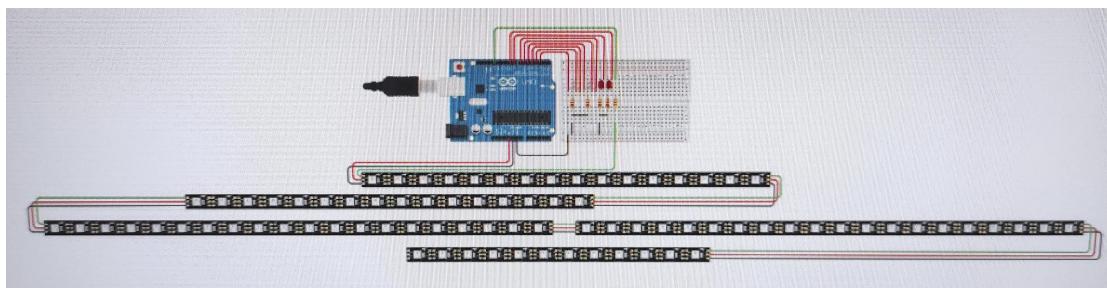
Fuentes de recursos:

<https://www.luisllamas.es/arduino-led-rgb-ws2812b/> (CONECTAR ARDUINO CON PANELES Y TIRAS LED RGB WS2812B (NEOPIXEL))

<https://www.youtube.com/watch?v=G-zCNkNp4RY> (Referencia para la animación led con el KY-037)

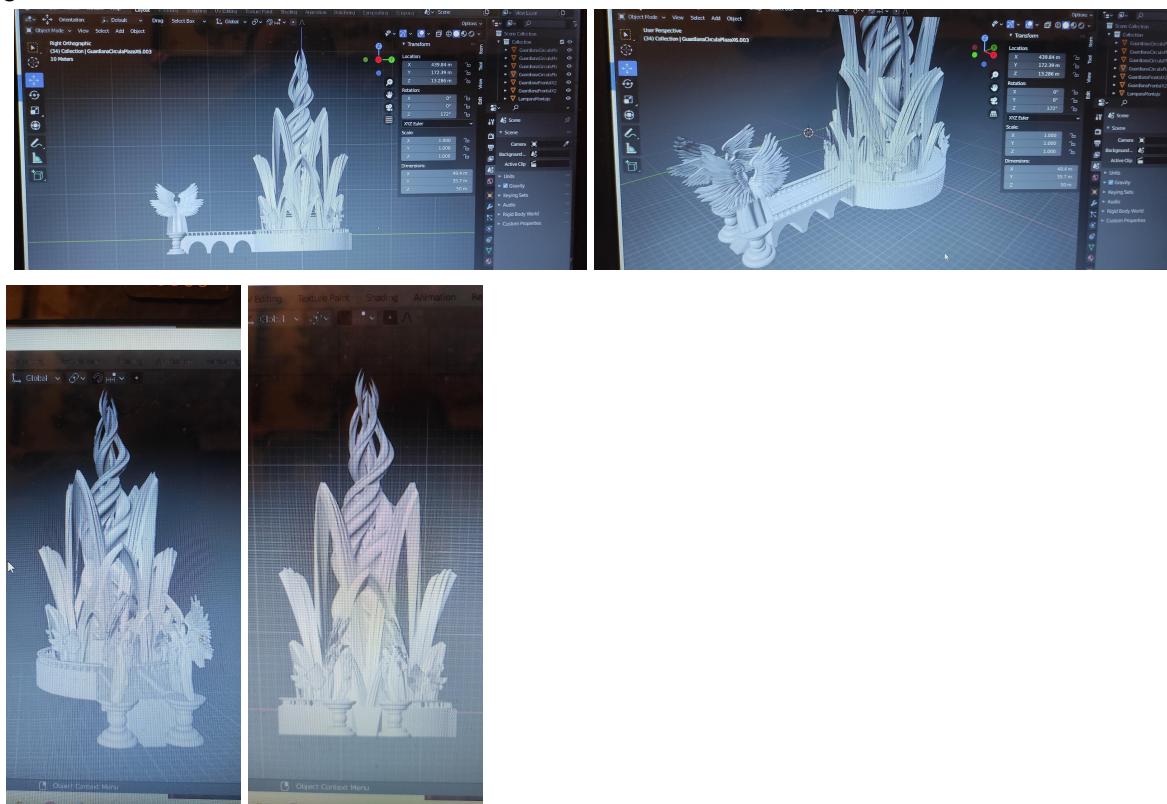
<https://programarfacil.com/blog/arduino-blog/buzzer-con-arduino-zumbador/> (Referencia para conectar el Buzzer)

Plano Electronico:



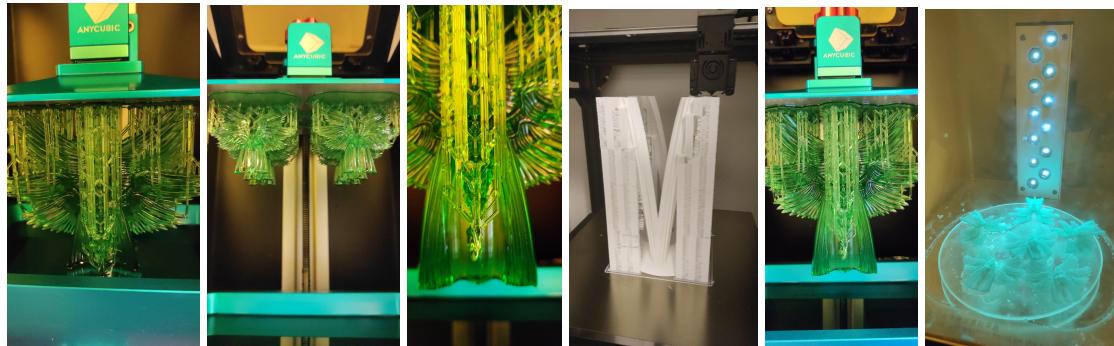
Montaje:

Se adapta el diseño hecho de otro proyecto para que contengan leds y electrónica con fusion 360 y montaje final maquetado en Blender ya que fusion 360 no puede con tanta geometría.



Se ha empezado imprimiendo todo el modelo en pla, petg y algunas partes en resina.

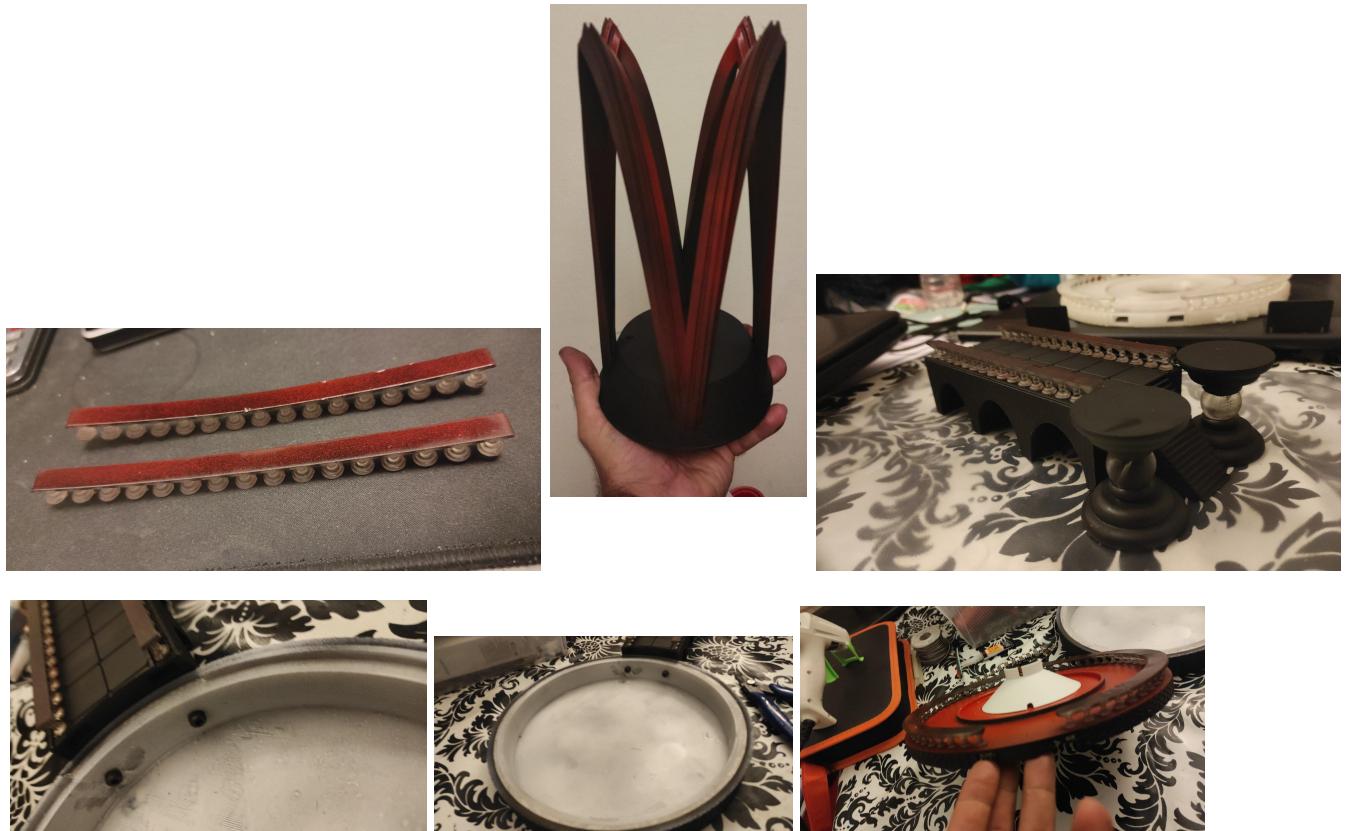




Una vez impreso limpiado los soportes en caso de PLA y PETG. En la resina una vez impreso se limpia con alcohol isopropílico se quitan los soportes y se cura durante 12 minutos en una máquina de curado con leds UV.

Se procede a la pintura de todas las piezas que lo necesitan.





En algunas partes calculé mal el espacio de la tira led y tuve que abrir espacio con los alicates de corte. al quitar algunas partes encaja bien la tira de led.



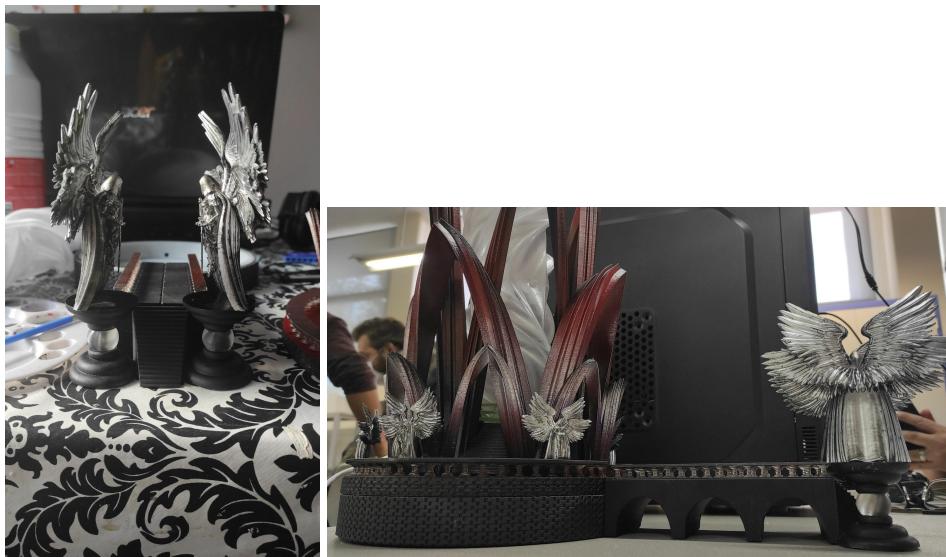
Se hace una pieza para que los leds que iluminan la llama desde la parte inferior.



La idea parece buena y encaja pero luego resulta que la luz que dan los leds es muy pequeña. Así que decidí agujerear la llama con una broca de 8mm y 40 cm de largo. Ahora la tira led irá por el centro de la llama para iluminar mejor.

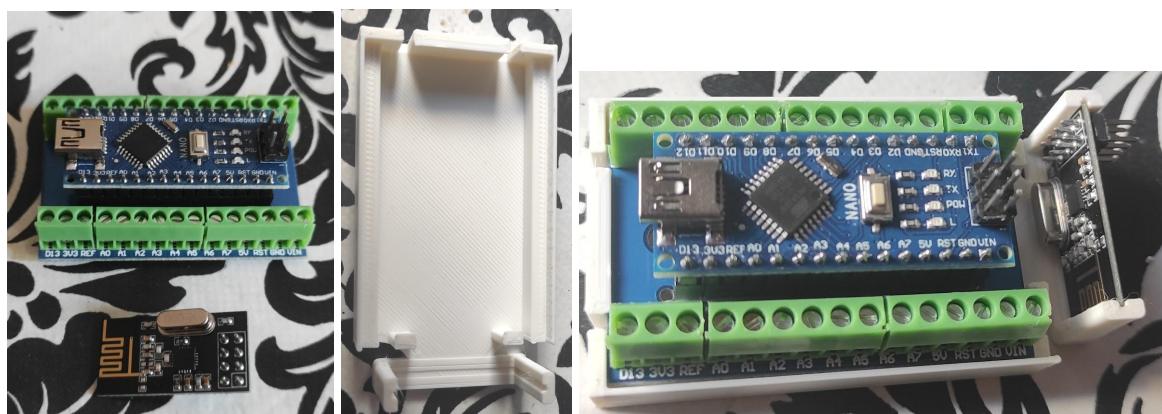


Se colocaron 2 leds RGB debajo de las guardianas de la entrada a los laterales de las escaleras.

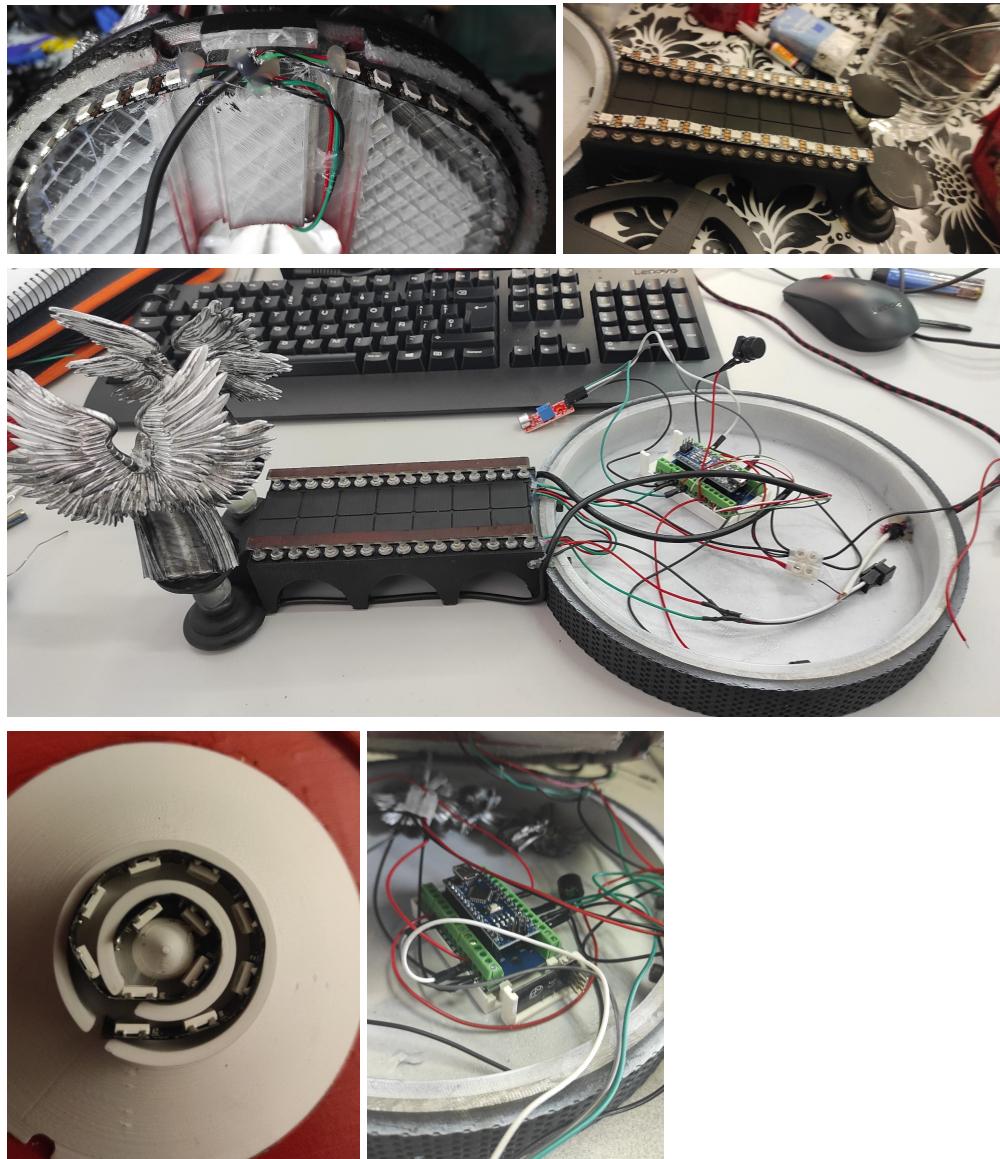


Se montaron todas las piezas, la idea es iluminar las barandillas tanto del puente como de la plaza del círculo central. Las guardianas de la entrada y la llama blanca.

Para la Placa arduino nano y el ESP8266 ESP-01 se diseña un soporte para que las placas queden lo mejor posicionado posible. Al final descarto el chip ESP8266 por falta de tiempo.

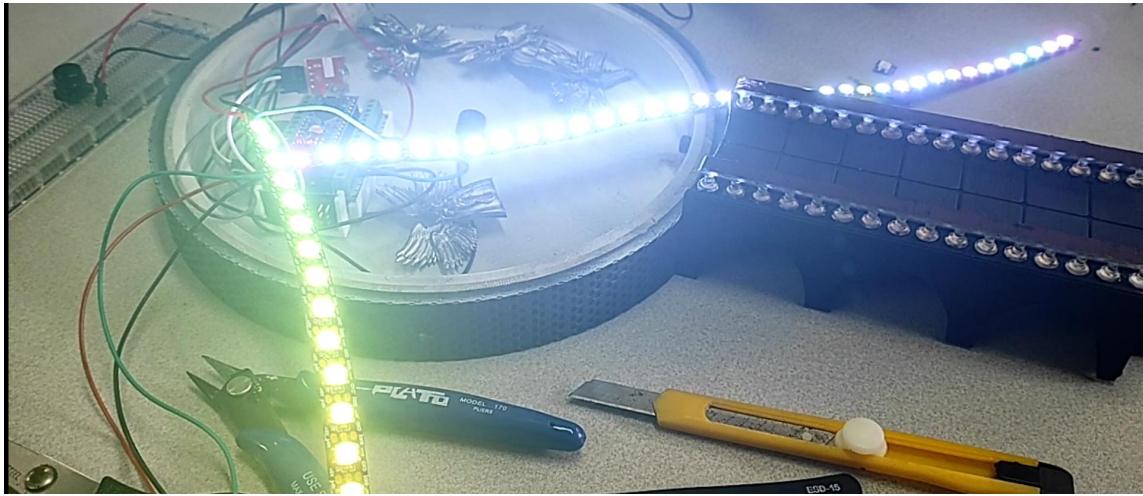


Empiezo a colocar todas las piezas dentro de la lámpara led.

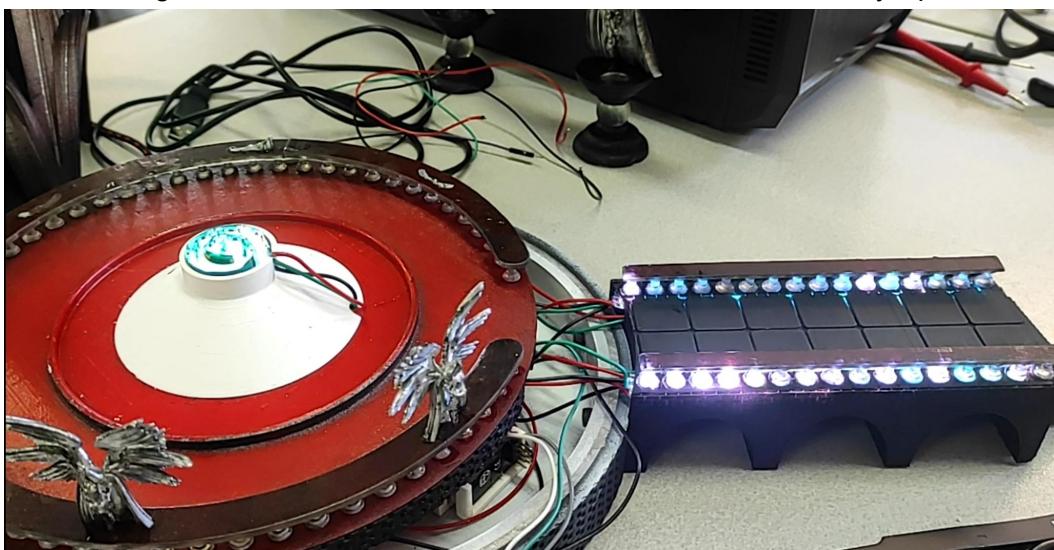


Se coloca un zumbador para que suene una vez arranca y cada vez que cambia de modo. También un botón para cambiar las diferentes formas de animación de la tira led. Se usa el módulo detector de ruido ambiente KY-037. Hay un pequeño lío de conceptos y no se sabe si detecta ruido o frecuencias. En internet se encuentran diferentes variantes así que nos ponemos a investigar el shield a ver cual es su verdadera función. Después de buscar con compañeros y el profesor se confirma que solo detecta Db de ruido y no frecuencias.

Surgen problemas que partes de la tira led no funcionan y me dedico a sanear el problema. Resulta que el corte entre leds de la tira led está mal hecho ya que la línea dibujada no es del todo correcta. Me dedico a testear partes de la tira y lo voy solucionando por sectores.



Se corren los problemas y la tira led de la plaza redonda y se añade una nueva. Se hacen algunos tests de animación led con la librería de Adafruit y Speed Led.



Una vez funciona nos encontramos con el problema que al conectar la tira led de 144 leds el alimentador de 5v 3,5A no da suficiente comida para todo los leds.

Calculamos el consumo total de leds y electrónica con el multímetro.

La primera prueba con solo los leds de la pasarela da un consumo de 0.76A al conectar el resto de tira led no arranca los leds ni el arduino. Con el multímetro nos da 3.2A.

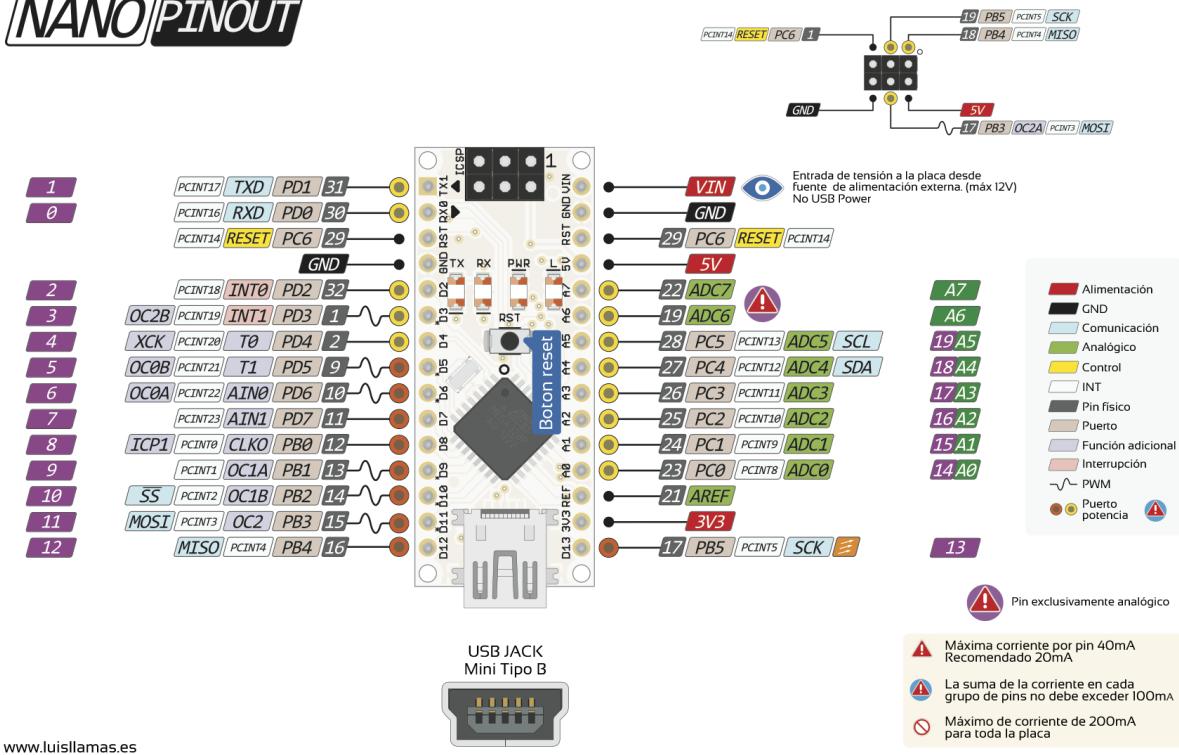
Después de hablar con David un compañero bastante avanzado en electrónica hacemos un cálculo teórico del consumo de todo el circuito.

Nos da 11,2 A de consumo para solucionar el problema de alimentación se procede a comprar una fuente de alimentación de 5V 12A.

Esperando a que llegue el material me dedico a preparar toda la programación en C++ para el funcionamiento correcto de los leds y los colores y animaciones que yo quiero usar.

Adjunto esquema para la explicación de conexión de los pines del arduino nano.

NANO PINOUT



www.luisllamas.es

El pin Tx y Rx lo reservamos para conectar el módulo wifi ESP8266 ESP-01. En caso que de tiempo.

D2: KY-037 Entrada Digital de datos

D3: Led RGB Guardiana Izquierda color R (Se usa este ya que se puede variar la intensidad de la luz del led)

D4: Salida datos de Tira led (Para controlar los colores e intensidad de la tira led WS2812B)

D5: Led RGB Guardiana Izquierda color G (Se usa este ya que se puede variar la intensidad de la luz del led)

D6: Led RGB Guardiana Izquierda color B (Se usa este ya que se puede variar la intensidad de la luz del led)

D7: Salida para el botón mecánico (Control de modos de animación led)

D8: Salida para el Zumbador.

D9: Led RGB Guardiana Derecha color R (Se usa este ya que se puede variar la intensidad de la luz del led)

D10: Led RGB Guardiana Derecha color G (Se usa este ya que se puede variar la intensidad de la luz del led)

D11: Led RGB Guardiana Derecha color B (Se usa este ya que se puede variar la intensidad de la luz del led)

D12: Led de luz blanca para las anillas alrededor de la flama.

A1: Umbral de frecuencias para el KY-037 (Da un valor analogico para saber la sensibilidad del sensor)

Aquí Adjunto link de Drive para ver un vídeo de animación de la lámpara led funcionando.

https://drive.google.com/file/d/1V8E0wgSDt0CFIZi1MVwlqv__dl94D5M/view?usp=sharing

Código:

```
#include <Adafruit_NeoPixel.h>
#include <FastLED.h>
#include <math.h>
#include <SoftwareSerial.h>
#include <EasyBuzzer.h>
#define N_PIXELS 176 // Número de píxeles en la tira
#define N_PIXELS_HALF (N_PIXELS/2)
#define MIC_PIN A1 // El micrófono está conectado a este pin analógico
#define LED_PIN 2 // La hebra de LED NeoPixel está conectada a este pin
#define SAMPLE_WINDOW 10 // Ventana de muestra para nivel medio
#define PEAK_HANG 24 //Tiempo de pausa antes de que caiga el punto máximo
#define PEAK_FALL 15 //Tasa de caída del punto máximo
#define PEAK_FALL2 8 //Tasa de caída del punto máximo
#define INPUT_FLOOR 10 //Rango más bajo de entrada de lectura analógica
#define INPUT_CEILING 1023 //Rango máximo de entrada de lectura analógica, cuanto menor sea el valor, más sensible (1023 = máx.) 300 (150)
#define DC_OFFSET 0 // Desviación de CC en la señal del micrófono: si no está en uso, deje 0
#define NOISE 10 // Ruido / zumbido / interferencia en la señal del micrófono
#define SAMPLES 6 // 60 Longitud del búfer para ajuste dinámico de nivel 60
#define TOP (N_PIXELS + 2) // Permita que el punto se salga ligeramente de la escala
#define SPEED .20 // Cantidad para incrementar el color RGB en cada ciclo
#define TOP2 (N_PIXELS + 1) // Permita que el punto se salga ligeramente de la escala
#define LAST_PIXEL_OFFSET N_PIXELS-1
#define PEAK_FALL_MILLIS 10 // Tasa de pico de caída del punto
#define POT_PIN 4 //4
#define BG 0
#define LAST_PIXEL_OFFSET N_PIXELS-1
#if FASTLED_VERSION < 3001000
#error "Requires FastLED 3.1 or later; check github for latest code."
#endif
#define BRIGHTNESS 255
#define LED_TYPE WS2812B // Solo use el LED_PIN para WS2812
#define COLOR_ORDER GRB
#define COLOR_MIN 0
#define COLOR_MAX 255
```

```

#define DRAW_MAX      100
#define SEGMENTS      2 // 4 Número de segmentos para tallar la barra de amplitud
en
#define COLOR_WAIT_CYCLES 10 // Ciclos de bucle para esperar entre el avance del
origen del píxel
#define qsubd(x, b) ((x>b)?b:0)
#define qsuba(x, b) ((x>b)?x-b:0) // Macro de resta analógica
sin signo. si resultado <0, entonces => 0. Por Andrew Tuline.
#define ARRAY_SIZE(A) (sizeof(A) / sizeof((A)[0]))

int Aros = 12;      //Definición del pin de los aros de la llama
int GuardianL_R = 4; //Difinición del led RGB de la guardiana de la izquierda
int GuardianL_G = 5; //Difinición del led RGB de la guardiana de la izquierda
int GuardianL_B = 6; //Difinición del led RGB de la guardiana de la izquierda
int GuardianR_R = 9; //Difinición del led RGB de la guardiana de la izquierda
int GuardianR_G = 10; //Difinición del led RGB de la guardiana de la izquierda
int GuardianR_B = 11; //Difinición del led RGB de la guardiana de la izquierda

struct CRGB leds[N_PIXELS];

Adafruit_NeoPixel strip = Adafruit_NeoPixel(N_PIXELS, LED_PIN, NEO_GRB +
NEO_KHZ800);

static uint16_t dist;      // Un número aleatorio para generador de ruido.
uint16_t scale = 10;       // 30 No recomendaría cambiar esto sobre la marcha, o la
animación será realmente bloqueada.
uint8_t maxChanges = 48;   // Valor para mezclar entre paletas.

CRGBPalette16 currentPalette(OceanColors_p);
CRGBPalette16 targetPalette(CloudColors_p);

//new ripple vu
uint8_t timeval = 20;      // Valor actual de 'retraso'. No, no uso retrasos, en su lugar uso
EVERY_N_MILLIS_I.
uint16_t loops = 0;         // Nuestro contador de bucles por segundo.
bool samplepeak = 0;        // Esta muestra está muy por encima del promedio y es un
'pico'.
uint16_t oldsample = 0;     // La muestra anterior se utiliza para la detección de picos y
para los valores "sobre la marcha".
bool thisdir = 0;
//new ripple vu

// Modos de animación
enum
{
} MODE;
bool reverse = true;

```

```

int BRIGHTNESS_MAX = 30;
int brightness = 10
;

byte
// peak    = 0,    // Utilizado para el punto que cae
// dotCount = 0,   // Contador de fotogramas para retrasar la velocidad de caída de puntos
    volCount = 0;   // Contador de fotogramas para almacenar datos de volúmenes pasados
int
reading,
vol[SAMPLES],    // Recolección de muestras de volumen anteriores
lvl    = 2,    // Nivel de audio "amortiguado" actual
minLvlAvg = 0,   // Para el ajuste dinámico del gráfico bajo y alto
maxLvlAvg = 512;
float
greenOffset = 30,
blueOffset = 150;
// ciclos variables

int CYCLE_MIN_MILLIS = 2;
int CYCLE_MAX_MILLIS = 1000;
int cycleMillis = 20;
bool paused = false;
long lastTime = 0;
bool boring = true;
bool gReverseDirection = false;
int      myhue =  0;
//vu ripple
uint8_t colour;
uint8_t myfade = 255;                                // Brillo inicial.
#define maxsteps 6                                     // 16 La declaración de caso no permitiría una
variable.
int peakspersec = 0;
int peakcount = 0;
uint8_t bgcol = 0;
int thisdelay = 20;
uint8_t max_bright = 255;

unsigned int sample;

//Samples
#define NSAMPLES 64
unsigned int samplearray[NSAMPLES];
unsigned long samplesum = 0;
unsigned int sampleavg = 0;
int samplecount = 0;

```

```

//unsigned int sample = 0;
unsigned long oldtime = 0;
unsigned long newtime = 0;

//Ripple variables
int color;
int center = 0;
int step = -1;
int maxSteps = 6; //16
float fadeRate = 0.80;
int diff;

//vu 8 variables
int
origin = 0,
color_wait_count = 0,
scroll_color = COLOR_MIN,
last_intensity = 0,
intensity_max = 0,
origin_at_flip = 0;
uint32_t
draw[DRAW_MAX];
boolean
growing = false,
fall_from_left = true;

//background color
uint32_t currentBg = random(256);
uint32_t nextBg = currentBg;
TBlendType currentBlending;

const int buttonPin = 5; // el número de la clavija del botón

//Variables will change:
int buttonPushCounter = 0; // contador del número de pulsaciones de botones
int buttonState = 0; // estado actual del botón
int lastButtonState = 0;

byte peak = 6; // 16 Nivel de pico de la columna; utilizado para la caída de puntos
// unsigned int sample;

byte dotCount = 0; //Contador de fotogramas para pico de puntos
byte dotHangCount = 0; //Contador de fotogramas para sujetar el punto pico

void sonidoTerminado(){

```

```

Serial.println ("Arrancada la placa");
}

void setup() {
  Serial.begin(9600);

  // Configuración del pin
  EasyBuzzer.setPin(8);
  // Configuración del beep
  EasyBuzzer.beep(
    2000,          // Frecuencia de herzios
    100,           // Duración beep en ms
    100,           // Duración silencio en ms
    2,              // Número de beeps por ciclos
    300,           // Duración de la pausa
    1,              // Número de ciclos
    sonidoTerminado // Función callback que es llamada cuando termina
  );

  //analogReference(EXTERNAL);
  pinMode (Aros, OUTPUT);
  pinMode (GuardianL_R, OUTPUT);
  pinMode (GuardianL_G, OUTPUT);
  pinMode (GuardianL_B, OUTPUT);
  pinMode (GuardianR_R, OUTPUT);
  pinMode (GuardianR_G, OUTPUT);
  pinMode (GuardianR_B, OUTPUT);
  pinMode(buttonPin, INPUT);

  //inicializar el buttonPin como salida
  digitalWrite(buttonPin, HIGH);

  // Enciende los leds
  digitalWrite(Aros, HIGH);
  digitalWrite(GuardianL_R, HIGH);
  digitalWrite(GuardianR_R, HIGH);

  // Serial.begin(9600);
  strip.begin();
  strip.show(); // todos los píxeles a 'off'

  Serial.begin(57600);
  delay(3000);

LEDS.addLeds<LED_TYPE,LED_PIN,COLOR_ORDER>(leds,N_PIXELS).setCorrection(Ty
picalLEDStrip);

```

```

LEDS.setBrightness(BRIGHTNESS);
dist = random16(12345);      // Un número semi-aleatorio para nuestro generador de
ruido

}

float fscale( float originalMin, float originalMax, float newBegin, float newEnd, float
inputValue, float curve){

    float OriginalRange = 0;
    float NewRange = 0;
    float zeroRefCurVal = 0;
    float normalizedCurVal = 0;
    float rangedValue = 0;
    boolean invFlag = 0;

    // condition curve parameter
    // limit range

    if (curve > 10) curve = 10;
    if (curve < -10) curve = -10;

    curve = (curve * -.1) ; // invertir y escalar - esto parece más intuitivo - los números
positivos dan más peso al extremo alto en la salida
    curve = pow(10, curve); // convertir la escala lineal en exponente logarítmico para otra
función pow

    // Compruebe valores de entrada fuera de rango
    if (inputValue < originalMin) {
        inputValue = originalMin;
    }
    if (inputValue > originalMax) {
        inputValue = originalMax;
    }

    // Zero Refference los valores
    OriginalRange = originalMax - originalMin;

    if (newEnd > newBegin){
        NewRange = newEnd - newBegin;
    }
    else
    {
        NewRange = newBegin - newEnd;
        invFlag = 1;
    }
}

```

```

zeroRefCurVal = inputValue - originalMin;
normalizedCurVal = zeroRefCurVal / OriginalRange; // normalizar a 0 - 1 float

// Verifique originalMin > originalMax: las matemáticas para todos los demás casos, es
// decir, los números negativos parecen funcionar bien
if (originalMin > originalMax ) {
    return 0;
}

if (invFlag == 0){
    rangedValue = (pow(normalizedCurVal, curve) * NewRange) + newBegin;

}
else // invierte los rangos
{
    rangedValue = newBegin - (pow(normalizedCurVal, curve) * NewRange);
}

return rangedValue;

}

void loop() {
    // función para que funcione la librería del buzzer
    EasyBuzzer.update();

    //para el microfono
    uint8_t i;
    uint16_t minLvl, maxLvl;
    int n, height;

    // fin del micrófono
    // lee el pin de entrada del botón pulsador:
    buttonState = digitalRead(buttonPin);
    //comparar el buttonState con su estado anterior
    if (buttonState != lastButtonState) {
        // si el estado ha cambiado, incrementa el contador
        if (buttonState == HIGH) {
            // si el estado actual es ALTO, entonces el botón
            // pasar de apagado a encendido:
            buttonPushCounter++;
            Serial.println("on");
            Serial.print("number of button pushes: ");
            Serial.println(buttonPushCounter);
            if(buttonPushCounter==16) {
                buttonPushCounter=1;
            }
        }
    }
}

```

```

    }
else {
    // si el estado actual es BAJO, entonces el botón
    // pasamos de encendido a apagado:
    Serial.println("off");
}
}

// guarda el estado actual como último estado,
// para la próxima vez a través del bucle
lastButtonState = buttonState;

switch (buttonPushCounter){

    case 1:
        buttonPushCounter==1; {
            All2(); // NORMAL
            break; }

    case 2:
        buttonPushCounter==2; {
            vu(); // NORMAL
            break; }

    case 3:
        buttonPushCounter==3; {
            vu1(); // Centro hacia fuera
            break; }

    case 4:
        buttonPushCounter==4; {
            vu2(); // Centro hacia adentro
            break; }

    case 5:
        buttonPushCounter==5; {
            Vu3(); // Arco iris normal
            break; }

    case 6:
        buttonPushCounter==6; {
            Vu4(); // Arco iris central
            break; }

    case 7:
        buttonPushCounter==7; {
            Vu5(); // Estrella fugaz
            break; }
}

```

```
    case 8:  
    buttonPushCounter==8; {  
    Vu6(); // Estrella fugaz  
    break;}  
  
    case 9:  
    buttonPushCounter==9; {  
    vu7(); // Ondulación con fondo  
    break;}  
  
    case 10:  
    buttonPushCounter==10; {  
    vu8(); // Romper  
    break;}  
  
    case 11:  
    buttonPushCounter==11; {  
    vu9(); // Pulso  
    break;}  
  
    case 12:  
    buttonPushCounter==12; {  
    vu10(); // Arroyo  
    break;}  
    case 13:  
    buttonPushCounter==13; {  
    vu11(); // Ondulación sin fondo  
    break;}  
  
    case 14:  
    buttonPushCounter==14; {  
    vu12(); // Ondulación sin fondo  
    break;}  
  
    case 15:  
    buttonPushCounter==15; {  
    vu13(); // Ondulación sin fondo  
    break;}  
  
    case 16:  
    buttonPushCounter==16; {  
    colorWipe(strip.Color(0, 0, 0), 10); // Negro  
    break;  
}  
}
```

```

void colorWipe(uint32_t c, uint8_t wait) {
    for(uint16_t i=0; i<strip.numPixels(); i++) {
        strip.setPixelColor(i, c);
        strip.show();
        if (digitalRead(buttonPin) != lastButtonState) // <----- add this
            return; // <----- and this
        delay(wait);
    }
}

void vu() {

    uint8_t i;
    uint16_t minLvl, maxLvl;
    int n, height;

    n = analogRead(MIC_PIN); // Lectura sin procesar del micrófono
    n = abs(n - 512 - DC_OFFSET); // Centro en cero
    n = (n <= NOISE) ? 0 : (n - NOISE); // Eliminar ruido / zumbido
    lvl = ((lvl * 7) + n) >> 3; //Lectura "amortiguada" (si no, parece nervioso)

    // Calcule la altura de la barra según los niveles dinámicos mínimo / máximo (punto fijo):
    height = TOP * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

    if(height < 0L) height = 0; // Salida de clip
    else if(height > TOP) height = TOP;
    if(height > peak) peak = height; // Mantenga el punto 'pico' en la parte superior

    // Píxeles de color basados en el degradado del arco iris
    for(i=0; i<N_PIXELS; i++) {
        if(i >= height) strip.setPixelColor(i, 0, 0, 0);
        else strip.setPixelColor(i,Wheel(map(i,0,strip.numPixels()-1,30,150)));
    }

    // Dibujar punto pico
    if(peak > 0 && peak <= N_PIXELS-1)
        strip.setPixelColor(peak,Wheel(map(peak,0,strip.numPixels()-1,30,150)));

    strip.show(); //Actualizar tira

    // Cada pocos fotogramas, haga que el píxel máximo se reduzca en 1:
    if(++dotCount >= PEAK_FALL) { //tasa de caída

```

```

    if(peak > 0) peak--;
    dotCount = 0;
}

vol[volCount] = n; // Guardar muestra para nivelación dinámica
if(++volCount >= SAMPLES) volCount = 0; // Contador de muestras de avance / traspaso

// Obtenga el rango de volumen de los fotogramas anteriores
minLvl = maxLvl = vol[0];
for(i=1; i<SAMPLES; i++) {
    if(vol[i] < minLvl) minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
}
// minLvl y maxLvl indican el rango de volumen sobre los fotogramas anteriores, utilizados
// para escalar verticalmente el gráfico de salida (por lo que parece interesante
// independientemente del nivel de volumen). Sin embargo, si están demasiado juntos
// (por ejemplo, a niveles de volumen muy bajos) el gráfico se vuelve muy grueso
// y 'nervioso' ... así que mantiene una distancia mínima entre ellos (esto
// también deja que el gráfico vaya a cero cuando no se reproduce ningún sonido):
if((maxLvl - minLvl) < TOP) maxLvl = minLvl + TOP;
minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; //Humedezca los niveles mínimo / máximo
maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (promedio móvil falso)

}

// Ingrese un valor de 0 a 255 para obtener un valor de color.
// Los colores son una transición r - g - b - de regreso a r.
uint32_t Wheel(byte WheelPos) {
    if(WheelPos < 85) {
        return strip.Color(WheelPos * 3, 255 - WheelPos * 3, 0);
    } else if(WheelPos < 170) {
        WheelPos -= 85;
        return strip.Color(255 - WheelPos * 3, 0, WheelPos * 3);
    } else {
        WheelPos -= 170;
        return strip.Color(0, WheelPos * 3, 0 - WheelPos * 3);
    }
}

void vu1() {

    uint8_t i;
    uint16_t minLvl, maxLvl;
    int n, height;
}

```

```

n = analogRead(MIC_PIN);           // Lectura sin procesar del micrófono
n = abs(n - 512 - DC_OFFSET); // Centro en cero
n = (n <= NOISE) ? 0 : (n - NOISE); // Eliminar ruido / zumbido
lvl = ((lvl * 7) + n) >> 3; // Lectura "amortiguada" (si no, parece nervioso)

// Calcule la altura de la barra según los niveles dinámicos mínimo / máximo (punto fijo):
height = TOP * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

if(height < 0L)    height = 0;    // Salida de clip
else if(height > TOP) height = TOP;
if(height > peak)  peak = height; // Mantenga el punto 'pico' en la parte superior

// Píxeles de color basados en el degradado del arco iris
for(i=0; i<N_PIXELS_HALF; i++) {
    if(i >= height) {
        strip.setPixelColor(N_PIXELS_HALF-i-1, 0, 0, 0);
        strip.setPixelColor(N_PIXELS_HALF+i, 0, 0, 0);
    }
    else {
        uint32_t color = Wheel(map(i,0,N_PIXELS_HALF-1,30,150));
        strip.setPixelColor(N_PIXELS_HALF-i-1,color);
        strip.setPixelColor(N_PIXELS_HALF+i,color);
    }
}

// Dibujar punto pico
if(peak > 0 && peak <= N_PIXELS_HALF-1) {
    uint32_t color = Wheel(map(peak,0,N_PIXELS_HALF-1,30,150));
    strip.setPixelColor(N_PIXELS_HALF-peak-1,color);
    strip.setPixelColor(N_PIXELS_HALF+peak,color);
}

strip.show(); // Actualizar tira

// Cada pocos fotogramas, haga que el píxel máximo se reduzca en 1:
if(++dotCount >= PEAK_FALL) { //tasa de caída

    if(peak > 0) peak--;
    dotCount = 0;
}

vol[volCount] = n;           // Guardar muestra para nivelación dinámica
if(++volCount >= SAMPLES) volCount = 0; // Contador de muestras de avance / traspaso

```

```

// Obtenga el rango de volumen de los fotogramas anteriores
minLvl = maxLvl = vol[0];
for(i=1; i<SAMPLES; i++) {
    if(vol[i] < minLvl)    minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
}
// minLvl y maxLvl indican el rango de volumen sobre los fotogramas anteriores, utilizados
// para escalar verticalmente el gráfico de salida (por lo que parece interesante
// independientemente del nivel de volumen). Sin embargo, si están demasiado juntos
// (por ejemplo, a niveles de volumen muy bajos) el gráfico se vuelve muy grueso
// y 'nervioso' ... así que mantén una distancia mínima entre ellos (esto
// también deja que el gráfico vaya a cero cuando no se reproduce ningún sonido):
if((maxLvl - minLvl) < TOP) maxLvl = minLvl + TOP;
minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Humedezca los niveles mínimo / máximo
maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (promedio móvil falso)

}

```

```

void vu2()
{
    unsigned long startMillis= millis(); // Inicio de la ventana de muestra
    float peakToPeak = 0; // nivel pico a pico

    unsigned int signalMax = 0;
    unsigned int signalMin = 1023;
    unsigned int c, y;

    while (millis() - startMillis < SAMPLE_WINDOW)
    {
        sample = analogRead(MIC_PIN);
        if (sample < 1024)
        {
            if (sample > signalMax)
            {
                signalMax = sample;
            }
            else if (sample < signalMin)
            {
                signalMin = sample;
            }
        }
        peakToPeak = signalMax - signalMin;

        // Serial.println(peakToPeak);
    }
}

```

```

for (int i=0;i<=N_PIXELS_HALF-1;i++){
    uint32_t color = Wheel(map(i,0,N_PIXELS_HALF-1,30,150));
    strip.setPixelColor(N_PIXELS-i,color);
    strip.setPixelColor(0+i,color);
}

c = fscale(INPUT_FLOOR, INPUT_CEILING, N_PIXELS_HALF, 0, peakToPeak, 2);

if(c < peak) {
    peak = c;      //Mantener el punto en la parte superior

    dotHangCount = 0; // haz que el punto cuelgue antes de caer
}
if (c <= strip.numPixels()) { // Fill partial column with off pixels
    drawLine(N_PIXELS_HALF, N_PIXELS_HALF-c, strip.Color(0, 0, 0));
    drawLine(N_PIXELS_HALF, N_PIXELS_HALF+c, strip.Color(0, 0, 0));
}

y = N_PIXELS_HALF - peak;
uint32_t color1 = Wheel(map(y,0,N_PIXELS_HALF-1,30,150));
strip.setPixelColor(y-1,color1);
//strip.setPixelColor(y-1,Wheel(map(y,0,N_PIXELS_HALF-1,30,150)));

y = N_PIXELS_HALF + peak;
strip.setPixelColor(y,color1);
//strip.setPixelColor(y+1,Wheel(map(y,0,N_PIXELS_HALF+1,30,150)));

strip.show();

// Animación de puntos pico basada en cuadros
if(dotHangCount > PEAK_HANG) { //Duración máxima de la pausa
    if(++dotCount >= PEAK_FALL2) { //Tasa de caída
        peak++;
        dotCount = 0;
    }
}
else {
    dotHangCount++;
}
}

```

```

void Vu3() {
    uint8_t i;
    uint16_t minLvl, maxLvl;
    int n, height;

    n = analogRead(MIC_PIN);          // Lectura sin procesar del micrófono
    n = abs(n - 512 - DC_OFFSET);    // Centro en cero
    n = (n <= NOISE) ? 0 : (n - NOISE); // Eliminar ruido / zumbido
    lvl = ((lvl * 7) + n) >> 3;    //Lectura "amortiguada" (si no, parece nervioso)

    // Calcule la altura de la barra según los niveles dinámicos mínimo / máximo (punto fijo):
    height = TOP * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

    if (height < 0L)    height = 0;    //Salida de clip
    else if (height > TOP) height = TOP;
    if (height > peak)  peak = height; // Mantenga el punto 'pico' en la parte superior

    greenOffset += SPEED;
    blueOffset += SPEED;
    if (greenOffset >= 255) greenOffset = 0;
    if (blueOffset >= 255) blueOffset = 0;

    // Píxeles de color basados en el degradado del arco iris
    for (i = 0; i < N_PIXELS; i++) {
        if (i >= height) {
            strip.setPixelColor(i, 0, 0, 0);
        } else {
            strip.setPixelColor(i, Wheel(
                map(i, 0, strip.numPixels() - 1, (int)greenOffset, (int)blueOffset)
            ));
        }
    }
    // Dibujar punto pico
    if(peak > 0 && peak <= N_PIXELS-1)
    strip.setPixelColor(peak,Wheel(map(peak,0,strip.numPixels()-1,30,150)));

    strip.show(); // Actualizar tira

    //Cada pocos fotogramas, haga que el píxel máximo se reduzca en 1:
    if(++dotCount >= PEAK_FALL) { //tasa de caída

        if(peak > 0) peak--;
        dotCount = 0;
    }
    strip.show(); // Actualizar tira

    vol[volCount] = n;
}

```

```

if (++volCount >= SAMPLES) {
    volCount = 0;
}

// Obtenga el rango de volumen de los fotogramas anteriores
minLvl = maxLvl = vol[0];
for (i = 1; i < SAMPLES; i++) {
    if (vol[i] < minLvl) {
        minLvl = vol[i];
    } else if (vol[i] > maxLvl) {
        maxLvl = vol[i];
    }
}

// minLvl y maxLvl indican el rango de volumen sobre los fotogramas anteriores, utilizados
// para escalar verticalmente el gráfico de salida (por lo que parece interesante
// independientemente del nivel de volumen). Sin embargo, si están demasiado juntos
// (por ejemplo, a niveles de volumen muy bajos) el gráfico se vuelve muy grueso
// y 'nervioso' ... así que mantiene una distancia mínima entre ellos (esto
// también deja que el gráfico vaya a cero cuando no se reproduce ningún sonido):
if ((maxLvl - minLvl) < TOP) {
    maxLvl = minLvl + TOP;
}
minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Dampen min/max levels
maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (fake rolling average)
}

```

```

void Vu4() {
    uint8_t i;
    uint16_t minLvl, maxLvl;
    int n, height;

    n = analogRead(MIC_PIN);           // Lectura sin procesar del micrófono
    n = abs(n - 512 - DC_OFFSET); // Centro en cero
    n = (n <= NOISE) ? 0 : (n - NOISE); // Eliminar ruido / zumbido
    lvl = ((lvl * 7) + n) >> 3; // Lectura "amortiguada" (si no, parece nervioso)

    // Calcule la altura de la barra según los niveles dinámicos mínimo / máximo (punto fijo):
    height = TOP * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

    if(height < 0L)    height = 0;    // Salida de clip
    else if(height > TOP) height = TOP;
    if(height > peak)  peak = height; // Mantenga el punto 'pico' en la parte superior
    greenOffset += SPEED;
    blueOffset += SPEED;
    if (greenOffset >= 255) greenOffset = 0;
    if (blueOffset >= 255) blueOffset = 0;
}

```

```

//Píxeles de color basados en el degradado del arco iris
for(i=0; i<N_PIXELS_HALF; i++) {
    if(i >= height) {
        strip.setPixelColor(N_PIXELS_HALF-i-1, 0, 0, 0);
        strip.setPixelColor(N_PIXELS_HALF+i, 0, 0, 0);
    }
    else {
        uint32_t color = Wheel(map(i,0,N_PIXELS_HALF-1,(int)greenOffset, (int)blueOffset));
        strip.setPixelColor(N_PIXELS_HALF-i-1,color);
        strip.setPixelColor(N_PIXELS_HALF+i,color);
    }
}

// Dibujar punto pico
if(peak > 0 && peak <= N_PIXELS_HALF-1) {
    uint32_t color = Wheel(map(peak,0,N_PIXELS_HALF-1,30,150));
    strip.setPixelColor(N_PIXELS_HALF-peak-1,color);
    strip.setPixelColor(N_PIXELS_HALF+peak,color);
}

strip.show(); // Actualizar tira

// Cada pocos fotogramas, haz que el píxel máximo se reduzca en 1:
if(++dotCount >= PEAK_FALL) { //tasa de caída

    if(peak > 0) peak--;
    dotCount = 0;
}

vol[volCount] = n;           // Guardar muestra para nivelación dinámica
if(++volCount >= SAMPLES) volCount = 0; // Contador de muestras de avance / traspaso

// Obtenga el rango de volumen de los fotogramas anteriores
minLvl = maxLvl = vol[0];
for(i=1; i<SAMPLES; i++) {
    if(vol[i] < minLvl)    minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
}
// minLvl y maxLvl indican el rango de volumen sobre los fotogramas anteriores, utilizados
// para escalar verticalmente el gráfico de salida (por lo que parece interesante
// independientemente del nivel de volumen). Sin embargo, si están demasiado juntos
// (por ejemplo, a niveles de volumen muy bajos) el gráfico se vuelve muy grueso
// y 'nervioso' ... así que mantén una distancia mínima entre ellos (esto
// también deja que el gráfico vaya a cero cuando no se reproduce ningún sonido):

```

```

if((maxLvl - minLvl) < TOP) maxLvl = minLvl + TOP;
minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Dampen min/max levels
maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (fake rolling average)

}

void Vu5()
{
    uint8_t i;
    uint16_t minLvl, maxLvl;
    int n, height;

    n = analogRead(MIC_PIN);           // Lectura sin procesar del micrófono
    n = abs(n - 512 - DC_OFFSET); // Centro en cero
    n = (n <= NOISE) ? 0 : (n - NOISE); // Eliminar ruido / zumbido
    lvl = ((lvl * 7) + n) >> 3; // "Lectura amortiguada (más se ve nervioso)"

    // Calcule la altura de la barra según los niveles dinámicos mínimo / máximo (punto fijo):
    height = TOP2 * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

    if(height < 0L)    height = 0;    // Salida de clip
    else if(height > TOP2) height = TOP2;
    if(height > peak)  peak = height; // Mantenga el punto 'pico' en la parte superior

#define CENTERED
// Píxeles de color basados en el degradado del arco iris
for(i=0; i<(N_PIXELS/2); i++) {
    if(((N_PIXELS/2)+i) >= height)
    {
        strip.setPixelColor(((N_PIXELS/2) + i), 0, 0, 0);
        strip.setPixelColor(((N_PIXELS/2) - i), 0, 0, 0);
    }
    else
    {
        strip.setPixelColor(((N_PIXELS/2) + i),Wheel(map(((N_PIXELS/2) +
i),0,strip.numPixels()-1,30,150)));
        strip.setPixelColor(((N_PIXELS/2) - i),Wheel(map(((N_PIXELS/2) -
i),0,strip.numPixels()-1,30,150)));
    }
}

// Dibujar punto pico
if(peak > 0 && peak <= LAST_PIXEL_OFFSET)
{
    strip.setPixelColor(((N_PIXELS/2) + peak),255,255,255); // (pico, Rueda (mapa (pico, 0,
    strip.numPixels () - 1,30,150)));
}

```

```

    strip.setPixelColor(((N_PIXELS/2) - peak),255,255,255); // (pico, Rueda (mapa (pico, 0,
strip.numPixels () - 1,30,150)));
}
#else
// Píxeles de color basados en el degradado del arco iris
for(i=0; i<N_PIXELS; i++)
{
    if(i >= height)
    {
        strip.setPixelColor(i, 0, 0, 0);
    }
    else
    {
        strip.setPixelColor(i,Wheel(map(i,0,strip.numPixels()-1,30,150)));
    }
}

// Dibujar punto pico
if(peak > 0 && peak <= LAST_PIXEL_OFFSET)
{
    strip.setPixelColor(peak,255,255,255); // (pico, Rueda (mapa (pico, 0, strip.numPixels () -
1,30,150)));
}

#endif

// Cada pocos fotogramas, haga que el píxel máximo se reduzca en 1:
if (millis() - lastTime >= PEAK_FALL_MILLIS)
{
    lastTime = millis();

    strip.show(); // Actualizar tira

    //fall rate
    if(peak > 0) peak--;
}

vol[volCount] = n;           // Guardar muestra para nivelación dinámica
if(++volCount >= SAMPLES) volCount = 0; // Contador de muestras de avance / traspaso

//Obtenga el rango de volumen de los fotogramas anteriores
minLvl = maxLvl = vol[0];
for(i=1; i<SAMPLES; i++)
{
    if(vol[i] < minLvl)    minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
}
// minLvl y maxLvl indican el rango de volumen sobre los fotogramas anteriores, utilizados

```

```

// para escalar verticalmente el gráfico de salida (por lo que parece interesante
// independientemente del nivel de volumen). Sin embargo, si están demasiado juntos
// (por ejemplo, a niveles de volumen muy bajos) el gráfico se vuelve muy grueso
// y 'nervioso' ... así que mantén una distancia mínima entre ellos (esto
// también deja que el gráfico vaya a cero cuando no se reproduce ningún sonido):
if((maxLvl - minLvl) < TOP2) maxLvl = minLvl + TOP2;
minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Dampen min/max levels
maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (fake rolling average)
}

void Vu6()
{
    uint8_t i;
    uint16_t minLvl, maxLvl;
    int n, height;

    n = analogRead(MIC_PIN);           //Lectura sin procesar del micrófono
    n = abs(n - 512 - DC_OFFSET); // Center on zero
    n = (n <= NOISE) ? 0 : (n - NOISE); // Eliminar ruido / zumbido
    lvl = ((lvl * 7) + n) >> 3; // Lectura "amortiguada" (si no, parece nervioso)
    // Calcular la altura de la barra en función de los niveles mínimos / máximos dinámicos
    // (punto fijo):
    height = TOP2 * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

    if(height < 0L)    height = 0;    // Salida de clip
    else if(height > TOP2) height = TOP2;
    if(height > peak)  peak = height; // Mantenga el punto 'pico' en la parte superior

#define CENTERED
//Dibujar punto pico
if(peak > 0 && peak <= LAST_PIXEL_OFFSET)
{
    strip.setPixelColor(((N_PIXELS/2) + peak),255,255,255); // (pico, Rueda (mapa (pico, 0,
    strip.numPixels () - 1,30,150)));
    strip.setPixelColor(((N_PIXELS/2) - peak),255,255,255); // (pico, Rueda (mapa (pico, 0,
    strip.numPixels () - 1,30,150)));
}
#else
// Píxeles de color basados en el degradado del arco iris
for(i=0; i<N_PIXELS; i++)
{
    if(i >= height)
    {
        strip.setPixelColor(i, 0, 0, 0);
    }
    else
    {

```

```

        }

    }

// Dibujar punto pico
if(peak > 0 && peak <= LAST_PIXEL_OFFSET)
{
    strip.setPixelColor(peak,0,0,255); // (pico, Rueda (mapa (pico, 0, strip.numPixels () -
1,30,150)));
}

#endif

// Cada pocos fotogramas, haga que el píxel máximo se reduzca en 1:

if (millis() - lastTime >= PEAK_FALL_MILLIS)
{
    lastTime = millis();

    strip.show(); //Actualizar tira

    //tasa de caída
    if(peak > 0) peak--;
}

vol[volCount] = n;           // Guardar muestra para nivelación dinámica
if(++volCount >= SAMPLES) volCount = 0; // Contador de muestras de avance / traspaso
// Get volume range of prior frames
minLvl = maxLvl = vol[0];
for(i=1; i<SAMPLES; i++)
{
    if(vol[i] < minLvl)   minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
}
// minLvl y maxLvl indican el rango de volumen sobre los fotogramas anteriores, utilizados
// para escalar verticalmente el gráfico de salida (por lo que parece interesante
// independientemente del nivel de volumen). Sin embargo, si están demasiado juntos
// (por ejemplo, a niveles de volumen muy bajos) el gráfico se vuelve muy grueso
// y 'nervioso' ... así que mantiene una distancia mínima entre ellos (esto
// también deja que el gráfico vaya a cero cuando no se reproduce ningún sonido):
if((maxLvl - minLvl) < TOP2) maxLvl = minLvl + TOP2;
minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Humea los niveles mínimo / máximo
maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (promedio móvil falso)
}

void vu7() {

EVERY_N_MILLISECONDS(1000) {

```

```

peaksperssec = peakcount; // Cuente los picos por segundo. Este
valor se convertirá en el tono de primer plano.
peakcount = 0; // Reinicia el contador cada segundo.
}

soundmems();

EVERY_N_MILLISECONDS(20) {
ripple3();
}

show_at_max_brightness_for_power();

} // loop()

```

```

void soundmems() { // Contador de promedio móvil: significa
que no tenemos que pasar por una matriz cada vez.
newtime = millis();
int tmp = analogRead(MIC_PIN) - 512;
sample = abs(tmp);

int potin = map(analogRead(POT_PIN), 0, 1023, 0, 60);

samplesum = samplesum + sample - samplearray[samplecount]; // Agregue la nueva
muestra y elimine la muestra más antigua de la matriz
sampleavg = samplesum / NSAMPLES; // Obtener un promedio
samplearray[samplecount] = sample; // Actualizar la muestra más
antigua de la matriz con una nueva muestra
samplecount = (samplecount + 1) % NSAMPLES; // Actualizar el contador de
la matriz

if (newtime > (oldtime + 200)) digitalWrite(13, LOW); // Apague el LED 200ms
después del último pico.

```

```

if ((sample > (sampleavg + potin)) && (newtime > (oldtime + 60))) { //Compruebe si hay un
pico, que es 30> el promedio, pero espere al menos 60 ms for another.
step = -1;
peakcount++;
digitalWrite(13, HIGH);
oldtime = newtime;
}
} // soundmems()

```

```

void ripple3() {

```

```

for (int i = 0; i < N_PIXELS; i++) leds[i] = CHSV(bgcol, 255, sampleavg*2); // Establece el
color de fondo.

switch (step) {

    case -1:                                // Inicializar variables de ondulación.
        center = random(N_PIXELS);
        colour = (peakspersec*10) % 255;           // Más picos / s = mayor
tono de color.
        step = 0;
        bgcol = bgcol+8;
        break;

    case 0:
        leds[center] = CHSV(colour, 255, 255);      // Muestra el primer píxel de la
ondulación.
        step++;
        break;

    case maxsteps:                           //Al final de las ondas.
        // step = -1;
        break;

    default:                                 // Media de las ondas.
        leds[(center + step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255, myfade/step*2);
//Envoltura simple de Marc Miller.
        leds[(center - step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255, myfade/step*2);
        step++;                               // Próximo paso.
        break;
    } // cambiar de paso
} // ripple()

void vu8() {
    int intensity = calculateIntensity();
    updateOrigin(intensity);
    assignDrawValues(intensity);
    writeSegmented();
    updateGlobals();
}

int calculateIntensity() {
    int    intensity;

    reading  = analogRead(MIC_PIN);           // Lectura sin procesar del micrófono
    reading  = abs(reading - 512 - DC_OFFSET); //Centro en cero
    reading  = (reading <= NOISE) ? 0 : (reading - NOISE); // Eliminar ruido / zumbido
    lvl = ((lvl * 7) + reading) >> 3; // Lectura "amortiguada" (si no, parece nervioso)
}

```

```

// Calcule la altura de la barra según los niveles dinámicos mínimo / máximo (punto fijo):
intensity = DRAW_MAX * (lvl - minLvlAvg) / (long)(maxLvlAvg - minLvlAvg);

return constrain(intensity, 0, DRAW_MAX-1);
}

void updateOrigin(int intensity) {
    // detectar el cambio de pico y guardar el origen en el vértice de la curva
    if (growing && intensity < last_intensity) {
        growing = false;
        intensity_max = last_intensity;
        fall_from_left = !fall_from_left;
        origin_at_flip = origin;
    } else if (intensity > last_intensity) {
        growing = true;
        origin_at_flip = origin;
    }
    last_intensity = intensity;

    // ajustar el origen si cae
    if (!growing) {
        if (fall_from_left) {
            origin = origin_at_flip + ((intensity_max - intensity) / 2);
        } else {
            origin = origin_at_flip - ((intensity_max - intensity) / 2);
        }
        // Correcta para el origen fuera de límites
        if (origin < 0) {
            origin = DRAW_MAX - abs(origin);
        } else if (origin > DRAW_MAX - 1) {
            origin = origin - DRAW_MAX - 1;
        }
    }
}

void assignDrawValues(int intensity) {
    // dibuja amplitud como 1/2 intensidad en ambas direcciones desde el origen
    int min_lit = origin - (intensity / 2);
    int max_lit = origin + (intensity / 2);
    if (min_lit < 0) {
        min_lit = min_lit + DRAW_MAX;
    }
    if (max_lit >= DRAW_MAX) {
        max_lit = max_lit - DRAW_MAX;
    }
    for (int i=0; i < DRAW_MAX; i++) {
        //si i está dentro del origen +/- 1/2 intensidad

```

```

    if (
        (min_lit < max_lit && min_lit < i && i < max_lit) // el rango está dentro de los límites e i
        está dentro del rango
        || (min_lit > max_lit && (i > min_lit || i < max_lit)) // el rango se ajusta fuera de los límites
        y yo está dentro de ese ajuste
    ) {
        draw[i] = Wheel(scroll_color);
    } else {
        draw[i] = 0;
    }
}

void writeSegmented() {
    int seg_len = N_PIXELS / SEGMENTS;

    for (int s = 0; s < SEGMENTS; s++) {
        for (int i = 0; i < seg_len; i++) {
            strip.setPixelColor(i + (s*seg_len), draw[map(i, 0, seg_len, 0, DRAW_MAX)]);
        }
    }
    strip.show();
}

uint32_t * segmentAndResize(uint32_t* draw) {
    int seg_len = N_PIXELS / SEGMENTS;

    uint32_t segmented[N_PIXELS];
    for (int s = 0; s < SEGMENTS; s++) {
        for (int i = 0; i < seg_len; i++) {
            segmented[i + (s * seg_len)] = draw[map(i, 0, seg_len, 0, DRAW_MAX)];
        }
    }

    return segmented;
}

void writeToStrip(uint32_t* draw) {
    for (int i = 0; i < N_PIXELS; i++) {
        strip.setPixelColor(i, draw[i]);
    }
    strip.show();
}

void updateGlobals() {
    uint16_t minLvl, maxLvl;

    //advance color wheel
}

```

```

color_wait_count++;
if (color_wait_count > COLOR_WAIT_CYCLES) {
    color_wait_count = 0;
    scroll_color++;
    if (scroll_color > COLOR_MAX) {
        scroll_color = COLOR_MIN;
    }
}

vol[volCount] = reading; // Guardar muestra para nivelación dinámica
if(++volCount >= SAMPLES) volCount = 0; // Contador de muestras de avance / traspaso

// Obtenga el rango de volumen de los fotogramas anteriores
minLvl = maxLvl = vol[0];
for(uint8_t i=1; i<SAMPLES; i++) {
    if(vol[i] < minLvl) minLvl = vol[i];
    else if(vol[i] > maxLvl) maxLvl = vol[i];
}
// minLvl y maxLvl indican el rango de volumen sobre los fotogramas anteriores, utilizados
// para escalar verticalmente el gráfico de salida (por lo que parece interesante
// independientemente del nivel de volumen). Sin embargo, si están demasiado juntos
// (por ejemplo, a niveles de volumen muy bajos) el gráfico se vuelve muy grueso
// y 'nervioso' ... así que mantén una distancia mínima entre ellos (esto
// también deja que el gráfico vaya a cero cuando no se reproduce ningún sonido):
if((maxLvl - minLvl) < N_PIXELS) maxLvl = minLvl + N_PIXELS;
minLvlAvg = (minLvlAvg * 63 + minLvl) >> 6; // Dampen min/max levels
maxLvlAvg = (maxLvlAvg * 63 + maxLvl) >> 6; // (fake rolling average)
}

```

```

void vu9() {
    //currentBlending = LINEARBLEND;
    currentPalette = OceanColors_p; // Paleta inicial.
    currentBlending = LINEARBLEND;
    EVERY_N_SECONDS(5); // Cambia la paleta cada 5 segundos.
    for (int i = 0; i < 16; i++) {
        targetPalette[i] = CHSV(random8(), 255, 255);
    }
}

EVERY_N_MILLISECONDS(100); // IMPRESIONANTE capacidad de
mezcla de paletas una vez que cambian.
uint8_t maxChanges = 24;
nblendPaletteTowardPalette(currentPalette, targetPalette, maxChanges);
}

```

```

EVERY_N_MILLIS_1(thistimer,20) { // Para divertirnos, hagamos que la
animación tenga un ritmo variable.
    uint8_t timeval = beatsin8(10,20,50); // Utilice una onda sinusoidal para la
línea de abajo. También podría utilizar la detección de picos / latidos.
    thistimer.setPeriod(timeval); // Le permite cambiar la frecuencia con la
que se ejecuta esta rutina.
    fadeToBlackBy(leds, N_PIXELS, 16); // 1 = lento, 255 = desvanecimiento
rápido. Dependiendo de la velocidad de desvanecimiento, los LED más alejados se
desvanecerán.
    sndwave();
    soundble();
}
FastLED.setBrightness(max_bright);
FastLED.show();

} // loop()

```

```

void soundble() { // Muestreo rápido y sucio del micrófono.

int tmp = analogRead(MIC_PIN) - 512 - DC_OFFSET;
sample = abs(tmp);

} // soundmems()

```

```

void sndwave() {

    leds[N_PIXELS/2] = ColorFromPalette(currentPalette, sample, sample*2, currentBlending);
// Ponga la muestra en el centro

    for (int i = N_PIXELS - 1; i > N_PIXELS/2; i--) { // moverse a la izquierda // Copie a la
izquierda y deje que el desvanecimiento haga el resto.
        leds[i] = leds[i - 1];
    }

    for (int i = 0; i < N_PIXELS/2; i++) { // mover a la derecha // Copie a la derecha
y deje que el resto se desvanezca.
        leds[i] = leds[i + 1];
    }
    addGlitter(sampleavg);
}

void vu10() {

```

```

EVERY_N_SECONDS(5) {                                // Cambie la paleta de destino a una
aleatoria cada 5 segundos.
    static uint8_t baseC = random8();                // Puede usar esto como un color de
línea de base si desea tonos similares en la siguiente línea.

    for (int i = 0; i < 16; i++) {
        targetPalette[i] = CHSV(random8(), 255, 255);
    }
}

EVERY_N_MILLISECONDS(100) {
    uint8_t maxChanges = 24;
    nblendPaletteTowardPalette(currentPalette, targetPalette, maxChanges); // IMPRESIONANTE capacidad de mezcla de paletas.
}

EVERY_N_MILLISECONDS(thisdelay) {                   // Retardo sin bloqueo basado en
FastLED para actualizar / mostrar la secuencia.
    soundtun();
    FastLED.setBrightness(max_bright);
    FastLED.show();
}
} // loop()

```

```

void soundtun() {

    int n;
    n = analogRead(MIC_PIN);                         // Raw leyendo del micrófono
    n = qsuba(abs(n-512), 10);                      // Centrarse en cero y deshacerse del ruido
de bajo nivel
    CRGB newcolour = ColorFromPalette(currentPalette, constrain(n,0,255),
constrain(n,0,255), currentBlending);
    nblend(leds[0], newcolour, 128);

    for (int i = N_PIXELS-1; i>0; i--) {
        leds[i] = leds[i-1];
    }
}

} // soundmems()

```

```

void vu11() {

EVERY_N_MILLISECONDS(1000) {
    peakspersec = peakcount;                      // Cuente los picos por segundo. Este
valor se convertirá en el tono de primer plano.
}

```

```

peakcount = 0;                                // Reinicia el contador cada segundo.
}

soundrip();

EVERY_N_MILLISECONDS(20) {
    rippled();
}

FastLED.show();

} // loop()

void soundrip() {                                // Contador de promedio móvil: significa que no
tenemos que pasar por una matriz cada vez.

    newtime = millis();
    int tmp = analogRead(MIC_PIN) - 512;
    sample = abs(tmp);

    int potin = map(analogRead(POT_PIN), 0, 1023, 0, 60);

    samplesum = samplesum + sample - samplearray[samplecount]; // Agregue la nueva
muestra y elimine la muestra más antigua de la matriz
    sampleavg = samplesum / NSAMPLES;                      // Obtener un promedio

    Serial.println(sampleavg);

    samplearray[samplecount] = sample;                      // Actualizar la muestra más antigua
de la matriz con una nueva muestra
    samplecount = (samplecount + 1) % NSAMPLES;           // Actualizar el contador de la
matriz

    if (newtime > (oldtime + 200)) digitalWrite(13, LOW); // Apague el LED 200ms después
del último pico.

    if ((sample > (sampleavg + potin)) && (newtime > (oldtime + 60))) { // Compruebe si hay un
pico, que es 30> el promedio, pero espere al menos 60 ms para otro.
        step = -1;
        peakcount++;
        oldtime = newtime;
    }

} // soundmems()

```

```

void rippled() {

    fadeToBlackBy(leds, N_PIXELS, 64); // 8 bit, 1 = lento, 255 = rapido

    switch (step) {

        case -1: // Inicializar variables de ondulación.
            center = random(N_PIXELS);
            colour = (peakspersec*10) % 255; // Más picos / s = mayor tono de color.
            step = 0;
            break;

        case 0: // Muestra el primer píxel de la
            leds[center] = CHSV(colour, 255, 255);
            ondulación.
            step++;
            break;

        case maxsteps: // Al final de las ondas.
            // step = -1;
            break;

        default: // Media de las ondas.
            leds[(center + step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255, myfade/step*2);
            // Envoltura simple de Marc Miller.
            leds[(center - step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255, myfade/step*2);
            step++; // Próximo paso.
            break;
    } // cambiar de paso

} // ripple()

//Se utiliza para dibujar una línea entre dos puntos de un color determinado.
void drawLine(uint8_t from, uint8_t to, uint32_t c) {
    uint8_t fromTemp;
    if (from > to) {
        fromTemp = from;
        from = to;
        to = fromTemp;
    }
    for(int i=from; i<=to; i++){
        strip.setPixelColor(i, c);
    }
}

void setPixel(int Pixel, byte red, byte green, byte blue) {

```

```

strip.setPixelColor(Pixel, strip.Color(red, green, blue));

}

void setAll(byte red, byte green, byte blue) {

for(int i = 0; i < N_PIXELS; i++) {

setPixel(i, red, green, blue);

}

strip.show();

}

void vu12() {

EVERY_N_MILLISECONDS(1000) {
    peakspersec = peakcount; // Cuente los picos por segundo. Este
valor se convertirá en el tono de primer plano.
    peakcount = 0; // RPonga el contador cada segundo.
}

soundripped();

EVERY_N_MILLISECONDS(20) {
    rippvu();
}

FastLED.show();

} // loop()

void soundripped() { // Contador de promedio móvil: significa que
no tenemos que pasar por una matriz cada vez.

newtime = millis();
int tmp = analogRead(MIC_PIN) - 512;
sample = abs(tmp);

int potin = map(analogRead(POT_PIN), 0, 1023, 0, 60);

samplesum = samplesum + sample - samplearray[samplecount]; // Agregue la nueva
muestra y elimine la muestra más antigua de la matriz
sampleavg = samplesum / NSAMPLES; // Obtener un promedio
Serial.println(sampleavg);
}

```

```

samplearray[samplecount] = sample;           // Actualizar la muestra más antigua
de la matriz con una nueva muestra
samplecount = (samplecount + 1) % NSAMPLES;    // Actualizar el contador de la
matriz

if (newtime > (oldtime + 200)) digitalWrite(13, LOW); // Apague el LED 200ms después
del último pico.

if ((sample > (sampleavg + potin)) && (newtime > (oldtime + 60)) ) { // Compruebe si hay un
pico, que es 30> el promedio, pero espere al menos 60 ms para otro.
step = -1;
peakcount++;
oldtime = newtime;
}

} // soundmems()
void rippvu() {                                // Mostrar ondas provocadas por
picos.

fadeToBlackBy(leds, N_PIXELS, 64);             // 8 bit, 1 = lento, 255 = rápido

switch (step) {

case -1:                                     // Inicializar variables de ondulación.
center = random(N_PIXELS);
colour = (peakspersec*10) % 255;              // Más picos / s = mayor tono de color.
step = 0;
break;

case 0:                                       // Muestra el primer píxel de la
onundación.
leds[center] = CHSV(colour, 255, 255);
step++;
break;

case maxsteps:                                // Al final de las ondas.
// step = -1;
break;

default:                                      // Media de las ondas.
leds[(center + step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255, myfade/step*2);
// Envoltura simple de Marc Miller.
leds[(center - step + N_PIXELS) % N_PIXELS] += CHSV(colour, 255, myfade/step*2);
step++;                                         // Next step.
break;
} // switch step

```

```

    addGlitter(sampleavg);
} // ripple()

void vu13() {                                // El >>>>>> L-O-O-P
<<<<<<<<<<<<<<<<<< está enterrado aquí !!! 11! 1!

EVERY_N_MILLISECONDS(1000) {
    peakspersec = peakcount;           // Cuente los picos por segundo. Este
valor se convertirá en el tono de primer plano.
    peakcount = 0;                   // Reinicia el contador cada segundo.
}

soundripper();

EVERY_N_MILLISECONDS(20) {
jugglep();
}

FastLED.show();

} // loop()

void soundripper() {                         // Contador de promedio móvil: significa que
no tenemos que pasar por una matriz cada vez.
    newtime = millis();
    int tmp = analogRead(MIC_PIN) - 512;
    sample = abs(tmp);

    int potin = map(analogRead(POT_PIN), 0, 1023, 0, 60);

    samplesum = samplesum + sample - samplearray[samplecount]; // Agregue la nueva
muestra y elimine la muestra más antigua de la matriz
    sampleavg = samplesum / NSAMPLES;           // Obtener un promedio
    Serial.println(sampleavg);

    samplearray[samplecount] = sample;           // Actualizar la muestra más antigua
de la matriz con una nueva muestra
    samplecount = (samplecount + 1) % NSAMPLES; // Update el contador de la
matriz

    if (newtime > (oldtime + 200)) digitalWrite(13, LOW); // Apague el LED 200ms después
del último pico.
}

```

```
if ((sample > (sampleavg + potin)) && (newtime > (oldtime + 60)) ) { // Compruebe si hay un pico, que es 30> el promedio, pero espere al menos 60 ms para otro.
```

```
    step = -1;
```

```
    peakcount++;
```

```
    oldtime = newtime;
```

```
                // Cambie la función del patrón actual periódicamente.
```

```
    jugglep();
```

```
}
```

```
} // loop()
```

```
void jugglep() { // Utilice la rutina de malabares, pero ajuste la base de tiempo en función de sampleavg para obtener cierta aleatoriedad.
```

```
// Persistent local variables
```

```
static uint8_t thishue=0;
```

```
timeval = 40; // Nuestro valor de temporizador
```

```
EVERY_N_MILLIS_I.
```

```
leds[0] = ColorFromPalette(currentPalette, thishue++, sampleavg, LINEARBLEND);
```

```
for (int i = N_PIXELS-1; i >0 ; i-- ) leds[i] = leds[i-1];
```

```
addGlitter(sampleavg/2); // Agregue brillo basado en sampleavg. Por Andrew Tuline.
```

```
} // matrix()
```

```
// Input a value 0 to 255 to get a color value.
```

```
// The colours are a transition r - g - b - back to r.
```

```
uint32_t Wheel(byte WheelPos, float opacity) {
```

```
    if(WheelPos < 85) {
```

```
        return strip.Color((WheelPos * 3) * opacity, (255 - WheelPos * 3) * opacity, 0);
```

```
}
```

```
    else if(WheelPos < 170) {
```

```
        WheelPos -= 85;
```

```
        return strip.Color((255 - WheelPos * 3) * opacity, 0, (WheelPos * 3) * opacity);
```

```
}
```

```
    else {
```

```
        WheelPos -= 170;
```

```
        return strip.Color(0, (WheelPos * 3) * opacity, (255 - WheelPos * 3) * opacity);
```

```
}
```

```
}
```

```
void addGlitter( fract8 chanceOfGlitter) { // Agreguemos un poco de  
brillo, gracias a Mark.  
  
if( random8() < chanceOfGlitter) {  
    leds[random16(N_PIXELS)] += CRGB::White;  
}  
  
} // addGlitter()  
//Lista de patrones para recorrer. Cada uno se define como una función separada a  
continuación.  
typedef void (*SimplePatternList[])();  
SimplePatternList qPatterns = {vu, vu1, vu2, Vu3, Vu4, Vu5, Vu6, vu7, vu8, vu9, vu10, vu11,  
vu12, vu13};  
uint8_t qCurrentPatternNumber = 0; // Número de índice del patrón actual  
  
void nextPattern2()  
{  
    // agregue uno al número de patrón actual y envuélvalo al final  
    qCurrentPatternNumber = (qCurrentPatternNumber + 1) % ARRAY_SIZE( qPatterns);  
}  
void All2()  
{  
    // Llame a la función de patrón actual una vez, actualizando la matriz 'leds'  
    qPatterns[qCurrentPatternNumber]();  
    EVERY_N_SECONDS( 30 ) { nextPattern2(); } // cambiar patrones periódicamente  
}
```

Menciones:

Joan Masdemont (Preciado profesor de placas robóticas)

*Ayuda en todos los puntos de electrónica y programación.

*Suministro de material necesario para que funcione fuente alimentación 5v 12A

Susana (Compañera de curso de placas robóticas)

*Ayuda en el entendimiento del código de adafruit.

Isa (Compañera de curso de placas robóticas)

*Ayuda en el entendimiento del funcionamiento real del sensor KY-037 y circuito con
leds.

Leonardo (Compañero de curso de placas robóticas)

*Apoyo moral cuando los leds no funcionaban con muy buen humor.

*Bromas varias cuando las cosas no entraban ayuda mucho a mantener la moral alta
y seguir adelante.

David (Compañero del curso de placas robóticas)

*Ayuda en el manejo, soldadura y cableado entre tiras leds.

*Ayuda en calculo de consumo de leds y saber que fuente de alimentación usar.