MINI RADAR

Integrantes: Thiago Arellano, Felipe Luchetta, Day Vila, Alejandro Alarcon

Profesor: Gonzalo Consorti

Año: 4to 1ra

Grupo 4 Computación



Thiago Arellano

Introducción

En nuestro grupo decidimos hacer el mini radar ya que es un proyecto innovador y bastante útil para funcionalidades de cosas a distancia. Deberemos de hacer un mini radar y que este lo interprete en una pantalla. Para esto hay que investigar los componentes que necesitaremos y para qué nos sirven cada uno de ellos, además de aprender a utilizar el código del lenguaje Processing.

Breve descripción del mini radar:

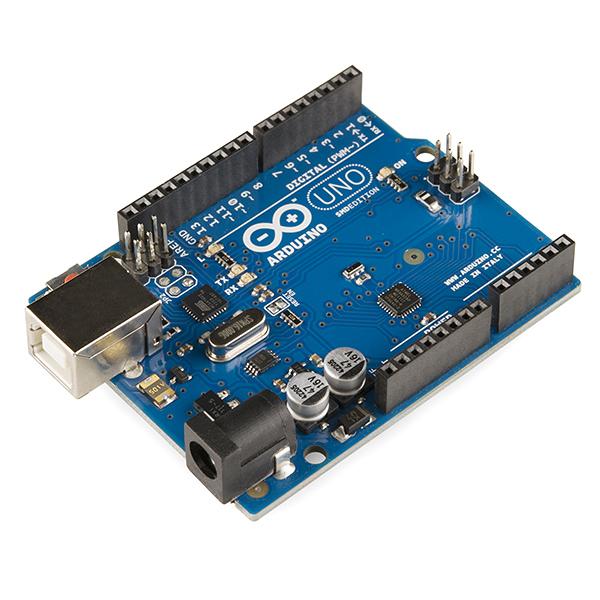
El mini radar es un dispositivo pequeño y fácil de usar que sirve para detectar objetos o movimientos en un área determinada, midiendo su distancia y velocidad. Usando tecnología de sensores y recursos disponibles, este detectará la posición y movimiento de objetos cercanos utilizando ondas de radio. Con este sistema, se podrá obtener información precisa del entorno sin necesidad de tocar los objetos detectados.

Vamos a construir un mini radar usando una placa de pruebas donde se conectarán diferentes componentes electrónicos, como un Arduino que se encargará de procesar los datos y controlar el sistema. El buzzer sonará para alertar sobre la presencia de objetos detectados, mientras que un sensor de proximidad medirá la distancia de los objetos en tiempo real. También vamos a añadir una pantalla para mostrar de manera visual la distancia de los objetos detectados y la velocidad a la que nos movemos. Por último, el servomotor se usará para mover el radar automáticamente y ajustar su área de detección, logrando una mejor cobertura.

Este mini radar es fácil de armar, e ideal para aprender sobre sensores, control de dispositivos electrónicos y el funcionamiento básico de sistemas de radar en un formato simple y práctico. Sin embargo, todas las instrucciones y todo el proceso digital se llevará a cabo en otro programa aparte de arduino, Processing es el programa donde más se va a trabajar para poder llegar a visualizar toda la parte gráfica del mini radar.

Componentes:

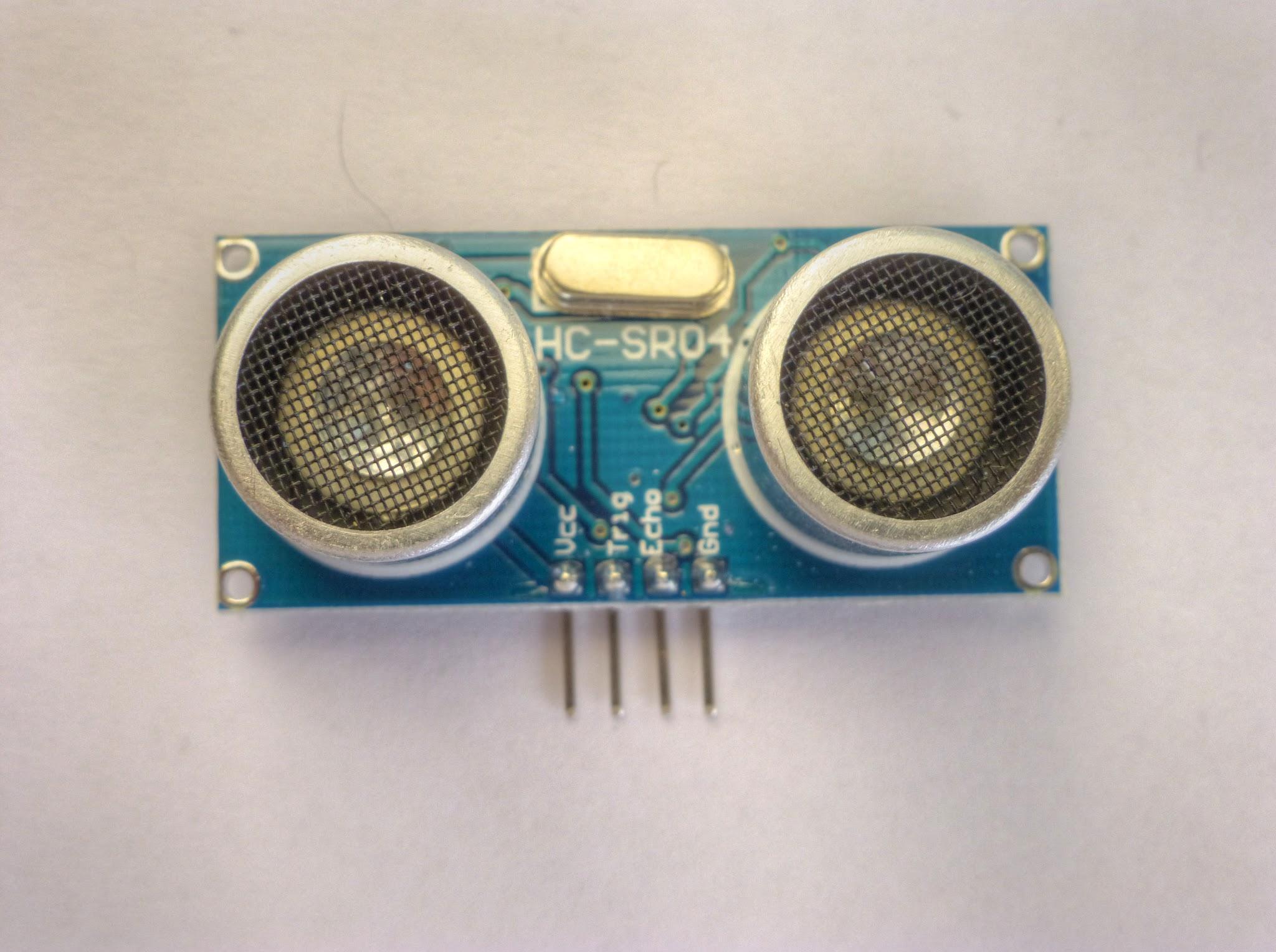
La placa de Arduino UNO, será el cerebro que se encargará de ejecutar cada parte digital y virtual que realizaremos durante la elaboración del código, también a su vez mantiene con sus pines la coneccion de los cables a la placa de pruebas, el sensor ultrasónico, el servomotor y el buzzer.



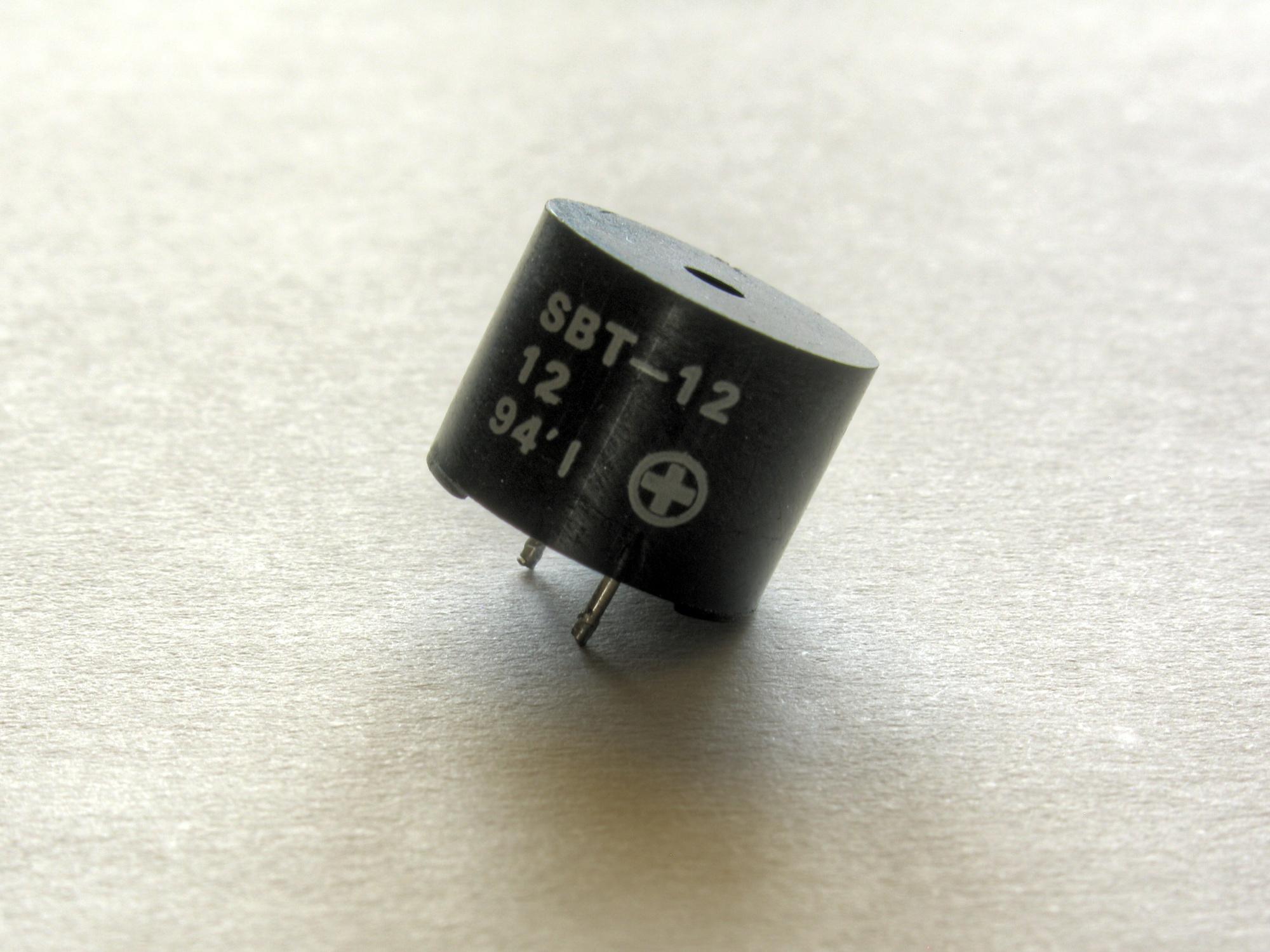
El Servomotor es una de las partes fundamentales para el mini radar, ya que este es la parte mecánica que nos sirve para mover el sensor permitiendo que rote el sensor donde escaneara el área en diferentes ángulos, donde se colocara el sensor ultrasónico para que rote en los ángulos que se establecerán después.



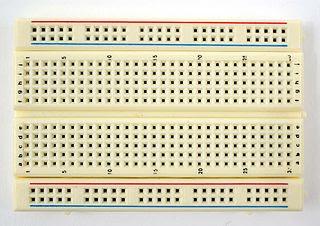
El Sensor ultrasónico, que es la parte más fundamental del proyecto, sin este no tendríamos quien se encargaría de detectar la distancia de los objetos o personas que estén más cercanos en su área.



El buzzer que va a ser la parte donde se accionara el ruido cuando el sensor ultrasónico detecte o capte algún movimiento, haciendo referencia a algún tipo de alarma de proximidad.

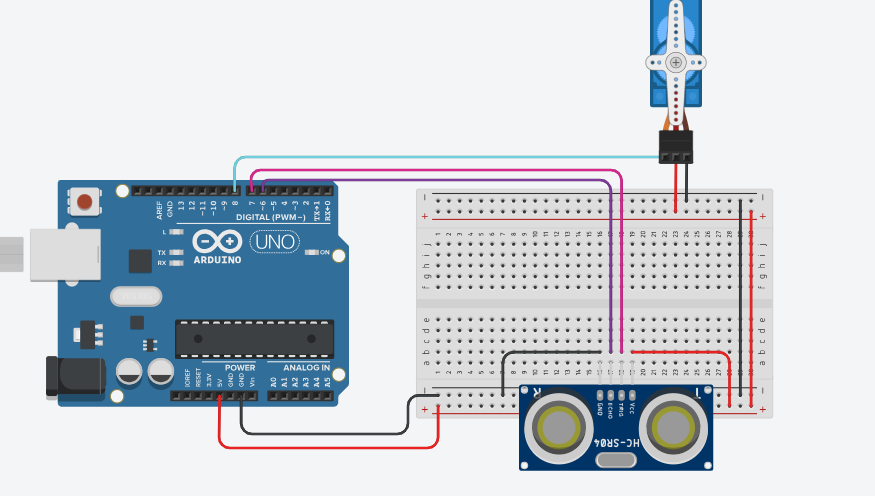


La placa de pruebas, yo creo es necesario destacar la zona de trabajo donde se establecerán todos los componentes que usaremos durante el proyecto del mini radar, donde se conectaran los cables tanto de GND como de potencia del el arduino, los cables del sensor ultrasónico, los del buzzer y del servomotor.

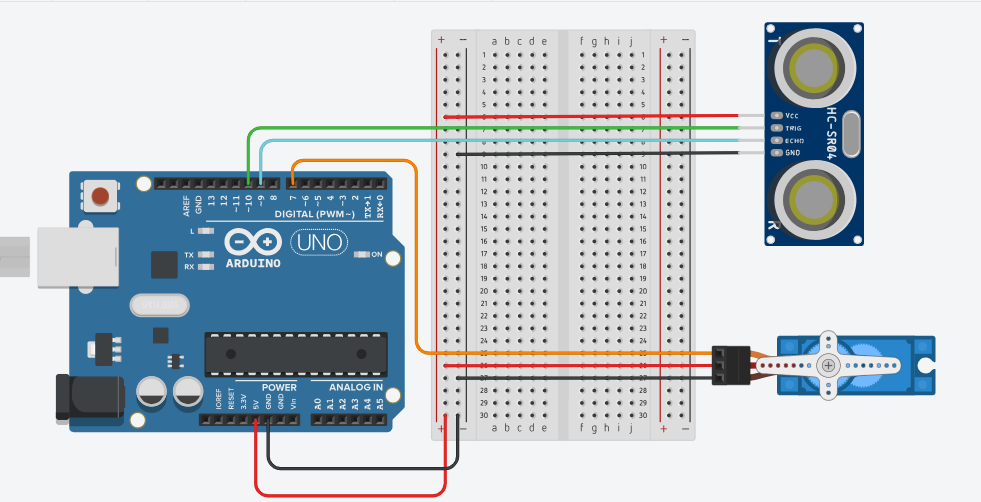


DISEÑO DEL PROYECTO:

Para guiarnos y tener una orientación más clara de como colocar cada componente nos guiamos en ciertas imágenes de internet, usando tinkercad practicamos las conexiones del proyecto tal y como se encuentra en las imágenes.

PRUEBA 1.

Empezamos con algo simple para tener más o menos una idea, pero como este no lograba aparentar una imagen tan clara decidimos cambiar y probar con algo más sofisticado.

PRUEBA 2.

Este fue el diseño que mejor nos atrajo ya que nos pareció más cómodo y la conexión de estos cables está mucho mejor ordenada junto con los componentes que presentan una mejor posición a la hora de visualizarse.

Explicación del diseño:

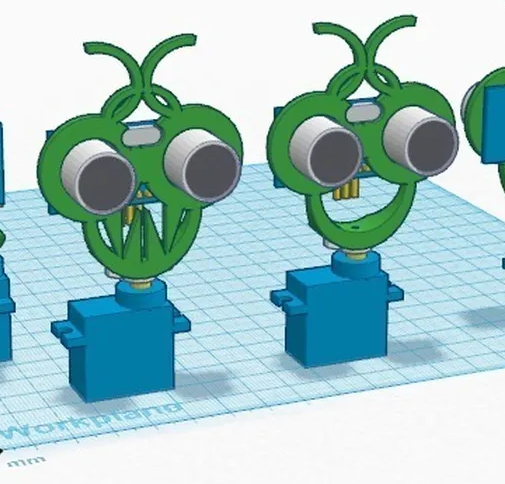
En este proyecto, utilizaremos el pin 10 del Arduino para emitir un pulso de ultrasonidos, mientras que el pin 9 será el encargado de recibirlo. Esta configuración nos permitirá medir distancias mediante el uso de un sensor ultrasónico. Además, para la conexión del servomotor, utilizaremos el pin 7, que será el encargado de controlar sus movimientos. Aunque el sensor ultrasónico y el servomotor se encuentren conectados por separado y en esas posiciones, esto simplemente es un ejemplo.

Teníamos pensado para poder colocar el sensor en el servomotor, crear manualmente algún tipo de soporte que lo mantenga pero como esto nos requerirá mucho tiempo optamos por investigar sobre impresiones 3D lo cual sea lo más acertado.

Impresión 3D de los soportes:

Para lograrlo deberemos de buscar o crear un soporte en una impresión 3d para poder conectar en la parte de arriba del servomotor (quitando la parte plástica y dejando el espacio libre) una estructura que se acople perfectamente al ancho y largo del tamaño de la forma del sensor ultrasónico.

Investigando bastante logramos dar con una página que fue de mucha ayuda para poder encontrar la pieza clave necesaria sobre la estructura del mini radar, esta fue [cults3d.com](https://cults3d.com/). En este se encuentra un modelo el cual lo sostiene desde la parte superior del servomotor enganchado a los tubos que posee con 2 grandes orificios en la misma estructura y así poder engancharse y gestionar la dirección y el eje donde este vaya apuntando.

Los soportes que mejor nos parecieron son los de las siguientes imágen

Cambio de planes.

Debido a la falta de tiempo y a mucha fila en las impresoras 3D de la escuela, al final se terminó optando en vez de crear un soporte físico con una impresora 3D, colocar silicona caliente en la manivela del servomotor y pegar en esa zona el sensor ultrasónico para que éste rote junto con el servomotor. Está demás decir que no usaremos los componentes de la escuela sino que usaremos unos componentes los cuales mi compañero de grupo: Alejandro Alarcon, compró exclusivamente para poder facilitar el trabajo fuera del horario escolar, y así también poder adelantar un poco del proceso del mini radar y a su permitiéndonos lograr la colocación de la silicona caliente ya que estos pertenecen a nuestro grupo.

### Creacion del codigo Arduino

### Es muy importante familiarizarnos con el funcionamiento básico del servomotor para poder entender lo que teníamos que hacer así que buscamos varias guías o fuentes de las cuales unos compañeros compartieron un tp que habían hecho en conjunto para tener una idea de lo que había que hacer. Un repaso rápido para es que este dispositivo tiene tres pines digitales y cada uno con una función específica:

* Tierra: conectado con el cable negro asegura el retorno de la corriente al circuito.
* Potencia: este conectado por el cable rojo, se conecta a una fuente de alimentación adecuada para el servomotor.
* Señal: representado por el cable naranja, este pin es el que recibe las órdenes desde el Arduino para controlar el movimiento del servo.

Para controlar el servomotor desde nuestro código, será necesario también incluir la biblioteca Servo, que facilita nos sirve para su configuración y manejo. Esto se hace con la siguiente línea de código:

| #Include <Servo.h> |
| --- |

Una vez que tenemos la biblioteca incluida, podemos utilizar diversas funciones que nos permitirán interactuar con el servomotor, varias de ella son:

* **Servo.attach(pin)**:  
  Esta función se utiliza para establecer la conexión entre el servomotor y un pin digital del Arduino. Al llamar a Servo.attach() con el número de pin como parámetro, el Arduino queda configurado para manejar el servomotor a través de ese pin específico. Esta conexión es necesaria para que el Arduino pueda enviar señales PWM (Modulación por Ancho de Pulso) al servomotor, permitiendo así su control y movimiento. Sin esta función, no sería posible interactuar con el servomotor desde el código.
* **Servo.write(ángulo)**:  
  Mediante esta función, podemos especificar el ángulo exacto al que deseamos que el servomotor se posicione. El parámetro de entrada es un valor en grados, donde típicamente 0° corresponde a la posición más a la izquierda, 90° al centro, y 180° a la posición más a la derecha, dependiendo de las características del servomotor. Por ejemplo, al usar Servo.write(90), el servomotor se alineará en su punto medio. Esta función es ideal para aplicaciones básicas donde se requiere controlar el movimiento en ángulos predefinidos.
* **Servo.writeMicroseconds(valor)**:  
  Para configuraciones más avanzadas, esta función permite ajustar el ancho del pulso enviado al servomotor en microsegundos. Esto resulta útil cuando se necesita mayor precisión o cuando se trabaja con servomotores no estándar que responden a rangos de pulsos personalizados. Generalmente, los valores van desde aproximadamente 1000 microsegundos (posición mínima) hasta 2000 microsegundos (posición máxima). Ajustar directamente el ancho del pulso brinda un control más fino que trabajar con ángulos predeterminados.
* **Servo.read()**:  
  La función Servo.read() es empleada para consultar el ángulo actual en el que se encuentra el servomotor. Esto es particularmente útil en proyectos donde es importante monitorear o registrar en tiempo real la posición del servo, ya sea para validaciones o ajustes automáticos en el sistema. Por ejemplo, al integrar sensores o mecanismos de retroalimentación, esta función facilita la sincronización entre las acciones del servomotor y otras partes del proyecto.
* **Servo.detach()**:  
  Finalmente, Servo.detach() se utiliza para liberar el control del servomotor. Al llamar a esta función, se interrumpe la comunicación entre el Arduino y el servomotor, dejando de enviar señales PWM al servo. Esto es útil cuando se desea apagar el servomotor para ahorrar energía, detener su movimiento o liberar el pin digital asignado para usarlo en otra tarea. Es importante tener en cuenta que, al desconectar el servo, éste permanecerá en su última posición.

Aunque varias de estas no las hayamos utilizado nos sirven de igual manera para orientarnos con los códigos del servomotor, una vez explicado todo esto definiremos los pines del sensor ultrasónico que tales son:

| const int trigPin = 10; const int echo = 11; |
| --- |

Y antes de configurar el void setup() declararemos la librería con servo myServo para crear el objeto que controle al servomotor, luego dentro del void setup() se encontrará el radar.attach(12) para conectarlo con el arduino, junto con los outputs del buzzer, del trigg y el input del echo, además del serial.begin(9600) que servirá para imprimir todo en pantalla.

Para poder lograr que el servomotor gire hay que crear dentro del void loop varios bucles for como se mostrará en el siguiente bloque de código:

for (int i = 15; i <= 180; i++) {  
 myServo.write(i);   
 delay(30);

}

Lo que hace este bucle for es: primero crear el ángulo en el que se encontrara el servo (la variable entera i), que se establecerá primeramente como 15, haciendo referencia a que si el ángulo en el que se encuentra el mini radar es menor o igual a 180, el ángulo aumentara y se escribirá con la función de myServo.write(i) la posición nueva junto con un delay que le dará más fluidez a la rotación del servomotor que sostendrá al sensor.

Sin embargo necesitábamos calcular la distancia para que este cumpla con la función de mini radar, por ello creamos una función específica que se encargue de medir la distancia.

| int **calculardistancia**() {  digitalWrite(trigPin, LOW);   delayMicroseconds(2);   digitalWrite(trigPin, HIGH);   delayMicroseconds(10);   digitalWrite(trigPin, LOW);   duracion = pulseIn(echoPin, HIGH);   distancia = duracion \* 0.034 / 2;  return distancia; } |
| --- |

Esta función lo que hace es básicamente que con los digitalWrite() de los triggs, se emite un pulso de 10 microsegundos para activar el sensor, lo cual la variable de duración que equivale a la función del pulseIn() del eco de cuando está activado que recibe y mide el tiempo de todas las señales que tardan en regresar calcula la distancia convertida en tiempo usando la velocidad del sonido (0.034 cm/µs). Se divide entre 2 porque la señal recorreria ida y vuelta. Siendo esto lo que se devolverá al final de esta función.

Ya con la función lista podríamos por fin llegar a hacer que el buzzer suena cuando el sensor capte un movimiento u objeto, para ello hay que agregarlo de igual forma al bucle for.

Pero para que esto funcione, osea para que el buzzer emita un ruido cuando se capte algo hay que crear una condición, la cual será: si la distancia captada es menor a 10 cm el buzzer emite un tono más agudo y rápido (haciendo una alusión a una gran cercanía), y si esta en vez de eso capta una cercanía entre 20 cm y 40 cm el tono del buzzer no será tan agudo pero llamará bastante la atención. Esto resultará en la siguiente línea de código:

| if (distancia < 10) {  tone(5, 500, 100);   } else if (distancia > 20 && distancia < 40) {  tone(5, 200, 100);  }  } |
| --- |

Se me olvido mencionar que antes del código del if, la distancia se establecio como equivalente de la función que creamos, osea distancia = **calculardistancia**(). Además también agregamos varios Serial.print() para poder imprimir en pantalla la proximidad de la distancia de lo que capte el sensor.

| Serial.print(i);   Serial.print(",");   Serial.print(distancia);   Serial.print("."); |
| --- |

Sin embargo todavía nos falta otro bucle for mas, ya que el que creamos antes era para que vaya de un lado para el otro, y que nos falta el bucle para que regrese.

El siguiente for es prácticamente igual al anterior, lo único que cambia es la creación de la variable que simboliza el ángulo, intercambiando 15 por 180 y así generando el ida y vuelta del servomotor. for (int i = 180; i > 15; i--)

Una vez con el código del arduino ya terminado proseguiremos con la parte de codificacion del interfaz gráfico para el mini radar.

Preparación del programa:

Comenzaremos con la instalación del dicho programa yendo a la página oficial de [Processing](https://processing.org/download), en este seleccionaremos la parte donde dice “Descargar Processing 4.3 para Windows”, se instalará un archivo .zip donde eventualmente tendremos que descomprimir el archivo para continuar con la descarga del programa.

Código Processing

Arduino processing es un lenguaje de programación y entorno de desarrollo integrado de código abierto basado en JAVA, está hecho para diseñadores quienes crean proyectos audiovisuales o multimedia, ya que se pueden realizar gráficos en el programa en formato 2D, 3D, así también como texturas y etc.

Una vez ya instalado el programa, empezaremos con la investigación para poder realizar el código de la interfaz gráfica del radar. Para ello busqué en google junto con mis compañeros ciertos videos o páginas que parecieran de utilidad, de los cuales encontré varias fuentes que fueron de gran ayuda para poder aprender los códigos y funciones básicas necesarias que usaremos en Processing.

Un video esencial fue este: <https://www.youtube.com/watch?v=N8qPvS2YTec>

En este video te explica para lo que sirve processing, que es básicamente un programa para crear interfaces gráficas, y en esta se usan ciertas funciones como por ejemplo: size(), que es la función en la que se establecen los parámetros para darle un ancho y largo a lo que vendría ser la pantalla, a su vez también explica la función de background(), que esta sirve para darle un color al fondo de pantalla y este se puede establecer de dos formas distintas, con solo un parámetro numérico donde el programa lo interpretara para calificarlo en una escala de grises ( 0 = negro y 255 = blanco), y con tres parámetros numéricos distintos que su rango limite es de 0 a 255 cada uno para referenciar a las escalas del RGB.

También encontré una página que fue realmente de mucha ayuda para poder aprender más de lo necesario y entrar mejor en el terreno de este lenguaje.

Esta pagina es: <https://learn.sparkfun.com/tutorials/connecting-arduino-to-processing/all>

Aquí se explican varias funciones y cómo empezar con este.

Primero que nada hay que importar la librería serial, es una herramienta que permite la comunicación entre dispositivos a través de un puerto serial en el que se transmiten y reciben datos, para importar esta librería hay que ir en el programa a la ventanita que dice sketch, luego importar biblioteca y después serial, de esta manera aparecerá un código que señalara que ya está incluida de esta manera: import processing.serial.\*;

También me olvidé de mencionar que en vez de un void loop en Arduino Processing se encuentra void draw que es bastante similar ya que es un bucle infinito del mismo tipo simplemente que se utiliza para dibujar las interfaces que nosotros deseemos.

También mis compañeros buscaron en google ciertas fuentes para el mini radar, cabe aclarar que estas simplemente nos sirvieron meramente como orientación al código que deseábamos crear en processing ya que no había ningún tipo de indicio para empezar.

Para poder escuchar cualquier comunicación serial, tenemos que conseguir un objeto Serial, que nos permita escuchar en un puerto serial de nuestra computadora cualquier dato entrante, para esto declararemos una variable global que será Serial myPort que gestionará toda la comunicación con el arduino, todo esto nos serviría para empezar con la parte principal del codigo, donde también se declaran las variables principales que se usarán para establecer los ángulos, índices, etc. Se declararon de la siguiente manera:

| String angulo = ""; String distancia = ""; String datos = ""; String sinObjeto; float pixelesDistancia; int iAngulo, iDistancia; int indice1 = 0; int indice2 = 0; |
| --- |

Primero crearemos unas variables globales de datos string que se usarán más tarde durante el código, estas son las de ángulo, distancia, datos y el sinObjeto que se usarán en las funciones que después crearemos. Junto con las variables de datos numéricas como int y float, que por ejemplo float pixelesDistancia se usará para medir todos los pixeles de distancia que detectara el sensor en la función en la que dibujaremos más tarde donde también se usará para hacer los cálculos matemáticos del mini radar, a su vez están int iAngulo y int iDistancia que ambas variables se complementaran en las funciones que se usarán para poder instanciar las condiciones donde tendremos que medir la distancia y ángulo para poder dibujar las líneas, el texto, el radar, etc.

Además tenemos los índices que se usarán para procesar y dividir los datos que se reciben a través del puerto serie desde Arduino, que son cadenas de texto en formato específico.

llegando a la parte de configuración del setup, aquí tendremos que establecer el tamaño de la pantalla que lo haremos relativamente grande, size(1200, 700); . Así como también se agregó smooth() para activar el suavizado de las líneas , figuras y textos que se dibujaran en la pantalla. Esto sirve más que nada para que las curvas y líneas que se dibujen se vean más suaves y menos "dentadas" o "pixeladas".

También agregaremos 2 lineas de codigo mas en el void setup:

| myPort = new Serial(this, "COM3", 9600); myPort .bufferUntil('.'); |
| --- |

En la primera línea de código aclaramos para que new Serial(this, "COM3", 9600) ; Haga referencia al objeto actual, en este caso, la instancia de Processing. Este es un parámetro requerido para indicar el contexto en el que se está utilizando la comunicación serie. junto con "COM3" que especifica el puerto de serie al que está conectado el dispositivo. "COM3" es el puerto de Windows que usaremos, 9600 será la velocidad de comunicación (en baudios). Este es la tasa más comúnmente utilizada para la comunicación entre Arduino y la computadora, Y ambos, el dispositivo y la aplicación que lee el puerto serie (en este caso, Processing), tienen que usar la misma tasa de baudios para que la comunicación funcione correctamente.

Utilizamos myPort.bufferUntil('.'); para configurar el puerto serie de manera que el programa almacene todos los datos entrantes hasta que encuentre el carácter especificado, en este caso, el punto. Esto es útil para delimitar la recepción de datos en situaciones donde se envían múltiples valores o mensajes a través del puerto serie.

Una vez configurado el setup del radar construiremos nuestro codigo basado en funciones de dibujo para poder hacer el radar, pero para ello primero habra que hacer que en el void draw la ventana se dibuje en cada fotograma con un color negro casi transparente, creando un efecto visual de desvanecimiento. Esto servirá para que las trazas antiguas del radar se "diluyan" gradualmente, simulando el barrido continuo de un radar real. Como se mostrará en el siguiente código:

| void **draw**() {  fill(98, 245, 31);  noStroke();  fill(0, 4);  rect(0, 0, width, height - height \* 0.065);  fill(98, 245, 31); } |
| --- |

Lo que hace la función de fill() es llenar toda la ventana de un verde fuerte (98,245,31), y junto con la función de noStroke() desactiva los bordes o contornos en las figuras para que se dibujen después de esta línea.

Aunque hayamos llenado toda la pantalla de verde, de igual manera tenemos que hacer otro fill mas pero este para que deje el efecto de desvanecimiento fill(0,4) contiene 2 parámetros: el primer parametro para el color, en este caso 0 que es negro y el segundo parametro para la opacidad de este, siendo de 0 totalmente transparente a 255 totalmente opaco, gracias a que ponemos en 4 es por que se da el efecto.

La siguiente función rect() es básicamente para hacer un rectángulo que cubra toda la ventana junto con el fill que ayude a cambiar de nuevo el color de relleno a verde brillante para los elementos que se dibujen después, esto nos asegura que las líneas y objetos en el radar se muestran claramente sobre el fondo translúcido.

ya con el void draw de processing preparado, simplemente nos quedaría empezar a crear las funciones con las que dibujaremos cada parte en el mini radar, ya sean el mismo radar, las líneas de este junto con los textos y los objetos que usaremos.

pero antes también crearemos un procedimiento más para que pueda interpretar todos los datos del sensor y así poder usarlos en nuestras variables globales que anteriormente declaramos, la idea la sacamos de internet y buscamos alguna fuente que nos explicara masomenos para que era asi que encontre esta fuente de processing: <https://processing.org/reference/libraries/serial/Serial_serialEvent_.html>

y con esto nos dimos masomenos una idea mientras que buscábamos guías de las cosas que teníamos que poner, mientras tanto hicimos esto.

| void **serialEvent**(Serial myPort) {  datos = myPort.readStringUntil('.');   datos = datos.substring(0, datos.length() + 1);   indice1 = datos.indexOf(",");   angulo = datos.substring(0, indice1);   distancia = datos.substring(indice1 + 1, datos.length());  iAngulo = int(angulo);   iDistancia = int(distancia);  } |
| --- |

El primer parámetro que se utiliza en este procedimiento es la variable global serial que gestiona todo los datos, por esto es que en la variable datos se utiliza myPort para leerlos hasta que se encuentre la condición especificada que es el punto, de igual manera se crea otra variable de datos más con la función de substring() que es utilizada para extraer una porción de una cadena de texto, esto se hace especificando un rango de índices de inicio y fin, como está establecido en el código, junto con la función length() para medir la cantidad de caracteres de los datos.

Para crear los índices simplemente usamos la función indexOf(",") para que saque los índices de los datos, que lo usamos para separar los valores del ángulo y la distancia, para sacar los angulos usamos la función substring() que anteriormente implementamos para extraer cadenas de textos con los datos pero esta vez con el índice que ya establecimos para poder sacar el angulo estableciendo antes de la coma en los parámetros.

Para la distancia es casi lo mismo solo que los valores se intercambian, osea lo que se hace en los parámetros del substring es para que tome el valor después de la coma.

Seguido de las variables de los valores de los ángulos y de las distancias que se establecen después de ya haber sacado todos los cálculos y códigos de estos.

Una vez finalizado el void serialEvent() ya podemos darnos el lujo de investigar de cómo sería poder comenzar a hacer todas las funciones que requerimos. Con bastante investigación que di de mi parte, pude llegar a entender que necesitariamos más de una función para nuestro código del mini radar.

Esto recae en que tendríamos que aprender a usar más funciones, de las cuales vi que son:

* translate(): La función [translate()](https://processing.org/reference/translate_.html) sirve para cambiar el origen del sistema de coordenadas, desplazándose de una posición a otra nueva posición. Por defecto, el origen (0, 0) está ubicado en la esquina superior izquierda de la ventana, pero con esta función podes moverlo a cualquier punto especificado en las coordenadas(x,y).
* arc(): La función [arc()](https://processing.org/reference/arc_.html) se utiliza para dibujar un segmento de un círculo o una elipse, definido por su posición, dimensiones y ángulos de inicio y final. Este sirve más que nada para crear formas parciales de círculos o curvas en aplicaciones gráficas, como medidores, gráficos o indicadores.
* pushMatrix(): La función [pushMatrix()](https://contribs.processing.org/reference/pushMatrix.html) sirve para guardar el estado actual del sistema de coordenadas en una pila (stack). Esto permite realizar transformaciones temporales (como traslaciones, rotaciones y escalados) sin afectar permanentemente el resto de los elementos en el trabajo. Cuando llamas a pushMatrix(), estás haciendo checkpoint en el que se guarda la posición, orientación y escala actuales del sistema de coordenadas. Seguido de eso, podes aplicar transformaciones que sólo afectan al contenido dibujado después de la llamada a esta función. Al usar popMatrix(), se restaurará el estado anterior guardado por pushMatrix()
* stroke(): La función [stroke()](https://processing.org/reference/stroke_.html) sirve para establecer el color de las líneas y los bordes de los objetos, formas o cosas que se dibujan. Este color se especifica en los valores RGB, en 3 parámetros distintos, también a su vez incluye opcionalmente un canal alfa para transparencia.
* line(): La función [line()](https://processing.org/reference/line_.html) en Processing se utiliza para dibujar una línea recta entre dos puntos en la ventana gráfica. Su sintaxis es: line(x1, y1, x2, y2), Esta función sirve bastante para las creaciones de conexiones visuales entre dos puntos, estructurar figuras geométricas más complejas o trazar trayectorias y referencias en la pantalla.

Con todas estas funciones es mucho más que necesario para poder armar lo que sea, por eso nos vamos a dar prisa con arrancar a tener todo listo con el mini radar. Primero que nada crearemos la función que genere todo el radar, empezaremos llamándola void **drawRadar**() y esta tendrá adentro como primer objeto un pushMatrix() para que las transformaciones aplicadas dentro de esta función (como un translate o un rotate) no afecten el resto del dibujo, por consiguiente se metera un translate que se mostrara a continuación:

| translate(width/2, height - height \* 0.074); |
| --- |

Esta línea especifica el traslado del origen del sistema de coordenadas al centro inferior de la ventana (ancho/2, cerca de la parte inferior). Esto facilita que el radar sea dibujado desde ese punto como base, y para que no se dibujen formas sólidas durante el proceso se implementara la función noFill(), para establecer el grosor de las líneas se utilizó strokeWeight(2), osea a 2 píxeles, más otro stroke(98, 245, 31) que definirá el color verde fuerte del que habíamos hablado antes para el trazo de las líneas. Ya teniendo los colores y parámetros establecidos tenemos que dibujar los arcos donde se simulará el radar, para ello usaremos la función arc() de esta manera:

| arc(0,0,(width-width\*0.0625),(width-width\*0.0625),PI,TWO\_PI); arc(0,0,(width-width\*0.27),(width-width\*0.27),PI,TWO\_PI); arc(0,0,(width-width\*0.479),(width-width\*0.479),PI,TWO\_PI); arc(0,0,(width-width\*0.687),(width-width\*0.687),PI,TWO\_PI); |
| --- |

Cada línea de código dibuja un arco en la mitad inferior del círculo, desde 180° (PI) hasta 360° (TWO\_PI). Las coordenadas (0, 0) hacen referencia al centro del arco (trasladado previamente con la función de translate). Los valores de width junto con los cálculos matemáticos determinan el diámetro de cada arco, decreciendo gradualmente para representar los anillos del radar.

Ya tenemos los arcos, qué nos falta? las líneas del radar, para esto usaremos la función line() que nos hará la mayor parte del trabajo. Primero, tenemos que hacer la línea horizontal en el medio del radar, desde el borde derecho hasta el borde izquierdo, como se mostrará en la siguiente línea de código:

| line(-width/2, 0, width/2, 0); |
| --- |

Con esto ya hecho, simplemente quedan las líneas en diferentes ángulos los cuales estarán colocados cada 30 grados de distancia cada uno.

| line(0,0,(-width/2)\*cos(radians(30)),(-width/2)\*sin(radians(30))); line(0,0,(-width/2)\*cos(radians(60)),(-width/2)\*sin(radians(60))); line(0,0,(-width/2)\*cos(radians(90)),(-width/2)\*sin(radians(90))); line(0,0,(-width/2)\*cos(radians(120)),(-width/2)\*sin(radians(120))); line(0,0,(-width/2)\*cos(radians(150)),(-width/2)\*sin(radians(150))); |
| --- |

Cada línea parte del centro (0, 0) hacia un punto determinado por las funciones trigonométricas cos() y sin(). A su vez, radians(ángulo) convierte los ángulos (en grados) a radianes necesarios para cos() y sin(), (-width/2) determina la longitud de las líneas, que le corresponden al radio del radar. Como última línea que me faltó aclarar, queda una línea más desde el ángulo de 30° hacia el borde derecho, para dar un refuerzo visual en el radar. line((-width/2) \* cos(radians(30)), 0, width/2, 0);.

Con todas las líneas ya finalizadas solo queda pedirle a la función que restaure el estado original del sistema de coordenadas, y deshaciendo las transformaciones aplicadas dentro de esta función, para esto usaremos la increíble función llamada popMatrix() que realizará y dará por finalizada toda esta función.

Con nuestra primera función ya creada, estamos más cerca de finalizar con el radar, aunque aún nos quedan bastantes cosas por hacer. Solo tenemos la función que dibuja al radar, ahora debemos de poder crear la función que detecte al objeto y lo dibuje en el radar, para ello empezaremos llamándola void **drawObject**() y como en todas las funciones que utilizaremos esta tendrá como primer paso un pushMatrix(), lo siguiente será trasladar el origen del sistema de coordenadas al punto con la siguiente línea de código:

| translate(width/2, height - height \* 0.074) |
| --- |

Siendo esto correspondiente al centro inferior de la ventana gráfica, todo esto para facilitar a que el dibujo del objeto esté alineado con el radar previamente dibujado. Continuando con dos funciones más, strokeWeight(9) para determinar el grosor en 9 píxeles y

stroke(255, 10, 10) para definirlo al objeto de color rojo para hacer referencia a que se detecto debido al contraste entre el verde y el rojo. Ahora necesitaremos convertir la distancia detectada por el sensor a píxeles, que son las unidades utilizadas en la ventana gráfica. Lo haremos declarando una variable nueva que será la que equivalga a estas unidades

| pixsDistance = iDistance \* ((height - height \* 0.1666) \* 0.025); |
| --- |

iDistance es la distancia detectada en cm por el sensor y (height - height \* 0.1666) calcula un rango proporcional basado en el tamaño de la ventana gráfica junto con un factor de escala que es (\* 0,025) que sirve para ajustar la conversión de centímetros a píxeles.

Una vez que tengamos esto, crearemos la condición que limite la visualización del objeto a una distancia menor a 40 cm. Si el objeto está fuera de este rango, no se dibuja.

| if(iDistance<40){ line(pixsDistance\*cos(radians(iAngle)),-pixsDistance\*sin(radians(iAngle)),(width-width\*0.505)\*cos(radians(iAngle)),-(width-width\*0.505)\*sin(radians(iAngle))); } |
| --- |

Puede que no se entienda mucho pero básicamente esta condición hace que cree la línea roja con la función line() gracias a la variable pixsDistance que ya antes declaramos si esta es menor a 40 cm, y como siempre un popMatrix() al final para dejarlo restaurado al estado anterior guardado por pushMatrix().

Ya casi estamos, tenemos las funciones para dibujar el radar y para dibujar el objeto detectado, nos toca hacer ahora la función que dibuje las líneas en el radar asi que para empezar ya con esto declararemos la nueva función void **drawLine**().

Como en todas las funciones que hicimos hasta ahora empezaremos colocando un pushMatrix(), seguido de un strokeWeight(9) del mismo grosor de pixeles que de las líneas del objeto detectado, junto con un stroke(30, 250, 60) que define nuevamente el color verde fuerte y brillante del radar. Con esto ya instanciado pasamos a hacer el traslado de las líneas como se ve en el siguiente código:

| translate(width/2, height - height \* 0.074) |
| --- |

Trasladando el origen del sistema de coordenadas al centro inferior de la ventana gráfica, alineándose con el radar. Ahora pasaremos a dibujar las líneas que parten del origen (0,0) hacia el punto determinado por el ángulo detectado (iAngle):

| line(0, 0, (height - height \* 0.12) \* cos(radians(iAngle)), -(height - height \* 0.12) \* sin(radians(iAngle)));  Los cálculos Cos(radians(iAngle)) y sin(radians(iAngle)) convierten el ángulo de grados a coordenadas polares, (height - height \* 0.12) es el radio (o longitud de la línea), y el signo negativo en sin() invierte la coordenada “y” para que los ángulos sean congruentes con el sistema de coordenadas gráficas. Finalizando una vez más con la función popMatrix() para tener terminada el void **drawLine**().  Ahora nos falta una función que nos escriba los textos que veremos en el radar, como por ejemplo, el estado del objeto captado ("detectado" o vacío si no hay detección), los valores actuales del ángulo y la distancia, junto con sus referencias de ángulos y distancias en el radar.  Empezaremos declarandola void **drawText**() y en ella meteremos una vez más pushMatrix() para que guarde el estado actual del sistema de coordenadas, y empezaremos diciendo que si el objeto está a más de 40 cm el radar dirá que está fuera de rango, o sí si lo está dira en rango. Para esto instanciamos un if que que condicione lo siguiente:  if(iDistance > 40){  noObject = "Out of Range";  }  else{  noObject = "In Range";  }  Ahora estableceremos el color de relleno del texto a negro fill(0,0,0) lo cual se usa para dibujar un fondo debajo de los textos, y añadiremos un noStroke() para que elimine el borde alrededor de las figuras para que sólo se dibuje el relleno. Necesitaremos tambien dibujar un rectángulo negro en la parte inferior de la pantalla, para usarlo como fondo para los textos que se mostrarán, la altura del rectángulo se ajusta en proporción a la resolución de la pantalla de la siguiente manera:  rect(0, height - height \* 0.0648, width, height);  Una vez hecho eso estableceremos el color del texto a verde claro con  fill(98, 245, 31) y el tamaño del texto con textSize(25), ya establecido esto pasaremos a dibujar los textos "10cm", "20cm", "30cm" y "40cm" en posiciones específicas en la parte inferior de la pantalla. Estos textos indicaran las distancias en centímetros en relación con las marcas del radar, y se mostrarán con el siguiente código:  text("10cm", width - width \* 0.3854, height - height \* 0.0833);  text("20cm", width - width \* 0.281, height - height \* 0.0833);  text("30cm", width - width \* 0.177, height - height \* 0.0833);  text("40cm", width - width \* 0.0729, height - height \* 0.0833);  Ya habiendo conseguido marcar los textos de la proximidad pasaremos a marcar los del ángulo, objeto y distancia, junto con un nuevo tamaño de texto para que sea más prominente.  textSize(40); text("objeto:" + noObject, width - width \* 0.875, height - height \* 0.0277); text("ángulo:" + iAngulo + " °", width - width \* 0.48, height - height \* 0.0277); text("distancia: " + iDistancia + " cm", width - width \* 0.26, height - height \* 0.0277);  Después de esto restablecemos el tamaño con textSize(25) a 25 píxeles y cambiamos el color del texto a un verde un poco más claro y brillante con fill(98, 245, 60),luego colocamos los textos posicionandolos a lo largo de un círculo para el radar con la función de translate() que la repetiremos 5 veces pero con distintas operaciones para poder poner el texto de cada ángulo en cada parte del radar:  bloque de código del texto para el ángulo 30:  translate((width - width \* 0.4994) + width / 2 \* cos(radians(30)), (height - height \* 0.0907) - width / 2 \* sin(radians(30))); rotate(-radians(-60)); text("30°", 0, 0); resetMatrix();  bloque de código del texto para el ángulo 60:  translate((width - width \* 0.503) + width / 2 \* cos(radians(60)), (height - height \* 0.0888) - width / 2 \* sin(radians(60))); rotate(-radians(-30)); text("60°", 0, 0); resetMatrix();  bloque de código del texto para el ángulo 90:  translate((width - width \* 0.507) + width / 2 \* cos(radians(90)), (height - height \* 0.0833) - width / 2 \* sin(radians(90))); rotate(radians(0)); text("90°", 0, 0); resetMatrix();  bloque de código del texto para el ángulo 120:  translate(width - width \* 0.513 + width / 2 \* cos(radians(120)), (height - height \* 0.07129) - width / 2 \* sin(radians(120))); rotate(radians(-30)); text("120°", 0, 0); resetMatrix();  bloque de código del texto para el ángulo 150:  translate((width - width \* 0.5104) + width / 2 \* cos(radians(150)), (height - height \* 0.0574) - width / 2 \* sin(radians(150))); rotate(radians(-60)); text("150°", 0, 0);  Ya con todo esto terminado, colocamos el legendario popMatrix() al final de la función y de esta manera concluimos con el void **drawText**().  Con todas las funciones ya creadas el código se supone que tendria que funcionar, por lo tanto las colocaremos en el void **draw**() para probarlo, y quedaría de esta manera:  void **draw**() { fill(98,245,31); noStroke(); fill(0,4); rect(0, 0, width, height-height\*0.065); fill(98,245,31); // green color drawRadar(); drawLine(); drawObject(); drawText(); }  Con el código en mano y con los componentes de Alarcon, decidimos probarlo en clase para ver qué onda y si funcionaba. Tristemente el código no andaba aunque era de esperar, estuvimos un rato viendo que habíamos puesto mal, pero los cálculos que habíamos hecho en las funciones estaban bien, buscamos en las variables globales si habia algun tipo de incongruencia pero no había nada raro, hasta que chequeamos en el void serialEvent(), teníamos mal expresado uno de los índices de nuestros datos, por lo cual provocaba que el programa intente acceder a un índice que está fuera de los límites de la cadena. Básicamente el error estaba en esta línea de código:  datos = datos.substring(0, datos.length() + 1);  Esto provocaba un error porque datos.length() + 1 supera el índice máximo de la cadena lo cual estaba mal porque había que eliminar el último carácter de la cadena de texto, de esta forma evitabamos acceder a índices fuera de los límites dado que el programa está diseñado para recibir datos en tiempo real desde un puerto serie, cualquier interrupción en el procesamiento que se genere de estos datos podía afectar a la funcionalidad general del sistema. Pero una vez corregido esto con el siguiente código:  datos = datos.substring(0, datos.length() - 1);    El programa no emitirá más errores por lo cual el proyecto del mini radar estaría completado y listo para el funcionamiento pleno de este.  Adjunto una foto de nuestro proyecto finalizado que sacó nuestra compañera durante el proceso del mismo. |
| --- |