

# Usando seno y coseno

## Créditos

- » **Autor:** Amarillion
- » **Traductor:** Carlos Gabriel Valentin
- » **Fecha:** 15 de Febrero del 2006

## Introducción

En este artículo discutiremos algunas técnicas de programación de juegos, todas en torno a un tema central: las funciones seno y coseno.

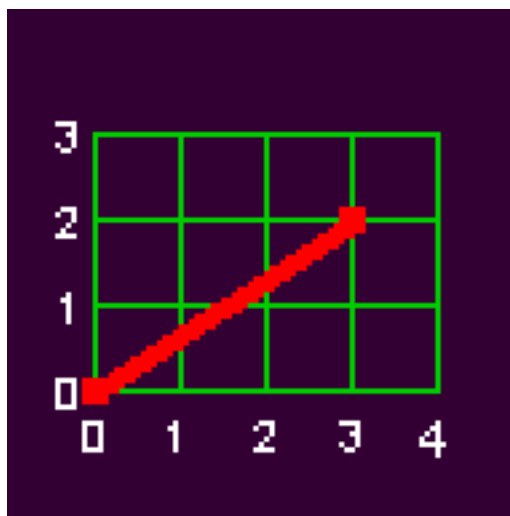
Se explicará el concepto de seno, coseno, vectores, tangentes y algunos efectos especiales. Veremos como hacer misiles direccionales y como trabajan las rotaciones de bitmap.

Comenzaré con lo más básico, pero luego cubriré técnicas de programación de juegos un poco más avanzadas. Puede descargar todos los códigos de ejemplo (12 en total) desde [aquí](#). Todos se verificaron con anterioridad utilizando el compilador DJGPP. Si usted tiene DJGPP, puede descomprimir los fuentes y el makefile en un directorio y correr "make".

## Vectores

Empecemos con algo que a veces es difícil de entender para los que recién comienzan, ya que es altamente abstracto, el vector. Este se puede visualizar de diferentes maneras.

Primero podemos imaginarnos que es una flecha a un punto en el espacio. En el caso de que sea en 2 dimensiones, necesitamos dos valores para definir el vector. Uno para las coordenada X y otro para la coordenada Y. En el caso de que estemos usando la tercera dimensión, necesitaremos un tercer valor para la coordenada Z. Aunque en este artículo solo trataremos la segunda dimensión.



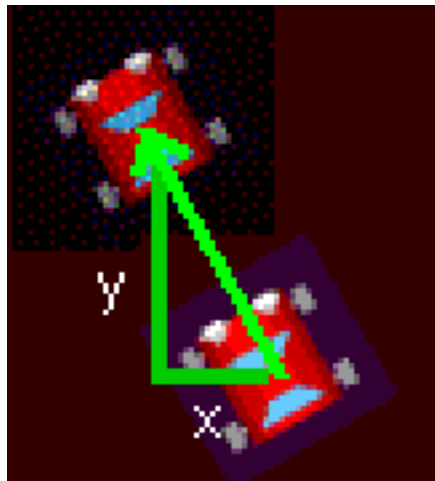
Representación de un vector

En la figura de arriba hay un vector dibujado con una coordenada X de valor 3 y una coordenada Y de valor 2. Pero estos dos valores no son el final de la historia. Por ejemplo si dibujamos este vector en papel, podemos medir su largo y obtener el ángulo que define con eje de coordenada X. Su largo será de 3.6 cm de largo y el ángulo será de 34 grados.

Si pensamos profundamente, podemos ver que no necesitamos el par (X,Y) del vector si ya conocemos su largo y el ángulo que genera con el X de coordenada. Es perfectamente posible definir un vector por su largo y su ángulo.

Con X e Y estamos usando coordenadas cartesianas. Si usamos el largo y el ángulo del vector estamos usando coordenadas polares.

Veamos un ejemplo. Supongamos que estamos escribiendo un juego de autos de carrera al estilo Top-down (algo así como Micro Machines). Necesitaremos alguna manera de guardar la velocidad del auto. ¿Como haremos eso? con un vector. Este vector velocidad es en realidad el cambio de posición del auto de un cuadro a otro. La pregunta es ¿deberíamos usar coordenadas cartesianas o coordenadas polares para este vector?.



Vector velocidad de un auto

Bueno, guardando solo las coordenadas cartesianas tiene la ventaja que es bastante fácil calcular la nueva posición del auto en cada instante. Supongamos que guardamos las coordenadas (X,Y) del vector velocidad en las variables 'vel\_x' y 'vel\_y', y la posición del coche en las variables 'pos\_x' y 'pos\_y'. Todo lo que tenemos que hacer en el bucle del juego es:

```
pos_x += vel_x;
pos_y += vel_y;
```

Por otro lado, guardando el ángulo y el largo del vector velocidad tiene la ventaja que hace más fácil el control del auto. Pensemos, si el jugador aprieta LEFT, quieres que el auto doble a la izquierda. Suponiendo que guardamos el ángulo en el entero 'car\_angle', podemos hacer uso del siguiente código:

```
if (key[KEY_LEFT])
{
    car_angle -= 1; // dobla un grado hacia la izquierda
}
if (key[KEY_RIGHT])
```

```
{  
  car_angle += 1; // dobla un grado hacia la derecha  
}
```

¿Como haríamos esto guardando el valor X e Y? tendríamos que cambiar las 2 variables y eso sería un poco más difícil que la solución anterior. Además si apretamos UP deseñaríamos que el auto vaya más rápido, Fácilmente podríamos solucionar esto incrementando el vector del auto.

## Seno y coseno

Ahora ya sabemos que hay 2 formas de guardar un vector (a travez de coordenadas polares y cartesianas) y que en este caso las 2 tienen sus ventajas. Entonces ¿Cual usamos? Bueno, no tendría que ser un problema si sabemos como calcular el ángulo y la velocidad desde la coordenada X y la coordenada Y o viceversa.

Primero hablaré de como convertir desde coordenadas polares a cartesianas. Por supuesto es posible la conversión en el otro sentido, pero hablaré de eso luego. Hay 2 funciones para realizar esto. Estas funciones son el seno y coseno. El seno se puede usar para calcular la coordenada Y del vector, y el coseno se puede usar para calcular la coordenada X. Las funciones seno (sin) y coseno (cos) solo admiten un parámetro: el ángulo. Ellas retornan números entre -1 y 1, si multiplicamos este numero por el largo del vector, obtendremos las coordenadas cartesianas exactas, del vector. Entonces nuestro código se verá así:

```
speed_x = speed_length * cos (speed_angle);  
speed_y = speed_length * sin (speed_angle);
```

Eso es todo, para un juego de carrera solo guardamos el ángulo y el largo del vector velocidad. Ajustamos esto en función de la entrada que nos de el jugador y calculemos las coordenadas cuando estemos listos para actualizar la posición del auto.

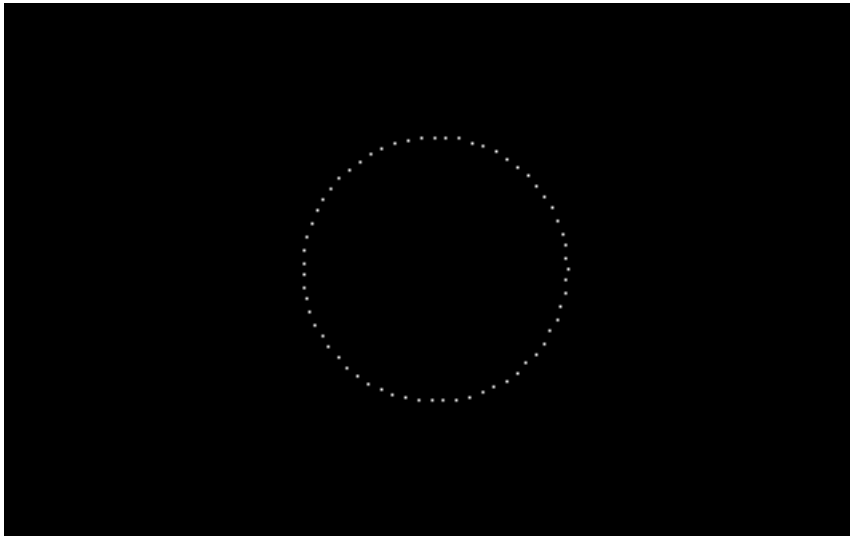
## dibujando un circulo

Daré un ejemplo simple de lo que hace sin y cos. En realidad, este es probablemente el programa más simple usando sin y cos que veremos. (ver circ1.c)

```
[...]  
void draw_circle ()  
{  
  int x, y;  
  int length = 50;  
  float angle = 0.0;  
  float angle_stepsize = 0.1;  
  
  // go through all angles from 0 to 2 * PI radians  
  while (angle < 2 * PI)  
  {  
    // calculate x, y from a vector with known length and angle  
    x = length * cos (angle);  
    y = length * sin (angle);  
  }  
}
```

```
    putpixel (screen,  
             x + SCREEN_W / 2, y + SCREEN_H / 2,  
             makecol (255, 255, 255));  
    angle += angle_stepsize;  
  }  
}  
[...]
```

Al ejecutar el programa verá:



Captura del programa funcionando

Lo que hace esta función es dibujar 60 puntos distanciados igualmente entre sí en la pantalla, que juntos forman un círculo perfecto. Como verá hay una variable llamada 'length' y otra llamada 'angle'. Estas dos representan el largo y el ángulo de un vector, respectivamente.

Primero calculamos la coordenada X y la coordenada Y, usando sin y cos. Después trazamos un pixel en las coordenadas X e Y calculada. Por último incrementamos el ángulo del vector un poco, pero no cambiamos el largo. Lo iteramos varias veces, desplazándonos a través de diferentes ángulos. Si dibujamos un punto a una distancia constante desde un punto fijo en diferentes direcciones, obtendremos un círculo.

### acerca de radianes

Primero veremos por qué la expresión dentro del ciclo while dice `angle < 2 * PI`. Y Segundo, por qué aunque 'angle\_increment' es un valor tan bajo los puntos no están cerca unos de otros.

La respuesta a lo primero es que sin y cos no toman grados como argumento. Hay  $2 * \text{PI}$  radianes en un círculo,  $\text{PI}$  siendo una constante matemática alrededor de 3,1415927. Entonces hay más o menos 6,282 radianes en un círculo. El siguiente código nos permitirá poder calcular el número de grados con el número de radianes y viceversa.

```
degrees = radians * 180 / PI;  
radians = degrees * PI / 180;
```

Consideremos que el incremento del ángulo es de 0,1 radianes.  $0,1 \text{ radianes} = 0,1 * 180 / 3.142 = 5,7 \text{ grados}$ . Si vemos la salida, del círculo dibujado anteriormente, notaremos que son los valores más o menos dibujados. Verdaderamente la razón de introducir radianes de esta manera es la siguiente, el largo de la circunferencia es exactamente  $2 * \text{PI}$ . Esto significa que el largo de la circunferencia es igual al número de radianes en un círculo completo.

### usar números fijos, no float

En computadoras nuevas, no hay mucha diferencia, pero en computadoras viejas la ganancia de velocidad usando números de puntos fijos es significativa frente al uso de floats. Aquí les mostraré la función `draw_circle`, pero ahora solo usando aritmética de números de puntos fijos.

Primero veremos una introducción al tema. Note que si usa C++ puede hacer uso de la clase "fix", la cual le aliviará un poco el trabajo, aunque no la explicaré aquí. Si programas en C++ y quieres usar la clase "fix", tendrás que buscarla en la documentación de `allegro`.

Regla #1: Podemos convertir de un float a un número de punto fijo o de int a un número de punto fijo, con las funciones `fixtoi`, `fixtof`, `itofix` y `ftofix`.

```
fixed_1 = itofix (int_1);
int_1 = fixtoi (fixed_1);
float_1 = fixtof (fixed_1);
```

Regla #2: podemos sumar y restar 2 números de puntos fijos, pero no un int y un número de punto fijo. Necesitaras convertir el int a un número de punto fijo.

```
fixed_3 = fixed_1 + fixed_2;
fixed_3 = fixed_1 - fixed_2;
fixed_3 = fixed_1 + itofix (int_2);
```

Regla #3: Podemos dividir y multiplicar por un int, pero no por otro número de punto fijo. En ese caso necesitaras usar las funciones `fmul ()` y `fdiv ()`.

```
fixed_3 = fixed_1 * int_2;
fixed_3 = fmul (fixed_1, fixed_2);
fixed_3 = fdiv (fixed_1, fixed_2);
```

Aquí les presento la nueva versión de la función `draw_circle` (`circ2.c`):

```
[...]
void draw_circle_fixed ()
{
    fixed x, y;
    int length = 50;
    fixed angle = 0;
    fixed angle_stepsize = itofix (5);

    // go through all angles from 0 to 255
```

```

while (fixtoi (angle) < 256)
{
    // calculate x, y from a vector with known length and angle
    x = length * fcos (angle);
    y = length * fsin (angle);

    putpixel (screen,
              fixtoi(x) + SCREEN_W / 2, fixtoi(y) + SCREEN_H / 2,
              makecol (255, 255, 255));
    angle += angle_stepsize;
}
}
[...]
```

Note que usamos fsin y fcos cuando usamos números de puntos fijos.

### introduciendo otra manera de representar ángulos

En la nueva versión de la función draw\_circle la condición del ciclo while ha cambiado por (fixtoi (angle) < 256 ). Veremos la manera en que los programadores prefieren usar ángulos: Ellos hacen uso de un círculo que esta dividido en 256 partes, con un rango de 0 a 255. Digamos que es parte de cómo maneja ángulos allegro. ¿Por que 256 en lugar de 360? ¿que pasará cuando tenga un ángulo de 361 grados?. Por definición un círculo es redondo, 361 grados representa el mismo punto como 1 grado. De la misma manera que  $3 * \text{PI}$  es lo mismo que  $1 * \text{PI}$  radianes, y 257 grados de allegro es lo mismo que 1 grado de allegro.

Para mantener el rango de valores de un ángulo dentro de los 0 y 360 grados deberemos hacerlo de la siguiente manera:

```

int angle_in_degrees;
while (angle_in_degrees > 360) angle_in_degrees -= 360;
while (angle_in_degrees < 0) angle_in_degrees += 360;
```

Pero como los grados de allegro tienen un rango de 0 a 255, y este rango puede guardarse en exactamente 8 bits, solo necesitamos reiniciar todos los otros bits y podemos estar seguro de tener un ángulo dentro del rango. Tenemos que desenmascarar todos los otros bits excepto los 8 bits menos significativos. Podemos hacer esto con el operador AND(&):

```

int allegro_degrees;
// guardar los 8 bits menos significativos
allegro_degrees &= 0xFF;
```

Para los que no entienden el operador AND: creanme, que es un modo bastante seguro de saber que el ángulo esta en el rango. Si usamos números de puntos fijos para representar grados debemos modificar un poco las cosas, ya que también tenemos 16 bits representando la parte derecha del punto. Entonces lo que tenemos que conservar es  $16 + 8 = 24$  bits. Esto es lo que hacemos:

```

fixed allegro_degrees;
// guardar los 24 bits menos significativos
```

```
allegro_degrees &= 0xFFFFFFFF;
```

Si entiendes esto, entonces entenderás porque la escala de 256 grados es usualmente mejor para programadores de juegos. Si usamos floats lo mejor es usar radianes, porque las funciones sin y cos tienen como parámetros radianes. Si usas números de puntos fijos, como usamos en este ejemplo, es mejor usar los grados de allegro, porque las funciones fsin() y fcos() los usan y es fácil mantener el ángulo dentro del rango con el operador AND.

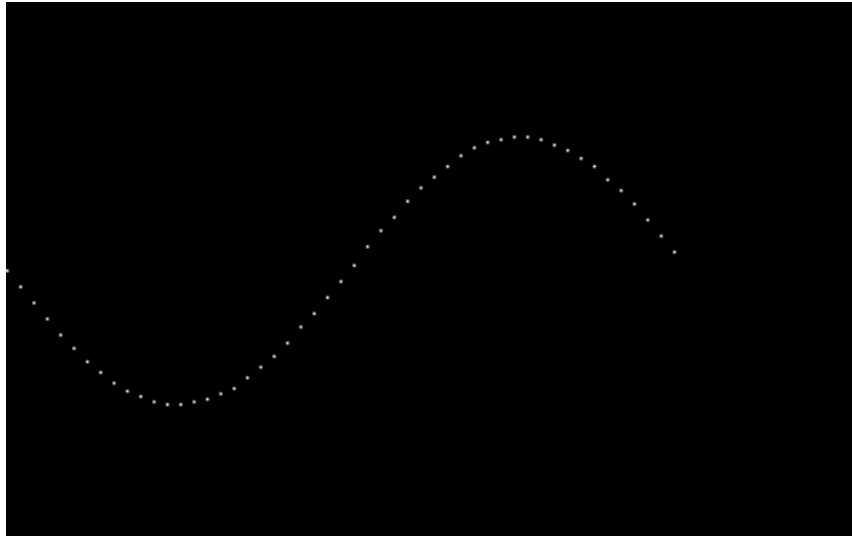
Como hicimos con radianes y grados, podemos calcular grados-allegro teniendo radianes o grados regulares. Aquí tienen el código para poder realizar ello:

```
allegro_degrees = regular_degrees * 256 / 360;  
allegro_degrees = radians * 128 / PI;
```

Para tener una idea de que hacen seno y coseno escribí la siguiente función (circ3.c)

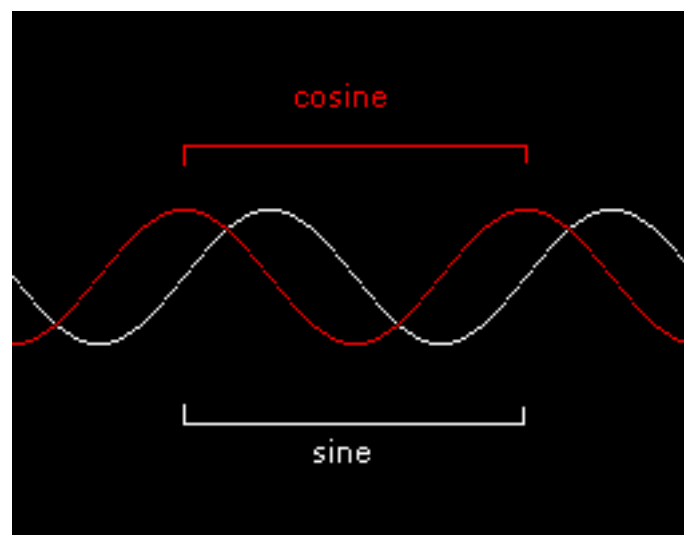
```
[...]  
void draw_sine ()  
{  
    int length = 50;  
    fixed x, y;  
    fixed angle = 0;  
    fixed angle_stepsize = itofix (5);  
  
    while (fixtoi(angle) < 256)  
    {  
        // the angle is plotted along the x-axis  
        x = angle;  
        // the sine function is plotted along the y-axis  
        y = length * fsin (angle);  
  
        putpixel (screen,  
                fixtoi (x), fixtoi (y) + SCREEN_H / 2,  
                makecol (255, 255, 255));  
  
        angle += angle_stepsize;  
    }  
[...]
```

Y verá en pantalla:



La función se ve más o menos parecida a la función `draw_circle`, pero hace algo diferente. Solo traza la función seno en la pantalla. Como podemos ver la función seno parece una ola. La función seno podría ser usada en nuestro juego para todos los movimientos con una forma oleada.

La imagen de abajo fue creada con una modificación en la versión `circle3.c`. Podemos ver que la función seno está trazada en blanco y la función coseno en rojo. Las funciones son continuas y repetitivas, no paran al llegar a los 256 grados-allegro. Si miramos atentamente veremos que las 2 funciones tienen la misma forma, la única diferencia es que la función coseno está desplazada un poco. El desplazamiento es exactamente 64 grados-allegro o 90 grados regulares.



En la tabla de abajo hay algunos valores claves de las funciones seno y coseno. Observemos que las dos funciones llegan a su máximo o su mínimo en los múltiplos de 90 grados regulares.

### un juego de autos de carrera

Hasta ahora conocemos 2 maneras de guardar un vector, ellas son mediante coordenadas cartesianas y polares. También aprendimos a calcular coordenadas cartesianas de un vector si sabemos sus coordenadas polares. Finalmente, vimos 3 modos diferentes de guardar ángulos: grados, radianes y grados-allegro. Pero

ahora, vamos al ejemplo con el que empezamos este artículo, el auto de carrera. En realidad el auto es un círculo con una línea representando la dirección de el auto, pero con un poco de imaginación puede ser un auto de carrera (ejemplo `circ4.c`):

```
[...]
void racing_car ()
{
    // length and angle of the racing car's velocity vector
    fixed angle = itofix (0);
    fixed length = itofix (0);
    // x- and y-coordinates of the velocity vector
    fixed vel_x, vel_y;

    // x- and y-position of the racing car
    fixed x = itofix (SCREEN_W / 2);
    fixed y = itofix (SCREEN_H / 2);

    while (!key[KEY_ESC])
    {
        // erase the old image
        circlefill (screen, fixtoi(x), fixtoi(y), 10, makecol (0, 0, 0));

        // check the keys and move the car
        if (key[KEY_UP] && length < itofix (2))
            length += ftofix (0.005);
        if (key[KEY_DOWN] && length > itofix (0))
            length -= ftofix (0.005);
        if (key[KEY_LEFT])
            angle = (angle - itofix (1)) & 0xFFFFFFFF;
        if (key[KEY_RIGHT])
            angle = (angle + itofix (1)) & 0xFFFFFFFF;

        // calculate the x- and y-coordinates of the velocity vector
        vel_x = fmul (length, fcos (angle));
        vel_y = fmul (length, fsin (angle));

        // move the car, and make sure it stays within the screen
        x += vel_x;
        if (x >= itofix (SCREEN_W)) x -= itofix(SCREEN_W);
        if (x < itofix (0)) x += itofix(SCREEN_W);
        y += vel_y;
        if (y >= itofix (SCREEN_H)) y -= itofix(SCREEN_H);
        if (y < itofix (0)) y += itofix(SCREEN_H);

        // draw the racing car
        circle (screen, fixtoi(x), fixtoi(y), 10, makecol (0, 0, 255));
        line (screen, fixtoi(x), fixtoi(y),
            fixtoi (x + 9 * fcos (angle)),
            fixtoi (y + 9 * fsin (angle)),

```

```

        makecol (255, 0, 0));

    // wait for 10 milliseconds, or else we'd go too fast
    rest (10);
}
}
[...]
```

La velocidad del auto de carrera está representada por el ángulo y el largo. Si el jugador aprieta UP, el largo del vector velocidad se incrementa; si aprieta DOWN, el largo del vector velocidad se decrementa, el ángulo cambia si el jugador aprieta LEFT o RIGHT. Con el enmascaramiento de los 24-bits nos aseguramos que el ángulo se mantenga en el rango. Después de que la dirección y la velocidad han sido ajustado, las coordenadas cartesianas 'vel\_x' y 'vel\_y' son calculadas con sin() y cos(). En cada iteración del bucle, estas coordenadas son sumadas a las coordenadas del auto.

### otra cosa importante que puedes hacer con seno y coseno

Si entiendes todo esto, no tendremos problema con el siguiente programa. Es otro pequeño ejemplo de lo que podemos hacer con seno y coseno. Esta vez usaremos seno y coseno para animar la orbita de un planeta. El planeta va a ser representado por un pequeño punto, que se moverá alrededor de un circulo. Aquí tenemos el código ( circ5.c):

```

[...]
```

```

void orbit ()
{
    int x = 0, y = 0;

    fixed angle = itofix (0);
    fixed angle_stepsize = itofix (1);

    // These determine the radius of the orbit.
    // See what happens if you change length_x to 100 :)
    int length_x = 50;
    int length_y = 50;

    // repeat this until a key is pressed
    while (!keypressed())
    {
        // erase the point from the old position
        putpixel (screen,
            fixtoi(x) + SCREEN_W / 2, fixtoi(y) + SCREEN_H / 2,
            makecol (0, 0, 0));

        // calculate the new position
        x = length_x * fcos (angle);
        y = length_y * fsin (angle);

        // draw the point in the new position
```

```

    putpixel (screen,
              fixtoi(x) + SCREEN_W / 2, fixtoi(y) + SCREEN_H / 2,
              makecol (255, 255, 255));

    // increment the angle so that the point moves around in circles
    angle += angle_stepsize;

    // make sure angle is in range
    angle &= 0xFFFFFFFF;

    // wait 10 milliseconds, or else it'd go too fast
    rest (10);
}
}
[...]
```

Trata experimentando con diferentes valores de 'length\_x' y 'length\_y'. Si estos 2 son diferentes, el resultado sera que el planeta no se moverá formando un circulo, sino que se moverá trazando una elipse.

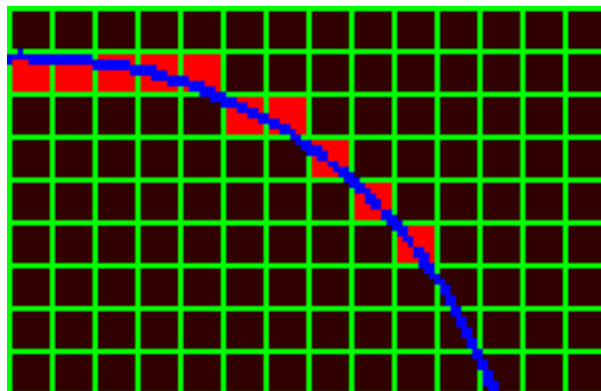
### dibujando un circulo de otra forma

En el primer capitulo expliqué que hay 2 maneras de dibujar un circulo, una usando floats y otra usando números de puntos fijos. Pero si miramos el archivo gfx.c en el directorio allegro/src/ veremos que el código fuente de la función circle() no es como el código fuente de la función draw\_circle que está aquí.

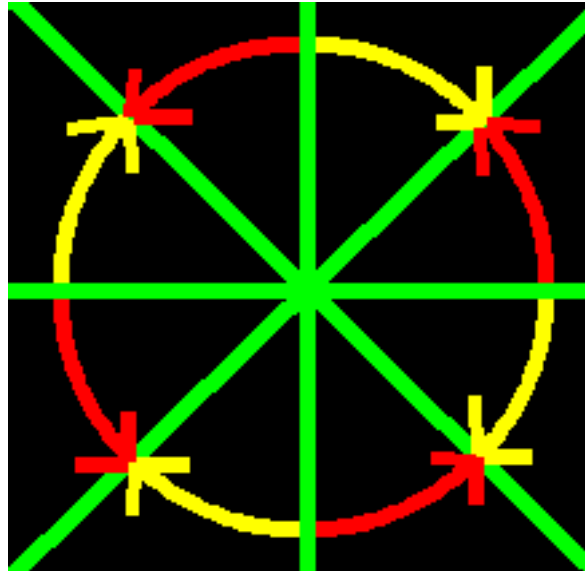
En realidad no encontraremos una sola función seno y coseno. El código hace uso de que todos los puntos están a la misma distancia del centro.

Digamos que empezamos en la parte superior del circulo. Las coordenadas en la parte superior son fácil de calcular: la coordenada X es 0 y la coordenada Y es igual que al radio del circulo (pero negativo en función de las coordenada de la pantalla). Entonces dibujamos un pixel en esa coordenada. Para el próximo pixel o podemos ir un pixel a la derecha, o un pixel abajo y luego un pixel a la derecha.

La solución es calcular para las 2 posibilidades la distancia al centro con el teorema de Pitagoras. Dibujamos el pixel cuya distancia desde el centro se aproxima más al radio del circulo.



Solo tenemos que hacer esto para la octava parte del círculo. El resto del círculo se puede dibujar haciendo uso de las líneas horizontales, verticales y diagonales de la simetría del círculo, como puedes ver en la figura de abajo. Observemos que podemos dibujar todas las secciones rojas y amarillas por el precio de una, simplemente reflejándolo a través de las líneas verdes.



Aquí está el código (circ6.c)

```
[...]
void my_draw_circle (BITMAP *bmp, int center_x, int center_y, int r, int color)
{
    // x and y are the current position in the circle.
    int x = 0, y = r;

    while (x <= y)
    {
        // We make use of 8 axes of symmetry in a circle.
        // This way we have fewer points to calculate on its circumference.
        putpixel (bmp, center_x + x, center_y + y, color);
        putpixel (bmp, center_x - x, center_y + y, color);
        putpixel (bmp, center_x + x, center_y - y, color);
        putpixel (bmp, center_x - x, center_y - y, color);
        putpixel (bmp, center_x + y, center_y + x, color);
        putpixel (bmp, center_x - y, center_y + x, color);
        putpixel (bmp, center_x + y, center_y - x, color);
        putpixel (bmp, center_x - y, center_y - x, color);

        // This is the most important part of the function.
        // We go to the right in all cases (x++).
        // We need to decide whether to go down (y--).
        // This depends on which point is
        // closest to the path of the circle.
        // Good old Pythagoras will tell us what to do.
```

```

    x++;
    if (abs (x*x + y*y - r*r) >
        abs (x*x + (y-1)*(y-1) - r*r))
        y--;
    }
}
[...]
```

Igual este código no se ve como el archivo gfx.c de allegro, pero esto es principalmente por lo siguiente:

```
if (abs (x*x + y*y - r*r) > abs (x*x + (y-1)*(y-1) - r*r))
```

Puede optimizarse mucho mas, de hecho si le hacemos todas las optimizaciones arribaremos a la función actual circle() de allegro.

### vectores por el otro lado

Hemos visto como ir desde coordenadas polares a cartesianas con sin y cos:

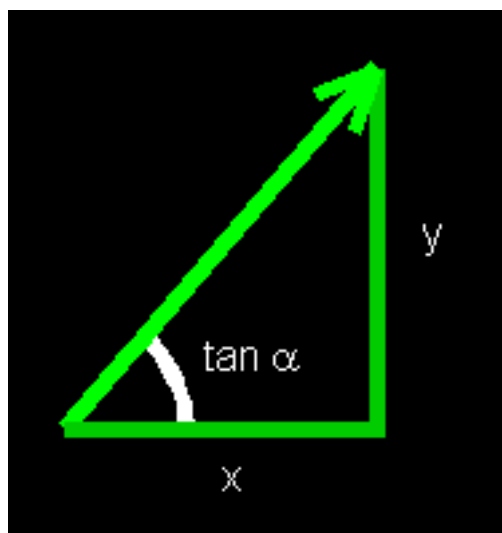
```
x = length * cos (angle)
y = length * sin (angle)
```

Ahora explicaré como ir en sentido contrario. Calcular el largo es la parte fácil, porque solo necesitamos el teorema de Pitagoras:  $a^2 + b^2 = c^2$  o más práctico:

```
length = sqrt (x * x + y * y)"
```

Calcular el ángulo es un poco mas difícil. Hay una función matemática llamada tangente, cuya implementación en C es la función tan(), que puede ser usada para calcular la proporción entre Y y X como se muestra a continuación:

```
tan (angle) = y / x
```



Esto puede ser escrito como:

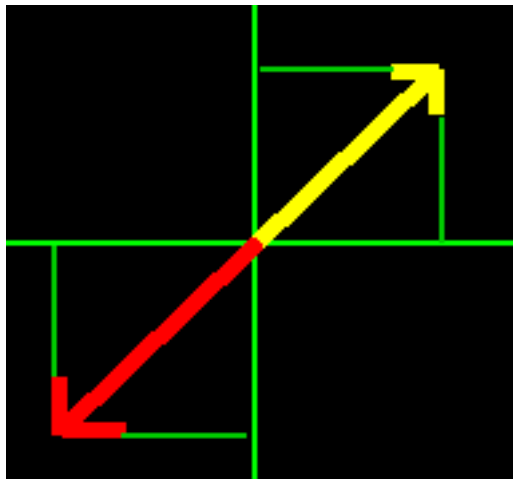
$$\tan(\text{angle}) = \sin(\text{angle}) / \cos(\text{angle})$$

Esto significa que la función "tan" es combinación de las funciones sin y cos. La función inversa de la tangente es llamada arcotangente; en C es atan(). Esta función puede ser usada para calcular el ángulo si conoces la proporción entre Y y X:

$$\text{angle} = \text{atan}(y / x)$$

Pero existe un problema menor: a veces este cálculo te dará un resultado incorrecto. En la figura de abajo se ven 2 vectores uno rojo y otro amarillo. Los dos tienen la misma proporción entre Y y X. Eso significa que si calculas el arcotangente de los 2 obtendrás el mismo resultado, el cual es 45 grados. Esto es correcto solo para el vector amarillo.

Además, tendrás que verificar los casos en donde X es 0, para evitar la división por 0.



Una solución parcial es la siguiente (parcial porque no comprobamos cuando X es 0).

```
if (x > 0)
    angle = atan (y / x);
else
    angle = PI + atan (y / x)
```

Pero para los programadores está la función atan2():

$$\text{angle} = \text{atan2}(y, x)$$

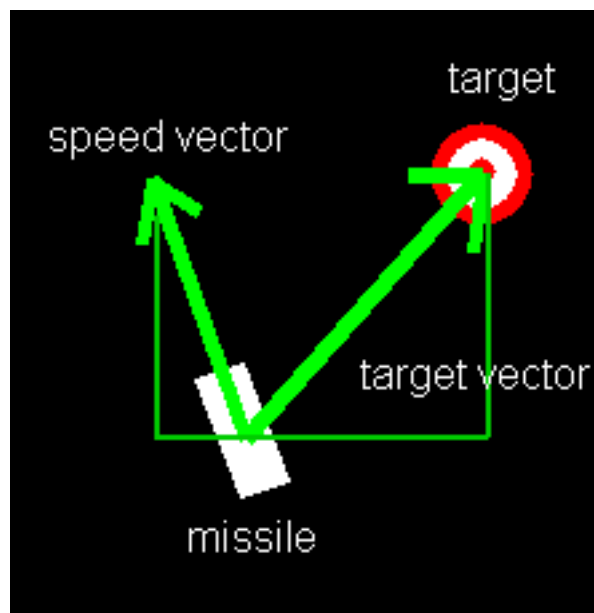
Esta función siempre producirá el ángulo correcto para cualquier par (x,y). Por supuesto que para números de puntos fijos, allegro provee la homologa fatan2().

**usando atan2()**

Supongamos que estamos escribiendo un juego en el que el jugador puede disparar misiles direccionales. Entonces decidimos hacerlo como dijimos.

Primero calculamos la dirección del objetivo como mirando desde el misil. Luego comparamos este ángulo con el ángulo actual del misil. Si el ángulo del objetivo es mayor que el ángulo actual, el ángulo tendría que incrementarse y viceversa.

Esto es una buena idea, pero ¿cómo calculamos la dirección del objetivo visto desde el misil? Podemos visualizar esto como un vector desde el misil hasta el objetivo. La coordenada X e Y del vector se pueden calcular muy fácil - solo restamos las coordenadas del vector del misil con las coordenadas del objetivo. Dadas las coordenadas X e Y del vector, podemos calcular el ángulo y el largo, usando la función `atan2()` como describimos antes. El largo no es importante pero el ángulo sí lo necesitaremos.



En el siguiente código, la posición del misil es representada por las variables 'x' e 'y'. La velocidad del misil está representada por las variables 'length' y 'angle'. Primero el programa determina si el objetivo ha sido fijado, en caso contrario, el programa elegirá uno al azar.

Finalmente el programa determina como debería cambiar el ángulo del misil. El ángulo hacia el objetivo se calcula en esta línea:

```
target_angle = fatan2 (target_y - y, target_x - x);
```

El programa usa este ángulo calculado para determinar si el ángulo dirección del misil debería incrementarse o decrementarse. Calcula la diferencia entre el ángulo del objetivo y el ángulo actual. Después se asegura de que esta diferencia este dentro del rango: `&0xFFFFFFFF`. Si el ángulo es menor que 128 grados-allegros (180 grados-normal), el ángulo dirección se decrementa. De otra manera se incrementa.

```
if (((angle-target_angle) & 0xFFFFFFFF) < itofix(128))
    angle = (angle - angle_stepsize) & 0xFFFFFFFF;
else
    angle = (angle + angle_stepsize) & 0xFFFFFFFF;");?>
```

Aquí tenemos todo el código (circ7.c)

```
[...]
void home_in ()
{
    // the x, y position of the homing missile
    fixed x = itofix(SCREEN_W / 2);
    fixed y = itofix(SCREEN_H / 2);
    // the angle and length of the missile's velocity vector
    fixed angle = 0;
    int length = 1;
    fixed angle_stepsize = itofix (3);
    // determines whether the missile has reached
    // the target and a new one should be chosen
    int new_target = TRUE;
    // angle to the target
    fixed target_angle;
    // position of the target
    fixed target_x, target_y;

    while (!keypressed())
    {
        clear (screen);
        // choose new target randomly when needed
        if (new_target)
        {
            target_x = itofix((SCREEN_W + rand() % (2 * SCREEN_W)) / 4);
            target_y = itofix((SCREEN_H + rand() % (2 * SCREEN_H)) / 4);
            new_target = FALSE;
        }

        // move the missile
        x += length * fcos (angle);
        y += length * fsin (angle);

        // if we are very close to the target, set a new target
        if (abs (x - target_x) + abs (y - target_y) < itofix(10))
            new_target = TRUE;

        // draw a pixel where the target is
        putpixel (screen, fixtoi(target_x), fixtoi(target_y),
            makecol (255, 255, 255));

        // draw the missile
        // (actually a circle with a line representing the angle)
        circle (screen, fixtoi(x), fixtoi(y), 10, makecol (0, 0, 255));
        line (screen, fixtoi(x), fixtoi(y),
            fixtoi(x) + fixtoi (9 * fcos (angle)),
            fixtoi(y) + fixtoi (9 * fsin (angle)),
```

```

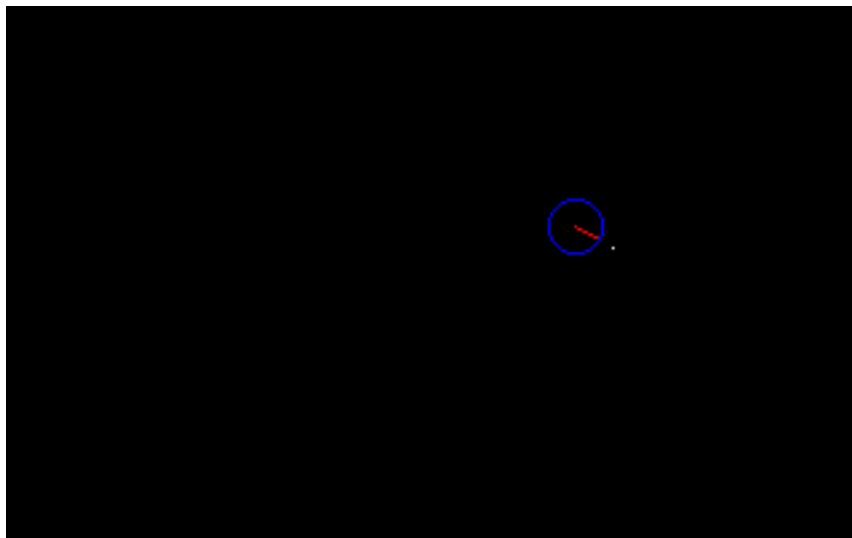
        makecol (255, 0, 0));

    // calculate the angle from the missile to the target
    target_angle = fatan2 (target_y - y, target_x - x);

    // Determine whether we should turn left or right.
    // Note that itofix (128) represents half a circle.
    // We use & 0xFFFFFFFF as a trick to get an angle
    // between 0 and 256.
    if (((angle-target_angle) & 0xFFFFFFFF) < itofix(128))
        angle = (angle - angle_stepsize) & 0xFFFFFFFF;
    else
        angle = (angle + angle_stepsize) & 0xFFFFFFFF;

    rest (10);
}
}
[...]
```

Aquí mostramos la salida. Como podemos ver, el misil es representado por un círculo azul con una línea roja dentro. El objetivo está representado por un punto blanco.



### usando el producto punto

La solución anterior enfrenta el problema bastante bien pero no quiere decir que no haya otra solución. En los libros de matemáticas puedes encontrar la siguiente fórmula para calcular el ángulo entre el vector  $a$  y  $b$ .

$$\cos(\text{angle}) = (x_a * x_b + y_a * y_b) / (\text{length}(a) * \text{length}(b))$$

La expresión  $(x_a * x_b + y_a * y_b)$  se llama producto punto y es igual al producto de los largos de los vectores multiplicado por el coseno del ángulo entre ellos. Queremos que nuestro misil vaya por un camino si el ángulo entre la dirección actual y el del objetivo está entre 0 y 180 grados, y que vaya para otro lado si el ángulo entre

esta entre 180 y 360 grados.

Dada su naturaleza el arcoseno no puede ser usado para determinar la diferencia entre el rango debajo de los 180 grados y por arriba de los 180 grados. Podemos determinar la diferencia entre el rango debajo de los 90 y por arriba de los 270, y el rango entre 90 y 270, porque el coseno es positivo en el primer caso y negativo en el segundo. Si no puedes ver esto, mira la imagen de la ola del coseno de nuevo.

Si rotamos un vector por 90 grados, podemos simplemente verificar si el resultado del producto punto es por debajo o por arriba del 0, para ver si tendríamos que doblar a la izquierda o a la derecha. Para hacer esto, hacemos uso de un pequeño truco: cambiamos las coordenadas y le cambiamos el signo de una de ellas. Dicho de otra manera, cambiamos 'xa' por 'ya' y 'ya' por '-xa'.

```
cos (angle) = (ya * xb - xa * yb) / (length (a) * length (b))
```

Ya que necesitamos saber si el resultado es positivo o negativo, y no necesitamos el valor actual del resultado, podemos dejar afuera el calculo del largo de los vectores.

```
result = ya * xb - xa * yb
```

Si el resultado es positivo doblamos hacia un lugar y si el resultado es negativo, doblamos hacia el otro. Aquí está el código:

```
if (fmul(dy,(target_x - x)) + fmul(-dx,(target_y - y)) > 0)
    angle = (angle - angle_stepsize) & 0xFFFFFF;
else
    angle = (angle + angle_stepsize) & 0xFFFFFF;");
```

En este código, 'dx' y 'dy' representan el vector velocidad del misil y "target\_x - x" e "target\_y - y" representan el vector hacia el objetivo. Aquí tenemos el ejemplo completo (circ8.c):

```
[...]
void dot_product_home_in ()
{
    // the position of the homing missile
    fixed x = itofix(SCREEN_W / 2);
    fixed y = itofix(SCREEN_H / 2);
    // the angle and length of the missile's velocity vector
    fixed angle = 0;
    int length = 1;
    fixed angle_stepsize = itofix (3);
    // determines whether the missile has reached
    // the target and a new one should be chosen
    int new_target = TRUE;
    // position of the target
    fixed target_x, target_y;
    // vector of missile movement
    fixed dx, dy;

    while (!keypressed())
```

```
{
    clear (screen);
    // choose new target randomly when needed
    if (new_target)
    {
        target_x = itofix((SCREEN_W + rand() % (2 * SCREEN_W)) / 4);
        target_y = itofix((SCREEN_H + rand() % (2 * SCREEN_H)) / 4);
        new_target = FALSE;
    }

    // Move the missile
    // We store dx and dy in variables so that
    // we can use them later on in the dot product.
    dx = length * fcos (angle);
    dy = length * fsin (angle);
    x += dx;
    y += dy;

    // if we are very close to the target, set a new target
    if (abs (x - target_x) + abs (y - target_y) < itofix(10))
        new_target = TRUE;

    // draw a pixel where the target is
    putpixel (screen, fixtoi(target_x), fixtoi(target_y),
              makecol (255, 255, 255));

    // draw the missile
    // (actually a circle with a line representing the angle)
    circle (screen, fixtoi(x), fixtoi(y), 10, makecol (0, 0, 255));
    line (screen, fixtoi(x), fixtoi(y),
          fixtoi(x) + fixtoi (9 * fcos (angle)),
          fixtoi(y) + fixtoi (9 * fsin (angle)),
          makecol (255, 0, 0));

    // Determine whether we should turn left or right
    // using the dot product.
    // We use & 0xFFFFFFFF as a trick to get an angle
    // between 0 and 256.
    if (fmul(dy,(target_x - x)) + fmul(-dx,(target_y - y)) > 0)
        angle = (angle - angle_stepsize) & 0xFFFFFFFF;
    else
        angle = (angle + angle_stepsize) & 0xFFFFFFFF;

    rest (10);
}
}
[...]
```

Por alguna razón algunos expertos programadores de juegos prefieren no usar atan2() y prefieren por alguna

razón el producto punto. ¿sera porque atan2() puede introducir error de redondeo? no estoy seguro. En el caso del misil ambos métodos trabajan bien, y es método a utilizar es una cuestión de preferencia.

### seno, coseno, mapas de bit y rotación

A lo largo del artículo hemos mencionado algunas aplicaciones que se le pueden dar a las funciones seno y coseno. Pero de hecho no hay fin para el uso de estas dos funciones. Daré otro ejemplo: rotación de sprites.

No debemos pensar: 'esto debe ser probablemente complicado y la biblioteca allegro ya nos provee de ello, entonces no lo necesito'. Pensemos en todas las modificaciones que podemos hacer si sabemos como trabaja la función de rotación de sprites - rotando mapa de tiles por ejemplo.

Entonces, ¿como trabaja? Existen 2 maneras de hacer esto. Una, la más obvia, es una iteración a través de todos los pixel del sprite que queremos rotar, calcular para cada pixel donde debe ir en pantalla, y después copiarlo. Esto es ciertamente posible, pero no habrá correspondencia de uno a uno de los pixeles de la pantalla. Entonces deberíamos buscar otra forma.

Una alternativa es que iteramos a través de todos los pixeles en el bitmap objetivo (casi siempre la pantalla) y calculamos que pixel del sprite debería ir ahí. Así nos aseguramos que cada pixel se pinta en pantalla, y que ninguno se imprime 2 veces (o mas).

Empecemos en la posición (0,0) de la pantalla. ¿Que pixel del bitmap debería ir ahí?. Hacer esto simple, ponemos la posición (0,0) del sprite. Luego nos movemos un lugar a la derecha de la pantalla, a la posición (1,0). ¿Que pixel del sprite debería ir allí?. Eso depende del ángulo que queremos rotar. Si rotamos a 0 grados, ponemos el pixel (1,0) del sprite allí. Si rotamos 270 grados, deberíamos poner el pixel (0,1) allí. Con esté código podemos realizar el cálculo para cualquier ángulo:

```
sprite_x = cos (angle);  
sprite_y = sin (angle);
```

Luego vamos una posición más a la derecha. La posición en el sprite que deberíamos usar ahora es:

```
sprite_x = 2 * cos (angle);  
sprite_y = 2 * sin (angle);
```

Así sucesivamente. Como estamos trabajando linealmente, simplemente podemos calcular el sin y cos una vez y sumarle esto a la posición en el sprite, cada vez que hacemos un pixel a la derecha en el destino. Mira el siguiente código (circ9.c):

```
[...]  
void my_rotate_sprite (BITMAP *dest_bmp, BITMAP *src_bmp,  
    fixed angle, fixed scale)  
{  
    // current position in the source bitmap  
    fixed src_x, src_y;  
  
    // current position in the destination bitmap  
    int dest_x, dest_y;
```

```
// src_x and src_y will change each time by dx and dy
fixed dx, dy;

// src_x and src_y will be initialized to start_x and start_y
// at the beginning of each new line
fixed start_x = 0, start_y = 0;

// We create a bit mask to make sure x and y are in bounds.
// Unexpected things will happen
// if the width or height are not powers of 2.
int x_mask = src_bmp->w - 1;
int y_mask = src_bmp->h - 1;

// calculate increments for the coordinates in the source bitmap
// for when we move right one pixel on the destination bitmap
dx = fmul (fcos (angle), scale);
dy = fmul (fsin (angle), scale);

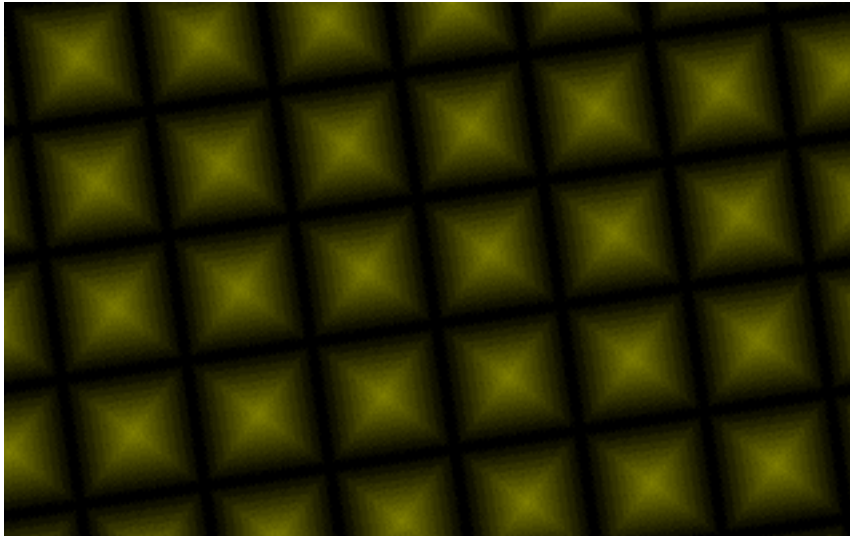
for (dest_y = 0; dest_y < dest_bmp->h; dest_y++)
{
    // set the position in the source bitmap to the
    // beginning of this line
    src_x = start_x;
    src_y = start_y;

    for (dest_x = 0; dest_x < dest_bmp->w; dest_x++)
    {
        // Copy a pixel.
        // This can be optimized a lot by using
        // direct bitmap access.
        putpixel (dest_bmp, dest_x, dest_y,
            getpixel (src_bmp,
                fixtoi (src_x) & x_mask,
                fixtoi (src_y) & y_mask));

        // advance the position in the source bitmap
        src_x += dx;
        src_y += dy;
    }

    // for the next line we have a different starting position
    start_x -= dy;
    start_y += dx;
}
[...]
```

Y su salida en pantalla:



Si miramos estas líneas:

```
dx = fmul (fcos (angle), scale);
dy = fmul (fsin (angle), scale);
```

Aquí calculamos el sin y cos del ángulo. El prefijo "d" en 'dx' y 'dy' hace referencia a delta. Esto representa el cambio en la posición en el sprite, a medida que avanzamos al próximo pixel en la pantalla. Como puedes ver, se introduce un factor de escala, así podemos hacer un acercamiento y un alejamiento de la imagen.

En las siguientes líneas:

```
putpixel (dest_bmp, dest_x, dest_y, getpixel (src_bmp,
    fixtoi (src_x) & x_mask,
    fixtoi (src_y) & y_mask))
```

...el pixel se copia desde el bitmap fuente (el sprite) al bitmap objetivo (la pantalla). Por supuesto que 'dest' hace referencia a destino (destination en inglés) y 'src' a fuente (source en inglés). Se utiliza mascara para asegurarnos que la posición en el bitmap fuente es válida, para no obtener un pixel que esta afuera del bitmap. Esto solo es aplicable si las dimensiones del bitmap fuente son potencia de 2, por ejemplo bitmaps de 32x32, o 64x256 funcionarían. Pero un bitmap de 100x100 no lo haría, porque 100 no es potencia de 2.

Con las siguientes líneas, nos movemos a la siguiente posición en la pantalla. 'dest\_x' es incrementada en el ciclo for, y 'src\_x' y 'src\_y' son incrementadas por 'dx' y 'dy' calculados anteriormente:

```
src_x += dx;
src_y += dy;
```

Después de que toda la línea se imprime, la posición en el bitmap fuente es devuelta a la posición del comienzo grabada en 'start\_x' y 'start\_y'. Por supuesto que 'start\_x' y 'start\_y' tienen que cambiar para poder ir una línea mas abajo. Al ser, un pixel para abajo, perpendicular al pixel que esta una posición a la derecha, utilizamos el mismo truco que usamos con el producto punto: reemplazamos 'dx' con '-dy' y 'dy' con 'dx'. Entonces aquí tenemos como la posición inicial debería cambiarse:

```
start_x -= dy;
start_y += dx;
```

## rotación

Supongamos que en cierto juego queremos rotar un punto alrededor de otro punto. Por ejemplo el jugador puede saltar de una cuerda y dar vueltas por ella y por ultimo saltar a una plataforma. Puede representar la vuelta como una rotación del jugador alrededor del punto donde la cuerda esta sujeta. Para poder hacer ello, necesitamos calcular el vector que va del jugador al centro de la rotación, donde la cuerda esta atada, tomamos el ángulo de este vector, incrementamos un poco, y recalculamos la posición del jugador.

Esto no es práctico aquí, porque la mayoría de las veces, guardamos la posición del jugador en coordenadas cartesianas. Tendríamos que calcular el ángulo del vector, del jugador hacia el centro de rotación, con `atan2()`. Después de incrementar el ángulo, podemos calcular las nuevas coordenadas `x` e `y` con `sin` y `cos`. Aquí esta el ejemplo:

```
angle = atan2 (y, x);
length = sqrt (x * x + y * y);
angle += 1;
new_x = length * cos (angle);
new_y = length * sin (angle);
```

Al convertir las coordenadas cartesianas a coordenadas polares y viceversa, podemos perder precisión. Existe una manera mejor; podemos hacer uso de una matriz de rotación. Las Matrices de rotación se utilizan frecuentemente en el mundo de gráficos 3D, pero también se pueden utilizar en 2D tranquilamente. Ellas proveen una manera de rotar un vector sin convertirlo a coordenadas polares. Aquí está la ecuación:

```
new_x = x * cos (angle) - y * sin (angle)
new_y = x * sin (angle) + y * cos (angle)
```

En este caso 'angle' es el ángulo con el que queremos rotar el vector. 'x' e 'y' son las coordenadas viejas del vector, y 'new\_x' y 'new\_y' son las nuevas coordenadas del vector. Con este método, podemos realizar rotaciones sin usar `atan2`. Es lógico precalcular `cos` y `sin`, ya que las necesitaremos 2 veces a cada una.

Aquí hay un ejemplo completo usando este método (`circ10.c`). Todo lo que se hace es rotar cuatro puntos alrededor del centro de la pantalla.

```
[...]
void projection_test()
{
    // initialize the coordinates of four dots
    fixed dot_x[4] = {itofix(-50), itofix(-50), itofix(50), itofix(50)};
    fixed dot_y[4] = {itofix(-50), itofix(50), itofix(50), itofix(-50)};

    fixed angle = 0;
    fixed angle_stepsize = itofix (1);

    // proj_x and proj_y will contain the projection of the dots
    fixed proj_x[4];
    fixed proj_y[4];

    int i;
```

```

// repeat this loop until Esc is pressed
while (!key[KEY_ESC])
{
    // project all the dots to their new positions after rotation
    for (i = 0; i < 4; i++)
    {
        proj_x[i] = fmul (dot_x[i], fcos (angle)) -
            fmul (dot_y[i], fsin (angle));
        proj_y[i] = fmul (dot_x[i], fsin (angle)) +
            fmul (dot_y[i], fcos (angle));
    }

    // draw the four dots
    for (i = 0; i < 4; i++)
    {
        putpixel (screen,
            fixtoi (proj_x[i]) + SCREEN_W / 2,
            fixtoi (proj_y[i]) + SCREEN_H / 2,
            makecol (255 ,255, 255));
    }

    rest (10);
    clear (screen);

    angle += angle_stepsize;
}
[...]
```

Para más información en matrices de rotación consulté este vínculo:  
[http://www.student.hk-r.se/pt93mm/thesis/techniques/3d\\_tutorial/3d.html](http://www.student.hk-r.se/pt93mm/thesis/techniques/3d_tutorial/3d.html)

Este sitio también contiene información acerca de proyecciones en 3D.

Existe un caso especial en rotaciones con matrices; una rotación de 90 grados. Para realizar este tipo de rotación hacemos lo siguiente: supongamos que tenemos un punto con coordenadas (4,8) y queremos rotarlo 90 grados alrededor del origen:

$$\begin{aligned} \text{new\_x} &= x * \cos (90) - y * \sin (90) \\ \text{new\_y} &= x * \sin (90) + y * \cos (90) \end{aligned}$$

cos (90) es 0 y sin (90) es 1, entonces simplifiquemos la formula así:

$$\begin{aligned} \text{new\_x} &= -y; \\ \text{new\_y} &= x; \end{aligned}$$

Entonces las nuevas coordenadas son (-8, 4). Usamos este truco 2 veces, ahora ya sabemos porque funciona. Si queremos rotar el punto (4, 8) por 180 grados alrededor del origen, obtenemos lo siguiente:

```
new_x = x * cos (180) - y * sin (180);  
new_y = x * sin (180) + y * cos (180);
```

o:

```
new_x = -x  
new_y = -y
```

Entonces la nueva coordenada es (-4, -8). En la tabla de abajo podemos ver como rotar por ángulos de 90, 180 y 270 grados.

### Información adicional acerca del documento

El artículo original (en inglés) fué realizado por Amarillion ([amarillion \\_arroba\\_ yahoo \\_dot\\_ com](mailto:amarillion_arroba_yahoo_dot_com)) para el número 5 de la revista electrónica Pixelate (<http://www.allegro.cc/pixelate>).

La presente versión no es una traducción literal del artículo original, existen ligeras modificaciones a fin de simplificar la redacción y adaptar los términos al español. Si tiene alguna duda por favor comuníquese con nosotros.

---

*Este documento ha sido generado automáticamente a partir del archivo 'seno\_coseno.xml' el Mon Jan 19 20:49:01 2009*

*La versión mas reciente de este documento se almacena en [www.losersjuegos.com.ar](http://www.losersjuegos.com.ar). Visitenos para obtener mas recursos y actualizaciones.*