main.c

```c
/* WELCOME TO THE PELAVERSE */
/* please use Linux Code::Blocks */

/* header files that call allegro libraries */
#include <allegro5/allegro5.h>
#include <allegro5/allegro_font.h>
#include <allegro5/allegro_ttf.h>
#include <allegro5/allegro_primitives.h>
#include <allegro5/allegro_audio.h>
#include <allegro5/allegro_acodec.h>
#include <allegro5/allegro_image.h>

/* header files for C library */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

/* header file for the game */
#include "main.h"

/* declaring and initialising global variables */
int frame = 0;
int hold_frame = 0;
bool music = false;
bool done = false;
bool redraw = true;
bool menu = true;
unsigned char key [ ALLEGRO_KEY_MAX ];
long score_display;

/* declaring and initialising structs. */
arrow_t arrow;
anti_token_t anti_token [ AT_N ];
blue_token_t b_token;
green_token_t g_token;
yellow_token_t y_token;
star_t star [ ST_N ];

/* initialising Allegro's native structs. */
ALLEGRO_DISPLAY * disp;
ALLEGRO_SAMPLE * song = NULL;
ALLEGRO_SAMPLE_INSTANCE * songInstance = NULL;
ALLEGRO_FONT * font;


/* ----------------- MAIN PROGRAM-------------------------------------------- */
int main ( )
{
        /* seeds the pseudo-random generator */
        srand ( time ( NULL ) );

        /* starts the program */
        must_init ( al_init( ), "allegro" );

        /* activates the keyboard peripheral */
        must_init ( al_install_keyboard( ), "keyboard" );

        /* Initialises the timer */
        ALLEGRO_TIMER* timer = al_create_timer ( 1.0 / 120 );
        must_init ( timer, "timer" );

        /* initialises the event queue */
        ALLEGRO_EVENT_QUEUE* queue = al_create_event_queue ( );
        must_init ( queue, "queue" );

        /* initialises allegro stock primitive functions */
        must_init ( al_init_primitives_addon ( ), "primitives" );

        /* audio initialiasation */
        audio_init ( );

        /* initialises native allegro display functions */
        disp_init ( );

        /* initialises native allegro keyboard functions */
        keyboard_init ( );

        /* initialises other custom functions */
        score_init ( );
        arrow_init ( );
        anti_token_init ( );
        blue_token_init ( );
        green_token_init ( );
        yellow_token_init ( );
        stars_init ( );

        /* make display, keyboard and timer a part of the program */
        al_register_event_source ( queue,
                al_get_keyboard_event_source ( ) );

        al_register_event_source( queue,
                al_get_display_event_source ( disp ) );

        al_register_event_source ( queue,
                al_get_timer_event_source ( timer ) );

        /* serves as continuous input for al_register_event_source */
        ALLEGRO_EVENT event;

        /* start timer before game begins */
        al_start_timer( timer );

        /* main game loop */
        while ( true )
        {
                al_wait_for_event ( queue, &event );

                /* makes sure the frame will either advance
                   according to specified frames per second
                   or will stop advancing and exit the program */
                switch( event.type )
                {
                        case ALLEGRO_EVENT_TIMER:

                                /* updates all the elements
                                 * in the game */
                                arrow_update ( );
                                anti_token_update ( );
                                green_token_update ( );
                                blue_token_update ( );
                                yellow_token_update ( );
                                stars_update ( );

                                /* Condition to exit the program */
                                if ( key [ ALLEGRO_KEY_ESCAPE ] )
                                        done = true;

                                /* condition to start the game */
                                if ( key [ ALLEGRO_KEY_Y ] )
                                {
                                        frame = 0;
                                        menu = false;
                                }

                                redraw = true;
                                frame++;
                                break;

                        /* Condition to exit the program */
                        case ALLEGRO_EVENT_DISPLAY_CLOSE:

                                done = true;
                                break;
                }
                /* exits the program */
                if ( done )
                        break;

                keyboard_update ( &event );

                /* if frame advances this conditional branch will execute */
                if ( redraw && al_is_event_queue_empty ( queue ) )
                {
                        /* clears the screen */
                        al_clear_to_color ( al_map_rgb ( 0, 0, 0 ) );

                        /* making sure game does not start until
                         * y is pressed */
                        if ( menu == true )
                        {
                                instruction_draw ( );
                                stars_draw ( );
                        }
                        else
                        {
                                /* starts the music when playing */
                                if ( !music )
                                {
                                        sample_trigger ( );
                                        music = true;
                                }

                                /* draws all the objects onto the screen */
                                score_draw ( );
                                arrow_draw ( );
                                anti_token_draw ( );
                                blue_token_draw ( );
                                green_token_draw ( );
                                yellow_token_draw ( );
                                stars_draw ( );
                        }
                        al_flip_display ( );
                        redraw = false;
                }
        }

        /* destroys all the stock allegro functions */
        audio_destroy ( );
        al_destroy_font ( font );
        al_destroy_display ( disp );
        al_destroy_timer ( timer );
        al_destroy_event_queue ( queue );

        return EXIT_SUCCESS;
}

/* ----------------- GENERAL FUNCTIONS --------------------------------------- */

/* tests each of allegro’s stock initialisation functions */
void must_init ( bool test, const char *description )
{
        if ( !test ) {
            printf ( "couldn’t initialise %s\n", description );
            exit( 1 );
        }
}

/* outputs an integer between a range of two values */
int between ( int min, int max )
{
        return min + ( rand( ) % ( max - min ) );
}

/* outputs a rational number between 0 and 1 */
double between_f ( double min, double max )
{
        return min + ( ( double ) rand ( ) / ( float ) RAND_MAX )
                * ( max - min );
}
/* calculates the magnitude of a 2D vector */
double vector_mag ( double x, double y )
{
        x *= x;
        y *= y;

        return sqrt ( x + y );
}

/* collision detection using square boundaries for each token */
bool collide ( int top_left_1x, int top_left_1y,
               int top_left_2x, int top_left_2y,
               int bottom_right_1x, int bottom_right_1y,
               int bottom_right_2x, int bottom_right_2y )
{
        if ( top_left_1x > bottom_right_2x ) return false;
        if ( top_left_2x < bottom_right_1x ) return false;
        if ( top_left_1y > bottom_right_2y ) return false;
        if ( top_left_2y < bottom_right_1y ) return false;

        return true;
}

/* boundary limit function */
double boundary ( double value )
{
        double value_max = MOMENTUM_MAX;
        double value_min = MOMENTUM_MIN;

        if ( value > value_max )
                return value_max;
        else if ( value < value_min )
                return value_min;

        return value;
}

/* ---------------- ALLEGRO FUNCTIONS ----------------------------------------- */

/* initialises the stock allegro display with antialiasing functionality */
void disp_init ( )
{
        al_set_new_display_option ( ALLEGRO_SAMPLE_BUFFERS, 1,
                ALLEGRO_SUGGEST );

        al_set_new_display_option ( ALLEGRO_SAMPLES, 8,
                ALLEGRO_SUGGEST );

        disp = al_create_display ( DISPLAY_W, DISPLAY_H );
        must_init ( disp, "display" );
}

/* Initialises audio and loads the background song onto the buffer */
void audio_init ( )
{
        al_install_audio ( );
        al_init_acodec_addon ( );
        al_reserve_samples ( 1 );

        song = al_load_sample ( "RYDEEN.ogg" );

        songInstance = al_create_sample_instance ( song );

        al_set_sample_instance_playmode ( songInstance,
                ALLEGRO_PLAYMODE_ONCE );

        al_attach_sample_instance_to_mixer ( songInstance,
                al_get_default_mixer ( ) );
}

/* triggers the loaded sample */
void sample_trigger ( )
{
        al_play_sample_instance ( songInstance );
}

/* destroys the audio sample */
void audio_destroy ( )
{
        al_destroy_sample ( song );
        al_destroy_sample_instance ( songInstance );
}

/* clearing the keyboard array */
void keyboard_init ( )
{
        memset ( key, 0, sizeof ( key ) );
}

/* updates the state of the keyboard while the game is being played */
void keyboard_update ( ALLEGRO_EVENT* event )
{
        switch ( event -> type )
        {
                case ALLEGRO_EVENT_TIMER:
                        for ( int i = 0; i < ALLEGRO_KEY_MAX; i++ )
                                key [ i ] &= KEY_SEEN;
                        break;

                case ALLEGRO_EVENT_KEY_DOWN:
                        key [ event -> keyboard.keycode ]
                                = KEY_SEEN | KEY_RELEASED;
                        break;

                case ALLEGRO_EVENT_KEY_UP:
                        key [ event -> keyboard.keycode ] &= KEY_RELEASED;
                        break;
                default:
                        break;
        }
}

/* -------------------------- FONT FUNCTIONS ---------------------------------- */

/* initialises the score of the game */
void score_init ( )
{
        al_init_font_addon ( );
        al_init_ttf_addon ( );
        font = al_load_font ( "game.ttf", 35, 0 );
        score_display = 0;
}

/* draws the current score of the game and displays the final score
 * when time is up. Then it waits a certain number of frames before
 * the game exits */
void score_draw ( )
{
        /* draws the current score */
        al_draw_textf ( font, al_map_rgb ( 0xFF, 0xFF, 0xFF ),
                0, 0, 0, "$%02ld", score_display );

        /* tells you the game is over and exits the game after
         * a certain number of frames */
        if ( frame > 19e3 )
        {
                al_draw_text ( font, al_map_rgb_f ( 1,1,1 ),
                DISPLAY_W / 2, DISPLAY_H / 2, ALLEGRO_ALIGN_CENTER,
                "TIME'S UP" );
                redraw = false;

                al_draw_textf ( font, al_map_rgb_f ( 1,1,1 ),
                DISPLAY_W / 2, DISPLAY_H / 1.5, ALLEGRO_ALIGN_CENTER,
                "YOU'VE WON %ld Pelah coins", score_display );

                redraw = false;

                if ( frame > 21e3 )
                        done = true;
        }
}

/* displays the instructions for the game before the game starts */
void instruction_draw ( )
{
```

```
         al_draw_text ( font, al_map_rgb_f ( 1,1,1 ),
                DISPLAY_W / 2, 20, ALLEGRO_ALIGN_CENTER,
               "Get the blue token to collide with the yellow token" );
        al_draw_text ( font, al_map_rgb_f ( 1,1,1 ),
                DISPLAY_W / 2, 120, ALLEGRO_ALIGN_CENTER,
                "Use the arrow keys and space bar to set the initial" );

        al_draw_text ( font, al_map_rgb_f ( 1,1,1 ),
                DISPLAY_W / 2, 220, ALLEGRO_ALIGN_CENTER,
                "position of the blue token. Use the momentum of the" );

        al_draw_text ( font, al_map_rgb_f ( 1,1,1 ),
                DISPLAY_W / 2, 320, ALLEGRO_ALIGN_CENTER,
                "green token to affect the trajectory of the blue token." );

        al_draw_text ( font, al_map_rgb_f ( 1,1,1 ),
                DISPLAY_W / 2, 420, ALLEGRO_ALIGN_CENTER,
                "Avoid the purple anti-tokens! For every successful" );

        al_draw_text ( font, al_map_rgb_f ( 1,1,1 ),
                DISPLAY_W / 2, 520, ALLEGRO_ALIGN_CENTER,
                "hit , you will receive one Pelah dollar. All clear?" );

        al_draw_text ( font, al_map_rgb_f ( 1,1,1 ),
                DISPLAY_W / 2, 620, ALLEGRO_ALIGN_CENTER,
                "Press Y to start!" );
}

/* ---------------- ANTI_TOKEN FUNCTIONS -------------------------------------- */

/* setting initial conditions for the anti-token */
void anti_token_init ( )
{
        for ( int i = 0; i < AT_N; i++ )
        {
                anti_token [ i ].x = between ( AT_R, DISPLAY_W - AT_R );
                anti_token [ i ].y = between ( AT_R, DISPLAY_H - AT_R );
                anti_token [ i ].momentum_x = 1;
                anti_token [ i ].momentum_y = 1;
                anti_token [ i ].force_x = 1;
                anti_token [ i ].force_y = 1;
                anti_token [ i ].stiff = 5e11;
                anti_token [ i ].kg = 100;
                anti_token [ i ].r = 255;
                anti_token [ i ].g = 0;
                anti_token [ i ].b = 255;
                anti_token [ i ].visible = false;
        }
        for ( int i = 0; i < 3; i++ )
                anti_token [ i ].visible = true;
}

/* updating the positions of the anti-token */
void anti_token_update ( )
{
        /* setting the time interval for the Euler Cromer Method */
        const long double dt = 1e-7;

        /* animation of the anti-tokens based on simple harmonic motion */
        for ( int i = 0; i < AT_N; i++ )
        {
                anti_token [ i ].force_x = -( anti_token [ i ].stiff )
                        * ( ( anti_token [ i ].x ) - DISPLAY_H / 2 );

                anti_token [ i ].momentum_x += anti_token [ i ].force_x * dt;

                anti_token [ i ].x += ( anti_token [ i ].momentum_x
                        / ( anti_token [ i ].kg ) ) * dt;

                anti_token [ i ].force_y = - ( anti_token [ i ].stiff )
                        * ( ( anti_token [ i ].y ) - DISPLAY_H / 2 );

                anti_token [ i ].momentum_y += anti_token [ i ].force_y * dt;

                anti_token [ i ].y += ( anti_token [ i ].momentum_y
```

```c
                        / ( anti_token [ i ].kg ) ) * dt;
        }

        /* flips the momentum of the anti-tokens whenever
         * they collide with the display boundaries */
        for ( int i = 0; i < AT_N; i++ )
        {
                if ( anti_token [ i ].x > DISPLAY_W - AT_R )
                        anti_token [ i ].momentum_x *= -1;
                if ( anti_token [ i ].x < AT_R )
                        anti_token [ i ].momentum_x *= -1;
                if ( anti_token [ i ].y > DISPLAY_H - AT_R )
                        anti_token [ i ].momentum_y *= -1;
                if ( anti_token [ i ].y < AT_R )
                        anti_token [ i ].momentum_y *= -1;
        }
}

/* draws the anti-tokens in their current positions */
void anti_token_draw ( )
{
        for ( int i = 0; i < AT_N; i++ )
                if ( anti_token [ i ].visible == true )
                        al_draw_filled_circle (
                                anti_token [ i ].x,
                                anti_token [ i ].y,
                                AT_R,
                                al_map_rgb (
                                        anti_token [ i ].r,
                                        anti_token [ i ].g,
                                        anti_token [ i ].b
                                )
                        );
}

/* -------------- BLUE TOKEN FUNCTIONS ---------------------------------------- */

/* setting initial conditions of the blue token */
void blue_token_init ( )
{
        b_token.live = false;
        b_token.x = arrow.x;
        b_token.y = arrow.y;
        b_token.momentum_x = 1;
        b_token.momentum_y = 1;
        b_token.force_x = 1;
        b_token.force_y = 1;
        b_token.x_hat = 1;
        b_token.y_hat = 1;
        b_token.kg= 1e21;
        b_token.r = 0;
        b_token.g = 0;
        b_token.b = 255;
}

/* releases the blue token at the current coordinates of the arrow */
void blue_token_trigger ( )
{
        if ( !b_token.live )
        {
                b_token.x = arrow.x;
                b_token.y = arrow.y;
                b_token.live = true;
        }
}

/* updates the position of the blue token */
void blue_token_update ( )
{
        /* adding time interval and position
         * vectors for the Euler Cromer Method */
        static double dt = 1e-5;
        double r1_x, r1_y;


        /* updates the position of the blue token with respect to the
         * green token using the Euler Cromer Method */
        if ( b_token.live )
        {
                r1_x = b_token.x - g_token.x;
                r1_y = b_token.y - g_token.y;

                b_token.x_hat = r1_x / vector_mag ( r1_x, r1_y );
                b_token.y_hat = r1_y / vector_mag ( r1_x, r1_y );

                b_token.force_x = G * ( b_token.kg * g_token.kg /
                        pow ( vector_mag ( r1_x, r1_y ), 2 ) )
                        * ( -b_token.x_hat );

                b_token.force_y = G * ( b_token.kg * g_token.kg /
                        pow ( vector_mag ( r1_x, r1_y ), 2 ) )
                        * ( -b_token.y_hat );

                b_token.momentum_x += ( boundary ( b_token.force_x ) ) * dt;

                b_token.momentum_y += ( boundary ( b_token.force_y ) ) * dt;

                b_token.x += ( b_token.momentum_x / b_token.kg ) * dt;
                b_token.y += ( b_token.momentum_y / b_token.kg ) * dt;


                /* flips the momentum of the blue token whenever it collides
                 * with the boundaries of the display */
                if ( b_token.x > DISPLAY_W - BT_R )
                        b_token.momentum_x *= -1;
                if ( b_token.x < BT_R )
                        b_token.momentum_x *= -1;
                if ( b_token.y > DISPLAY_H - BT_R )
                        b_token.momentum_y *= -1;
                if ( b_token.y < BT_R )
                        b_token.momentum_y *= -1;

                /* if the blue token collides with the anti-token, the
                 * anti-token dissapears and the blue token returns to the
                 * current coordinates of the arrow */
                for ( int i = 0; i < AT_N; i++ )
                {
                        if ( collide ( b_token.x - BT_R, b_token.y - BT_R,
                                b_token.x + BT_R, b_token.y + BT_R,
                                anti_token[i].x - AT_R, anti_token[i].y - AT_R,
                                anti_token[i].x + AT_R, anti_token[i].y + AT_R )
                                && anti_token[i].visible == true )
                        {
                                b_token.live = false;
                                b_token.momentum_x = 0;
                                b_token.momentum_y = 0;
                                anti_token[i].visible = false;
                        }
                }
                /* if the blue token is in contact with the tip of the arrow,
                 * it will stick to it */
                if ( collide ( b_token.x - BT_R, b_token.y - BT_R,
                        b_token.x + BT_R, b_token.y + BT_R,
                        arrow.x, arrow.y,
                        arrow.x, arrow.y ) && frame > ++hold_frame )
                {
                        b_token.live = false;
                        b_token.momentum_x = 0;
                        b_token.momentum_y = 0;
                }
        }
}

/* draws the current position of the blue token */
void blue_token_draw ( )
{
        if ( b_token.live )
                al_draw_filled_circle ( b_token.x, b_token.y, BT_R,
                        al_map_rgb ( b_token.r, b_token.g, b_token.b ) );
}
```

```

/* ---------------- GREEN TOKEN FUNCTIONS ------------------------------------- */

/* sets the initial conditions of the green token */
void green_token_init ( )
{

        g_token.x = between ( GT_R, DISPLAY_W - GT_R );
        g_token.y = between ( DISPLAY_H >> 1, DISPLAY_H - GT_R );
        g_token.momentum_x = 1;
        g_token.momentum_y = 1;
        g_token.force_x = 1;
        g_token.force_y = 1;
        g_token.x_hat = 1;
        g_token.y_hat = 1;
        g_token.kg = 5e22;
        g_token.r = 0;
        g_token.g = 255;
        g_token.b = 0;

}
/* updates the position of the green token */
void green_token_update ( )
{
        /* setting the time interval for the Euler-Cromer Method */
        const double dt = 0.00001;

        g_token.momentum_x += ( boundary ( -b_token.force_x ) ) * dt;
        g_token.momentum_y += ( boundary ( -b_token.force_y ) ) * dt;
        g_token.x += ( g_token.momentum_x / g_token.kg ) * dt;
        g_token.y += ( g_token.momentum_y / g_token.kg ) * dt;

        /* flips the momentum of the green token whenever it collides
         * with the boundaries of the display screen */
        if ( g_token.x > DISPLAY_W - GT_R )
                g_token.momentum_x *= -1;
        if ( g_token.x < GT_R )
                g_token.momentum_x *= -1;
        if ( g_token.y > DISPLAY_H - GT_R )
                g_token.momentum_y *= -1;
        if ( g_token.y < GT_R )
                g_token.momentum_y *= -1;
}

/* draws the green token at its current position */
void green_token_draw ( )
{
        al_draw_filled_circle ( g_token.x, g_token.y, GT_R,
                al_map_rgb ( g_token.r, g_token.g, g_token.b ) );
}

/* -------------- YELLOW TOKEN FUNCTIONS -------------------------------------- */

/* sets the initial conditions of the yellow token */
void yellow_token_init ( )
{
        y_token.x = between ( YT_R, DISPLAY_W );
        y_token.y = between ( YT_R, DISPLAY_H >> 2 );
        y_token.r = 0xFF, y_token.g = 0xDF, y_token.b = 0x00;
}

/* if the blue token collides with the yellow token then the location of the new
 * yellow token and green token changes. Whenever the yellow token is hit, a new
 * anti-token is generated */
void yellow_token_update ( )
{
        /* checks whether the blue token collides with the yellow token */
        if ( collide ( y_token.x - YT_R, y_token.y - YT_R, y_token.x + YT_R,
                y_token.y + YT_R, b_token.x - BT_R, b_token.y + BT_R,
                b_token.x + BT_R, b_token.y + BT_R ) ) {
                        y_token.x = between ( YT_R, DISPLAY_W );
                        y_token.y = between ( YT_R, DISPLAY_H >> 2 );
                        score_display++;
                        green_token_init ( );
                        /* Generates an anti-token whenever
```

```c
                         * the blue token collides with the yellow token */
                        for ( int i = 0; i < AT_N; i++ )
                        {
                                if ( anti_token[i].visible == false )
                                {
                                        anti_token[i].visible = true;
                                        break;
                                }
                        }
        }
}

/* draws the current position of the yellow token */
void yellow_token_draw ( )
{
        al_draw_filled_circle ( y_token.x, y_token.y, YT_R,
        al_map_rgb ( y_token.r, y_token.g, y_token.b ) );
}
/* ---------------------- ARROW FUNCTIONS ------------------------------------- */

/* initialising the position of the arrow */
void arrow_init ( )
{
        arrow.x = 0;
        arrow.y = 0;
        arrow.theta = PI / 2;
        arrow.mag = 25;
        arrow.r = 0xFF;
        arrow.g = 0xFF;
        arrow.b = 0xFF;
}

/* updates the position of the arrow according to the key commands
 * and releases the blue token whenever the space bar is pressed */
void arrow_update ( )
{
        if ( key [ ALLEGRO_KEY_LEFT ] )
                arrow.theta += 0.025;

        if ( key [ ALLEGRO_KEY_RIGHT ] )
                arrow.theta -= 0.025;

        if ( key [ ALLEGRO_KEY_UP ] )
                arrow.mag += ARROW_SPEED;

        if ( key [ ALLEGRO_KEY_DOWN ] )
                arrow.mag -= ARROW_SPEED;

        arrow.x = ( DISPLAY_W >> 1 ) + ( arrow.mag  * cos ( arrow.theta ) );
        arrow.y = ( DISPLAY_H ) - ( arrow.mag * sin ( arrow.theta ) );

        if ( arrow.theta < 0 )
                arrow.theta = 0;

        if ( arrow.mag < 0 )
                arrow.mag = 0;

        if ( arrow.theta > PI )
                arrow.theta = PI;

        if ( arrow.mag > ( DISPLAY_H / sqrt ( 2 ) ) )
                arrow.mag = ( DISPLAY_H / sqrt ( 2 ) );

        if ( key [ ALLEGRO_KEY_SPACE ] )
        {
                blue_token_trigger ( );
                hold_frame = frame;
        }
}

/* draws the current position of the arrow */
void arrow_draw ( )
{
        al_draw_line ( DISPLAY_W / 2, DISPLAY_H, arrow.x, arrow.y,
                al_map_rgb ( arrow.r, arrow.g, arrow.b ), 15 );

        if ( !b_token.live )
                al_draw_filled_circle ( arrow.x, arrow.y, BT_R,
                        al_map_rgb ( b_token.r, b_token.g, b_token.b ) );
}

/* ---- STARS FUNCTIONS ( modified from Allegro Vivace tutorial )
 * https://github.com/liballeg/allegro_wiki/wiki/Allegro-Vivace --------- */

/* initialises the positions of the stars */
void stars_init ( )
{
        for ( int i = 0; i < ST_N; i++ )
        {
                star [ i ].y = between_f ( 0, DISPLAY_H );
                star [ i ].speed = between_f ( 0.1, 1 );
        }
}

/* updates the position of the stars */
void stars_update ( )
{
        for (int i = 0; i < ST_N; i++ )
        {
                star [ i ].y += star [ i ].speed;
                if ( star [ i ].y >= DISPLAY_H )
                {
                        star [ i ].y = 0;
                        star [ i ].speed = between_f ( 0.1, 1 );
                }
        }
}

/* draws the stars at their current position */
void stars_draw ( )
{
        const float step_x = 6;
        const float speed_factor = 2.4;
        float star_x = 4.5;

        for ( int i = 0; i < ST_N; i++ )
        {
                float l = star [ i ].speed * speed_factor;
                al_draw_filled_circle ( star_x, star [ i ].y, 2,
                        al_map_rgb_f ( l, l, l ) );

                star_x += step_x;
        }
}


/* ---END------------------------------------------------------------------ */

```