main.c

```
1  /* WELCOME TO THE PELAVERSE */
2  /* please use Linux Code::Blocks */
3
4  /* header files that call allegro libraries */
5  #include <allegro5/allegro5.h>
6  #include <allegro5/allegro_font.h>
7  #include <allegro5/allegro_ttf.h>
8  #include <allegro5/allegro_primitives.h>
9  #include <allegro5/allegro_audio.h>
10 #include <allegro5/allegro_acodec.h>
11 #include <allegro5/allegro_image.h>
12
13 /* header files for C library */
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <math.h>
17 #include <time.h>
18
19 /* header file for the game */
20 #include "main.h"
21
22 /* declaring and initialising global variables */
23 int frame = 0;
24 int hold_frame = 0;
25 bool music = false;
26 bool done = false;
27 bool redraw = true;
28 bool menu = true;
29 unsigned char key [ ALLEGRO_KEY_MAX ];
30 long score_display;
31
32 /* declaring and initialising structs. */
33 arrow_t arrow;
34 anti_token_t anti_token [ AT_N ];
35 blue_token_t b_token;
36 green_token_t g_token;
37 yellow_token_t y_token;
38 star_t star [ ST_N ];
39
40 /* initialising Allegro's native structs. */
41 ALLEGRO_DISPLAY * disp;
42 ALLEGRO_SAMPLE * song = NULL;
43 ALLEGRO_SAMPLE_INSTANCE * songInstance = NULL;
44 ALLEGRO_FONT * font;
45
46
47 /* ----------------- MAIN PROGRAM----------------------------------------- */
48 int main ( )
49 {
50         /* seeds the pseudo-random generator */
51         srand ( time ( NULL ) );
52
53         /* starts the program */
54         must_init ( al_init( ), "allegro" );
55
56         /* activates the keyboard peripheral */
57         must_init ( al_install_keyboard( ), "keyboard" );
58
59         /* Initialises the timer */
60         ALLEGRO_TIMER* timer = al_create_timer ( 1.0 / 120 );
61         must_init ( timer, "timer" );
62
63         /* initialises the event queue */
64         ALLEGRO_EVENT_QUEUE* queue = al_create_event_queue ( );
65         must_init ( queue, "queue" );
66
67         /* initialises allegro stock primitive functions */
68         must_init ( al_init_primitives_addon ( ), "primitives" );
69
70         /* audio initialiasation */
71         audio_init ( );
72
73         /* initialises native allegro display functions */
74         disp_init ( );
75
```

```
76          /* initialises native allegro keyboard functions */
77          keyboard_init ( );
78
79          /* initialises other custom functions */
80          score_init ( );
81          arrow_init ( );
82          anti_token_init ( );
83          blue_token_init ( );
84          green_token_init ( );
85          yellow_token_init ( );
86          stars_init ( );
87
88          /* make display, keyboard and timer a part of the program */
89          al_register_event_source ( queue,
90                  al_get_keyboard_event_source ( ) );
91
92          al_register_event_source( queue,
93                  al_get_display_event_source ( disp ) );
94
95          al_register_event_source ( queue,
96                  al_get_timer_event_source ( timer ) );
97
98          /* serves as continuous input for al_register_event_source */
99          ALLEGRO_EVENT event;
100
101         /* start timer before game begins */
102         al_start_timer( timer );
103
104         /* main game loop */
105         while ( true )
106         {
107                 al_wait_for_event ( queue, &event );
108
109                 /* makes sure the frame will either advance
110                    according to specified frames per second
111                    or will stop advancing and exit the program */
112                 switch( event.type )
113                 {
114                         case ALLEGRO_EVENT_TIMER:
115
116                                 /* updates all the elements
117                                  * in the game */
118                                 arrow_update ( );
119                                 anti_token_update ( );
120                                 green_token_update ( );
121                                 blue_token_update ( );
122                                 yellow_token_update ( );
123                                 stars_update ( );
124
125                                 /* Condition to exit the program */
126                                 if ( key [ ALLEGRO_KEY_ESCAPE ] )
127                                         done = true;
128
129                                 /* condition to start the game */
130                                 if ( key [ ALLEGRO_KEY_Y ] )
131                                 {
132                                         frame = 0;
133                                         menu = false;
134                                 }
135
136                                 redraw = true;
137                                 frame++;
138                                 break;
139
140                         /* Condition to exit the program */
141                         case ALLEGRO_EVENT_DISPLAY_CLOSE:
142
143                                 done = true;
144                                 break;
145                 }
146                 /* exits the program */
147                 if ( done )
148                         break;
149
150                 keyboard_update ( &event );
```

```
151
152                 /* if frame advances this conditional branch will execute */
153                 if ( redraw && al_is_event_queue_empty ( queue ) )
154                 {
155                         /* clears the screen */
156                         al_clear_to_color ( al_map_rgb ( 0, 0, 0 ) );
157
158                         /* making sure game does not start until
159                          * y is pressed */
160                         if ( menu == true )
161                         {
162                                 instruction_draw ( );
163                                 stars_draw ( );
164                         }
165                         else
166                         {
167                                 /* starts the music when playing */
168                                 if ( !music )
169                                 {
170                                         sample_trigger ( );
171                                         music = true;
172                                 }
173
174                                 /* draws all the objects onto the screen */
175                                 score_draw ( );
176                                 arrow_draw ( );
177                                 anti_token_draw ( );
178                                 blue_token_draw ( );
179                                 green_token_draw ( );
180                                 yellow_token_draw ( );
181                                 stars_draw ( );
182                         }
183                         al_flip_display ( );
184                         redraw = false;
185                 }
186         }
187
188         /* destroys all the stock allegro functions */
189         audio_destroy ( );
190         al_destroy_font ( font );
191         al_destroy_display ( disp );
192         al_destroy_timer ( timer );
193         al_destroy_event_queue ( queue );
194
195         return EXIT_SUCCESS;
196 }
197
198 /* ----------------- GENERAL FUNCTIONS ------------------------------------- */
199
200 /* tests each of allegro's stock initialisation functions */
201 void must_init ( bool test, const char *description )
202 {
203         if ( !test ) {
204             printf ( "couldn't initialise %s\n", description );
205             exit( 1 );
206         }
207 }
208
209 /* outputs an integer between a range of two values */
210 int between ( int min, int max )
211 {
212         return min + ( rand( ) % ( max - min ) );
213 }
214
215 /* outputs a rational number between 0 and 1 */
216 double between_f ( double min, double max )
217 {
218         return min + ( ( double ) rand ( ) / ( float ) RAND_MAX )
219                 * ( max - min );
220 }
221 /* calculates the magnitude of a 2D vector */
222 double vector_mag ( double x, double y )
223 {
224         x *= x;
225         y *= y;
```

```
226
227         return sqrt ( x + y );
228 }
229
230 /* collision detection using square boundaries for each token */
231 bool collide ( int top_left_1x, int top_left_1y,
232                int top_left_2x, int top_left_2y,
233                int bottom_right_1x, int bottom_right_1y,
234                int bottom_right_2x, int bottom_right_2y )
235 {
236         if ( top_left_1x > bottom_right_2x ) return false;
237         if ( top_left_2x < bottom_right_1x ) return false;
238         if ( top_left_1y > bottom_right_2y ) return false;
239         if ( top_left_2y < bottom_right_1y ) return false;
240
241         return true;
242 }
243
244 /* boundary limit function */
245 double boundary ( double value )
246 {
247         double value_max = MOMENTUM_MAX;
248         double value_min = MOMENTUM_MIN;
249
250         if ( value > value_max )
251                 return value_max;
252         else if ( value < value_min )
253                 return value_min;
254
255         return value;
256 }
257
258 /* ---------------- ALLEGRO FUNCTIONS ---------------------------------------- */
259
260 /* initialises the stock allegro display with antialiasing functionality */
261 void disp_init ( )
262 {
263         al_set_new_display_option ( ALLEGRO_SAMPLE_BUFFERS, 1,
264                 ALLEGRO_SUGGEST );
265
266         al_set_new_display_option ( ALLEGRO_SAMPLES, 8,
267                 ALLEGRO_SUGGEST );
268
269         disp = al_create_display ( DISPLAY_W, DISPLAY_H );
270         must_init ( disp, "display" );
271 }
272
273 /* Initialises audio and loads the background song onto the buffer */
274 void audio_init ( )
275 {
276         al_install_audio ( );
277         al_init_acodec_addon ( );
278         al_reserve_samples ( 1 );
279
280         song = al_load_sample ( "RYDEEN.ogg" );
281
282         songInstance = al_create_sample_instance ( song );
283
284         al_set_sample_instance_playmode ( songInstance,
285                 ALLEGRO_PLAYMODE_ONCE );
286
287         al_attach_sample_instance_to_mixer ( songInstance,
288                 al_get_default_mixer ( ) );
289 }
290
291 /* triggers the loaded sample */
292 void sample_trigger ( )
293 {
294         al_play_sample_instance ( songInstance );
295 }
296
297 /* destroys the audio sample */
298 void audio_destroy ( )
299 {
300         al_destroy_sample ( song );
```

```
301         al_destroy_sample_instance ( songInstance );
302 }
303
304 /* clearing the keyboard array */
305 void keyboard_init ( )
306 {
307         memset ( key, 0, sizeof ( key ) );
308 }
309
310 /* updates the state of the keyboard while the game is being played */
311 void keyboard_update ( ALLEGRO_EVENT* event )
312 {
313         switch ( event -> type )
314         {
315                 case ALLEGRO_EVENT_TIMER:
316                         for ( int i = 0; i < ALLEGRO_KEY_MAX; i++ )
317                                 key [ i ] &= KEY_SEEN;
318                         break;
319
320                 case ALLEGRO_EVENT_KEY_DOWN:
321                         key [ event -> keyboard.keycode ]
322                                 = KEY_SEEN | KEY_RELEASED;
323                         break;
324
325                 case ALLEGRO_EVENT_KEY_UP:
326                         key [ event -> keyboard.keycode ] &= KEY_RELEASED;
327                         break;
328                 default:
329                         break;
330         }
331 }
332
333 /* --------------------------- FONT FUNCTIONS ---------------------------------- */
334
335 /* initialises the score of the game */
336 void score_init ( )
337 {
338         al_init_font_addon ( );
339         al_init_ttf_addon ( );
340         font = al_load_font ( "game.ttf", 35, 0 );
341         score_display = 0;
342 }
343
344 /* draws the current score of the game and displays the final score
345  * when time is up. Then it waits a certain number of frames before
346  * the game exits */
347 void score_draw ( )
348 {
349         /* draws the current score */
350         al_draw_textf ( font, al_map_rgb ( 0xFF, 0xFF, 0xFF ),
351                 0, 0, 0, "$%02ld", score_display );
352
353         /* tells you the game is over and exits the game after
354          * a certain number of frames */
355         if ( frame > 19e3 )
356         {
357                 al_draw_text ( font, al_map_rgb_f ( 1,1,1 ),
358                 DISPLAY_W / 2, DISPLAY_H / 2, ALLEGRO_ALIGN_CENTER,
359                 "TIME'S UP" );
360                 redraw = false;
361
362                 al_draw_textf ( font, al_map_rgb_f ( 1,1,1 ),
363                 DISPLAY_W / 2, DISPLAY_H / 1.5, ALLEGRO_ALIGN_CENTER,
364                 "YOU'VE WON %ld Pelah coins", score_display );
365
366                 redraw = false;
367
368                 if ( frame > 21e3 )
369                         done = true;
370         }
371 }
372
373 /* displays the instructions for the game before the game starts */
374 void instruction_draw ( )
375 {
```

```
            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
```

```
451                             / ( anti_token [ i ].kg ) ) * dt;
452         }
453
454         /* flips the momentum of the anti-tokens whenever
455          * they collide with the display boundaries */
456         for ( int i = 0; i < AT_N; i++ )
457         {
458                 if ( anti_token [ i ].x > DISPLAY_W - AT_R )
459                         anti_token [ i ].momentum_x *= -1;
460                 if ( anti_token [ i ].x < AT_R )
461                         anti_token [ i ].momentum_x *= -1;
462                 if ( anti_token [ i ].y > DISPLAY_H - AT_R )
463                         anti_token [ i ].momentum_y *= -1;
464                 if ( anti_token [ i ].y < AT_R )
465                         anti_token [ i ].momentum_y *= -1;
466         }
467 }
468
469 /* draws the anti-tokens in their current positions */
470 void anti_token_draw ( )
471 {
472         for ( int i = 0; i < AT_N; i++ )
473                 if ( anti_token [ i ].visible == true )
474                         al_draw_filled_circle (
475                                 anti_token [ i ].x,
476                                 anti_token [ i ].y,
477                                 AT_R,
478                                 al_map_rgb (
479                                         anti_token [ i ].r,
480                                         anti_token [ i ].g,
481                                         anti_token [ i ].b
482                                 )
483                         );
484 }
485
486 /* -------------- BLUE TOKEN FUNCTIONS ------------------------------------- */
487
488 /* setting initial conditions of the blue token */
489 void blue_token_init ( )
490 {
491         b_token.live = false;
492         b_token.x = arrow.x;
493         b_token.y = arrow.y;
494         b_token.momentum_x = 1;
495         b_token.momentum_y = 1;
496         b_token.force_x = 1;
497         b_token.force_y = 1;
498         b_token.x_hat = 1;
499         b_token.y_hat = 1;
500         b_token.kg= 1e21;
501         b_token.r = 0;
502         b_token.g = 0;
503         b_token.b = 255;
504 }
505
506 /* releases the blue token at the current coordinates of the arrow */
507 void blue_token_trigger ( )
508 {
509         if ( !b_token.live )
510         {
511                 b_token.x = arrow.x;
512                 b_token.y = arrow.y;
513                 b_token.live = true;
514         }
515 }
516
517 /* updates the position of the blue token */
518 void blue_token_update ( )
519 {
520         /* adding time interval and position
521          * vectors for the Euler Cromer Method */
522         static double dt = 1e-5;
523         double r1_x, r1_y;
524
525
```

```
526         /* updates the position of the blue token with respect to the
527          * green token using the Euler Cromer Method */
528         if ( b_token.live )
529         {
530                 r1_x = b_token.x - g_token.x;
531                 r1_y = b_token.y - g_token.y;
532
533                 b_token.x_hat = r1_x / vector_mag ( r1_x, r1_y );
534                 b_token.y_hat = r1_y / vector_mag ( r1_x, r1_y );
535
536                 b_token.force_x = G * ( b_token.kg * g_token.kg /
537                         pow ( vector_mag ( r1_x, r1_y ), 2 ) )
538                         * ( -b_token.x_hat );
539
540                 b_token.force_y = G * ( b_token.kg * g_token.kg /
541                         pow ( vector_mag ( r1_x, r1_y ), 2 ) )
542                         * ( -b_token.y_hat );
543
544                 b_token.momentum_x += ( boundary ( b_token.force_x ) ) * dt;
545
546                 b_token.momentum_y += ( boundary ( b_token.force_y ) ) * dt;
547
548                 b_token.x += ( b_token.momentum_x / b_token.kg ) * dt;
549                 b_token.y += ( b_token.momentum_y / b_token.kg ) * dt;
550
551
552                 /* flips the momentum of the blue token whenever it collides
553                  * with the boundaries of the display */
554                 if ( b_token.x > DISPLAY_W - BT_R )
555                         b_token.momentum_x *= -1;
556                 if ( b_token.x < BT_R )
557                         b_token.momentum_x *= -1;
558                 if ( b_token.y > DISPLAY_H - BT_R )
559                         b_token.momentum_y *= -1;
560                 if ( b_token.y < BT_R )
561                         b_token.momentum_y *= -1;
562
563                 /* if the blue token collides with the anti-token, the
564                  * anti-token dissapears and the blue token returns to the
565                  * current coordinates of the arrow */
566                 for ( int i = 0; i < AT_N; i++ )
567                 {
568                         if ( collide ( b_token.x - BT_R, b_token.y - BT_R,
569                                 b_token.x + BT_R, b_token.y + BT_R,
570                                 anti_token[i].x - AT_R, anti_token[i].y - AT_R,
571                                 anti_token[i].x + AT_R, anti_token[i].y + AT_R )
572                                 && anti_token[i].visible == true )
573                         {
574                                 b_token.live = false;
575                                 b_token.momentum_x = 0;
576                                 b_token.momentum_y = 0;
577                                 anti_token[i].visible = false;
578                         }
579                 }
580                 /* if the blue token is in contact with the tip of the arrow,
581                  * it will stick to it */
582                 if ( collide ( b_token.x - BT_R, b_token.y - BT_R,
583                         b_token.x + BT_R, b_token.y + BT_R,
584                         arrow.x, arrow.y,
585                         arrow.x, arrow.y ) && frame > ++hold_frame )
586                 {
587                         b_token.live = false;
588                         b_token.momentum_x = 0;
589                         b_token.momentum_y = 0;
590                 }
591         }
592 }
593
594 /* draws the current position of the blue token */
595 void blue_token_draw ( )
596 {
597         if ( b_token.live )
598                 al_draw_filled_circle ( b_token.x, b_token.y, BT_R,
599                         al_map_rgb ( b_token.r, b_token.g, b_token.b ) );
600 }
```

```
601
602 /* ---------------- GREEN TOKEN FUNCTIONS ------------------------------------ */
603
604 /* sets the initial conditions of the green token */
605 void green_token_init ( )
606 {
607
608         g_token.x = between ( GT_R, DISPLAY_W - GT_R );
609         g_token.y = between ( DISPLAY_H >> 1, DISPLAY_H - GT_R );
610         g_token.momentum_x = 1;
611         g_token.momentum_y = 1;
612         g_token.force_x = 1;
613         g_token.force_y = 1;
614         g_token.x_hat = 1;
615         g_token.y_hat = 1;
616         g_token.kg = 5e22;
617         g_token.r = 0;
618         g_token.g = 255;
619         g_token.b = 0;
620
621 }
622 /* updates the position of the green token */
623 void green_token_update ( )
624 {
625         /* setting the time interval for the Euler-Cromer Method */
626         const double dt = 0.00001;
627
628         g_token.momentum_x += ( boundary ( -b_token.force_x ) ) * dt;
629         g_token.momentum_y += ( boundary ( -b_token.force_y ) ) * dt;
630         g_token.x += ( g_token.momentum_x / g_token.kg ) * dt;
631         g_token.y += ( g_token.momentum_y / g_token.kg ) * dt;
632
633         /* flips the momentum of the green token whenever it collides
634          * with the boundaries of the display screen */
635         if ( g_token.x > DISPLAY_W - GT_R )
636                 g_token.momentum_x *= -1;
637         if ( g_token.x < GT_R )
638                 g_token.momentum_x *= -1;
639         if ( g_token.y > DISPLAY_H - GT_R )
640                 g_token.momentum_y *= -1;
641         if ( g_token.y < GT_R )
642                 g_token.momentum_y *= -1;
643 }
644
645 /* draws the green token at its current position */
646 void green_token_draw ( )
647 {
648         al_draw_filled_circle ( g_token.x, g_token.y, GT_R,
649                 al_map_rgb ( g_token.r, g_token.g, g_token.b ) );
650 }
651
652 /* -------------- YELLOW TOKEN FUNCTIONS ----------------------------------- */
653
654 /* sets the initial conditions of the yellow token */
655 void yellow_token_init ( )
656 {
657         y_token.x = between ( YT_R, DISPLAY_W );
658         y_token.y = between ( YT_R, DISPLAY_H >> 2 );
659         y_token.r = 0xFF, y_token.g = 0xDF, y_token.b = 0x00;
660 }
661
662 /* if the blue token collides with the yellow token then the location of the new
663  * yellow token and green token changes. Whenever the yellow token is hit, a new
664  * anti-token is generated */
665 void yellow_token_update ( )
666 {
667         /* checks whether the blue token collides with the yellow token */
668         if ( collide ( y_token.x - YT_R, y_token.y - YT_R, y_token.x + YT_R,
669                 y_token.y + YT_R, b_token.x - BT_R, b_token.y + BT_R,
670                 b_token.x + BT_R, b_token.y + BT_R ) ) {
671                         y_token.x = between ( YT_R, DISPLAY_W );
672                         y_token.y = between ( YT_R, DISPLAY_H >> 2 );
673                         score_display++;
674                         green_token_init ( );
675                         /* Generates an anti-token whenever
```

```
676                              * the blue token collides with the yellow token */
677                             for ( int i = 0; i < AT_N; i++ )
678                             {
679                                     if ( anti_token[i].visible == false )
680                                     {
681                                             anti_token[i].visible = true;
682                                             break;
683                                     }
684                             }
685             }
686     }
687
688     /* draws the current position of the yellow token */
689     void yellow_token_draw ( )
690     {
691             al_draw_filled_circle ( y_token.x, y_token.y, YT_R,
692             al_map_rgb ( y_token.r, y_token.g, y_token.b ) );
693     }
694     /* ---------------------- ARROW FUNCTIONS ----------------------------------- */
695
696     /* initialising the position of the arrow */
697     void arrow_init ( )
698     {
699             arrow.x = 0;
700             arrow.y = 0;
701             arrow.theta = PI / 2;
702             arrow.mag = 25;
703             arrow.r = 0xFF;
704             arrow.g = 0xFF;
705             arrow.b = 0xFF;
706     }
707
708     /* updates the position of the arrow according to the key commands
709      * and releases the blue token whenever the space bar is pressed */
710     void arrow_update ( )
711     {
712             if ( key [ ALLEGRO_KEY_LEFT ] )
713                     arrow.theta += 0.025;
714
715             if ( key [ ALLEGRO_KEY_RIGHT ] )
716                     arrow.theta -= 0.025;
717
718             if ( key [ ALLEGRO_KEY_UP ] )
719                     arrow.mag += ARROW_SPEED;
720
721             if ( key [ ALLEGRO_KEY_DOWN ] )
722                     arrow.mag -= ARROW_SPEED;
723
724             arrow.x = ( DISPLAY_W >> 1 ) + ( arrow.mag  * cos ( arrow.theta ) );
725             arrow.y = ( DISPLAY_H ) - ( arrow.mag * sin ( arrow.theta ) );
726
727             if ( arrow.theta < 0 )
728                     arrow.theta = 0;
729
730             if ( arrow.mag < 0 )
731                     arrow.mag = 0;
732
733             if ( arrow.theta > PI )
734                     arrow.theta = PI;
735
736             if ( arrow.mag > ( DISPLAY_H / sqrt ( 2 ) ) )
737                     arrow.mag = ( DISPLAY_H / sqrt ( 2 ) );
738
739             if ( key [ ALLEGRO_KEY_SPACE ] )
740             {
741                     blue_token_trigger ( );
742                     hold_frame = frame;
743             }
744     }
745
746     /* draws the current position of the arrow */
747     void arrow_draw ( )
748     {
749             al_draw_line ( DISPLAY_W / 2, DISPLAY_H, arrow.x, arrow.y,
750                     al_map_rgb ( arrow.r, arrow.g, arrow.b ), 15 );
```

```
751
752         if ( !b_token.live )
753                 al_draw_filled_circle ( arrow.x, arrow.y, BT_R,
754                         al_map_rgb ( b_token.r, b_token.g, b_token.b ) );
755 }
756
757 /* ---- STARS FUNCTIONS ( modified from Allegro Vivace tutorial )
758  * https://github.com/liballeg/allegro_wiki/wiki/Allegro-Vivace --------- */
759
760 /* initialises the positions of the stars */
761 void stars_init ( )
762 {
763         for ( int i = 0; i < ST_N; i++ )
764         {
765                 star [ i ].y = between_f ( 0, DISPLAY_H );
766                 star [ i ].speed = between_f ( 0.1, 1 );
767         }
768 }
769
770 /* updates the position of the stars */
771 void stars_update ( )
772 {
773         for (int i = 0; i < ST_N; i++ )
774         {
775                 star [ i ].y += star [ i ].speed;
776                 if ( star [ i ].y >= DISPLAY_H )
777                 {
778                         star [ i ].y = 0;
779                         star [ i ].speed = between_f ( 0.1, 1 );
780                 }
781         }
782 }
783
784 /* draws the stars at their current position */
785 void stars_draw ( )
786 {
787         const float step_x = 6;
788         const float speed_factor = 2.4;
789         float star_x = 4.5;
790
791         for ( int i = 0; i < ST_N; i++ )
792         {
793                 float l = star [ i ].speed * speed_factor;
794                 al_draw_filled_circle ( star_x, star [ i ].y, 2,
795                         al_map_rgb_f ( l, l, l ) );
796
797                 star_x += step_x;
798         }
799 }
800
801
802 /* ---END---------------------------------------------------------------------- */
803
```