Real-Time Systems

Project Report

Simulating a Jumping Balls with Collisions

RAWAN BINFARDAN

AALA ALKHAYER

Introduction

In simulating Jumping Balls with Collisions, the following problems have to be taken into consideration:

- Collision detection of a ball to the wall.
- The collision handling of a ball to the wall.
- The collision detection of a ball to another ball.
- The collision handling of a ball to another ball.

General description

- Each ball is animated by the dedicated periodic task that updates its state every period.
- The user can interact with the system and create new balls.
- The user can drop an obstacle.
- The system provides information on the number of active balls (tasks).

User interface

The user can interact with the application with the following keys:

- SPACE: create a new ball
- Mouse click: create a new obstacle;
- ESC: exit the program.

Tasks description

The project includes the following main tasks:

Ball

Creates a ball whenever the user enters space and updates its state every period.

Collision

Detects possible crashes between one ball and another and between the ball and the obstacle.

Draw_obstcale

Draws the obstacle and updated its position.

Collision processing

In this section, collision processing is discussed. The first part discusses collision detection, while the second part discusses how the collision is handled once it is detected.

Collision Detection

Initially, the collision is detected by calculating the distance between all the balls in the scene. Since the diameter of the ball is equal, d, the minimum distance that is allowed between two balls without colliding is 2d, as shown in Figure below:

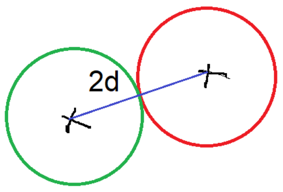


Figure 1: Colliding balls.

As Figure 1 illustrates, if the distance between the two balls is less than 2 d, the collision must be handled. The following equation shows how to find the distance between two balls:

$$distance = \sqrt{\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2} \qquad .... \; eq1$$

Collision handling

To handle the collision, several parameters must be taken in consideration:

1. The position of the two collided balls.
2. The direction of each ball.
3. The speed of the two balls.

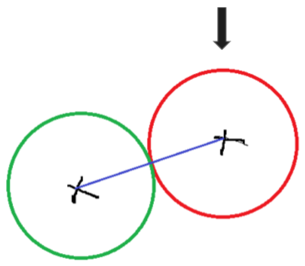


Figure 2: The first case of collision handling

The first case of collision handling processes the collision when one of the balls is positioned at one of the window borders. In this case, the ball positioned at the window border will not change its direction. However, if both balls are positioned away from the window border, then booth balls will change their direction as shown in the Figure below:

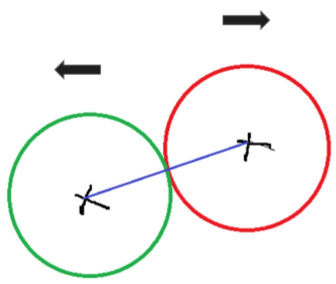


Figure 3: The second case of collision handling

Obstacles simulation

In this part, the obstacle task is discussed .at. First, the movement of the obstacle will be discussed. After that, handling collusion between the obstacle and any ball will be discussed. The user can only create one obstacle.

Obstacles movement:

The obstacle movement relays on the coordinates entered by the user. Therefore, whenever the mouse clicks, the obstacle will be drawn, and its coordinates will be updated.

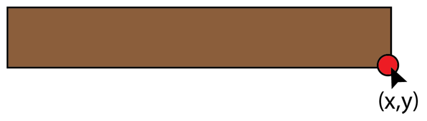


Figure 4: obstacle movement

Ball collusion with the Obstacle:

To detect collision with the obstacle, we defined four sides for the obstacle Up, Down, right, and left inside the obstacle struct. They are updated whenever the obstacle's coordinates are updated. Therefore, the program will calculate the distance between the ball centre and the four sides of the obstacle when the distance equals the ball's radius and the obstacle width, which means collision was detected. It has continuously processed whenever there is a drawn obstacle in the game.

Project Implementation

Allegro:

In this project, Allegro 4.4 was used to implement the graphic side and to design an user interface.

Header files

The project includes the following header files:

```
#include "pmutex.h"
#include "ptask.h"
#include "tstat.h"
#include <allegro.h>
#include <math.h>
#include <pthread.h>
#include <sched.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

Global data

In the project, several variables have been defined as global constant variables:

```
#define XWIN       720
#define YWIN       520
#define PI         3.1415
#define BGC        0
#define L          15
#define G          9.8
#define BASE       15
#define TOP        520
#define XMIN       15
#define XMAX       720
#define VELX       10.
#define VMIN       9.
#define PER        20
#define DREL       20
#define PRIO       80
#define DPER       12
```

The window border size is defined using the XWIN and YWIN variables. The background colour is set to black (BGC=0). The required physical quantities are also defined (PI= 3.1415) and the acceleration of gravity (G= 9.8). The dimension of the ball (L) is set to 15. The initial height of the ball (TOP) is 520. The minimum (XMIN) and maximum (XMAX) position of the x coordinate of the ball is set to 15 and 720, respectively. The horizontal ball velocity (VELX) is set to 10 whereas the minimum velocity (VMIN) is set to 9. The required variables for the scheduling are also defined. To define the task period in ms, the variable PER is set to 20. The priority of the ball task is set to 80. The priority of the drawing obstacle task is set to 12.

Some further variables have been utilized as it is shown:

```
float v0[MAX_TASKS];
int x_coor[MAX_TASKS];
int y_coor[MAX_TASKS];
```

The MAX_TASKS is set to 50 by the ptask library. So, the three abovementioned matrices are deployed to store specific information on each task. For instance, v0 matrix is used to define the impact velocity of the ball with the floor. However, the x_coor and the y_coor are used to store the x position and y position of each ball.

Init Function

The initial step in the Init function is to initialize the allegro environment. Subsequently, the window, the background, the keyboard, and the mouse are initialized.

```
allegro_init();
set_gfx_mode(GFX_AUTODETECT_WINDOWED, XWIN, YWIN, 0, 0);
clear_to_color(screen, BGC);
install_keyboard();
install_mouse();
show_mouse(screen);
srand(time(NULL));
```

Object Animation

1. Drawing approach

The ball is drawn using a simple circlefill function given its x and y coordinate as well as its colour as described below:

```
void draw_ball(int x, int y, int c) { circlefill(screen, x, y, L, c); }
```

While the obstacle is drawn using draw_obstaclel( ) function. it simply takes the mouse coordinates as (rx1,ry1) and then obtains the other point coordinates (rx2,ry2). The function will check the visibility of the obstacle that can tell if there is an old obstacle that needs to be erased or not, .then pass points by rectfill( ) function.

```
if ((ob.visible = 0)){
        pthread_mutex_lock(&mdraw);
        rectfill(screen, rx1, ry1, rx2, ry2, COLBROWN);
        pthread_mutex_unlock(&mdraw);
    }
    else{
        pthread_mutex_lock(&mdraw);
        rectfill(screen, rx01, ry01, rx02, ry02, COLBROWN);
        /* erase the old rectangle */
        rectfill(screen, rx01, ry01, rx02, ry02, BGC);
        rectfill(screen, rx1, ry1, rx2, ry2, COLBROWN);
        pthread_mutex_unlock(&mdraw);
    }
```

2. Handling background

The background colour has been defined as a constant BGC, and it is chosen to be black. Therefore, it is equal to zero due to the black colour represented as (0,0,0) in the RGB colour model

```
#define BGC        0
```

3. Handling borders

The height and width of the Screen are set to 720 ×520, and they are defined as constant XWIN and YWIN. They are used to create an allegro graphical screen.

```
set_gfx_mode(GFX_AUTODETECT_WINDOWED, XWIN, YWIN, 0, 0);
```

Also, in the user interface design to determine the position of the components on the screen. On the other hand, BASE and TOP constants were defined as a boundary for the ball's motion.

Collision detection

The function `collisionDetection` has been implemented to process the collision between balls. Initially, a for loop is utilized to loop through all the tasks (balls). Since the x and y coordinates of the ball are initialized to zero, thus, in order to check whether a ball is colliding with other balls is defined as follows:

1. The x and y coordinates are not zero → the ball is used in the scene.
2. The i and j are not equal → the ball can not collide with itself.

If the aforementioned conditions are met, the collision is detected as discussed in the previous section. Firstly, the distance between the two balls is calculated. Then, the distance is checked whether it is less than or equal to L. If this condition is true, the function will return the value j which represents the ID of the collided ball. However, the function returns -1, if the for loop went through all the tasks, indicating that there is no collision.

```
int collisionDetection (int i, int x1, int y1)
{
    //check for collission
          for (int j = 0; j < MAX_TASKS; j++)
          {
              //get the distance between the two ball's centers
             if(y_coor[j]!=0 && x_coor[j]!=0 && j!=i){
                 float gdistance =
((x_coor[j]-x1)*(x_coor[j]-x1))+((y_coor[j]-y1)*(y_coor[j]-y1));
                 gdistance = sqrt(gdistance);
                 if(gdistance <= L)
                 {
                     //there is a collision with ball j
                     return j;
                 }
             }
          }
          // there is no collision
          return -1;
}
```

Obstacle Detection

A special data structure has been defined to create the obstacle as shown below:

```
struct obstacle
```

```
{
    int x;
    int y;
    int up;
    int down;
    int right;
    int left;
    int visible;
};
```

The x and y variables define the x and y position of the obstacle. The up, down, left, and right variables define the sides of the obstacle. Finally, the variable visible is initiated to 0 (the obstacle is defined but not displayed on the screen) and it is set to 1 once the user locates the position of the obstacle by clicking on the screen.
For detecting the collision of the balls with the obstacle, we did expert the following condition whenever the visibility value of the obstacle is equal one ,when the condition is satisfied, that means there is a ball has collided with the obstacle

```
if(ob.visible ==1)
       {
        if (((x >= ob.left) && (x <= ob.right + L + 5 )) || ((x + L + 5 >= ob.left) &&
(x + L + 5 <= ob.right)))
```

We determine a range for x positions from the right side of the obstacle to the left. When the ball's x is in this range, that means there is a collision.

Ball Task

In implementing the ball task the following variables were defined:

```
    int i= 0;
    int x, y;
    int ox, oy;
    int x0;
    float vx, vy;
    float dt;
```

The i indicates the task (ball) index. The x and y represent the ball coordinates whereas the ox and oy represent the old coordinates of the ball. The variable x0 is the initial position of the ball and it is calculated using the random function as follows:

```
x0 = 20 + rand() % (SCREEN_H-L)
```

The vx and vy represent the ball's speed. Finally, the dt variable is utilized to calculate the time increment.

The initial step in the ball task is to process the collision with any obstacle as discussed in a previous section. Subsequently, the ball collision is detected by invoking the `collisionDetection` function. If the `collisionDetection` function returns -1, which indicates no collision, the following action is being implemented:

```
x = x0 + vx * tx;
y = BASE + vy * t - .5 * G * t * t;
y_coor[i]=y;
x_coor[i]=x;
tx += dt;
```

The new x coordinate is calculated by adding the initial x coordinate (x0) and the ball's speed on the x-axis and the time increment. For the y position, the bouncing ball animation is implemented by employing the base, the ball's speed on the y-axis, the gravity, and the time. The x and y coordinates of the ball are stored in the z_coor and y_coor arrays respectively.

However, if the `collisionDetection` function returns a value other than -1, a collision is detected and must be handled. The following code discusses how the collision handling is processed.

```
    // the ball at the left of the collied object
                if(x < x_coor[result])
                {
                    tx = 0.4;
                    x0 = L-x_coor[result];
                    vx =  vx - 1 % 10;                  /*  -vx; */
                    x = x0 + vx * tx;
                    x_coor[i]=x;
                }
                // the ball at the right of the collied object
                else
                {
                    tx = 0.0;
                    x0 = L+x_coor[result];
                    vx = -vx;
                    x = x0 + vx * tx;
                    x_coor[i]=x;
                }
```

The collision is handeled in two scenarios:

1. If the collide ball is in the left of the current ball. In this case, the current ball direction will be changed to the right.
2. If the collide ball is in the right of the current ball. In this case, the current ball direction will be changed to the left.

The ball task further handles the window borders as follows:

```
        if (y < BASE) {
            t = 0.0;
            pthread_mutex_lock(&mxv);
            v0[i] = .9 * v0[i];
            vy = v0[i];
            pthread_mutex_unlock(&mxv);
            y = BASE + vy * t - .5 * G * t * t;
            y_coor[i]=y;
        }
        if (x > XMAX) {
            tx = 0.0;
            x0 = XMAX;
            vx = -vx;
            x = x0 + vx * tx;
            x_coor[i]=x;
        }
        if (x < XMIN) {
            tx = 0.0;
            x0 = XMIN;
            vx = -vx;
            x = x0 + vx * tx;
            x_coor[i]=x;
        }
```

If the ball reaches the floor (BASE), the ball will bounce up. If the ball reaches the left border of the window, the ball will move back in the opposite direction to the right. In counter measure, the ball reaches the right border of the window, the ball will move back in the opposite direction to the left.

Additionally, the ball task draws the ball by invoking the `draw_ball` function. To assure that this function is invoked by more than on task, the function is locked.

```
pthread_mutex_lock(&mxa);
draw_ball(ox, SmallW - oy, BGC);
draw_ball(x, SmallW - y, 4);
pthread_mutex_unlock(&mxa);
```

As shown in the code above, the drawing is implemented in two steps:

1. A black background is drown in the old postion of the ball in order to delete the old ball from the screen.
2. The new ball is drawn with new calculated coordinates.

Finally, the ball task checks for any deadline misses by invoking the `ptask_deadline_miss` function.

Main task

In the main(), the following variables were defined and assigned an initiate values

```
int main(void) {
   int c;                           /* character from keyboard      */
   int i, j, k, o;                  /* number of tasks created      */
   double a;                        /* temporary variable           */
   int h;                           /* temporary variable           */
   int ntasks = 0;                  /* total number of created tasks*/
   int last_proc = 0;               /* last assigned processor      */
   int max_proc = ptask_getnumcores(); /* max number of procs          */

   init();

  a = 2. * G * (float)TOP;          /*acceleration of the ball task  */
```

Integer C variable is to save the pressed key from the keyboard to indicate the proper interaction with the user. The other integer values were defined as i, j, k, and o to index the tasks and store the values to print the number of executed task. While the integer value ntasks was created to store the number of tasks that were created.

Double a defined variable and assign the acceleration value. Last_proce variable and max_proce are to manage the processor utilization.

```
init();
   for (i = 0; i < MAX_TASKS; i++)
       {
           v0[i] = sqrt(a);
           x_coor[i]= 0;
           y_coor[i]=0;
       }

   i = 0;
   do {
       k = 0;
       if (keypressed()) {
           c = readkey();
           k = c >> 8;
       }
       if ((ntasks == 0) && (k == KEY_SPACE)) {
           clear_to_color(screen, BGC);

       }
       if ((ntasks == 0) && (k == KEY_SPACE)) {
           clear_to_color(screen, BGC);}
```

Starting the program by calling init() function that activates the allegro functions and the designed user interface.then initiate a for loop that's responsible to keep the created tasks less than the maximum number of tasks and store the task into the v[ ] array.after that in the if statement check if the user pressed any key or not cause the game will start if and only the user had interact with the program then if the pressed key was space and the tasks are equal to zero that means the the game just start the perogram will remove the user interface by makes a black background.

```
if (((ntasks == 0) || (ntasks < MAX_TASKS)) && (mouse_b & 1)){
           tpars params1 = TASK_SPEC_DFL;
           params1.period = tspec_from(DPER, MILLI);
           params1.rdline = tspec_from(DREL, MILLI);
           params1.priority = PRIO - i;
           params1.measure_flag = 1;
           params1.act_flag = NOW;
```

```
            /* a round robin assignment */
            params1.processor = last_proc++;
            if (last_proc >= max_proc)
                last_proc = 0;
            o = ptask_create_param(draw_obstacle, &params1);
      }
```

Now, based on the pressed key, the program will continue executing, in the condition that the tasks are less than the maximum number of tasks and the mouse was clicked, the obstacle task will be created and assigned its parameters.
Furthermore, if the tasks are still beyond the maximum limit and the space key is pressed, the ball task will be created.

```
if ((ntasks < MAX_TASKS) && (k == KEY_SPACE)) {
            tpars params = TASK_SPEC_DFL;
            params.period = tspec_from(PER, MILLI);
            params.rdline = tspec_from(DREL, MILLI);
            params.priority = PRIO - i;
            params.measure_flag = 1;
            params.act_flag = NOW;
            /* a round robin assignment */
            params.processor = last_proc++;
            if (last_proc >= max_proc)
                last_proc = 0;
```

the game will keep running unless the esc key was pressed , the number of tasks will printed and deadline misses the worst case of each created task.

Results and Discussion

In this section, several screenshots of running the project are illustrated along with some statistics.

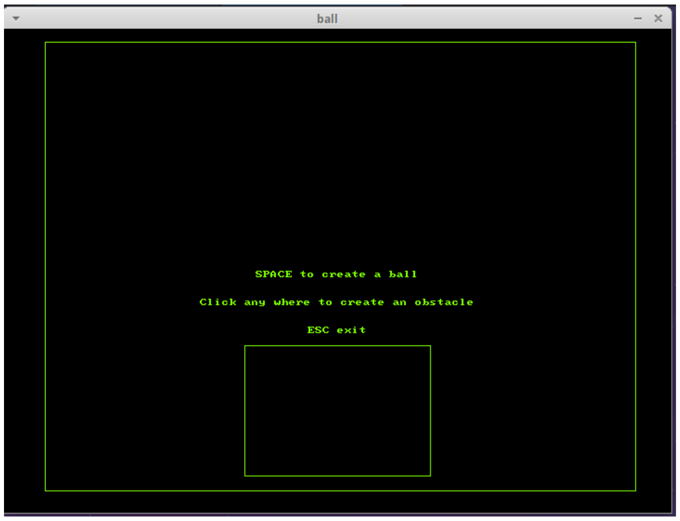


Figure 5: Welcoming screen

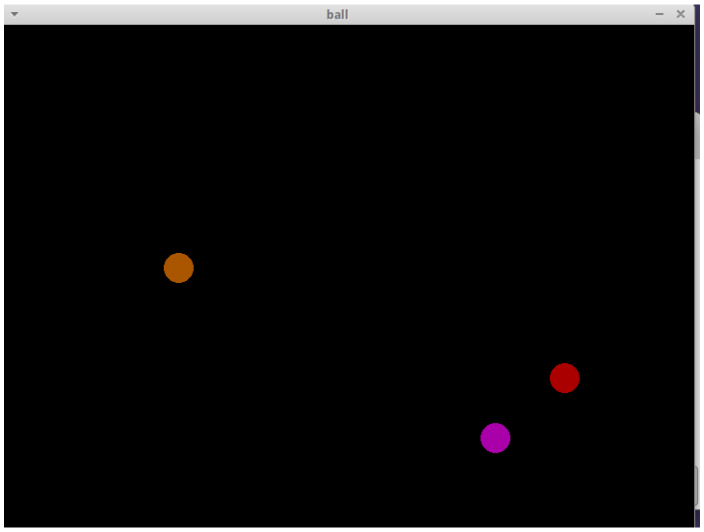
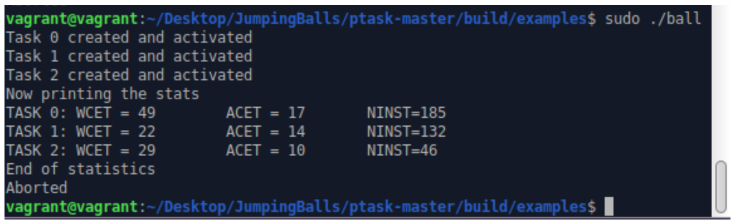


Figure 6: Adding three balls to the scene.



```
vagrant@vagrant:~/Desktop/JumpingBalls/ptask-master/build/examples$ sudo ./ball
Task 0 created and activated
Task 1 created and activated
Task 2 created and activated
Now printing the stats
TASK 0: WCET = 49        ACET = 17       NINST=185
TASK 1: WCET = 22        ACET = 14       NINST=132
TASK 2: WCET = 29        ACET = 10       NINST=46
End of statistics
Aborted
vagrant@vagrant:~/Desktop/JumpingBalls/ptask-master/build/examples$
```

Figure 7: The resulting statistics

As shown in Figure 7, The worst-case execution times of task1, task2 and task 3 are 49, 22, 29 respectively.

References:

1. Buttazzo, G. C. (2011). Hard real-time computing systems: predictable scheduling algorithms and applications (Vol. 24). Springer Science & Business Media.
2. Caselli, F., & Frémond, M. (2009). Collision of three balls on a plane. Computational Mechanics, 43(6), 743-754.
3. Hedrih, K. R. S. (2017). Central collision of two rolling balls: theory and examples. Adv. Theor. Appl. Mech., 10, 33-79.
4. Bélanger, J., Venne, P., & Paquin, J. N. (2010). The what, where and why of real-time simulation. Planet Rt, 1(1), 25-29.
5. Laplante, P. A. (2004). Real-time systems design and analysis (p. xxi). New York: Wiley.