ELE00029C - C Assignment Report

Y3903727

May 13, 2022

1 Abstract

This report describes an implementation of a physics-based game based around projectiles, where the user is scored based on their throws. The implementation itself has been Given an initial description of the game and its target audience, this report shows the steps to implement the game, along with justifying any design decisions along the way.

The report is divided into 4 sections: A breakdown of the initial problem, algorithmic solutions to the problem, evaluation of the final product, and the asset and code listing.

2 Problem Analysis

2.1 Brief

The brief consists of a physics-based game about launching a projectile onto a target. The projectile must be affected by at least 2 forces, gravity and wind. The user is then scored on how well they did with the throw. Other mechanics and features are left to the programmer but must be relevant to the target demographic.

The target demographic are users of a high-street bank who need something to entertain themselves while waiting for service. Therefore, the game must be short and simple. Also, the game may be ported to multiple different platforms.

2.2 Implementation of Game

In accordance with the brief above, the game will consist of multi-level puzzle game, where the aim of each level is to hit a chest with your projectile. The user will be scored based on how many tries it took to hit the chest. The game will be called 'The Money Dungeon'. Each level will be able to have different gravity and wind conditions, and multiple obstacles.

Each level will consist of some 'puzzle' where the user will throw the projectile at a given angle and magnitude (given by the position of the mouse) in order to bounce on jump pads, rebound off walls and hit the chest. The position of the player and the target change per level, and some levels will be played from right-to-left.

2.3 Breakdown of Requirements

1. Calculate position of projectile in 2d space
   (a) Adjust position based on forces acting on projectile (gravity, wind)
   (b) Calculate when a projectile has collided with an obstacle
   (c) Calculate when to rebound or bounce from a jump pad
2. Calculate when the projectile hits the target
3. Loading and unloading of levels
4. How to score the user on each level
5. Drawing each level
   (a) Drawing the projectile and its path
   (b) Drawing the obstacles

2.4 Game Physics

As the physics of the game is based around throwing a projectile, it would be best to use SUVAT equations to model them. Given a starting position, component velocity, and a given time the position of the projectile at any time can be found with:

$$\text{pos}_x = \text{start pos}_x + (\text{velocity}_x + \text{gravity}_x + \text{wind}_x) * \text{time}$$

$$\text{pos}_y = \text{start pos}_y + (\text{velocity}_y + \text{wind}_y) * \text{time} + \text{gravity}_y * \text{time}^2$$

2.5 Flow of Gameplay

2.6 Limitations

There are a few limitiations described in the assignment. Namely that the program must be written in C and must be compilable using either CodeBlocks v20 or v17 on Linux or Windows. Along with this, the program must use Allegro 5 for the graphics.

3 Specification

3.1 Technical Design Decisions

The game will be designed with object-orientation in mind. Each level, projectile, obstacle and tilemap will be their own struct (object), storing data necessary to the object. Each object will have their own functions related to them, such as constructors, destructors and functions to modify the object. For example:

Level Object	
DATA	METHODS
Player's start pos Target pos Wind components Gravity components Obstacle objects in Level Tilemap objects in Level Level no.	initialize_level(); [constructor] add_object_to_level(); [modifier] free_level(); [destructor]

Figure 1: Example Level object layout

Other objects include:

Level stores data about the currently loaded level

Tilemap stores an array of tiles (hex values from 0-F) for use when drawing

Position stores an (x,y) position or anything that has an x,y component (velocity, width, height, etc.)

Obstacle stores data about an obstacle (its position, dimensions, and what happens when the projectile hits it)

Projectile stores data about a given projectile on the screen (its position, velocity, time of flight, whether it is active or not, etc.)

Along with object orientation, functions will be split into their own library files. Grouping functions into libraries allows for a degree of modularity in the code, along with keeping certain functions that should only be accessible inside the library private. Along with this, the files themselves will be neater due to the smaller file length. The functions will be split into these libraries:

- draw.c: *for all drawing-related functions*
- physics.c: *for all physics calculations*
- structures.c: *for all objects and their related functions*

3.2 User Input

As described in the brief, this program must be able to be ported to multiple platforms, mobile included. As such, the user input will be limited to as few buttons as possible. Aiming will be done with mouse movement, where the displacement of the mouse position determines the velocity components. Firing a projectile will be done with the left mouse button. Having the main controls of the game done with the mouse prioritizes ease-of-use and allows for easier porting in the future for devices with simpler input devices (i.e. mobile and tablet devices).

3.3 Graphics

The graphics will be done with allegro, as described in the brief. the window size will be limited to 640x480 as it is a small enough screen size to be supported on a variety of display sizes, along with being a common aspect ratio (4:3 ratio). The graphics themselves will be limited to pixel-art as pixel-art graphics are easy to implement and have an abundance of free game art available (see section 5). The graphics will be tile-based as it is easier to implement due to everything being on a fixed-grid, along with being less intensive to draw.

3.4 Key Algorithms

Collision Collision is done through a quarter-step method. Every frame a new position for the projectile is calculated using the equations described in subsection 2.4. The new position and the old position are then split into quarter-step intervals. At each interval, if the position is within the bounds of an obstacle, the new position is set to the previous quarter-step position. This is done to prevent the projectile from getting stuck within the obstacle. When checking the collision, the type is returned if a collision occurs. The type dictates what to do when a projectile collides with the obstacle (e.g. rebound off of wall). This code can also be repurposed for checking whether the projectile has hit the target.

```
FUNC do_qstep_coll(in new_p, in old_p,
                   out type)
{
  qstep_p = (new_p - old_p)/4;
  for i in [1,2,3,4] {
    temp_p = old_p + qstep_p * i;
    if is_coll(temp_p, type) {
      //collides at step i,
      //return pos at step i-1
      new_p = temp_p - qstep_p;
      return new_p;
    }
  }
  //does not collide at any steps
  return new_p;
}
```

```
FUNC is_coll(in pos, out type)
{
  for obstacle in level {
    //if inside obstacle, true
    if (pos.x >= obstacle.min_x and
        pos.x <= obstacle.max_x) or
       (pos.y >= obstacle.min_y and
        pos.y <= obstacle.max.y) {
        type = obstacle.type;
        return true;
    }
  }
  return false;
}
```

Figure 2: Pseudocode for collision

Drawing obstacles Obstacles are not drawn to the screen. Rather, a representation of the position of the obstacles is drawn instead. this representation is stored as a Tilemap object (see subsection 3.1). Tiles are stored as hex values (as a single char) representing their position on the tileset bitmap. The tileset bitmap's layout is designed so that the x direction represents what tile is which, and the y direction represents which tileset to use (for bg tiles, fg tiles, etc.).

Level Loading/Unloading Level Unloading is easy to implement due to the code being object-orientated. To unload the level, `free_level()` should be called. This function will free any allocated memory to the Level object (i.e. the object array described in subsection 3.1).

Level loading consists of loading the data needed from files and creating the Level object with the data from the files. Levels consist of 4 files: 3 files for tilemaps (fg, bg, dec), and one file for obstacle data and initial values described in subsection 3.1. Tilemap data is stored as one long line of characters, so data is easy to parse and store into an array (stored as Tilemap object). Level and obstacle data is stored in CSV format, where each line corresponds to a new field in the Level object.

```
100,416          //player pos
512,384          //target pos
0,0              //wind components
0,100            //gravity components
0,480,640,64,0   //obstacle 1 (ground)
0,32,640,32,0    //obstacle 2 (ceiling)
448,416,192,32,0 //obstacle 3
```

Figure 3: Example level.txt file for loading

Rebounding and Jump Pads

4 Evaluation

5 Appendix