Apple Thrower

Software Design Document

July 3 2021

Authors

梁潘钰曈 (2019080106)

Overview

Apple Thrower is a 2D infinite shooting game built using liquidfun, an extension of the Box2D physics engine. The objective of the game is to increase your score by defeating successive waves of enemies, navigating around a particle system below by using limitless, but slow jumps. To defeat enemies, the player must throw apples at them. The number of apples a player has is limited, and apple regeneration is slow. When enemies are defeated, players gain apples. When a player comes in contact with an enemy, they will begin to lose health. When their health drops to zero, they player dies, and the game ends.

Usage

To play the game, first clone into the public repository available at:
https://github.com/kevinpliang/apple-thrower-liquidfun
Then, navigate to the testcase folder and run the bash script inside.

The game uses the keyboard and mouse. The controls are listed below:

W	Jump Up
A	Jump Left
D	Jump Right
Left-Mouse	Shoot
P	Pause
R	Restart
Esc	Quit

File Structure

The games code is contained entirely within the Testbed folder of liquidfun. To get there, navigate to src/liquidfun/Box2D/Testbed. Here, there are three folders of note:

```
Testbed
├─[…]          Other folders
├─Framework    Contains Testbed framework (UI, rendering, settings etc.)
├─Release      Contains executable game file
└─Tests        Contains bulk of games code (classes, logic, etc.)
```

Within Tests, TestEntries.cpp contains only one test, Apple.h. Apple.h is the main file of Apple Thrower, handling the word, object creation, and game logic. The remaining files, Entity.h, Player.h, Enemy.h, Contact.h, and Bullet.h, contain various useful class definitions.

Code Design - Classes

The most important part of Apple Thrower's design is the way it handles entities. Both Player and Enemy inherit from a parent class, Entity.

Entity's member variables are split into three sections. First are the body variables, objects necessary for any body to exist in liquidfun's physics system. Second is the Entity's type, represented by an enumerator. There are only two kinds of entities, players and enemies. Lastly, there are the physics variables, variables that dictate how the entity behaves in the game, like its health, speed, velocity, and number of contact points. It should be noted that contact points denote the number of things contacting an entity that reduce its health.

The only constructor for Entity takes in a b2World pointer for context. It then sets up the entity as a box with a specified fixture and default physics constants.

Entity also has a number of class methods. They are all relatively straightforward and are explained below:

`setVel(b2Vec2 v)`	Sets the entities velocity to a directional vector given by v.
`process()`	Updates the entities rendering and checks its contacts. Called every timeframe.
`render()`	Draws the entity.
`renderAtBody()`	Calls render at the correct position so the rendering follows the body.
`startContact(b2Body* c)`	Called when the entity comes into contact with another body.
`endContact(b2Body* c)`	Called when the entity leaves contact with another body.
`randomFloat(int a, int b)`	Returns a random float between a and b.

Player and Enemy both have unique constructors that update their unique physics constants (for example, the Player class has a higher speed value than the Enemy class). They have unique render functions, as they look different and thus must be rendered differently. They also have unique contact functions, as players and enemies must respond differently to contacting different things. For example, enemies should take damage from apples whereas players should not.

The Bullet class is much like the Entity class, but since it has a constant velocity and does not need to be rendered or care when it is contacted, it is missing a few functions.

The ContactListener class, found in Contact.h, overrides liquidfun's b2Contact Listener. In it, it declares BeginContact and EndContact, both of which are defined by liquidfun and return a b2Contact pointer. The b2Contact pointer gives us information about the two objects that have been contacted. We use this information to determine if either object is an entity, and if so, we call its corresponding start or end contact function.

Code Design - Game

As mentioned above, the majority of the gameplay logic can be found in Apple.h, which defines a class Apple that inherits from Test.

First, I will explain the global variables. The player is a global variable, as well as its alive state. The ContactListener must have global scope to interact with the player, and the enemies vector contains a list of all the enemies currently present in the game. I keep track of an enemyThreshold, which determines how likely an enemy is to spawn on any timeframe, and the enemyHealth, which progressively gets higher as the game goes on. The score is self-explanatory.

Let's now look at the default constructor. First, I declare the various world variables, overriding m_world's default gravity and contact listener with my own. I then set the game bounds with multiple large b2bodies defined as boxes. Finally, I create a player and the particle system, which is a simple rectangle that crashes onto the ground.

Keyboard, MouseDown, and MouseUp override Test's virtual functions and allow for player movement. Specifically, using the keyboard sets the players velocity, and the mouse spawns a bullet object with velocity vector pointing from the player to the position of the mouse click.

Finally, the most interesting method is Step, which is called every timeframe by the main Testbed function. First, I update each of the status strings present in the games UI. Then, I process each of the entities, including the player and all of the enemies, deciding whether entities should be spawned, killed, or changed in any way. When an entity dies, it must first be removed from processing and then deleted. Additionally, when enemies die, they drop apples, increase the players score, and increase the health of all further enemies. When the player dies, the game ceases to process until the player chooses to restart.