**Report for Project 3, “RoboBlast”**

Design decisions and description of non-trivial functions:

The first thing I want to say in this report is, DESIGN DECISIONS SHOULD ALWAYS BE IN THE FIRST PLACE FOR BIG APPLICATIONS.

* Game frame: StudentWorld:

Design decision: make functions less than 50 lines. For very long functions, split it by functionality.

* + Data structure:
    - I have been used a vector to store the actors in each of sublevel, to be precisely, vector of pointers to actors.
    - And three vectors for storing data structures of each level: pointers pointing to the vectors above; pointers to players, pointers to the Level class.
    - And status variables to keep track with the environmental records / status of current level / sublevel.
  + init(): I split the functionality of this function to three public functions. Well, actually I can put them in private members, as nobody will call them outside. However, I think it would be very necessary to describe them.
    - mapStatus(): load map and prompt load status.
    - loadMap(): for each actor, put them into data structure in the StudentWorld.
    - putActors(): give each actor a correct parameter in its own data structure.
  + move() and : nothing big difference from the spec.
    - In removing dead members in vectors, I choose the slow but reliable method: copy everything except the one you wanted to remove, to a new vector, and return the new vector.
      * Well, if more time given, I will use list. Removing members from a list is much faster.
    - In detecting dead actors, do the act when actor dead respectively, such as add a Waterpool.
  + cleanUp(): trivial…
  + Rest of non-important but non-trivial functions:
    - std::string getFilename(int level, int subLevel); // return correct filename for each level and sublevel
    - bool createWaterpool(int X, int Y); // as Gangster died, create a Waterpool there
    - bool createHostage(int X, int Y); // as Robot Boss died, create a Hostage thereInheritance: Actor family.

Design decision: make objects having common functionalities share a same parent.

* + Actor:
    - Player:
    - Bullet:
    - Fixed non-passable objects: Wall, BullyNest
    - Fixed passable objects:
      * FakeWall
      * Goodies: ExtraLife, RestoreHealth, Ammo, FarplaneGun, Gold, Hostage, Gate, Exit
    - Moveable actors: , Gangster, Bully, Robot Boss
  + Non-trivial functionality of actors:
    - checkStuck: for enemies, check whether it hits a wall and needs to turn around.
    - checkShot: for bullets, check whether it hits some objects. // actually it should be in Bullet objects
    - checkDir: helper function for functions below, to check whether a certain object exists on the view of a coordinate on certain direction (use “view” below) // actually those three should be in MO objects
    - checkPlayer: to see whether player is in the view, take the obstacle into consideration.
    - stuckPlayer: to see whether the enemy is stuck by the player, which forbids the enemy from moving and turning.
    - Pickup: for Bully.
    - Walk & firePlayer: for Enemies.

Test of classes:

Briefly, testing in part 1 is a tremendously important work, while in most cases testing each Actor’s behavior can be done in real-time playing. So in this part I will focus on testing in the StudentWorld frame as well as the non-trivial behaviors of Actors.

But for certain specific functionalities, I used cerr to help monitoring parameters. These functionalities include:

* cleanUp()
* checkPlayer() series – that’s a tricky one, for the “view” – gangsters must not be able to see me over a wall.
* loadLevel() series
* conditions for exit revealing

Known issues:

* the tick might be a bit slower than 20/sec. – due to the algorithms.
* When enemies just next square of the player and firing to the player, player doesn’t lose HP.

Comments on this project:

Well, it’s just long and tedious, with little thinking, while a lot of tricks from software engineering techniques are required. I will just list some vital ones:

* Refactoring (!)
* Unit test (!)
* Integrated test (!)