Underperforming Students versus Teachers - Documentation

We do our version control on Github. Many details, markdowns and implementation history are included on the website. For more details, please click here.

```
+-- QObject
|   +-- QWidget
|       +-- QLabel
|           +-- ClickableLabel
|               +-- SpriteCard
|   +-- TimeVariant
|       +-- Human
|           +-- Student
|               +-- AttackStudent
|                   +-- SleepDeprivedStudent
|                   +-- DeadlineFighter
|                   +-- CgaGod
|               +-- SupportStudent
|                   +-- ShamelessStudent
|                   +-- TeachersPet
|                   +-- GbusStudent
|           +-- Teacher
|               +-- OverworkedTA
|               +-- Kelvin
|               +-- Pang
|               +-- Desmond
|       +-- Item
|           +-- Redbull
|           +-- Assignment
|           +-- VendingMachine
|   +-- Game
|   +-- Row
```

Class Design

Our objects can be divided into three main parts: TimeVariant objects, UI elements and game controllers.

1. For *TimeVariant* objects, they are all the objects that change with time. It includes most of the elements in the game, such as students, teachers and different items like Redbull and assignments.

   TimeVariant has a pure virtual function: update(). All TimeVariant objects must implement this interface. And it will be used for game controllers to make everything move.

2. We made use of the four main OOP concepts: inheritance, encapsulation, polymorphism and abstract.
   a. Inheritance: "*OverworkedTA* is a *Human*", "*Redbull* is an *Item*"
   b. Encapsulation: elements in the class are made private and not visible to outside
   c. Polymorphism: In the game controllers, objects are often saved as Teacher or Student pointers, and their corresponding update() is called. And they will behave differently because they have different underlying implementation
   d. Abstract: We have some pure virtual abstractions like class Student and Teacher. They can never be instantiated but plays an important role in class management.

3. The second type is UI elements. Since Qt does not provide some useful UI components e.g. (clickable label), we implemented some of these by ourselves. Some UI components are also customized for our game, such as the *SpriteCard* class, which is the object that shows up on the left-hand side of the game (for student selection).

4. The third type is game managers. The class *Game* is a singleton class responsible for managing timer counting, scene switching, teacher generation and Redbull (energy) number management. The class *Row* is responsible for managing objects in the single row, including their generation, destruction and collision detection.

Integration

Our code is mainly integrated through the game controllers. Most game objects such as defenders, attackers, Redbulls (energy) and assignments(bullets) are *TimeVariant* and they have an update() interface. The game controller saves a reference to every game object. It also manages several *QTimer* to call the update() function of all game objects at a rate of 1 call per 20 ms.

In different classes, we implement their update() functions to develop most of the dynamics in the game. For example, students (defenders) will count the time and shoot assignments. Similarly, teachers (attacks) will change their x position according to their speed in its update(). It also applies to other objects like Redbull and assignment. All the movements are done in update() of different classes. Collision detection are also performed in each objects' update(). To avoid two objects doubly reacting to the same event, we developed a "Teacher-first principle", which is, teachers (attackers) are responsible for the reaction more compared to items and defenders. For example, when an assignment hits a teacher, the assignment object need not reduce the teacher's hp, but the teacher will reduce its own hp.

The object instances are managed in rows. We assume that every row is independent of other rows, i.e. an object in row i will never interact with objects in row j. Given this assumption, all objects in the same row are managed together using advanced data structures (priority queue and vector). We implemented fast collision detection based on the priority queue. (details will be covered in the next section) All of these managements are packed in a class *Row*.

In our singleton game manager, we keep track of 5 *Row* objects. With utilities implemented in Row objects and the update() functions, the game manager can monitor the game seamlessly and conveniently. When the game termination is triggered (Timer ends or attacker reached the edge of a row), the game is over and it switches scenes / play different background music and so on.

Reusability, Efficiency and Data Structures

Our code is highly extensible and reusable. The abstraction of *TimeVariance* and *Row independence* allows us to add new content to the game conveniently. For example, when a new type of defender needs to be added, we only need to create a new class and implement its update(), and all other things will automatically be done. It is also easy for us to implement some complex functionalities in the game. For example, we realize many complex skills based on the time variance and collision detection system (Desmond increases the speed of all attackers, teacher's pet can turn teachers around and deadline fighters will shoot faster when teachers are closer). It is another proof that our code is highly flexible and reusable.

In terms of efficiency, thanks to the use of different data structures and some efficiency improvement techniques, our game run at a very low latency rate. In *Row* objects, we used priority queues to manage students and assignments. We used the *distance to the left* as the sorting criteria of the priority queues so that when teachers try to detect collisions, they can get the leftmost student or assignment in constant time. (since teachers will only collide with the leftmost objects in a row) Other than data structures, some other techniques are also applied. For illustration, all the pixmaps are preloaded before the game starts instead of loading during update() because we found that it would slow down our game. Generally, we put emphasis on efficiency and latency management.