Project design choices

Vladimir Jovicic

15.05.2020

1 Animations

One of my customized features in the project is the sprite animation implementation. I wanted to use some quick and efficient way of implementing not too complex animations without having to use more complex tools of some sort in the Java API, like calling the timer to stop when needed for example.

The basic goal was to find a way of changing the sprite's images at each "frame" within the scope of a delay. Each frame represents the state of one loop of the method `step` in the main class, and at each frame, the image of a sprite is supposed to change according to the frame. The frame is therefore a *count* remembering which image was the previous one in the previous `step` call. This is why the images in *.png* format are ending with an integer, the integer representing the frame. Each time the frame increases, the image is set to the next one. The frame count has to be stopped and reset in a way that the last image of the sprite sheet must reset to the first one in order to make an animated loop.

For that I defined an interface `Animated` implemented by `Sprite` and giving the option to use the following methods: `setAnimation(int delay)` is supposed to set a frame field to 1 and to determine how much frames there should be according to how much images there is in the animation loop. I called this number "delay". The method `delayIsReached()` verifies if the frame count has reached its maximum, i.e. if frame $<$ delay, so that the frame can be reset with `resetFrame()`. The method `nextFrame()` increases the count of the frame and returns the count before increment.

1.1 Animated Shots

There are two types of animated Shots: the bombs and the lasers. They differentiate through shot damage. The bombs are supposed to be thrown by more powerful enemies, the spaceships, and have a damage of 2, whereas lasers are shot by regular aliens. I harmonized the number of images for the shots and decided to go with the classic number of 4 images per loop to keep everything simple. The method calling the implemented animation methods is the `move()` method, because this method naturally involves the motion animation.

1.2 Animated Enemies

I quickly began to feel the consequences of the simplicity of the animation design choice when I started implementing the enemies. I realized that a given animation would only work for one type of animation. What if a given sprite has to implement several animations under different conditions? The frame count and the delay would for example not be the same for a motion animation and for an explosion animation.

1.2.1 Motion of enemies

The motion function is assigned to each concrete type of enemy in its specific `move()` method. No problem with that since it is analogous to the shot motion.

1.2.2 Explosion of enemies

The problem of exploding enemies is to find a way to dissociate dead enemies from exploding enemies in the list of enemies in the `SpaceInvaders` class. It is obvious that the method triggering the enemies' explosions is the `resolveShotCollisionsAndDeadEnemies()` (that I refactored to `resolveCollisionsAndExplosions`) since there is a shot involved in the triggering of an explosion. I considered 2 options for the explosion animation.

(a) Either the death of an enemy triggers its image to be set to an exploding enemy image with the same coordinates and this enemy with health points = 0 would have a limited lifetime in the enemy group through the methods of `Animated`.

The problem with this implementation is that I could not use the methods of the `Animated` interface because there would be no way of knowing the frame count and the lifetime of the explosion would be synchronized or even interfere with the motion loop. Moreover, the classic retro graphics of Space Invaders implementation freezes somehow the enemies during a couple of milliseconds (but not the player) when an enemy is hit by the player. I figured out another way.

(b) I created a new type of enemy called `ExplodingEnemy` that has the exploding image of an enemy. In that way I could create a new animation with a new delay for the explosion lifetime and place it in the group of enemies with the coordinates of the previous one.

However I decided to implement the classical retro feel of the enemies freezing by creating a new group in `SpaceInvaders` called `explodingEnemies`, so that each time an enemy would be marked as dead, its coordinates would be transferred to a new exploding enemy and that enemy would be added to the exploding enemies group. The dead enemy would be of course removed from the living enemies' group. All this is implemented in the `Group` class in `retrieveDeadEnemiesCoordinates()` and `resolveHitEnemies()`. For the explosion lifetime, I implemented it in the `Enemy` class. I should have maybe implemented it only in the exploding enemy class but then I could not use the for-each loop in the group's methods, so I left it as is. These methods would then be called in `resolveShotCollisionsAndDeadEnemies().`

In that way, I made some sort of an "exploding mode" in the `step` method by first verifying if the dead enemies' group is empty or not. If it is empty, then the enemies would freeze for a certain amount of time. This gave me an idea of “game modes” where every loop in `step` would be “geared in” by the state of the game (explosion mode, normal shooting mode, etc.).

1.3 Animated player

I wanted the player to look damaged in an enemy-freeze-like behavior when hit but also to have the option to recover after the freeze in order not to be shot immediately after an unescapable shot (leading to the rage of the player ;)). Here I used a similar option described higher for the enemies in (a): when the player is hit by a shot, the `resolveShotCollisionsAndDeadEnemies()` would mark the player as damaged, which would trigger the player-damaged mode, and when this mode is finished running, the player would not be damaged and set to recovering mode. The damage animation and the recovery animation have the same duration since they are all based on the same frame-delay implementation (option (a)).

Side note: the alien spaceship is actually a hybrid of (a) and (b). Since the alien spaceship is a more powerful enemy with 2 lives, it uses a transform into damaged method for the first hit and behaves like other enemies with the second hit.

I now realize that all these methods are the reason why the `SpaceInvaders` class itself implements `Animated` (!). The class `Player` would have been better suited to implement those modes since it is an animated sprite, but at that time I was focused on this “playing mode” aspect. I am aware that there are certainly some methods in the Java API for these animations especially that this class extends an animation class.

2 Enemy anger modes

All this “game modes mindset” gave me the idea to implement angrier enemies at a certain stage of the game to spice the game up, but instead of the classical enemy speed increase, I have opted for a “Super Mario-like” boss changing state.

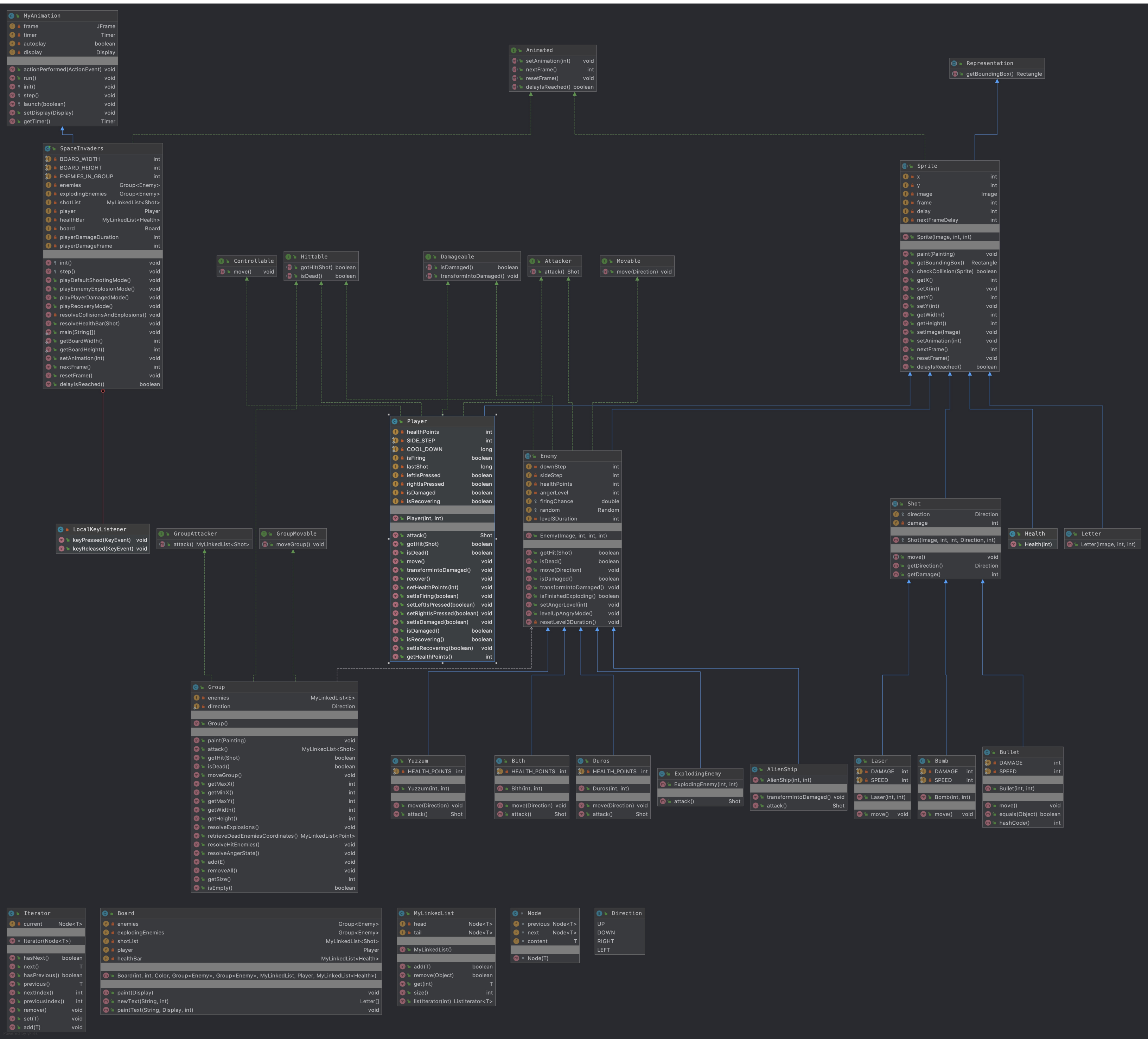
The modes are implemented in `Enemy` in `levelUpAnryMode()` that is called in `Group` in `resolveAngerState()` each time a certain number of enemies in the group has changed. There are 4 levels plus the first one (0) being the default one at the start of the game. Each level increases or decreases the speed of the enemies but also their shooting rate (the less they are, the more they shoot). The most interesting modes are the 3rd level and the 4th level. The 3rd level moves the remaining enemies really fast for a couple of seconds and then stops them and make them rain bombs on the player (the player can hide between the bomb “ropes”) before going crazy again and stopping again, all this until only one enemy remains, the final boss of the 4th level which goes very fast and drops a lot of shots.

For the 3rd level animation I used a similar method compared to the `Animated` implementation without implementing it (because it is already used for the motion). In other words, I implemented the same method but more simplified and with a difference: each enemy knows its state thanks to the countdown of `level3Duration`. It is set to 90 and until it reaches 0, the enemies are in bomb-rain mode, and when it reaches 0, the enemies switch to crazy mode, until `level3Duration` reaches -70 and then it is reset to 90.

3 Conclusion

All in all, I am satisfied with my design choices. I can see however that I used in total more than 3 different implementations for the animations, some of which caused some problems sometimes. For example, the creation of the Exploding Enemy class took me some time to figure out why the exploding enemy would not get painted on the coordinates of the dead enemy until I found out that all enemies would be set to a specific location the first time they are created because of the Enemy constructor (which was intended for the appearance of the enemy in the `init()` method...

However, I have learned to perform my own simple animations using different methods for different purposes and solving the problems anyway for each case.


MyAnimation
frame JFrame
timer Timer
autoplay boolean
display Display
actionPerformed(ActionEvent) void
run() void
init() void
step() void
launch(boolean) void
setDisplay(Display) void
getTimer() Timer
Animated
setAnimation(int) void
nextFrame() int
resetFrame() void
delayIsReached() boolean
Representation
getBoundingBox() Rectangle
SpaceInvaders
BOARD_WIDTH int
BOARD_HEIGHT int
ENEMIES_IN_GROUP int
enemies Group<Enemy>
explodingEnemies Group<Enemy>
shotList MyLinkedList<Shot>
player Player
healthBar MyLinkedList<Health>
board Board
playerDamageDuration int
playerDamageFrame int
init() void
step() void
playDefaultShootingMode() void
playEnnemyExplosionMode() void
playPlayerDamagedMode() void
playRecoveryMode() void
resolveCollisionsAndExplosions() void
resolveHealthBar(Shot) void
main(String[]) void
getBoardWidth() int
getBoardHeight() int
setAnimation(int) void
nextFrame() int
resetFrame() void
delayIsReached() boolean
Sprite
x int
y int
image Image
frame int
delay int
nextFrameDelay int
Sprite(Image, int, int)
paint(Painting) void
getBoundingBox() Rectangle
checkCollision(Sprite) boolean
getX() int
setX(int) void
getY() int
setY(int) void
getWidth() int
getHeight() int
setImage(Image) void
setAnimation(int) void
nextFrame() int
resetFrame() void
delayIsReached() boolean
Controllable
move() void
Hittable
gotHit(Shot) boolean
isDead() boolean
Damageable
isDamaged() boolean
transformIntoDamaged() void
Attacker
attack() Shot
Movable
move(Direction) void
Player
healthPoints int
SIDE_STEP int
COOL_DOWN long
isFiring boolean
lastShot long
leftIsPressed boolean
rightIsPressed boolean
isDamaged boolean
isRecovering boolean
Player(int, int)
attack() Shot
gotHit(Shot) boolean
isDead() boolean
move() void
transformIntoDamaged() void
recover() void
setHealthPoints(int) void
setIsFiring(boolean) void
setLeftIsPressed(boolean) void
setRightIsPressed(boolean) void
setIsDamaged(boolean) void
isDamaged() boolean
isRecovering() boolean
setIsRecovering(boolean) void
getHealthPoints() int
Enemy
downStep int
sideStep int
healthPoints int
angerLevel int
firingChance double
random Random
level3Duration int
Enemy(Image, int, int, int)
gotHit(Shot) boolean
isDead() boolean
move(Direction) void
isDamaged() boolean
transformIntoDamaged() void
isFinishedExploding() boolean
setAngerLevel(int) void
levelUpAngryMode() void
resetLevel3Duration() void
Shot
direction Direction
damage int
Shot(Image, int, int, Direction, int)
move() void
getDirection() Direction
getDamage() int
Health
Health(int)
Letter
Letter(Image, int, int)
LocalKeyListener
keyPressed(KeyEvent) void
keyReleased(KeyEvent) void
GroupAttacker
attack() MyLinkedList<Shot>
GroupMovable
moveGroup() void
Group
enemies MyLinkedList<E>
direction Direction
Group()
paint(Painting) void
attack() MyLinkedList<Shot>
gotHit(Shot) boolean
isDead() boolean
moveGroup() void
getMaxX() int
getMinX() int
getMaxY() int
getWidth() int
getHeight() int
resolveExplosions() void
retrieveDeadEnemiesCoordinates() MyLinkedList<Point>
resolveHitEnemies() void
resolveAngerState() void
add(E) void
removeAll() void
getSize() int
isEmpty() boolean
Yuzzum
HEALTH_POINTS int
Yuzzum(int, int)
move(Direction) void
attack() Shot
Bith
HEALTH_POINTS int
Bith(int, int)
move(Direction) void
attack() Shot
Duros
HEALTH_POINTS int
Duros(int, int)
move(Direction) void
attack() Shot
ExplodingEnemy
ExplodingEnemy(int, int)
attack() Shot
AlienShip
AlienShip(int, int)
transformIntoDamaged() void
attack() Shot
Laser
DAMAGE int
SPEED int
Laser(int, int)
move() void
Bomb
DAMAGE int
SPEED int
Bomb(int, int)
move() void
Bullet
DAMAGE int
SPEED int
Bullet(int, int)
move() void
equals(Object) boolean
hashCode() int
Iterator
current Node<T>
Iterator(Node<T>)
hasNext() boolean
next() T
hasPrevious() boolean
previous() T
nextIndex() int
previousIndex() int
remove() void
set(T) void
add(T) void
Board
enemies Group<Enemy>
explodingEnemies Group<Enemy>
shotList MyLinkedList<Shot>
player Player
healthBar MyLinkedList<Health>
Board(int, int, Color, Group<Enemy>, Group<Enemy>, MyLinkedList, Player, MyLinkedList<Health>)
paint(Display) void
newText(String, int) Letter[]
paintText(String, Display, int) void
MyLinkedList
head Node<T>
tail Node<T>
MyLinkedList()
add(T) boolean
remove(Object) boolean
get(int) T
size() int
listIterator(int) ListIterator<T>
Node
previous Node<T>
next Node<T>
content T
Node(T)
Direction
UP
DOWN
RIGHT
LEFT