Alissa Niewiadomski

704794074

CS 32 W18

**Project 3 Report**

1. **High Level Descriptions of Public Member Functions**

Actor

* virtual void doSomething() = 0 – I chose to make this pure virtual because every Actor in the game is called upon to doSomething, and every class is derived from Actor.
* bool isDead() const – Returns whether or not the Actor is dead (i.e., should be removed from the game).
* void setDead() – Sets the Actor to be dead (private m\_dead variable is true).
* bool isDamageable() – Returns whether or not the Actor is damageable (i.e., can be damaged or hit by a projectile/ship. I chose this so I could more easily access the Actors to be damaged by other Actors in my StudentWorld functions.
* StudentWorld\* getWorld() const – Returns a pointer to the StudentWorld that the Actor lives in. I believed that it would be good practice to have every Actor knowledgeable about the StudentWorld that it was placed in.
* virtual void sufferDamage(int hp) – Subtracts the Actor’s HP by the hp passed in, lowering its HP.
* virtual int getHP() const – Returns the actor’s HP.
* virtual int getScore() const – Returns the amount of damage that that Actor is capable of doing.

Star Actor < Star

* virtual void doSomething() – Moves the star one pixel to the left, and checks if it’s gone off the screen. If it has, the star is set to dead.

Explosion Actor < Explosion

* virtual void doSomething() – Increases the size to 1.5 time its current size. It also checks whether or not it has been alive for more than four ticks; if it has, it should be set to Dead.
* int getCount() const – Returns the number of ticks the Explosion has been instantiated for.
* int incCount() – Adds one to the Explosion’s count.

Projectile Actor < Projectile

* virtual void doSomething() = 0 – I left this function pure as well because each Projectile does something, and the ABC Projectile should not be instantiated.
* int getDirection() const – Returns the projectile’s current direction (rotation).
* int setThisDirection(int dir) – Allows one to increment to the projectile’s m\_direction quantity, i.e. rotate the projectile.

Cabbage Actor < Projectile < Cabbage

* virtual void doSomething()– Checks if projectile flew off right side of screen/is dead, then checks if it hit an alien. If it did, it returns, otherwise, it moves appropriately and checks if it hit aliens again. The reason I decided not to abstract most of this into Projectile’s doSomething() is because Turnip 1) checks if it flew off the other side of the screen, 2) checks if it hit the Nachenblaster, and 3) moves a different amount of pixels/direction than Cabbages. Ideally, I would have abstracted this, But because there wasn’t too much code and I decided to use two different functions in StudentWorld for hitting ships/hitting the NachenBlaster, I kept it separated.

Turnip Actor < Projectile < Turnip

* virtual void doSomething()– Checks if projectile flew off left side of screen/is dead, then checks if it hit the NachenBlaster. If it did, it returns, otherwise, it moves appropriately and checks if it hit the NachenBlaster again.

Torpedo Actor < Projectile < Torpedo

* virtual void doSomething()– Checks if the torpedo is dead or if it flew off the screen. It then calls the specializedAttack() function, which executes what the specific torpedo should do. Both types of torpedoes do the same for the beginning of the more abstract doSomething() (i.e., both check if dead, then they do their respective different actions).
* virtual void specializedAttack() = 0 – A pure virtual function. There are two types of Torpedoes, Alien and NachenBlaster-fired, so I did not want to allow the program to instantiate a Torpedo by itself.

AlienTorpedo Actor < Projectile < Torpedo < AlienTorpedo

* virtual void specializedAttack()– Checks if it hit the NachenBlaster; if it didn’t, it moves appropriately, and checks again. Most of this logic is handled in the StudentWorld class.

NachTorpedo Actor < Projectile < Torpedo < NachTorpedo

* virtual void specializedAttack()– Checks if it hit an alien ship; if it did, it’ll move appropriately and check again. Again, this logic is handled in the StudentWorld class.

Ship Actor < Ship

* virtual void doSomething() = 0 – Kept this pure virtual because a ship needs to be either an Alien or NachenBlaster. Both have hp, so this class deals with anything that has to do with modifying hp.
* void sufferDamage(int hp) – Decrease the ship’s hp by hp.
* double getHP() const – Return the ship’s hp.
* void addHP(int hp) const – Add hp to the ship’s hp.
* void setHP(int num) – Set the ship’s hp to hp.

NachenBlaster Actor < Ship < NachenBlaster

* virtual void doSomething() – Check if the NachenBlaster is dead, otherwise, get the input from the user and move accordingly, or shoot a torpedo/cabbage (if available). After this, refill 1 cabbage energy point.
* void increaseHP(int hp) – Add to the NachenBlaster’s m\_hp by hp.
* int getCabbagePercent() const – Return the % of cabbage energy points available.
* int getNumTorpedoes() const – Return the number of torpedoes the NachenBlaster has.
* void setNumTorpedoes(int num) – Increase the number of torpedoes the NachenBlaster has by num.
* void decTorpedoes() – Subtract one torpedo from the NachenBlaster.

Alien Actor < Ship <Alien

* virtual void doSomething() – Check if the alien is dead, and if it is decrease the number of aliens in StudentWorld. If there was a collision with the NachenBlaster, don’t do anything else. Otherwise, set a new travel direction if needed, then move as indicated by the current travel direction. Check again for a collision.
* int getFlightPlan() const – return the current flight plan length.
* void setFlightPlan(int length) – set the flight plan length.
* void decrementFlight() – subtract one from the flight plan length.
* void setFlightDirection(int dir) – set the Alien’s flight direction, encoded by an integer 0 to 2.
* void getFlightDirection() const – return the Alien’s flight direction.
* void setTravelSpeed(double speed) – set the Alien’s speed.
* double getTravelSpeed() const – return the Alien’s speed.
* void getScore() const – return the amount of points that the user gets by destroying this Alien.
* virtual void dropSomething() = 0 – Allows Aliens to drop a goodie. Did not want to instantiate Alien class, and all Aliens might (or might not) drop something.
* virtual void specializedAttack() = 0 – Allows the Alien to do shoot or attack accordingly. I set this to be pure virtual because each type of Alien has its own unique attack method.

Smallgon Actor < Ship < Alien < Smallgon

* virtual void specializedAttack() – Checks if the NachenBlaster is in range to shoot, and possibly shoots a Turnip.
* virtual void dropSomething() – Smallgons do not drop any goodies, so this function was left empty.

Smoregon Actor < Ship < Alien < Smoregon

* virtual void specializedAttack() – Checks if the NachenBlaster is in range to shoot, and possibly shoots a Turnip. It also may possibly start ramming forward at an increased speed if within range.
* virtual void dropSomething() – Checks if the Smoregon should drop a new goodie, and whether or not it is a Torpedo or Repair goodie.

Snagglegon Actor < Ship < Alien < Snagglegon

* virtual void specializedAttack() – Checks if the NachenBlaster is in range to shoot, and possibly shoots a Torpedo.
* virtual void dropSomething() – Checks if the Snagglegon should drop an extra life goodie (1 in 6 probability).

Goodie Actor < Goodie

* virtual void doSomething() – Checks if it’s dead or out of range; if so returns. If alive, it checks whether or not the NachenBlaster grabbed the goodie; if not, it moves appropriately, and then checks again if it collided with the NachenBlaster.
* virtual void specialized() = 0 – Each goodie gives a unique benefit to the NachenBlaster, so I kept this as pure virtual to make this an ABC (each goodie must be specialized).

ExtraLifeGoodie Actor < Goodie < ExtraLifeGoodie

* virtual void specialized() – Increments the number of lives in the game.

RepairGoodie Actor < Goodie < RepairGoodie

* virtual void specialized() – Increments the HP of the NachenBlaster by 10.

TorpedoGoodie Actor < Goodie < TorpedoGoodie

* virtual void specialized() – Adds 5 torpedoes to the NachenBlaster.

StudentWorld

* ~StudentWorld() – removes all Actors from the game; deletes dynamically allocated variables. Executed by calling cleanup()
* init() – Places 30 stars in the game, and also initializes the NachenBlaster. Returns to continue game as a status, allowing the user to continue playing.
* move() –
  + Determines the number of aliens to be destroyed to pass the level.
  + Iterates through each actor in the StudentWorld’s actor vector and allows it to do something, if it’s not dead. If the NachenBlaster is dead, it returns that the player died and decrements the lives. If the number of aliens destroyed is equal to the number to pass the level, it returns that you finished the level.
  + All actors are iterated through once more; if an actor is marked as dead, it is deleted in memory and removed from the vector.
  + Might place a new star on the side of the screen.
  + Might place a new alien on the screen.
  + Prints out game stat text at the top of the screen.
  + If nothing had returned, then return to continue the game.
* cleanup() –
  + Iterates through each actor and deletes it from memory and removes it from the array.
  + Deletes the NachenBlaster and sets it pointer to nullptr.
* NachenBlaster\* getNachBlaster() – returns the pointer to the game’s NachenBlaster.
* void addActor(Actor\* actor) – adds a pointer to a new actor to the StudentWorld’s Actor vector and increases the game’s actor count.
* int getNumAliensDestroyed() const – Returns the number of aliens the nachenBlaster destroyed.
* void incrementNumAliensDestroyed() – increases the number of aliens killed by one.
* void decrementNumAliens() – deletes one from the alien count on screen.
* bool hitNachBlaster(Actor\* colliding, int hp) – Checks if a collision occurred between an enemy projectile and the NachenBlaster. If so, it reduces the NachenBlaster’s hp by the specified amount, and checks if the NachenBlaster died. It also sets the projectile to dead and returns true if this happened. If no collision occurred, it returns false.
* bool hitDamageableActors(Actor\* colling, int hp) – Checks if a NachenBlaster’s projectile hit an Alien ship. It runs through the vector of actors to see if a collision occurred between an Alien and the colliding object. If so, it causes the alien to suffer damage, and checks if it should be set dead. If so, it’ll call the Alien’s dropSomething() function to potentially drop a goodie, and adjust the alien counts of the StudentWorld accordingly. If this happens, it returns true (that a collision happened), and if no collision happened it returns false.
* bool goodieReceived(Goodie\* goodie) – if a collision occurred between the NachenBlaster and the given goodie, it increases the game’s score, sets the goodie to dead, and calls Goodie’s specialized() function to give benefits accordingly. If the collision occurred, it return true, otherwise, it returns false.
* bool shipCollision(Actor\* alien, NachenBlaster\* nach) – Checks if a collision occurred between the alien ship and NachenBlaster. If so, it causes the NachenBlaster to lose 10hp, and then checks if it should be set dead. It then sets the alien to dead, calls the Alien’s dropSomething() function, and increases score appropriately while also adjusing the alien count variables accordingly. It then introduces an explosion to the game and returns true. Otherwise, this function returns false.
* double euclideanDistance(int x1, int x2, int y1, int y2) – Returns the Euclidean distance between two coordinate points.
* bool collisionOccurred(Actor\* one, Actor\* two) – Returns true if the Euclidean distance between two Actors is less than 0.75 \* sum of the actor’s radii. Otherwise, returns false.

1. **Bugs/Missing Functionality**

I don’t have any known bugs or functionality that I knowingly wasn’t able to implement.

1. **Design Decisions and Assumptions**

I decided to keep my design like the recommended one. Initially, I decided to not make a base class for the Nachenblaster and the Aliens (i.e., no Ship class) because I figured that the two were different other than the fact that they share hp. After finishing, I realized that it would be good practice to abstract them into a higher Ship class, so I implemented that to keep the HP functions out of the NachenBlaster/Alien classes.

I decided to keep all of my pointer to actors EXCEPT the NachenBlaster in their own vector. The reason I wanted to keep the NachenBlaster out of the Actor vector was because I wanted to quickly and easily reference the NachenBlaster pointer without iterating through the entire vector.

I decided to implement all collisions and interactions in the StudentWorld class, whereas the Actor class simply calls the StudentWorld class functions to check for collisions.

1. **Testing Classes**

Star/Actor – To test Actor, I first created the Star class (thereby testing both Actor and Star). In my first few runs, I first tested that with the GWSTATUS\_PLAYER\_DIED, I was able to see stars for a brief moment, and that my Actors were being cleaned up properly. To do this, I put in my destructors print statements that would print out if that Actor was being destroyed. I then added functionality that stars would move across the screen. Since all of the Actors were stored in the same vector (except the NachenBlaster), I was sure that if that code worked, then it should be trivial to delete other new Actors dynamically.

NachenBlaster – Again, I wanted to first be sure that I could get the NachenBlaster to appear at the right coordinates, so without adding any key press functionality, I initialized a pointer to a new NachenBlaster in the game (without adding it to the vector of actors), and made sure that it was deleted after the game was over.

As I kept developing later (adding the possibility of dying, continuing levels, etc.), I kept getting errors that I was trying to double delete pointers—this was due to the fact that the destructor for StudentWorld was also calling cleanUp() after it had already been called by the game framework. To solve this, I had to set the NachenBlaster pointer to null in cleanUp(), then make sure delete was only being called if it wasn’t a null. Again, to make sure things were being deleted, I’d have destructors print out statements that they were being run.

Cabbages – I then started working on cabbages. I initially had trouble trying to determine how I could use the pointer to StudentWorld to dynamically add Actors into its member vector variable, but I eventually came up with a function that would pass in a pointer to an actor (new Cabbage()) from the Actor.cpp and use that to instantiate Cabbages.

To enure that they were being cleaned up when they flew off the screen, I added statements to ensure that the destructor was being called as soon as they were out of range. At this point, I still was not factoring in anything to do with collisions.

Aliens – I first created the Smallgon class without abstracting the Alien class, and got them to show up on the screen. Figuring out the flight path controls were a bit complicated for me, but after I did that, I tried to get collisions to work. To do this, I used the Euclidean distance function to return whether or not two objects collided; if they did, I’d print “collided” to make sure that the collisions were being registered properly.

To test that the proper number of aliens were appearing (because I didn’t get this right the first few times), I’d actually count the number of aliens that I had shot, I had to still shoot, etc. and have those printed out every tick.

Explosions – This wasn’t too difficult to test—I had explosions set themselves dead after four ticks, and would print out in the destructor that they were getting cleaned up. I simply just had to call the addActor() function whenever there was an alien death, and add a new Explosion into the game.

Projectiles + Goodies – These were the last two types of classes that I had to implement. I first tested that I could get these to show up on the screen, without moving, and then that they’d move and set themselves dead after going off the screen. I initially didn’t have Cabbages as projectiles, but I found that this abstraction allowed me to make the code a lot cleaner, especially when it came to Torpedoes. Goodies were fairly straightforward to test—as long as I had the stats printed out on the screen, I could see if the specialized tasks that each goodie had to do were being executed if the score/lives/health etc. changed appropriately.