Matthew Dalton

Project 3 Report

**1) A high-level description of each of your public member functions in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual.**

*Actor:*

virtual void doSomething() = 0;

Declared as public so that my Actor\* list in StudentWorld (m\_actors) can call do something on each of its Actor pointers. Declared as pure virtual because an actor is an abstract thing; there are no actual actors – only objects with an actor base part. Therefore, it does not make sense for a plain actor to do anything.

bool isAlien() const;

Declared as public so that my StudentWorld functions could determine if an Actor pointer was pointing to an alien or not (without using dynamic casting).

bool isDead() const;

Declared as public so that my StudentWorld functions could determine if an Actor was dead (for a multitude of reasons, e.g. whether or not to call *doSomething()*).

*Alien:*

virtual void doSomething();

Declared as public for the same reason as Actor::doSomething(). Declared as virtual because it inherits the function name from its base part (Actor). I decided to implement doSomething in the Alien base class, because the same sequence of steps are to be performed on each alien, but with different specifics; those specifics are protected virtual (some pure, some not) in the Alien class.

void decAlienHealth(int howMuch);

Declared as public so that my StudentWorld class can use it when accounting for collisions between aliens and projectiles or the NachenBlaster.

*Smallgon:*

Smallgon(StudentWorld\* world, double startX, double startY);

Declared as public so that it can be constructed.

*Smoregon:*

Smoregon(StudentWorld\* world, double startX, double startY);

Declared as public so that it can be constructed.

*Snagglegon:*

Snagglegon(StudentWorld\* world, double startX, double startY);

Declared as public so that it can be constructed.

*Explosion:*

Explosion(StudentWorld\* world, double startX, double startY);

Declared as public so that it can be constructed.

virtual void doSomething();

Declared as public for the same reason as Actor::doSomething(). Declared as virtual because it inherits the name from its base part (Actor). I decided to implement doSomething because an explosion is an object that is used throughout the game and it needs its own implementation to act accordingly each tick.

*Goodie:*

virtual void doSomething();

Declared as public for the same reason as Actor::doSomething(). Declared as virtual because it inherits the function name from its base part (Actor). I decided to implement doSomething in the Goodie base class, for the same sequence of steps are to be performed on each goodie, but with a different specific; that specific is a protected pure virtual in the Goodie class.

*ExtraLife:*

ExtraLife(StudentWorld\* world, double startX, double startY);

Declared as public so that it can be constructed.

*Repair:*

Repair(StudentWorld\* world, double startX, double startY);

Declared as public so that it can be constructed.

*TorpedoGoodie:*

TorpedoGoodie(StudentWorld\* world, double startX, double startY);

Declared as public so that it can be constructed.

*NachenBlaster:*

NachenBlaster(StudentWorld\* world);

Declared as public so that it can be constructed.

virtual void doSomething();

Declared as public for the same reason as Actor::doSomething(). Declared as virtual because it inherits the name from its base part (Actor). I decided to implement doSomething because a NachenBlaster is an object that needs its own implementation to act accordingly each tick.

int getHealth() const;

Declared as public so that the StudentWorld class could have access to how many hit points the NachenBlaster has left (so that it can appropriately set the health of the NachenBlaster and so that it can update the status bar.

int getCabbages() const;

Declared as public so that the StudentWorld class can know how many cabbage points the NachenBlaster has in order to update the status bar.

int getTorpedoes() const;

Declared as public so that the StudentWorld class can know how many torpedoes the NachenBlaster has in order to update the status bar.

void setHealth(int amount);

Declared as public so that the StudentWorld class can set the health of the NachenBlaster, as might happen when it calls doSomething on a projectile and then that projectile tells the StudentWorld that it crashed into the NachenBlaster.

void addTorpedoes(int howMany);

Declared as public so that the StudentWorld class can add Torpedoes to the NachenBlaster’s torpedo count (as when a TorpedoGoodie is picked up).

*Projectile:*

virtual void doSomething();

Declared as public for the same reason as Actor::doSomething(). Declared as virtual because it inherits the function name from its base part (Actor). I decided to implement doSomething in the Projectile base class, for the same sequence of steps are to be performed on each Projectile, but with a different specific; that specific is a protected pure virtual in the Projectile class.

*Cabbage:*

Cabbage(StudentWorld\* world, double startX, double startY);

Declared as public so that it can be constructed.

*TorpedoProjectile:*

TorpedoProjectile(StudentWorld\* world, double startX, double startY);

Declared as public so that it can be constructed.

*Turnip:*

Turnip(StudentWorld\* world, double startX, double startY);

Declared as public so that it can be constructed.

*Star:*

Star(StudentWorld\* world, double size, double startY, double startX = VIEW\_WIDTH-1);

Declared as public so that it can be constructed. The last parameter is default because most of the time (save for StudentWorld::init()), when a star is created, it is created at the right-hand-side of the screen.

virtual void doSomething();

Declared as public for the same reason as Actor::doSomething(). Declared as virtual because it inherits the name from its base part (Actor). I decided to implement doSomething because a Star is an object that needs its own implementation to act accordingly each tick.

*StudentWorld:*

virtual int init();

Declared as public to be used by the GameController. Initializes the GameWorld.

virtual int move();

Declared as public to be called by the GameController every tick. Calls doSomething on each actor, determines if the level is over, removes dead actors from the list, and determines whether or not to add a new star or alien to the screen.

virtual void cleanUp();

Declared as public to be used by the GameController. Deletes all data members.

void incNachenBlasterHealth(int howMuch);

Declared as public so that Actor objects (RepairGoodies) can raise the NachenBlaster’s health.

void decNachenBlasterHealth(int howMuch);

Declared as public so that Actor objects (aliens and projectiles) can lower the NachenBlaster’s health.

void incNachenBlasterTorpedoes(int howMuch)

Declared as public so that Actor objects (TorpedoGoodies) can raise the NachenBlaster’s torpedo count.

bool collidedWithNachenBlaster(Actor\* one)

Declared as public so that an actor object can determine if it collided with the NachenBlaster and act accordingly (aliens or projectiles).

bool collidedWithAlien(Actor\* one, int howMuch);

Declared as public so that an actor object can determine if it collided with an alien and act accordingly (NachenBlaster or projectiles).

void addActor(Actor\* newGuy, bool atFront)

Declared as public so that an actor object can add a new object to the m\_actors data member (a projectile or explosion).

bool NachenBlasterInRange(double x, double y)

Declared as public so that an alien can determine whether or not the NachenBlaster is in front of them (then decide whether or not to act upon it).

void incDestroyedAlienCount();

Declared as public so that when an alien dies, it can tell the StudentWorld object that it is one alien close to completing the level.

~StudentWorld();

Declared as public for obvious reasons. It’s only line of code is to call cleanUp().

**2) A list of all functionality that you failed to finish as well as known bugs in your classes.**

As far as I know, my NachenBlaster game is completely functional (as per the spec and provided executable). There are no known bugs in my classes. However, one thing that doesn’t work (in mine or the provided) is play SOUND\_FINSIHED\_LEVEL. I suspect it is overridden by the SOUND\_DEATH or SOUND\_BLAST that inherently precedes it.

**3) A list of other design decisions and assumptions you made.**

I chose to ignore the provided class hierarchy for Project 3 and continue with what I was planning to do (for better or for worse). Most of my design decisions adhere strictly to what was specified in the script, but for certain cases, I had to either refer to the provided executable or make an executive decision. Here are design decisions and assumptions I made (other than the public virtual/non-virtual functions already described above):

1) Perhaps the most blaring “why is this not virtual everywhere” function in my implementation is the protected/private one, “collisionCheck()”. This function exists for all objects that interact with others (NachenBlaster, Aliens, Projectiles, and Goodies). Although all of these classes and their descendants use this function, I did not think it a sufficient reason to factor it out and create a new base class for all of these classes. Besides, the name is sort of generic; what actually goes on in that function differs widely from class to class.

2) I declared all base class constructors as protected because no outside classes should ever call them.

3) I used the virtual function “outOfBounds” in my actor class so that an object could easily tell if it flew off the screen in the x direction. The function was declared virtual so that goodies could have their own personal implementation that checked for the y-direction as well.

4) I kept 3 data members for all Actors: two bools (m\_alien and m\_dead) and a StudentWorld\* (m\_world). The first two are pretty self-explanatory. The third was used to allow for two-way interactions between StudentWorld and its objects.

5) I have a move function for each class that had to move (named move*ClassName*()). I didn’t factor this out into a base class (and make it virtual) for the same reason as collisionCheck().

6) I have three data members for my NachenBlaster class (m\_hitPoints, m\_cabbageEnergy, m\_torpedoCount). Their names are pretty self-explanatory. My alien class also has a hit points data member, but I did not see this a sufficient reason to create a base class between the two (no code was duplicated, just a [private] variable name).

7) I kept a data member for explosions (m\_tickNumber) which keeps track of how long it’s been alive so that it knows when to die.

8) My alien base class is interesting. While reading the spec, I noticed many similarities between two of the aliens (Smallgon and Smoregon) that differed in Snagglegons. Instead of making another base class for those two aliens, I decided to keep a data member (m\_isSnagglegon). This way, I was able to generalize all but two functions for the three aliens, even when certain things differed (e.g. how many hit points to start out with). Other data members include ones for flight direction, speed, and plan length, the last of which was only significant to the first two aliens, and so was maintained at 1 for the Snagglegon (I apologize for such atrocities, but I thought they made the layout cleaner and easier to grasp).

9) Adding to the alien class, perhaps the most important function is decAlienHealth(int howMuch). The way I implemented the game meant that this function was to be called by the StudentWorld class whenever a projectile or the NachenBlaster collided with an alien. However, different things were to happen depending on which object the alien collided with (NachenBlaster or projectile). I found an issue with how to determine how much health to take away from the NachenBlaster when it collided with an alien, for it differed depending on whether or not it was a Snagglegon. I could have made a function akin to isAlien() but for Snagglegons, but I decided against that. Thus, when a NachenBlaster collided with an alien, this function was called with the parameter SHIP\_KILL\_ALIEN, which is a const int set to 10 million. This way, the alien will never have more hit points than 10 million (and will always die) and it is an easy way to check if it was a NachenBlaster that killed the alien (instead of maybe passing a bool parameter).

10) As far as flying off of the screen goes, I chose to not check for if the (x,y) coordinates left the screen, but accounted for the radius as well (as seen in isOutOfBounds()).

11) I kept no other data members for my derived actor objects, but there were still some interesting things: Each goodie had its own implementation of benefit(), which was called in the case a goodie collided with the NachenBlaster, and each projectile had its own method for moving and colliding with enemies. The cabbages and turnips were easy. However, for torpedoes, I added the extra check (using its direction) to see if it was aiming for aliens or the NachenBlaster (a fairly easy check, that is).

12) An important decision I made regarding how to deal with checking for collisions is as follows: Originally, I didn’t think it necessary for the NachenBlaster to check for colliding with anything (the other objects would take care of that), but then I thought of the case when a projectile is added to the screen on top of a NachenBlaster, and then it moved before the projectile was asked to do something. Fortunately, I avoided this issue by using a list container for my objects and a separate data member for my NachenBlaster. As the list iterates through, if an object is added to the end of the list by another object, then that added object will get a chance to do something before the loop is over. However, the reverse case of a NachenBlaster adding a projectile on top of an alien ship would not work the same (the added projectile would go after the alien), and so I originally thought that aliens would have to check for projectiles. However, I came up with the idea of pushing projectiles added by the NachenBlaster on to the front of the list. This way they would check for a collision with aliens before the aliens could move. Therefore, Aliens need only check for collisions with the NachenBlaster and vice versa.

13) I chose to have a relatively self-explanatory interface for my StudentWorld class, with many public function that performed one line of code that did exactly as they said (usually by calling a corresponding function on one of its members). I kept four data members: m\_actors (list<Actor\*>), m\_ship (NachenBlaster\*), m\_destroyedAliens (int), and m\_aliensToKill (int). These are pretty self-explanatory; the latter two are used in determining whether or not the level is over and how many aliens to display on the screen. All StudentWorld private functions were implemented to make the code cleaner on its more important functions (and avoid repeat code).

14) I removed many comments that were originally in the source code to make it more aesthetically pleasing. However, I left in ones that I felt were important; most of my original ones were left over from when I commented in my plans before implementations. There are no difficult algorithms to understand in this program. Difficulty of comprehension lies in understanding how the functions interact with each other; that is what this report is for.

**4) A description of how you tested each of your classes (1-2 paragraphs per class).**

I will try to be as extensive as I can for how I tested each of my classes, but the truth is that there just wasn’t much testing to be done for some of them, and a lot of the same methods were repeated for classes. Also, I tried to extensively plan out classes and their interactions with each other before implementing them, so as to eliminate a need for special tests on certain things (i.e. in addition to normal gameplay). Testing through gameplay included many uses of the ‘f’ key to freeze and break down certain movements, as well as comparisons to the provided executable. Many of these tests involved checking for sounds, and so headphones were used (a good chunk of the work was done in Powell Library). Furthermore, testing on abstract base classes was more abstract (a thought process), and therefore the testing of their capabilities was done through their derived classes, and so I will not write separate descriptions for them.

*Smallgon:*

As with all of my classes, I implemented their interactions with other classes last, and so my original testing included adding code in my StudentWorld::move() function to add a new Smallgon as often as it added a new star. This way, I was able to make sure that the graphics were correct and that it moved in the correct ways. Next I added its shooting capabilities, and I spawned many of them directly across from the NachenBlaster to get it to shoot at it. Lastly, I added the capability to crash into the NachenBlaster; this was tested more so through gameplay then anything specific (and it surfaced a bug – the level wasn’t ending when the last alien was killed by a collision with the NachenBlaster – which was easily rectifiable).

*Smoregon:*

I tested the Smoregon much in the same way I tested the Smallgon: adding a bunch onto the screen and making sure their actions were in order. The main difference is that the Smoregon could drop a goodie, and so I had to make sure that that was properly taken care of.

*Snagglegon:*

Snagglegons didn’t really require much testing besides an equivalent of what was performed on the other aliens. As with the others, I took note of how the movements should be and made sure they matched up (which was easier to do with a Snagglegon as there is no flight plan length to be taken into account).

*Explosion:*

Explosions aren’t very complex. Simple testing was performed by hitting the ‘f’ key right before they were about to happen and then go through them tick by tick, making sure they worked exactly the same as those in the provided executable.

*ExtraLife:*

As with all goodies, the ExtraLife was tested by dropping it with a fixed chance at the top center of the screen, so that it could easily be grabbed by the user. Then, I made sure that the status bar was properly updated and that I received the benefits that I should.

*Repair:*

This testing was painfully similar to that above and so I shan’t write anymore.

*TorpedoGoodie:*

This was the same as the other two goodies, except that it sort of went hand in hand with testing torpedo projectiles, for I could receive them through the goodies and then shoot them.

*NachenBlaster:*

There were many tests I had to perform on the NachenBlaster that were unnecessary on other objects. I tested keyboard input as one might imagine and made sure that the boundaries it could travel were the same as in the provided executable. Like the other classes, implemented collisions last. This was done easily by randomly adding projectiles on the screen and making sure they did the right thing when colliding, as well as adding a bunch of ships so that I could ram into them. Its data members were tested via the status bar, which kept me updated on them.

*Cabbage:*

Cabbages inherently tested in the tests on aliens, except that the aliens weren’t *really* handling the collisions, the cabbages were. Cabbages don’t really have much else to do besides move across the screen and check for collisions (as defined in Projectile::doSomething()) or if it’s out-of-bounds.

*TorpedoProjectile:*

The torpedo projectile had to be tested slightly differently, for my implementation had projectiles that could either be attacking aliens or attacking the NachenBlaster. Although it sounds slightly more complex, the testing was as easy as adding torpedoes to m\_actors at random locations with different directions and making sure they acted properly.

*Turnip:*

The turnip was tested very similarly to the cabbage, except that it attacks the NachenBlaster instead of aliens.

*Star:*

The testing for stars was the easiest of them all, because the testing was how they would actually be implemented (i.e. I didn’t have to remove the code after testing). Basically, I just added stars to the screen and watched them drift, making sure they did so in the same manner as in the provided executable.

*StudentWorld:*

My student world class was tested through my actor objects. Indeed, the actor objects couldn’t do anything without my StudentWorld class, and so any test I performed on them was really a test on my StudentWorld class. I did not perform any additional testing, although a bug was picked up very early on (before part 1): when deleting objects from the game, I forgot to set the iterator to the return value of erase(), and so I was performing undefined behavior in trying to access dangling pointers (this was solved, of course, by a glance at my solution for the Project 3 warmup).