Yash Moondhra

Project 3 – Computer Science 32

Due Date: 1 March 2018

Part 1: Higher-Level Definitions

*Write 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. For example, “I chose to define a pure virtual version of the sneeze() function in my base Actor class because all actors in NachenBlaster are able to sneeze, and each type of actor sneezes in a different way.”*

# StudentWorld : public GameWorld

StudentWorld(std::string assetDir);

Constructor for the student world creates a new StudentWorld with a GameWorld base. Necessary to create a StudentWorld object.

virtual int init();

1. Initializes the data structures used to keep track of the game’s world (i.e. aliens destroyed / need to be destroyed, maxShips)
2. Allocates and inserts a valid NachenBlaster object
3. Allocates and inserts 30 stars into the world

I chose to define this because it is a pure virtual function of GameWorld and is called by the GameController class.

virtual int move();

1. For each alive actor, including the NachenBlaster, call doSomething(). Check if the NachenBlaster died. Check if the number of aliens need to be destroyed is 0, meaning the level is over
2. Remove the dead objects in the game
3. Calculate if an alien needs to be added to the game
   1. If so, use probability and the randInt() function to decide which type of alien needs to be added
4. Update the status text on the top of the screen with the latest information (score, level, health percentage, cabbage points percentage, and number of torpedoes) using a string stream
5. Using probability, calculate whether or not a star must be added to the game
6. Return the status signifying that the game must continue

I chose to define this because it is a pure virtual function of GameWorld and is called by the GameController class.

virtual void cleanUp();

1. If the pointer to the NachenBlaster is not null, delete the NachenBlaster and set the pointer to null
2. If the list of actors (actorList) is not empty, delete all the actors if their pointers are not pointing to null

I chose to define this because it is a pure virtual function of GameWorld and is called by the GameController class.

void addActor(Actor\* a);

Pushes an actor onto the list of actors (actorList).

I chose to define this as public so that actors can add other types of actors to the game (i.e. NachenBlaster adds cabbage to the game).

void increaseNumAliensDestroyed(int i);

Increases the variable “numAliensDestroyed” by one.

I chose to define this as public so that when an alien gets killed by a NachenBlaster or projectile of the NachenBlaster, it can call this function with parameter equal to 1.

NachenBlaster\* getNach() const;

Returns the pointer to the NachenBlaster

I chose to define this as public so that actors can see if it collided with the NachenBlaster by comparing the actor pointer (i.e. Actor\* collidedWithThis) which it collided with to the pointer to the NachenBlaster (i.e. if(collidedWithThis == getWorld()->getNach())

bool allCollisions(Actor\* a);

1. By using the euclidean distance function (which is private), this function checks if the inputted actor (\*a) collided with the NachenBlaster or each of the actors on the actorList. If so, it calls the function “collide” for each actor it collided with, except itself. If the collide function returns true, meaning that the object collided with something relevant (i.e. an alien colliding with a NachenBlaster), then allCollisions will be set to true. Otherwise, it will return false. The purpose of this is so that projectiles and goodies can return immediately if they collide with relevant objects.

I chose to define this as public so that each actor can call this function in its doSomething() function, rather than just calling its own “collide” function. The purpose of this is that it can collide with other objects that are not relevant in the same tick (i.e. a goodie colliding with a star and the NachenBlaster at the same time).

virtual ~StudentWorld();

This function deletes the StudentWorld object and all of its dynamically allocated storage by calling cleanUp().

I chose to make this public because another class calls “delete gw” where gw is an object of type StudentWorld. It is virtual because it must delete the GameWorld base part of itself along with any variables.

# Actor – Abstract Base Class

NOTE: All of the following functions are public because the different types of actors need to be able to access and/or change the private variables of Actor, which is their base.

Actor(StudentWorld\* s, int health, int imageID, double startX, double startY, int dir, double size, int depth);

Constructor for the actor creates a new Actor with a GraphObject base. Necessary to create an Actor object.

virtual ~Actor();

Destructor for the Actor is set to virtual so that it deletes the GraphObject base along with all variables of Actor.

void nowDead();

Sets the bool variable “dead” of the Actor to true, meaning that the actor is now dead.

I chose to make this not virtual because all types of actors can die.

bool isDead() const;

Returns the bool variable “dead” of the Actor.

I chose to make this not virtual because it is simply an accessor method for a private member of Actor.

int getHealth() const;

Returns the integer variable “health” of the Actor.

I chose to make this not virtual because it is simply an accessor method for a private member of Actor.

void subtractHealth(int h); //sets health to [health - h] and if object dies, calls nowDead

If the actor is a ship, this function sets the private integer variable “health” to health-h. If the object’s health goes below zero, it calls kill(). If the health goes above 50 and it is a NachenBlaster, then recursion is used to subtract the required amount of health from the NachenBlaster.

I chose to make this public so that other actors can alter the health of the ships. I chose to not make it virtual because I added a check in the function to see if the Actor it is subtracting the health from is an alien or NachenBlaster. If it is not, then it won’t do anything.

void makeAlienTrue();

Sets the bool variable “alien” of the Actor to true, meaning that the actor is an alien.

I chose to not make this virtual because all actors either are or aren’t aliens.

void makeAlienFalse();

Sets the bool variable “alien” of the Actor to false, meaning that the actor is not an alien.

I chose to not make this virtual because all actors either are or aren’t aliens.

bool isAlien() const;

Returns the bool variable “alien” of the Actor.

I chose to make this not virtual because it is simply an accessor method for a private member of Actor.

bool isSnagglegon() const;

Returns the bool variable “snagglie” of the Actor, which holds whether or not the Actor is a snagglegon. The purpose of having this is to know how much to increase the score by (250 if false or 1000 if true).

I chose to make this not virtual because it is simply an accessor method for a private member of Actor.

void makeSnagglegonTrue();

Sets the bool variable “snagglie” of the Actor to true, meaning that the actor is a snagglegon.

I chose to not make this virtual because all actors either are or aren’t snagglegons.

bool flownOffScreen() const;

Returns whether or not the object flew off the screen (X or Y is < 0 or >= VIEW\_WIDTH or VIEW\_HEIGHT).

I chose to make this not virtual because all actors can fly off the screen in this way except ships (they bounce off the top and bottom of the screen.

virtual void kill(); //different than move off screen

Returns return. Does nothing else. This is different than moving off a screen. This is murder.

I chose to make this virtual because I wanted it to only apply for ships, but I wanted other actors to be able to call the function using only an Actor pointer (i.e. Actor\* a; a->kill()) for when they collide with the ship.

StudentWorld\* getWorld();

Returns a pointer to the StudentWorld object it is in.

I chose to make this not virtual because I wanted all objects to be able to access the “allCollisions()” and “getNach()” functions of StudentWorld.

virtual void move() = 0;

A pure virtual function that defines the movement of an actor in one tick.

I chose to make this pure virtual because I wanted each actor to move in a different way (i.e. cabbage moves some amount of pixels to the right while a snagglegon bounces off the top and bottom of the screen.

virtual bool collide(Actor\* a) = 0;

A pure virtual function that defines what happens once a collision between another actor and the current actor occurs. It is called in the allCollisions function of StudentWorld.

I chose to make this pure virtual because I wanted each actor to perform a unique set of actions once it collided with something. Also, each actor has different relevant things it can collide with (i.e. cabbage with aliens, goodies with NachenBlaster).

virtual void doSomething() = 0;

A pure virtual function that defines what each type of actor should do in one tick. It calls allCollisions to see if that actor collided with any relevant objects.

I chose to make this pure virtual because I wanted each actor to perform a unique set of actions in each tick. All types of actors call this function.

# AlienOrNach – Abstract Base Class: Inherits Actor

AlienOrNach(StudentWorld\* s, int health, int imageID, double startX, double startY, int dir, double size, int depth, int f, double t);

Constructor for the AlienOrNach creates a new AlienOrNach with a Actor base. Necessary to create an AlienOrNach object. Calls makeAlienTrue(), sets the flightPlanLength to 0, and sets travelSpeed and flightPlan equal to the input.

virtual ~AlienOrNach();

Destructor for the AlienOrNach is set to virtual so that it deletes the Actor base along with all variables of AlienOrNach.

int getFlightPlan();

Returns the integer variable “flightPlan” of the AlienOrNach.

I chose to make this not virtual because it is simply an accessor method for a private member of AlienOrNach.

void setFlightPlan(int f);

Sets the integer variable “flightPlan” of the AlienOrNach to int f.

I chose to not make this virtual because all AlienOrNach objects should have a flightPlan (except NachenBlaster, it is 0).

int getFlightPlanLength();

Returns the integer variable “flightPlanLength” of the AlienOrNach.

I chose to make this not virtual because it is simply an accessor method for a private member of AlienOrNach.

void setFlightPlanLength(int fl);

Sets the integer variable “flightPlanLength” of the AlienOrNach to int fl.

I chose to not make this virtual because all AlienOrNach objects should have a flightPlanLength (except NachenBlaster, it is 0).

double getTravelSpeed();

Returns the integer variable “travelSpeed” of the AlienOrNach.

I chose to make this not virtual because it is simply an accessor method for a private member of AlienOrNach.

void setTravelSpeed(double t);

Sets the integer variable “travelSpeed” of the AlienOrNach to double t.

I chose to not make this virtual because all AlienOrNach objects should have a travelSpeed (except NachenBlaster, it is 0).

void shoot();

1. If the AlienOrNach is an alien
   1. If it is a snagglegon, torpedo sound will be played and torpedo will be added
   2. Else a turnip will be added and sound\_alien\_shoot will be played.
2. Else if it is the NachenBlaster
   1. New cabbage will be added, sound\_player\_shoot will be played, and the appropriate amount of cabbage points will be subtracted using the function “addCabbagePoints()”.

I chose to not make this virtual because the only types of actors that can shoot are AlienOrNach objects.

virtual void kill();

If the AlienOrNach is not dead, it is set to dead. If the AlienOrNach is an alien, then a new explosion is added, dropGoodie() is called, sound\_death is played, increaseNumAliensDestroyed(1) is called, and the appropriate score is added to the current score.

I chose to not make this virtual because all AlienOrNach objects should have a travelSpeed (except NachenBlaster, it is 0).

virtual void dropGoodie() = 0;

This function is only for aliens and has the purpose of dropping a goodie, if the probability calls for such.

I chose to not make this pure virtual because it should do nothing for the NachenBlaster, and each of the three types of aliens should have its own unique way to drop goodies.

## NachenBlaster – Inherits AlienOrNach

NachenBlaster(StudentWorld\* s);

Constructor for the NachenBlaster creates a new NachenBlaster with an AlienOrNach base. Necessary to create an NachenBlaster object. Sets cabbagePoints to 30, calls makeAlienFalse(), and sets the amount of torpedoes to 0.

virtual ~NachenBlaster();

Destructor for the NachenBlaster is set to virtual so that it deletes the AlienOrNach base along with all variables of NachenBlaster.

virtual void doSomething();

If the actor is not dead:

1. Add one cabbagePoint using the addCabbagePoints() function
2. Call move()
3. If the health is <= 0 call nowDead() which sets the NachenBlaster to dead.

virtual void move();

Receives an input of a key

* 1. If an arrow key, move appropriately
  2. If spacebar, call shoot() which tries to shoot a cabbage
  3. If tab, call shootTorpedo() which tries to shoot a torpedo

int getCabbagePoints();

Returns the integer variable cabbagePoints

void addCabbagePoints(int num);

Adds num to the integer cabbagePoints.

If cabbagePoints is < 0, set it to 0.

Else if cabbagePoints is > 30, set it to 30.

void shootTorpedo();

If the integer variable “torpedoes” is greater than 0, then create a new torpedo, add it to the actorList, play sound\_torpedo, and decremenet the torpedoes variable by 1.

void incTorpedoes(int num);

Increase the integer “torpedoes” variable by num.

int getNumTorpedoes();

Return the integer variable “torpedoes”.

virtual void dropGoodie();

Return. A NachenBlaster cannot drop goodies.

virtual bool collide(Actor\* a);

Return. All of the collisions with the NachenBlaster are taken care of in other types of actors’ collide functions.

## Smallgon – Inherits AlienOrNach

Smallgon(StudentWorld\* s, int health);

Constructor for the Smallgon creates a new Smallgon with an AlienOrNach base. Necessary to create an Smallgon object.

virtual ~Smallgon();

Destructor for the Smallgon is set to virtual so that it deletes the AlienOrNach base along with all variables of Smallgon.

virtual void doSomething();

If it’s dead or flew off the left side of the screen, return.

Call the allCollisions function with the parameter (this).

If the alien died, return immediately.

If it hits the top of the screen, make its flight plan down left.

If it hits the bottom of the screen, make its flight plan up left.

If the flightPlanLength is 0, randomly reset to either left, down left, or up left.

If any of the three statements above are true, randomly reset the flightPlanLength between 1 and 32.

If the NachenBlaster is to the left of this alien and is within 4 y-coordinates, there is a chance that it will fire a turnip at the NachenBlaster.

Call move() and then call allCollisions(this).

virtual void move();

If it flew off the left side of the screen, call nowDead() and return.

Move in the direction of the flight plan by getTravelSpeed() pixels.

Decrement the flightPlanLength by 1.

virtual bool collide(Actor\* a);

If this alien collided with a NachenBlaster, subtract 5 health from the NachenBlaster using subtractHealth(5), call kill() on the alien, and return true. Otherwise, return false, meaning that the alien collided with nothing relevant.

virtual void dropGoodie();

Return, because smallgons cannot drop goodies.

## Smoregon – Inherits AlienOrNach

Smoregon(StudentWorld\* s, int health);

Constructor for the Smoregon creates a new Smoregon with an AlienOrNach base. Necessary to create an Smoregon object.

virtual ~Smoregon();

Destructor for the Smoregon is set to virtual so that it deletes the AlienOrNach base along with all variables of Smoregon.

virtual void doSomething();

If it’s dead or flew off the left side of the screen, return.

Call the allCollisions function with the parameter (this).

If the alien died, return immediately.

If it hits the top of the screen, make its flight plan down left.

If it hits the bottom of the screen, make its flight plan up left.

If the flightPlanLength is 0, randomly reset to either left, down left, or up left.

If any of the three statements above are true, randomly reset the flightPlanLength between 1 and 32.

If the NachenBlaster is to the left of this alien and is within 4 y-coordinates, there is a chance that it will fire a turnip at the NachenBlaster. There is also a chance that it will change its flight direction.

Call move() and then call allCollisions(this).

virtual void move();

If it flew off the left side of the screen, call nowDead() and return.

Move in the direction of the flight plan by getTravelSpeed() pixels.

Decrement the flightPlanLength by 1.

virtual bool collide(Actor\* a);

If this alien collided with a NachenBlaster, subtract 5 health from the NachenBlaster using subtractHealth(5), call kill() on the alien, and return true. Otherwise, return false, meaning that the alien collided with nothing relevant.

virtual void dropGoodie();

There is a 1/3 chance it will drop a goodie. If it will drop a goodie, there is a 50% chance it will be a FlatulenceTorpedoGoodie and a 50% chance it will be a RepairGoodie.

## Snagglegon – Inherits AlienOrNach

Snagglegon(StudentWorld\* s, int health);

Constructor for the Snagglegon creates a new Snagglegon with an AlienOrNach base. Necessary to create an Snagglegon object. Calls makeSnagglegonTrue(), which makes the private bool variable of Actor called “snagglie” true.

virtual ~Snagglegon();

Destructor for the Snagglegon is set to virtual so that it deletes the AlienOrNach base along with all variables of Snagglegon.

virtual void doSomething();

If it’s dead or flew off the left side of the screen, return.

Call the allCollisions function with the parameter (this).

If the alien died, return immediately.

If it hits the top of the screen, make its flight plan down left.

If it hits the bottom of the screen, make its flight plan up left.

If the NachenBlaster is to the left of this alien and is within 4 y-coordinates, there is a chance that it will fire a torpedo at the NachenBlaster.

Call move() and then call allCollisions(this).

virtual void move();

If it flew off the left side of the screen, call nowDead() and return.

Move in the direction of the flight plan by getTravelSpeed() pixels.

virtual bool collide(Actor\* a);

If this alien collided with a NachenBlaster, subtract 15 health from the NachenBlaster using subtractHealth(15), call kill() on the alien, and return true. Otherwise, return false, meaning that the alien collided with nothing relevant.

virtual void dropGoodie();

There is a 1/6 chance it will drop an ExtraLifeGoodie.

## Star – Inherits Actor

//creating a new star at beginning of game

Star(StudentWorld\* s);

Constructor for the Star creates a new Star with an Actor base. Necessary to create a star object.

//creating a new star in the middle of the game. No matter what x value is inputted, it will always be 0

Star(StudentWorld\* s, double startX);

Constructor for the Star creates a new Star with an Actor base. Necessary to create a star object for when it is added during or after the first tick (not in the StudentWorld::init() function. This is because the startX is always set to VIEW\_WIDTH-1.

virtual ~Star();

Destructor for the Star is set to virtual so that it deletes the Actor base along with all variables of Star.

virtual void doSomething();

If the star hasn’t flown off the left side of the screen, call move(). Otherwise, call nowDead() on the star.

virtual void move();

If the star has flown off the screen, call nowDead() and return. Otherwise, move one coordinate to the left (getX() – 1).

virtual bool collide(Actor\* a);

Return false because stars cannot collide with any actors.

## Explosion – Inherits Actor

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

Constructor for the Explosion creates a new Explosion with an Actor base. Necessary to create a Explosion object. Set the numTicksAlive to 0.

virtual ~Explosion();

Destructor for the Explosion is set to virtual so that it deletes the Actor base along with all variables of Explosion.

virtual void doSomething();

The explosion must show for four ticks and then disappear.

virtual void move();

Return. Explosions can’t move.

virtual bool collide(Actor\* a);

Return false. Explosions cannot collide with any actors.

# Projectile – Abstract Base Class: Inherits Actor

Projectile(StudentWorld\* s, int imageID, double startX, double startY, int dir, double size, int depth);

Constructor for the Projectile creates a new Projectile with an Actor base. Necessary to create a Projectile object.

virtual ~Projectile();

Destructor for the Projectile is set to virtual so that it deletes the Actor base, which it would do even if it’s not virtual.

virtual void doSomething();

If it is dead or has flown off the right side of the screen, return. If it collided with a relevant object (an alien) return. Call move(), and then if it collided with a relevant object (an alien), return.

virtual void move() = 0;

Defines the movement of a projectile.

I made this function pure virtual because each projectile moves in a different way.

## Cabbage: Inherits Projectile

Cabbage(StudentWorld\* s, int health);

Constructor for the Cabbage creates a new Cabbage with an Projectile base. Necessary to create an Cabbage object.

virtual ~Cabbage();

Destructor for the Cabbage is set to virtual so that it deletes the Projectile base along with all variables of Cabbage.

virtual bool collide(Actor\* a);

If the cabbage collided with an alien and is not dead and the alien isn’t dead, subtract 2 health from the alien. If the alien did not die, play sound\_blast. Mark the cabbage as dead and return whether or not it collided with a relevant object.

virtual void move();

If it flew off the screen, set itself as dead and return. Otherwise, move 8 pixels to the right and tilt itself 20 degrees counter-clockwise.

## Turnip: Inherits Projectile

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

Constructor for the Turnip creates a new Turnip with an Projectile base. Necessary to create an Turnip object.

virtual ~Turnip();

Destructor for the Turnip is set to virtual so that it deletes the Projectile base along with all variables of Turnip.

bool collide(Actor\* a);

If the turnip collided with the NachenBlaster and is not dead and the NachenBlaster isn’t dead, subtract 2 health from the NachenBlaster. Play sound\_blast. Mark the turnip as dead and return whether or not it collided with a relevant object.

virtual void move();

If it flew off the screen, set itself as dead and return. Otherwise, move 6 pixels to the left.

## FlatulenceTorpedo: Inherits Projectile

FlatulenceTorpedo(StudentWorld\* s, double startX, double startY, int dir);

Constructor for the FlatulenceTorpedo creates a new FlatulenceTorpedo with an Projectile base. Necessary to create an FlatulenceTorpedo object.

virtual ~FlatulenceTorpedo();

Destructor for the FlatulenceTorpedo is set to virtual so that it deletes the Projectile base along with all variables of FlatulenceTorpedo.

virtual bool collide(Actor\* a);

If it is not dead and the thing it collided with is not dead

If the direction is zero and it collided with an alien OR if the direction is 180 and it collided with a NachenBlaster

Subtract 8 health, set itself to nowDead. If it collided with a Nachenblaster, play the sound\_blast. Otherwise, if it collided with an alien that is not dead, play sound\_blast.

Return whether or not it collided with something relevant.

virtual void move();

If it flew off the screen, set itself as dead and return. Otherwise, move 8 pixels to the left or right depending on the direction.

# Goodie – Abstract Base Class: Inherits Actor

Goodie(StudentWorld\* s, int imageID, double startX, double startY, int dir, double size, int depth);

Constructor for the Goodie creates a new Goodie with an Actor base. Necessary to create a Goodie object.

virtual ~Goodie();

Destructor for the Goodie is set to virtual so that it deletes the Actor base, which it would do even if it’s not virtual.

virtual void move();

If it flew off the screen, set itself as dead and return. Otherwise, move down and left.

virtual void doSomething();

If it flew off the screen or is dead, return. If it collided with a relevant object, return. Call move(), then check again if it collided with a relevant object by calling allCollisions.

virtual void collide(Actor\* a)

Returns whether or not it collided with a relevant object (NachenBlaster). Call activate() if this is true.

virtual void activate() = 0;

This function activates the goodie.

I chose to make it pure virtual because each goodie benefits the NachenBlaster in a different way.

## ExtraLifeGoodie: Inherits Goodie

ExtraLifeGoodie(StudentWorld\* s, double startX, double startY);

Constructor for the ExtraLifeGoodie creates a new ExtraLifeGoodie with an Goodie base. Necessary to create a ExtraLifeGoodie object.

virtual ~ExtraLifeGoodie();

Destructor for the Goodie is set to virtual so that it deletes the Actor base, which it would do even if it’s not virtual.

virtual void activate();

Increases the score by 100, plays sound\_goodie, increases the number of lives, and sets itself to dead using nowDead().

## RepairGoodie: Inherits Goodie

RepairGoodie(StudentWorld\* s, double startX, double startY);

Constructor for the RepairGoodie creates a new RepairGoodie with an Goodie base. Necessary to create a RepairGoodie object.

virtual ~RepairGoodie();

Destructor for the RepairGoodie is set to virtual so that it deletes the Goodie base, which it would do even if it’s not virtual.

virtual void activate();

Increases the score by 100, plays sound\_goodie, subtracts -10 health (basically increasing the health by 10), and sets itself to dead using nowDead().

## FlatulenceTorpedoGoodie: Inherits Goodie

FlatulenceTorpedoGoodie(StudentWorld\* s, double startX, double startY);

Constructor for the FlatulenceTorpedoGoodie creates a new FlatulenceTorpedoGoodie with an Goodie base. Necessary to create a FlatulenceTorpedoGoodie object.

virtual ~FlatulenceTorpedoGoodie();

Destructor for the FlatulenceTorpedoGoodie is set to virtual so that it deletes the Goodie base, which it would do even if it’s not virtual.

virtual void activate();

Increases the score by 100, plays sound\_goodie, increases the number of torpedoes by 5, and sets itself to dead using nowDead().

Part 2: Bugs

*Write a list of all functionality that you failed to finish as well as known bugs in your classes, e.g. “I didn’t implement the Turnip class.” or “My Snagglegon doesn’t work correctly yet so I treat it like a Smoregon right now.”*

I finished everything. There was nothing I did not complete. All test cases should work. There are no bugs in my classes. The only thing that could slightly be seen as a “bug” is that the probability of creating a new star is to the accuracy of 6 decimal places,

Part 3: Design Decisions and Assumptions

*A list of other design decisions and assumptions you made; e.g., “It was not specified what to do in situation X, so this is what I decided to do.”*

The spec was very details, so most of the questions I had were answered by it, but here is a list of assumptions and design choices I made:

Before delving into the actual program:

* I made the decision to have an overarching base class “Actor”, which all classes (except Student World) were in some form derived from.
* I assumed that the ships (3 types of aliens and the NachenBlaster) would have similar characteristics, so I created the base class AlienOrNach.
* I assumed that the three types of goodies would have similar characteristics and three types of projectiles would have similar characteristics, so I made the base classes Goodie and Projectile.

During Programming

* In the spec details about the init() function, it said to initialize any data structures that we were going to use, but I didn’t have any data structures that needed to be initialized that couldn’t or shouldn’t have been initialized in the constructor. I instead initialized a few members such as an integer variable.
* I assumed that we were allowed to have public member functions in StudentWorld
* The probabilities of almost all events occurring (i.e. which type of alien to create every tick) required probabilities using doubles, but we had to use randInt. I found a way around this, by multiplying the ratio for everything that required doubles by the level, but this method did not work for creating stars.

Part 4: Testing

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

# StudentWorld

To test student world, I ran the game, clicked enter to see if a new level started and the NachenBlaster was displayed along with 30 stars, waited a few seconds to see if more stars and aliens started appearing.

I moved around using arrow keys, clicked the space-bar to see if cabbages came, and clicked the tab button to see that no torpedoes came out. I saw that there was a lot of smallgons, decent amount of smoregons, and few snagglegons, which roughly falls in line with what is supposed to happen.

Then, I completed a level and saw that a new screen appeared that I completed a level. I saw that my health, cabbagePoints, and number of torpedoes reset. I also died and saw that everything reset.

I restarted the game, and killed 9 aliens on Round 1. I saw that only one alien came at me at a time, meaning that the adding-an-alien part of my move() function worked.

Moreover, I went line-by-line through the entire class and compared it with the spec multiple times to see if there were any mistakes.

# Actor

I made sure that all the types of actors were constructed properly, and that there were no memory leaks. Then, I checked each function by going to where the other actors called these functions and see if they returned the correct values by using breakpoints and “cout” statements. I also checked if the Actor was abstract by trying to create a new Actor object in the initialize function of StudentWorld, which failed, which is what is supposed to happen.

Moreover, I went line-by-line through the entire class and compared it with the spec multiple times to see if there were any mistakes.

# AlienOrNach

I made sure that all the aliens and the NachenBlaster were constructed properly. I tested each function by going to where the other actors called these functions and saw that they returned the proper values and altered the private variables of the AlienOrNach class in the appropriate way. I also checked if the AlienOrNach class was abstract by trying to create a new AlienOrNach object in the initialize function of StudentWorld, which failed, which is what is supposed to happen.

# NachenBlaster

I made sure that when the game started and I clicked enter, a ship appeared. Then, I tried to move the ship around using the arrow keys, clicked space bar to see if the percentage of cabbages remaining changed and went back up to 100% and saw that a cabbage was visually displayed. Also, once I picked up a FlatulenceTorpedoGoodie, I shot the torpedoes and saw that my number of torpedoes changed appropriately and the torpedo was displayed properly. Moreover, when a new level was started or I died, all the displayed variables at the top reset except my score.

Moreover, I went line-by-line through the entire class and compared it with the spec multiple times to see if there were any mistakes.

# Smallgon/Smoregon/Snagglegon (Aliens)

I checked to see that when I created a new level, each of these ships were displayed with the correct image. I saw that when each alien collided with the NachenBlaster, it would create an explosion, the sound was played, the alien disappeared, the score was increased, and the NachenBlaster lost health. I also saw that when their health became zero after being hit with the NachenBlaster’s torpedo or cabbage, the same thing happened as above except the NachenBlaster’s health didn’t change. The appropriate sounds were played.

I checked to see that each alien moved properly. Smallgons and Smoregons randomly changed their flight plan, the Smoregons randomly went at ramming speed, and the Snagglegons bounced off the sides. The Small/Smoregons shot turnips appropriately, and the Snagglegon occasionally shot torpedoes at the NachenBlaster.

When an alien died, it dropped its appropriate goodie.

Moreover, I went line-by-line through the entire class and compared it with the spec multiple times to see if there were any mistakes.

# Star

I checked to see that at the beginning of the game, a lot of stars were randomly displayed on the screen with different sizes. Then, as the game continued, these stars all moved to the left at the same rate, and new stars were created at the right side of the screen, meaning my second constructor functioned properly. Once they reached the left side of the screen, they disappeared because they were marked as dead.

Moreover, I went line-by-line through the entire class and compared it with the spec multiple times to see if there were any mistakes.

# Explosion

I checked to see that when an alien was killed by the NachenBlaster, directly or indirectly, an explosion was created. I actually originally had this code wrong because the doSomething() for NachenBlaster was called before it was displayed, meaning that it started at 1.5 size rather than 1. A revision of my code with if and if-else statements fixed this issue. I made sure that it was displayed for four ticks by using the “f” and “r” keys and comparing it with screenshots I took of the actual NachenBlaster game.

Moreover, I went line-by-line through the entire class and compared it with the spec multiple times to see if there were any mistakes.

# Projectile

My cabbage, turnip, and flatulenceTorpedo are derived from this class. I made sure that this class was pure virtual by trying to create an object of type Projectile in the init() function of StudentWorld, which didn’t work, which is what is supposed to happen. I tested the doSomething() by pressing “f” when each type of projectile came onto the screen and clicked space bar to see how it moved and how it interacted with other objects (i.e. the NachenBlaster).

Moreover, I went line-by-line through the entire class and compared it with the spec multiple times to see if there were any mistakes.

# Cabbage

I released a cabbage by pressing space bar, and clicked the “f” key. I went tick by tick and saw the it was spinning counter-clockwise by roughly 20 degrees. I saw that it only died once it collided with an alien, and it died when it moved off the right side of the screen.

Moreover, I went line-by-line through the entire class and compared it with the spec multiple times to see if there were any mistakes.

# Turnip

I moved the NachenBlaster in front of Small/Smoregons and saw that these aliens occasionally released turnips. I saw that these turnips only interacted with the NachenBlaster and decreased its health by 4%, meaning that their collide function was working. They were moving to the left, by roughly 5-10 pixels so I made the assumption that the move function works.

# FlatulenceTorpedo

I moved the NachenBlaster in front of Snagglegons and saw that they occasionally released torpedoes. I saw that these turnips only interacted with the NachenBlaster and decreased its health by a significant, meaning that Snagglegon’s collide function was working. They were moving to the left, by roughly 5-10 pixels so I made the assumption that the move function works.

Then, once I received torpedoes, I shot them and saw that they were facing the right direction and only interacted with aliens. The aliens would die when hit with enough torpedoes.

Moreover, I went line-by-line through the entire class and compared it with the spec multiple times to see if there were any mistakes.

# Goodie

I saw that each of the three types of goodies were created properly (i.e. correct image ID and correct start point). They all moved the same way (down and left 0.75 pixels). They all disappeared (died) after colliding with the NachenBlaster, not any other type of object. I tried to create an object of type Goodie in the init() function of StudentWorld and saw that it didn’t work, which is what should happen, because Goodie is an abstract class.

# ExtraLifeGoodie

I saw that this goodie gave the NachenBlaster +100 to the score, an extra life when it collided with it, played the sound\_goodie, and disappeared after the collision. It did not interact with any other type of object and was displayed with the correct depth.

# RepairGoodie

I saw that this goodie gave the NachenBlaster +100 to the score, increased health when it collided with it, played the sound\_goodie, and disappeared after the collision. It did not interact with any other type of object and was displayed with the correct depth.

# FlatulenceTorpedoGoodie

I saw that this goodie gave the NachenBlaster +100 to the score, 5 torpedoes when it collided with it, played the sound\_goodie, and disappeared after the collision. It did not interact with any other type of object and was displayed with the correct depth.