1. **Descriptions of all public member functions**

**virtual int StudentWorld::init();**

Init recreates each level by creating a new NachenBlaster object, regardless of whether or not the NachenBlaster lost a life and restarted the level or beat the previous level, resetting game stats (how many Aliens were destroyed, how many aliens are on the screen, how many aliens need to be destroyed in order to advance, and the max number of aliens that can be on screen at one time). This function was defined as virtual in the spec.

**virtual int StudentWorld::move();**

Move essentially calls doSomething for every actor in the game, checks to see if the NachenBlaster player is still alive, checks to see if the player has successfully destroyed the required number of aliens in order to advance, removes dead actors from the game, updates the status bar at the top of the screen, and adds new Aliens to the game when applicable. This function was defined as virtual in the spec, and is defined in the StudentWorld class because this function essentially relates all the actors’ independent movements with the game play.

**virtual void StudentWorld::cleanUp();**

Cleanup removes all actors from the game, including the NachenBlaster. This is declared in the StudentWorld class because StudentWorld oversees all actors and the game world, and it is virtual because that was what was provided in the spec.

**void StudentWorld::addActorToList(Actor\* a);**

addActorToList is a function that allows other classes to pass on newly created objects of that class type to be passed to StudentWorld and kept track of in the list of actors. This function is declared in StudentWorld because StudentWorld is responsible for keeping track of all actors, and this functions allows other classes to add to the list of actors, which is a private data member of StudentWorld, without compromising the integrity of the list. This function is not virtual because StudentWorld has no derived classes, and all actors are added to the list in the same class.

**bool StudentWorld::playerInLineOfFire(const Actor\* a);**

playerInLineOfFire allows aliens to determine whether the NachenBlaster is in its line of fire. This function was declared in the StudentWorld class because Aliens do not have access to the NachenBlaster’s coordinates; only through the StudentWorld class can the coordinates of both an Alien and the NachenBlaster be compared. This function is not virtual because StudentWorld has no derived classes.

**void StudentWorld::checkForCollisions(Alien\* a);**

This function checks if the Alien object passed in as the parameter has collided with either the NachenBlaster or a NachenBlaster-fired projectile (cabbages or flatulence torpedoes), and if so, calls a function called sufferdamage in the Alien class. This function was declared in StudentWorld because that is the only way to get access to the NachenBlaster/list of NachenBlaster-fired projectiles. This function is not virtual because StudentWorld has no derived classes.

**int StudentWorld::checkForCollisions(Projectile\* p);**

This function checks if an Alien-fired projectile (passed in as the parameter) has collided with the NachenBlaster. If so, it returns an int that indicates what type of projectile was fired (Turnip or Torpedo), otherwise it returns an int that indicates no collision occurred. It then calls sufferDamage in the Projectile class. This function is declared in StudentWorld because this is the only way to access the NachenBlaster’s coordinates, and it is not virtual because StudentWorld has no derived classes.

**NachenBlaster\* StudentWorld::checkForCollisions(Goodie \* g);**

This function checks if any goodie (which is passed in as the parameter) has collided with the NachenBlaster. If so, it returns a pointer to the NachenBlaster; otherwise it returns nullptr. This function is declared in StudentWorld because the Goodie class cannot access NachenBlaster directly, and it is not virtual because StudentWorld has no derived classes.

**void StudentWorld::killedAnAlien();**

This function performs the steps necessary when an Alien has been killed- it plays the appropriate sound, increments a private data member that stores the number of Aliens killed, an decrements a private data member that stores the number of Aliens currently on the screen. This function is in StudentWorld because StudentWorld is responsible for keeping track of Aliens and their status within the game, the number of Aliens killed, and the number of Aliens currently on screen. I made this function not virtual because StudentWorld has no derived classes.

**void StudentWorld::decAliensOnScreen();**

decAliensOnScreen decrements the number of Aliens on screen after it flies off the edge of the screen. It is declared in StudentWorld because StudentWorld is responsible for keeping track of the numbers of Aliens, and it is declared not virtual because StudentWorld has no derived classes.

**void virtual Actor::doSomething() = 0;**

doSomething is a pure virtual function declared in the Actor class because every single Actor does something different for each tick, and an object of an Actor type is not allowed for the game.

**StudentWorld\* Actor::getWorld();**

getWorld is an accessor function declared in the Actor class that allows Actor objects to call functions in the StudentWorld class by returning a pointer to the StudentWorld the Actor object is currently in. This function is declared non virtual in the Actor class because all Actors will need access to the StudentWorld they are currently on, and the implementation of it is the same for all Actors.

**bool Actor::isAlive() const;**

isAlive is an accessor that returns a boolean that indicates whether the current Actor is alive or not. It is declared non virtual in the Actor class because all Actor objects need to either be in a dead or alive state, but the implementation of it is the same for all Actors.

**void Actor::setAlive(bool alive);**

setAlive is a modifier function declared in the Actor class that changes the dead/alive state of the current Actor object. This function is declared non-virtual in the Actor class because all Actors have to be able to change their dead/alive state, but the implementation of this feature is the same for all Actor classes.

**bool virtual Actor::isAlien() const;**

isAlien is a function that returns a boolean depending on whether or not the actor is an Alien. This function is declared virtual in the Actor class because all Actors are either Aliens or not Aliens. Most objects of derived classes from the Actor class would return false, except for objects of the Alien class, in which case the function is overridden.

**bool virtual Actor::isProjectile() const;**

isProjectile is a function that returns a boolean depending on whether or not the actor is a Projectile. This function is declared virtual in the Actor class because all Actors are either Projectiles or not Projectiles. Most objects of derived classes from the Actor class would return false, except for objects of the Projectile class, in which case the function is overridden.

**bool virtual Actor::isStar() const;**

isStar is a function that returns a boolean depending on whether or not the actor is a Star. This function is declared virtual in the Actor class because all Actors are either Stars or not Stars. Most objects of derived classes from the Actor class would return false, except for objects of the Star class, in which case the function is overridden.

**void NachenBlaster::doSomething();**

doSomething in the NachenBlaster class adds a cabbage point for every tick, returns immediately if the NachenBlaster’s hit points reach 0, and makes a move depending on what key the character presses (including shooting a cabbage/torpedo). This function is declared non-virtual in the NachenBlaster class because NachenBlaster has no derived classes, and this implementation of doSomething is unique to the NachenBlaster.

**int NachenBlaster::hitPoints() const;**

hitPoints is an accessor function that returns an int that represents the number of hitPoints the NachenBlaster currently has. This function is declared non-virtual in the NachenBlaster class because NachenBlaster does not have any derived classes, and because NachenBlaster is only one of two Actors that have hitPoints.

**void NachenBlaster::sufferDamage(int damage);**

sufferDamage subtracts the amount of damage the NachenBlaster has taken, provided by the parameter, from its total number of hitPoints. This function is declared non-virtual in NachenBlaster because NachenBlaster does not have any derived classes, and because the implementation of this feature is unique to this class.

**void NachenBlaster::recoverHitPoints(int amt);**

recoverHitPoints increases the amount of hitPoints NachenBlaster has, depending on the number provided in the parameter, provided it doesn’t exceed the cap of 50 hitPoints. This function is declared non-virtual in NachenBlaster because only NachenBlaster objects can recover hitPoints, and this class has no derived classes.

**int NachenBlaster::cabbagePoints() const;**

cabbagePoints is an accessor function that returns the number of cabbagePoints NachenBlaster currently has. It is declared non-virtual in the NachenBlaster class because only NachenBlasters have cabbagePoints, and NachenBlaster does not have any derived classes.

**int NachenBlaster::torpedoInventory() const;**

torpedoInventory is an accessor function that returns the number of torpedoes an object of the NachenBlaster class currently has. This function is declared non-virtual in the NachenBlaster class because only NachenBlasters have a limited number of torpedoes, and NachenBlaster has no derived classes.

**void NachenBlaster::getTorpedoes(int amt);**

getTorpedoes is a modifier that increases the number of torpedoes the NachenBlaster object has by the amount specified in the parameter. This function is declared non-virtual in the NachenBlaster class because this feature is unique to objects of this class, and this class has no derived classes.

**void Star::doSomething();**

Star’s implementation of doSomething simple moves each star left by one pixel. This implementation is unique to the Star class, and the Star class has no derived classes, which is why the function was declared non-virtual in the Star class.

**bool Star::isStar() const;**

This function returns true for this implementation because all objects in the Star class are stars. This function is declared non-virtual in the Star class because the Star class does not have any derived classes, and the implementation of this function is unique to this class, although the function itself is not.

**void Alien::doSomething();**

doSomething checks to see if the Alien is alive, if the Alien has flown off the side of the screen, if the Alien has collided with either the NachenBlaster or a projectile fired by the NachenBlaster, and if the NachenBlaster is in its line of fire, the Alien has a chance of firing a projectile or moving along its flightplan. This function also checks to see if the Alien needs a new flightplan, and if so, calls a modifier function to do so. This function is declared non-virtual in the Alien class because this implementation of doSomething is unique to Aliens, but all Aliens behave the same way.

**bool Alien::isAlien() const;**

This function returns true for this implementation because all objects in the Alien class are Aliens. This function is declared non-virtual in the Alien class because the implementation of this function is unique to this class, although the function itself is not.

**int Alien::hitPoints() const;**

hitPoints is an accessor function that returns an int that represents the number of hitPoints the Alien currently has. This function is declared non-virtual in the Alien class because this class is only one of two Actors that have hitPoints, but all Aliens calculate their hitPoints the same way.

**void Alien::sufferDamage(int cause, Actor\* a);**

sufferDamage as declared in the Alien class first determines whether a collision with the player, a cabbage, or a NachenBlaster-fired torpedo has occurred, based on the parameter, and then either calls the appropriate functions to damage the NachenBlaster, sets the Alien or NachenBlaster-fired projectile to dead, or decreases the Alien’s hitPoints, depending on the type of collision. This function was declared non-virtual in the Alien class because this implementation is unique to Aliens, but all Aliens sufferDamage the same way.

**bool virtual Alien::isSnagglegon() const;**

This function returns a boolean depending on whether the Alien is a Snagglegon; it is declared virtual in the Alien class because while all Aliens have to be either a Snagglegon or not a Snagglegon, not all Aliens have the same implementation of this function.

**void virtual Alien::possiblyDropGoodie() = 0;**

This function calculates the probability of an Alien dropping a goodie when it is destroyed- and if so, drops the appropriate goodie. This function is pure virtual in the Alien class because every Alien implements this function differently, and an object of an Alien type is not allowed in this game.

**void Smallgon::possiblyDropGoodie();**

This function simply returns because Smallgons do not drop goodies. Smallgons do not have any derived classes, which is why this function was declared non-virtual.

**void Smoregon::possiblyDropGoodie();**

This function uses a probability unique to Smoregons to determine whether to drop a Repair Goodie or an Extra Life Goodie, which can only be dropped by Smoregons. Smallgons do not have any derived classes, which is why this function was declared non-virtual.

**bool Snagglegon::isSnagglegon() const;**

This functions returns true for all objects of the Snagglegon class. Snagglegons do not have any derived classes, which is why this function was declared non-virtual.

**void virtual Snagglegon::possiblyDropGoodie();**

This function uses a probability unique to Snagglegons to determine whether to drop an Extra Life Goodie or not, which can only be Snagglegons. Snagglegons do not have any derived classes, which is why this function was declared non-virtual.

**void Projectile::doSomething();**

doSomething as declared in the Projectile class checks to see whether the projectile is currently alive, is currently out of bounds, and if it was shot by an Alien, whether a collision with the NachenBlaster has occurred. If no collision has occurred, the projectile moves, rotates, and checks again, otherwise the function returns. This function is declared non-virtual in the Projectile class because this implementation is the same for all Projectiles.

**bool Projectile::isProjectile() const;**

This function returns true for all objects of the Projectile class, which is why it is non-virtual.

**void Projectile::sufferDamage(int cause, Actor\* a);**

sufferDamage calls NachenBlaster’s sufferDamage function, depending on which type of projectile collided with the NachenBlaster, which is provided by the parameter, then sets the projectile to dead. This implementation is the same for all Projectiles, which is why it is non-virtual.

**bool virtual Projectile::isTorpedo() const;**

This function returns false for all objects of the Projectile class, which is why it is non-virtual.

**bool virtual Projectile::shotByAlien() const = 0;**

This function returns a boolean that indicates whether it was shot by an Alien, or the NachenBlaster (returns false). This function is pure virtual because all Projectiles have different characteristics that cause their implementations of this function to be different, and because an object of the Projectile type is not allowed for this game.

**bool Cabbage::shotByAlien() const;**

This function returns false for all objects of the Cabbage class, which is why the function is non-virtual in the Cabbage class.

**bool Turnip::shotByAlien() const;**

This function returns true for all objects of the Turnip class, which is why the function is non-virtual in the Turnip class.

**bool Flatulence\_Torpedo::isTorpedo() const;**

This function returns true for all objects of the Flatulence Torpedo class, which is why the function is non-virtual in this class.

**bool Flatulence\_Torpedo::shotByAlien() const;**

This function returns a boolean that indicates whether or not it was shot by an Alien, which can be determined from a private data member of the Flatulence Torpedo class. This class has no derived classes, which is why the function is non-virtual in this class.

**void Explosion::doSomething();**

doSomething as implemented in the Explosion class determines whether 4 ticks have passed- if so, it sets the Explosion to dead and returns, otherwise, it increases the size of the Explosion object by a factor of 1.5 and increments the number of ticks passed. The Explosion class has no derived classes, which is why the function is non-virtual.

**void Goodie::doSomething();**

doSomething as implemented in the Goodie class determines whether the Goodie is currently alive, and if so, then checks the Goodie has flown off the side of the screen. If the Goodie is still alive and within view, it then checks to see if a collision with the NachenBlaster has occurred, and if so, performs the appropriate actions on the NachenBlaster and increases the score, otherwise, it moves, then checks again to see if a collision with the NachenBlaster has occurred.

1. **A list of all functionality that you failed to finish/known bugs**

**N/A**

1. **Assumptions made**

I assumed that the order of the actions performed in StudentWorld’s move function did not have to follow the list in the spec, since that was not clear. Instead of removing dead actors from the list at the end of the move function, I added that to the loop that iterated through the list in order to call doSomething, so that only one loop was necessary.

1. **How classes were tested**

**StudentWorld class:** This class was constantly tested while developing the game. Debugging statements were used to test whether the functions that determined whether to add a new Alien to the game (including, but not limited to, how many aliens were killed, the max number of Aliens on the screen, and the current number of Aliens on the screen), and if so, what kind, and were also used to test that the collisions were being logged, and if so, that the correct sufferDamage function was being called, and if the player was in the line of fire of the Aliens.

Debugging statements were also used to test the implementation of a new feature, the status bar. Debugging statement printed out the values that should be printed out in the health bar in order to make sure that the percentages of hitPoints and cabbagePoints the NachenBlaster had were being calculated correctly.

**Actor class:** This class was being constantly tested while developing the game. Since this class does not have that many public non-virtual member functions aside from identifiers, this class was tested more indirectly, through the testing of its derived classes and during gameplay.

**NachenBlaster class:** The doSomething function in this class was tested by playing the game and pressing each of the keys required for gameplay. The other functions, which are mostly accessors and modifiers for some private data members, were tested during gameplay, and by other class/functions calling these functions, and through the occasional use of debugging statements.

**Star class:** This function was not tested much after Part 1, since this was necessary for Part 1. During gameplay, it is also easy to tell whether or not the functions were working based on the movement of the stars in the background.

**Alien Class**: This class was tested during gameplay, and through the extensive use of debugging statements when testing sufferDamage and doSomething, in order to ensure that certain actions were being logged and the function that was supposed to be called in response was being called, even though an action may not always result from the function call. Other functions such as identifiers and accessors were not tested much, except through the function calls from other classes.

**Smallgon class:** Aside from the constructor (which would provide compile-time and run-time errors if implemented incorrectly), the possiblyDropGoodie was tested through debugging statements, since just because a goodie wasn’t drop doesn’t mean that the function was implemented incorrectly, since the probability of dropping a goodie isn’t 100%.

**Smoregon class:** Aside from the constructor (which would provide compile-time and run-time errors if implemented incorrectly), the possiblyDropGoodie was tested through debugging statements, since just because a goodie wasn’t drop doesn’t mean that the function was implemented incorrectly, since the probability of dropping a goodie isn’t 100%.

**Snagglegon class:** Aside from the constructor (which would provide compile-time and run-time errors if implemented incorrectly), the possiblyDropGoodie was tested through debugging statements, since just because a goodie wasn’t drop doesn’t mean that the function was implemented incorrectly, since the probability of dropping a goodie isn’t 100%. The identifier function isSnagglegon was tested through other classes’ function calls and during gameplay.

**Projectile class:** This class was tested by freezing the game to make sure that the projectiles were moving and rotating correctly, and in order to make sure that collisions were properly logged. Debugging statements were used in order to make sure that sufferDamage was functioning properly, and during gameplay, it was easy to tell whether the objects behaved appropriately.

**Cabbage class:** This class was mainly tested during gameplay, when it was easy to tell whether cabbages were being fired from only the NachenBlaster, and whether they disappeared when they flew off the screen or collided with an Alien. The shotByAlien function was mainly tested through other functions’ calls and during gameplay.

**Turnip class:** This class was mainly tested during gameplay, when it was easy to tell whether turnips were being fired from only Aliens, and whether they disappeared when they flew off the screen or collided with the NachenBlaster. The shotByAlien function was mainly tested through other functions’ calls and during gameplay.

**Flatulence\_Torpedo class:** This class was mainly tested during gameplay, when it was easy to tell whether Flatulence Torpedoes were being shot when the tab key was pressed, whether the torpedoes were travelling in the right direction were being fired from only Aliens, and whether they disappeared when they flew off the screen or collided with the another object. The shotByAlien and isTorpedo function was mainly tested through other functions’ calls and during gameplay.

**Explosion class:** This class was mainly tested by playing the game in slow motion, in order to test that the explosions were only alive for 4 ticks and were increasing by the proper factor. During gameplay, it is also easy to tell if explosions are created at the proper location and when Aliens are destroyed.

**Goodie class:** This class was tested during gameplay, as it is easy to see if the Goodies are moving the way they’re supposed, if the score increases, and (not) interacting with other actors like they’re supposed to, except when they collide with the NachenBlaster. Debugging statements were also used to make sure that collisions were being properly detected.

**Repair\_Goodie class:** The only member function in this class is the goodiePickedUp function, which I tested by printing debugging statements and also during gameplay, to see if the player’s score increased as well as their hitpoints. The constructor was not tested separately, but any errors should be apparent during gameplay.

**Extra\_Life\_Goodie class:** The only member function in this class is the goodiePickedUp function, which I tested by printing debugging statements and also during gameplay, to see if the player’s score increased as well as their lives. The constructor was not tested separately, but any errors should be apparent during gameplay.

**Flatulence\_Torpedo\_Goodie class:** The only member function in this class is the goodiePickedUp function, which I tested by printing debugging statements and also during gameplay, to see if the player’s score increased as well as their hitpoints/torpedo count/lives. The constructor was not tested separately, but any errors should be apparent during gameplay.