Actor:

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

Actor is derived from the class GraphObject. When it is first created, it passes the required parameter required to make the GraphObject first. Then, it sets its pointer to StudentWorld and indicate that the actor is alive.

The actor has a pure virtual function doSomething() because the objects in the game will be derived from this class. Each actor will be doing something different during each tick. Actor is an abstract class because it is just used to create common functions that all objects in the game will carry out such as checking whether it is still alive (using alive() ) or get to the world it lives in (getWorld()). I tested this class by checking if initializing an object of this class causes a compiler error.

//virtual destructor for the actor class, so all classes derived from actor can be destructed properly when a level is completed or when the game ends

virtual ~Actor();

//for actors that cannot suffer damage, the function just returns and do nothing

virtual void sufferDamage(double amt, int cause);

Throughout the game, Nachenblaster and aliens would suffer damage. Because StudentWorld only contains a list of pointers to all Actors in the game, and StudentWorld is responsible for indicating whether damageable objects( those derived from class Ship mentioned below) has been damaged by another ship or a projectile, sufferDamage is a virtual function and simply returns if actors such as cabbages, stars, and goodie cannot sufferDamage.

virtual void doSomething() = 0; //make Actor an ABC

//check if the actor is an alien (smallgon, smoregon, snagglegon)

virtual bool isAliens() const;

This function would should only true for the class Aliens and classes derived from it, so I made this a virtual function that returns false for all non-Alien actors.

//check if actor is still alive

bool alive() const;

//if the actor died, set itself dead

void setDead();

//returns a pointer to the world the actor lives in

StudentWorld\* getWorld();

//checks if the actor went off the screen

bool isInBound(double x, double y) const;

Ship:

Ship::Ship(StudentWorld\* world, double startX, double startY, int imageID, double hitpoints, int dir, double size, int depth): Actor(world, startX, startY, imageID, dir, size, depth), m\_hit(hitpoints){}

Ship is a derived class of Actor. Ships are damageable, but since it does not have its own implementation of the doSomething function, it is also set as an abstract base class. Aliens and Nachenblaster are ships because they are damageable and need to keep track of their hitpoints throughout the game. I tested this class by checking if initializing an object of this class causes a compiler error.

//returns the number of hitpoints

double getHitPoints() const;

//increase or decrease number of hitpoints by the "hitpoints" passed into the parameter

void setHitPoints (double hitpoints);

Nachenblaster:

Nachenblaster::Nachenblaster(StudentWorld\* world): Ship(world, 0, 128, IID\_NACHENBLASTER, 50, 0, 1.0, 0), m\_cabEnergy(30), m\_torpedoes(0) {}

When Nachenblaster is constructed when StudentWorld’s init is called, it sets its cabbage energy points to 30 and number of torpedoes it has to 0.

virtual ~Nachenblaster();

virtual void doSomething();

Nachenblaster has a virtual doSomething function because it will carry out a move (move in a direction, fire cabbage/torpedoes) if the player pressed the corresponding key for the move during each tick. It will also check its dead/alive status and track its cabbage and torpedoes count in this function.

virtual void sufferDamage(double amt, int cause);

Unlike other actors who can’t incur damage, Nachenblaster would suffer damage if its been hit by an alien-fired projectile or collided with an alien. This method is therefore virtual and would be call by an alien-fired projectile or an alien when they collided with Nachenblaster. When these objects call this function, they will pass in the amount of damage they cause and the cause of damage. Alien-fired projectile will indicate to nachenblaster the cause is HIT\_BY\_PROJECTILE, and aliens will indicate to nachenblaster the cause is HIT\_BY\_SHIP. If the

//Accessors:

//return the number of torpedoes player has

int numTorpedoes() const;

// Get the player's health percentage.

int healthPct() const;

// Get the player's cabbage energy percentage.

int cabbagePct() const;

//Mutators:

//increases player's hitpoints by "num" in parameter

void increaseHitPoints(double num);

// Incease the number of torpedoes the player(Nachenblaster) has by amt.

void increaseTorpedoes(int amt);

I tested this class by printing out the new hitpoints of Nachenblaster if its sufferDamage function is called. I check if the number of torpedoes and cabbage count is valid by directly checking the game stats bar on the screen since the setting the game stats call the Accessors numTorpedoes, healthPct(), and cabbagePct().

Aliens:

Aliens(StudentWorld\* world, int imageID, double hitpoints, double damageAmt, double speed, unsigned int scoreValue, double startX, double startY);

Aliens are also derived from the class Ship since aliens can be damaged by nachenblaster projectile or nachenblaster. When Aliens are added to the game, it will also be required to indicate the amount of damage they would cause to nachenblaster and score value if the alien were to collide with nachenblaster.

virtual void doSomething();

For Aliens doSomething function, it also first check if it has died or moved off the screen. It then check whether it has collided with nachenblaster with the function “collideWNach” (described below), which will cause the alien to damage nachenblaster and suffer damage from collision with nachenblaster accordingly. The checking for whether alien has collided with a nachenblaster-fired projectile will be done in projectile’s doSomething() function. Aliens will then determine whether it needs a new flightplan or make an attack (shoot turnip/flatulence torpedo or ram towards Nachenblaster). If the alien did not make an attack, it will continue to move in its current travel direction and check if it has collided with Nachenblaster again.

virtual bool isAliens() const;

Unlike the class Actor and the classes derived from Actor, this function returns true for Aliens.

virtual void sufferDamage(double amt, int cause);

Aliens can also suffer damage like Nachenblaster. It cause either suffer damage from Nachenblaster fired projectile or Nachenblaster, which will be indicated in “cause” in the parameter by whichever object that calls the function. Alien’s actual suffering will be done by calling the function “alienDied()” (described below).

//determine if aliens will fire projectiles or perform some other form of attack to Nachenblaster

virtual bool attack();

This is a virtual function because different aliens will cause different attacks to Nachenblaster. In alien’s version of attack, there is a 1/(20/CurrentLevel+5) chance the alien will fire a turnip at Nachenblaster. For aliens that fire turnips (Smallgon and Smoregon), they can just call this version of attack. The function returns true if an attack has been carried out, so the Alien’s doSomething function will be notified to not do anything else during the current tick since an attack has been carried out.

//damages Nachenblaster and suffer damage from Nachenblaster if collided with Nachenblaster

void collideWNach();

This function will be called in Alien’s doSomething() to damage Nachenblaster if collided with Nachenblaster and cause Alien to suffer damage incurred by collision with Nachenblaster.

// If this alien collided with the player, damage the player and return

// true; otherwise, return false.

bool damageCollidingPlayer();

This function checks if alien has collided with nachenblaster by passing a pointer to itself to StudentWorld. StudentWorld will return a pointer to nachenblaster only if a collision has occured. Alien can then use this pointer to cause damage to Nachenblaster.

//impose damage and carry out actions caused by death of alien

void alienDied();

If an alien dies, the sound SOUND\_DEATH would be played and the alien will be set to dead. This function will also indicate to StudentWorld that an Alien has been destroyed, thus causing StudentWorld to update the remaining aliens needed to be destoryed at current level. As a result of the death, the player’s score increase and an explosion is added. If an alien drops a goodie when it dies, it might also drop a goodie through the method “possiblyDropGoodie()” (described below).

//maybe drop a goodie

virtual void possiblyDropGoodie();

This is a virtual function because not all aliens drop goodies. For aliens that do drop goodie, they drop it at different possibilities and drop different goodies. The Alien’s class version simply returns and does nothing for classes derived from aliens that do not drop goodie.

//sets a new flightplan if alien has reached top or bottom of screen or flight plan length has reached zero

virtual void setNewFlight();

This is a virtual function called in the Alien’s doSomething function. It is virtual because Smallgon and Smoregon have to keep track of their flight plan length, while Snagglegon doesn’t. Snagglegon has its own version of setNewFlight that does not keep track or reset its flight plan length.

//move in current travel direction, decrease flight plan length accordingly

virtual void moveCurrent();

This is virtual because while Smallgon and Smoregon need to keep track and update its flight plan length as they move in their current travel direction, Snagglegon does not. Snagglegon therefore has its own version of moveCurrent that does not consider flight plan length in moving in its current direction.

//Accessors:

//return travel speed

double getSpeed() const;

//return amount of damage alien would cause if collided with Nachenblaster

double damageAmt() const;

//return amount of score that shoud be increased if alien collided with Nachenblaster

int scoreValue() const;

//return current travel direction of the alien

char getTravDir() const;

//Mutators:

//changes travel speed to "speed" in parameter

void setSpeed(double speed);

//changes flight length to "fl" in parameter

void setFlightLength(double fl);

//changed travel direction to "dir" in parameter

void setTravDir(char dir);

//returns a random travel direction (U, D, OR, L)

char randomDir();

I tested this class by printing out the message “Alien has died!” every time the alienDied() function is called. In the sufferDamage function, I print out Aliens hitpoints every time after setHitPoints is called to see if hitpoints of the aliens are being subtracted. This also helps me test if a collision among projectiles and ships are detected throughout the game.

Smallgon:

Because a smallgon basically does everything an alien does, it only has a constructor and virtual destructor.

Smallgon(StudentWorld\* world);

virtual ~Smallgon();

Throughout the game, I also observe how smallgon moves and make sure it is moving only left, up and left or down and left. I also set the probability so that it will always fire a turnip when nachenblaster is in line of fire to make sure it conducts its attack correctly.

Smoregon:

Smoregon(StudentWorld\* world);

virtual ~Smoregon();

//returns true if a Smoregon attack has been made

virtual bool attack();

This virtual function calls Alien’s version of attack to see if this Smoregon will fire a turnip. If a turnip has been fired, this function simply returns true so in the doSomething() function, nothing else will be carried out by this Smoregon during the current tick. If a turnip was not fired, Aliens’ version of attack will return false. In 1/(20/current\_level+5) chance, the smoregon will set its current direction to left and increases it speeds and ram towards a Nachenblaster.

//possibly drop a repair or flatulence torpedo goodie

virtual void possiblyDropGoodie();

This is a virtual function because unlike smallgon, smoregon might drop a goodie when it dies. There is 1/3 chance a goodie will be dropped. If a goodie is dropped, it will either be a repair or a flatulence torpedo goodie.

I check if smoregon is behaving correctly by setting the probability so that smoregon will always drop a goodie when it is destroyed. This allows me to see if it is dropping the correct goodie. I also set the probability so that smoregon will always make an attack when it is created to see if it conducts the appropriate attack. Throughout the game, I also observe how smoregon moves and make sure it is moving only left, up and left or down and left.

Snagglegon:

Snagglegon(StudentWorld\* world);

virtual ~Snagglegon();

virtual bool attack();

virtual void setNewFlight();

virtual void possiblyDropGoodie();

virtual void moveCurrent();

Because Snagglegon does not keep track of a flightlength when it moves. It has its own version of setNewFlight and moveCurrent function that simply does not consider the flightlength in determining a new travel direction or move in its current direction.

A Snagglegon only drops an extra life goodie if it decides to drop a goodie (in 1/6 chance). When a snagglegon makes an attack, it will launch an alienlaunched flatulence torpedo instead of a turnip.

I check if snagglegon is behaving correctly by setting the probability so that snagglegon will always drop a goodie when it is destroyed. This allows me to see if it is dropping the correct goodie. I also set the probability so that snagglegon will always make an attack when it is created to see if it conducts the appropriate attack. Throughout the game, I also observe how snagglegon moves and make sure it is moving only up and left or down and left.

Goodie:

Goodie(StudentWorld\* world, double startX, double startY, int imageID);

virtual ~Goodie();

virtual void doSomething();

Goodie’s doSomething function will check if the goodie is alive and inbound. Afterwards, it will check if it collided with Nachenblaster with the function “collideWNach()” described below and move by 0.75 pixels for every tick.

//different goodies do different things when collided with Nachenblaster

virtual Nachenblaster\* collideWNach();

This is a virtual function because all goodies will play the sound SOUND\_GOODIE when collided with Nachenblaster and increase the Nachenblaster’s score by 100 points, but they will also do differentiated things depending on the type of goodie. This function returns a pointer to nachenblaster to its doSomething() function if a collision with nachenblaster occured. Nachenblaster’s doSomething function can thus use it to determine whether to set the goodie’s state to dead during the current tick.

I checked whether goodie is behaving correctly by checking whether Nachenblaster’s points have been increased when goodie is caught from the status bar. When goodie appears on the screen, it should also be moving downward and leftward continuously. To make the checking process easier, I increased the probability of an alien dropping a goodie in Smoregon’s and Snagglegon’s “possiblyDropGoodie()” function.

ExtraLifeGoodie:

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

virtual ~ExtraLifeGoodie();

//give Nachenblaster one extra life if collided

virtual Nachenblaster\* collideWNach();

In this virtual function, ExtraLifeGoodie calls Goodie’s version of collideWNach to play the sound SOUND\_GOODIE and increase Nachenblaster’s score if Nachenblaster has caught the goodie. It then increases the life and return a pointer to Nachenblaster back to the doSomething() function so the doSomething function will set this goodie as dead.

I check whether the extra life goodie has increased the life from the status bar and also check whether 100 points gets added when a goodie is caught.

Repair Goodie:

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

virtual ~RepairGoodie();

virtual Nachenblaster\* collideWNach();

In addition to playing sound and increasing player’s score, repair goodie will also increase Nachenblaster’s hitpoints in its collideWNach function.

I checked if Nachenblaster’s hitpoints increased in the game status bar after it has caught a repair goodie. To make the checking process easier, I first set the probability in Smoregon’s possiblyDropGoodie() so that it will always drop a repair goodie and will always drop a goodie when collided with Nachenblaster.

FlatulenceTorpedoGoodie:

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

virtual ~FlatulenceTorpedoGoodie();

virtual Nachenblaster\* collideWNach();

Like the other goodies, FlatulenceTorpedoGoodie will first call Goodie’s version of collideWNach. It will then increase the torpedo count of Nachenblaster.

Projectile:

Projectiles::Projectiles(StudentWorld\* world, double startX, double startY, int dir, double deltaX, int imageID, double size, int depth, double damageAmt, bool rotates): Actor(world, startX, startY, imageID, dir, size, depth), m\_damageAmt(damageAmt), m\_deltX(deltaX), m\_rotates(rotates){}

When a projectile is created by Nachenblaster or an alien, they must pass in the parameters required for actor and the amount of damage the projectile will make, the amount it travels, and whether the projectile rotates. This is an abstract base class because different projectile, depending on whether it is launched by Nachenblaster or an alien, will do different damages.

virtual ~Projectiles();

virtual void doSomething();

In projectiles doSomething function, it will check whether the projectile is alive or offscreen. Regardless of whether the projectile is launched by Nachenblaster or an alien, it will also move in the x direction it’s supposed to move in and check if it has collided with its target. Collide is a pure virtual function so classes derived from projectiles can determine in different ways whether a collision is significant. The function also check if the projectile should rotate and does so accordingly.

//abstract base class

virtual bool collide() = 0;

//returns the amount of damage projectile will do

double damageAmt() const;

//returns how much the projectile moves in the x direction for each tick

double deltX() const;

//returns whether the projectile rotates

bool rotates() const;

I check whether damage by projectile has been done by printing message “Alien got hit by projectile!” in Alien’s sufferDamage function and “Nachenblaster got hit by projectile!” in Nachenblaster’s sufferDamage function as I play the game. Since the “collide()” function of each class derived from projectiles will be calling the sufferDamage function if collided with its designated target.

Cabbage:

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

virtual ~Cabbage();

virtual bool collide();

The collide function of cabbage is virtual so that cabbage can be set to only damage an Alien by calling the sufferDamage function for alien. In collide(), cabbage will call the “getOneCollidingAlien(const Actor\* a)” from StudentWorld to check if it has collided with an alien and cause damage as a result. If a collision occur, collide() returns true to the projectile’s doSomething function so the cabbage would no nothing else during this tick.

I checked whether cabbage is rotating correctly by freezing the screen and checking how it moves a tick at a time. I also printed out Alien’s hit points in Alien’s sufferDamage() function to see if alien’s hitpoints get deducted by 2 everytime a cabbage collided with an alien.

Turnip:

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

virtual ~Turnip();

virtual bool collide();

The collide function for turnip is virtual so it will only impose damage on Nachenblaster. In this function, the turnip will call StudentWorld’s getCollidingPlayer function to check if it has collided with Nachenblaster. If collision occured, it will call the sufferDamage method of Nachenblaster and incur the damage. If a collision occur, collide() returns true to the projectile’s doSomething function so the cabbage would no nothing else during this tick.

I checked whether turnip is rotating correctly by freezing the screen and checking how it moves a tick at a time. I also printed out Nachenblaster’s hit points in Nachenblaster’s sufferDamage() function to see if nachenblaster’s hitpoints get deducted by 2 everytime a turnip collided with nachenblaster.

Flatulence Torpedo:

FlatulenceTorpedo(StudentWorld\* world, double startX, double startY, int imageDir, double deltaX);

virtual ~FlatulenceTorpedo();

Flatulence Torpedo is an abstract base class derived from projectile since it does not have implementations for the collide() function in projectiles. It is made an abstract class because different flatulence torpedo, depending on whether it’s fired from the player or an alien(Snagglegon) will only cause damage if it were its “enemy”. The amount of damage a flatulence torpedo will make is always set to 8 hitpoints when it is first created.

Player Launched Torpedo:

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

virtual ~PlayerLaunchedTorpedo();

virtual bool collide(); //check for collision with aliens

Player Launched Torpedo check for collision with alien in its collide function and impose damage (decreases hitpoints) by the damage amount specified when it’s first created.

I check if player launched torpedo is actually launched from Nachenblaster and moves toward the right for 8 pixels every tick. To check whether appropriate damage is made, I used the same method I used for other projectiles (printing out resulting hitpoints in sufferDamage) and see if the hitpoints subtracted for nachenblaster is 8.

Alien Launched Torpedo:

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

virtual ~AlienLaunchedTorpedo();

virtual bool collide(); //check for collision with nachenblaster

Alien launched torpedo check for collision with nachenblaster in its collide function and subtracts hitpoints by the damage amount specifed when it’s first created.

I check if alien launched torpedo is actually launched from Nachenblaster and moves toward the left for 8 pixels every tick. To check whether appropriate damage is made, I used the same method I used for other projectiles (printing out resulting hitpoints in sufferDamage) and see if the hitpoints subtracted for the alien is 8.

Explosion:

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

virtual ~Explosion();

virtual void doSomething();

Explosion is derived from the Actor class because its collision with other actors are insignificant. An explosion is created everytime a collision between Nachenblaster and an alien occured. After it’s creation, explosion will grow in size by a factor for 1.5 and disappear 4 ticks after its creation. Every explosion have a private member variable that serves as a counter that will make sure explosion disappears when it’s supposed to.

In explosion doSomething function, I used a counter (first set to zero) to ensure the explosion only starts increasing size after the first tick it has been created. The count will increment by one everytime doSomething() is called, so by the time counter reached 4, the explosion will set itself to dead.

I check whether explosion is increasing its size properly by freezing the screen when a collision occur and move one tick at a time. I compared the initial size and the increased size of the explosion to one shown in the executable file and make sure the explosion disappears after the same amount of tick in the executable file.

Star:

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

virtual ~Star();

virtual void doSomething();

Star is derived from the class actor. Stars do not interact with any other object in the game in any significant way. In its doSomething method, it simply move towards the left every tick. If the star moved offscreen, it will set itself to dead.

I checked if stars are created correctly by checking if after StudentWorld’s init() is called, there are 30 items in the list. I also made sure stars do not pile up at the left of my screen as the game continues.

StudentWorld:

StudentWorld(std::string assetDir);

~StudentWorld();

virtual int init();

This function is virtual because it is derived from gameworld. It will calculate some private member variable Student World has such as T (total number needed to be destoryed per level), M (maximum number of aliens that should be on the screen at a time), and S2, S3, and S. S2, S3, and S will be used in the private member function newShip to determine the kind of Alien that should be added to the game. It also sets D, CUR, and R. D is number of aliens destoryed, which is zero at beginning of game. CUR is current aliens on screen, which should also be 0 at start of game, R is the remaining aliens that need to be killed, which is T-D. It will also create 30 stars randomly distributed on the screen and continues the game.

virtual int move();

The move function calls the doSomething function for each actor in the game. It will then determine if move result in death of nachenblaster and remove all dead actors from the list. If an alien has died, CUR will be updated as it gets removed from the list. New stars and aliens might be introduced into the game through function call newStars() and newAliens() and game statistic will be updated by updateGameStats().

virtual void cleanUp();

The function deletes Studentworld’s pointer to nachenblaster and all other actors in the game, leaving an empty list of actor’s list.

//allow student world to get objects newly added to the game

void getObjects(Actor\* obj);

Add “obj” into the list of actors

//returns a pointer to nachenblaster if "a" collided with nachenblaster

Nachenblaster\* getCollidingPlayer(const Actor\* a) const;

This function calls collide, which will use the euclidian distance approach to check whether “a” has collided with nachenblaster, for which student keep as a pointer to as private member variable. The function returns a pointer to nachenblaster collide returns true and returns a nullptr to indicate a collision did not occur to whoever calls this function.

//returns a pointer to an alien that has collided with "a"

Actor\* getOneCollidingAlien(const Actor\* a) const;

The function iterates through the list that contains all actors in the game. If it encounters an actor in the list that is an alien (by calling the isAliens() method), it will then check if the alien has collided with “a”. If both conditions are true, the function returns a pointer to the actor in the list, which should be an alien.

//determine y distance is within range for aliens to make an attack

bool playerInLineOfFire(const Actor\* a) const;

The function checks if the magnitude of the y distance between a and nachenblaster is smaller than or equal to 4. If so, nachenblaster is in line of fire for a and the function will return true.

//check if actor1 and actor2 has collided using euclidian distance

bool collide(const Actor \* actor1, const Actor \* actor2) const;

The function returns true if collision occured as defined by euclidian distance, returns false otherwise.

//update the number of aliens destoryed (D)

void destroyedAliens();

Everytime an alien died, Alien’s alienDied() function will call this function so StudentWorld can update D (number of alien destroyed).

//update game statistic

void updateGameStats();

The function uses ostringstream to create a string of text that represents the game statistic. It then passes this string to the setGameStatText function from gameworld. This function will be call every move in StudentWorld’s move function so the game statistic will get updated throughout the game.

I checked if student world is updating its game statistic correctly through the game status bar shown on screen. To check if its member variables that keep track of whether a level has been completed or whether the game is over such as D, M, CUR, and R, I printed out their values when I quit the game to see if the values corresponds to what is shown on the screen. To check if the algorithm for adding new alien is implemented correctly, I print out the CUR and R value everytime the newAliens function is called to make sure the game knows when is adding and not adding aliens correctly.