DESIGN of the actors:

1. Actor
   1. NachenBlaster
   2. Stars
   3. Aliens
      1. Smallgon
      2. Smoregon
      3. Snagglegon
   4. Goodies
      1. ExtraLife
      2. Repair
      3. Flatulence Torpedo
   5. Projectiles
      1. Cabbage
      2. Turnip
      3. Flatulence Torpedo

**Actor:**

doSomething: pure virtual “Because based on what type of actor it is, this function needs to be overridden as per specs of that particular type of actor. Also the function will never be called on an actor itself, cause we always create a type of actor, not an actor”

getWorld() “returns a pointer to the student world (SW) that the actors are located in.’

To modify Dead/alive Status of all the Actors, so that SW can remove them when required

void setState(string state)

bool alive()

For keeping track of hit points of Nachenblaster and Alien Ships. An important consideration here is that hit points only make sense as integers but setting hit points for aliens require us to multiple .5\*getLevel() and that gives a double value and there is a possible loss of value as it changes to int but makes sense and would not make a difference.

void change\_hit\_pts(int pts)

void set\_hit\_pts(int hit)

int hit\_pts()

virtual void projResult() : Damage from projectiles to alien ships upon collision. Not made pure virtual as not goodies or nachenblaster would call but need to be a member function for aliens to use

//Damage from Projectiles to Nachenblaster

virtual void damageNB() : not made pure virtual as goodies will not use them but all the projectiles shall use

Bool Alien(): virtual ‘this returns true if the actor is an alien. We need it before checking for a collision with other objects so. It by default returns false but overridden in Alien class so return true, hence virtual’

bool collision(Actor\* first, Actor\* second): Measure Euclidian Distance between various actors. Not virtual as the formula is same for all classes.

To test this class, i would mostly add debugging lines in between. And tried creating a pointer to new Actor and since that gave me an error, it was proven that it was a pure virtual abstract class. The hit points could be seen changing on the screen as required, and so the effects of setting the state to dead or alive. While iterating through the vector separately and using Alien() was effective for checking, while upon collision, using assert statements on the game score, lives, level et cetera were able to check the class properly.

**NachenBlaster:**

None of the public functions (except doSomething() of course) are virtual in NachenBlaster(NB) as they are there just to keep tracks of or modify the number of cabbages and flatulence torpedos left to shoot with NB, hence never used by other classes

void incFlatTorps(int n)

int cabScore()

int flatScore()

doSomething(): behaves according to the input given by the user. Fires cabbages and flatulence torpedoes and responsible for directional movement of the nachenblaster. Also increases cabbage pts by 1 every tick (upto 30)

Apart from debug comments, Nachenblaster was tested with intentional collision and checking assert messages on the cabbage points as when were required. Most of it was visual since we have directional commands.

**Star:** doSomething(): drifts the stars towards the left and declares them dead if off screen.

This was tested with occasionally commenting out setState(‘Dead”) and seeing that the stars clutter up at the left end of the screen and hence confirming that they are not getting eliminated by Student world::move()

**Alien:** doSomething(): responsible for checking the alive status of all alien ships and check for their collision with NB. it uses the following public functions (none of them virtual) of Alien to implement various portions:

newPlan(): this is needed by functions to determine a new flight plan, when it reaches the top or bottom of the screen or the flight plan reduces to 0

toFire: depending on the type of alien ship, this fires various projectiles at NB

set\_Traveldir: for modifying the travel direction, where 0: DUE LEFT; 134: UP AND LEFT 224: DOWN AND LEFT

travelDir: depending on the direction of the alien ship, and its speed, this function facilitates the movements

For modifying and tracking flight plan (not virtual):

int flightPlan()

void set\_flightPlan(int fp)

For modifying and tracking speed(not virtual):

double speed()

void setSpeed(double n)

virtual int AlienType() = 0; to identify different types of alien ships; pure virtual so that it returns SMALLGON=1, SMOREGON=2, SNAGGLEGON=3, and we never make the mistake of never calling it on a regular alien class

The three types of alien classes all rely on doSomething() provided by alien class and just return their respective ints for the AlienType class

All the alien classes can be tested by debugging and commenting out different parts of the code. Modifying the various travel directions can give us different movements visually. On commenting out TravelDir() from dosomething, we can see that the alien ships do not move as required. Commenting out toFire, we see no signs of the goodies anywhere in the game when the alien ship dies. We could see visually that upon dying, the correct goodies are getting dropped. By putting in extreme values for the speed, we were able to see (or not see more than a peek) the effect on the screen. We tried creating an Alien object too which threw error hence fulfilling our purpose.

**Projectile:**

The doSomething() (virtual) is responsible for seeing if the projectiles clash with any other actors, and if they do they depending on their spec, they damage the alien ships/NB accordingly

virtual int projType() = 0; to identify different types of projectiles; pure virtual so that it returns the correct int and we never make the mistake of never calling it on a regular projectile class

The three types of projectile classes rely on doSomething provided here and and just return different ints for the projType()

All the alien classes can be tested by debugging and commenting out different parts of the code. We tried creating a new Projectile object too which threw error hence fulfilling our purpose. Varying their speeds and the inputs for moveTo() we can see the expected differences visually. Occasionally to check for collision with nachenblaster/alien ships, i put in explosions and see the game stats to see if it is behaving accordingly.

**Goodies:**

The doSomething() (virtual, for clarity) is responsible for declaring the goodie dead if it is off screen, and check for a collision with a Nachenblaster, and as per the specs of different goodies, give extra life/scores.

All the alien classes can be tested by debugging and commenting out different parts of the code. We tried creating a new Projectile object too which threw error hence fulfilling our purpose. Varying their speeds and the inputs for moveTo() we can see the expected differences visually. Occasionally to check for collision with nachenblaster, i put in explosions and see the game stats to see if it is behaving accordingly.

**StudentWorld**

The Student World class is responsible for creating the game world, and calling upon the animation on all the actors on screen, and removing them as needed. It also contains a vector of pointers to all the actors. One important note here is that instead of using vector iterator, i used indexing since addition to vector invalidates the iterator, hence causes errors.

To check if the indexing were working properly, the moment the game began, we spammed it with new cabbages, increasing the capacity of the vector at a high speed. Since that did not give us incompatible vector errors, we are fine. Also, every new projectile or goodie can be seen on the screen whenever required so that serves well. Increase in score was tested visually with the game stats text but couting the cabbage points at every line shower its progress as well.

Also leaving the game unattended, we could see how the student world would keep on adding objects and animating and deleting as needed, and eventually increment the levels. Additionally, by modifying the probabilities to extreme values, we could visually see how certain actors would either swarm the screen or not show up at all, confirming our probabilities are working. Even changing randIt(,)<=n to randInt()==n can show the difference.