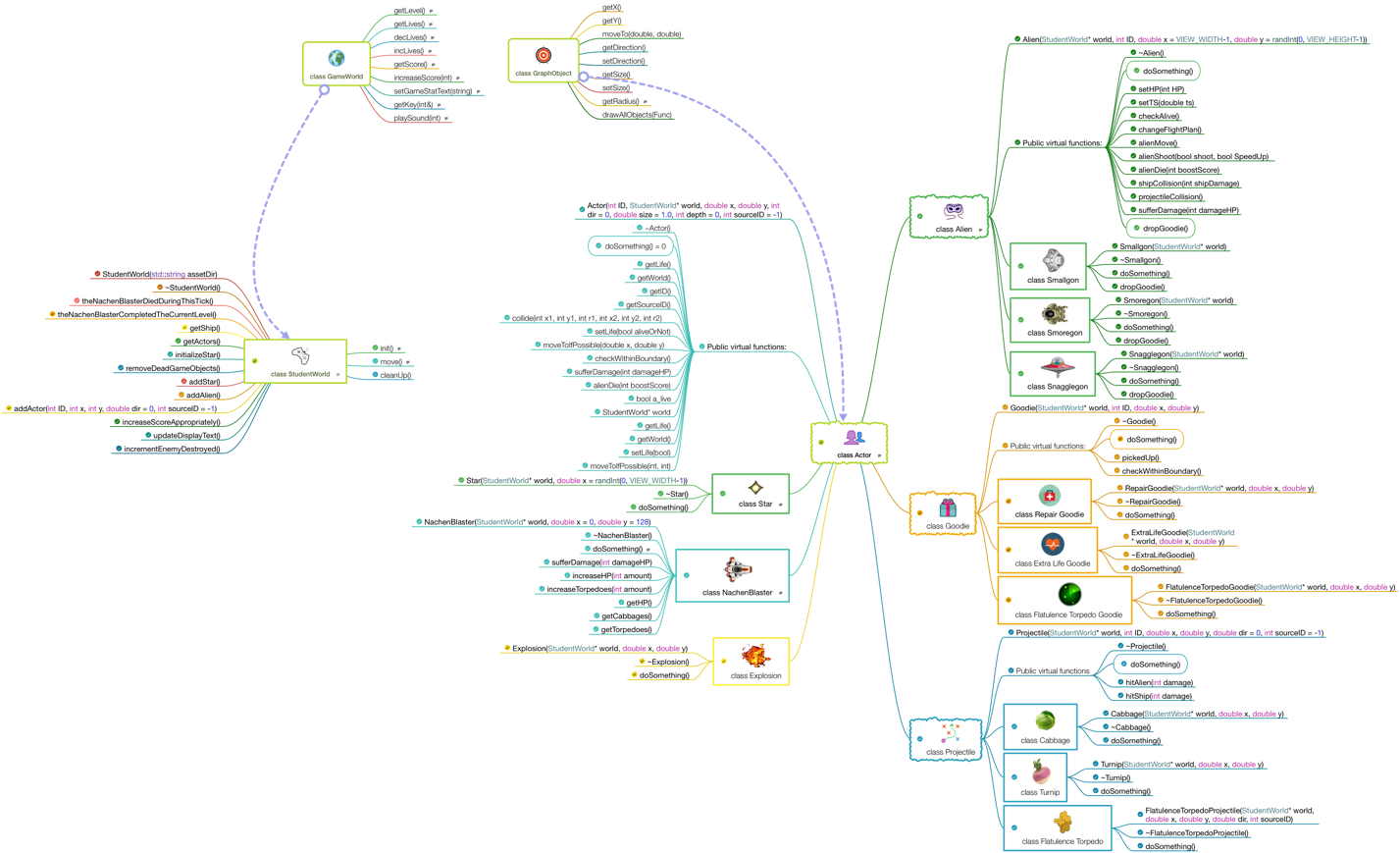
CS 32 Project 3 Report

1. Above is a visualization of the inheritance & polymorphism structure of my project 3 (it has high resolution so feel free to zoom in), which contains the base classes and derived classes plus the constructors, destructors, and all the public functions. All the public functions are marked “virtual”, including the destructor. Abstract classes are in “cloud” shapes with irregular borders, while concrete classes are in “rectangle” shapes. Pure virtual functions are circles, while others are only underlined.

Below is the higher-level description for each function:

// StudentWorld

Public functions:

    // constructor & destructor

StudentWorld(std::string assetDir);

virtual ~StudentWorld();

    // fundamental functions

virtual int init() {

Dynamically allocate a new NachenBlaster ship;

(Load a vector of actors and push data into this STL structure)

Initialize a field of 30 moving stars (call initializeStar());

Initialize several private member variables (for tracking purposes) to 0;

}

virtual int move() {

// Give each actor a chance to do something

For each actor:

Call its doSomething() function;

(update the STL structure correspondingly)

Call ship’s doSomething() function;

 // Check status

If the NachenBlaster ship dies (call theNachenBlasterDiedDuringThisTick()),

Then decrement life by 1;

Return DIED message;

If the NachenBlaster ship completes the current level (call theNachenBlasterCompletedTheCurrentLevel()),

Then increase scrore appropriately;

Return FINISHED message;

 // Clean up and update

Remove newly dead actors after each tick (call removeDeadGameObject());

Introduce new stars by a chance of 1/15 (call addStar());

Introduce new aliens if necessary (call addAlien());

Update display text by string literal (call updateDisplayText());

}

virtual void cleanUp() {

For each actor:

Delete the actor;

Erase the actor from vector;

If the ship pointer is not NULL,

Then delete the pointer to the NachenBlaster ship;

Set the ship pointer to NULL;

}

// accessors

// …status check

virtual bool theNachenBlasterDiedDuringThisTick() {

Call ship’s getLife() function;

If getLife() returns true,

Then return false;

Otherwise,

Return true;

}

virtual bool theNachenBlasterCompletedTheCurrentLevel() {

If # of enemy destroyed is >= the intended # for this level,

Then return true;

Otherwise,

Return false;

}

    // ...whenever called

virtual NachenBlaster\* getShip() { return ship pointer; }

virtual vector<Actor\*> getActors() { return the vector sequence of Actor pointers; }

    // mutators

    // ...only in init()

virtual void initializeStar() {

For 30 times,

push\_back() a newly dynamically allocated star into the actors vector;

}

    // ...at each tick

virtual void removeDeadGameObjects() {

For each actor,

If the actor is “dead”,

If the actor is an alien,

Decrement the # of current aliens;

Delete this actor;

Erase the corresponding actor from vector;

}

virtual void addStar() {

Introduce a new star with random y value by a chance of 1/15;

}

virtual void addAlien() {

According to the spec, do the calculations and dynamically allocate a new alien properly if necessary;

}

virtual void addActor(int ID, int x, int y, double dir = 0, int sourceID = -1) {

push\_back() a newly dynamically allocated actor into the actors vectos, according to the ID that is passed in;

}

    virtual void increaseScoreAppropriately() { Nothing; }

virtual void updateDisplayText() {

Using string literal technique, update the text properly by calling its getLives(), getScore(), getLevel() and ship’s getHP(), getCabbage(), getTorpedoes() functions;

}

    // ...whenever called

    virtual void incrementEnemyDestroyed() { increment the # of destroyed enemy (private member variable to keep track of level completion); }

The student world functions do not involve further inheritance and polymorphism, but all functions are set to virtual for furture development. Since there is no class deriving from StudentWorld and thus StudentWorld has to be a concrete class, there is no pure virtual function in StudentWorld. The functions except for init(), move(), and cleanUp() are implemented to either support the three fundamental functions or to support the doSomething() function of each actor.

// Actor

public functions:

// constructor & destructor

Actor(int ID, StudentWorld\* world, double x, double y, int dir = 0, double size = 1.0, int depth = 0, int sourceID = -1);

virtual ~Actor();

    // pure virtual function

virtual void doSomething() = 0;

I chose to define a pure virtual version of the doSomething() function in my base Actor class because all actors in NachenBlaster should be able to do something, and each type of actor does something in a different way.

    // accessor

virtual bool getLife() const { return status of the actor, true for alive and false for dead; }

This is defined as virtual, but not pure virtual, since every actor has a status of either alive or dead, and they are all alive or dead in a same way.

virtual StudentWorld\* getWorld() const { return the world in which this actor is located; }

This is defined as virtual, but not pure virtual, since every actor has pointer to the world in which it is located, and they all locate in the world in a same way.

virtual int getID() const { return the ID of the actor; }

virtual int getSourceID() const { return the sourceID from which the object is fired; }

These two functions are defined as virtual, since each actor access ID and sourceID in the same way.

virtual bool collide(int x1, int y1, int r1, int x2, int y2, int r2) const {

If the euclidian distance between two actors is less than 0.75 times the sum of the radii of the two actors,

Then return true;

Return false;

}

This is defined as virtual, rather than pure virtual, since given the x, y coordinate and the radius of two actors, there is universal way of figuring out if they are collided.

    // mutator

virtual void setLife(bool aliveOrNot) { set the living status of the actor to alive or dead; }

This is defined as virtual, but not pure virtual, since every actor has a status of either alive or dead, and they are all alive or dead in a same way.

virtual bool moveToIfPossible(double x, double y) { if the passed in x, y coordinate is within the boundary of the game field, then move the actor to this designated coordinate by the moveTo() function; }

This is defined as virtual, but not pure virtual, since every actor can move and thus has a passed in destination coordinate, the coordinate is either within the boundary or without. Each actor are the same with regard to move to a designated coordinate.

virtual bool checkWithinBoundary() {

Call the x coordinate of this actor by getX();

If the x coordinate is within the boundary of the game field;

Then set the living status of the actor to false;

Return false;

Otherwise,

Return true;

}

This is defined as virtual, but not pure virtual, since every actor has a coordinate on which they are currently located, and they either within the boundary or without. Each actor are the same with regard to the boundary. This function only check the x coordinate since most actors do not move in the y direction or that their movement has already been held within the boundary by moveToIfPossible. Later, the function checkWithinBoundary() is overrided in class Goodie due to a need to check both x and y.

virtual void sufferDamage(int damageHP); { Nothing; }

virtual bool alienDie(int boostScore);   {  Nothing; }

These two functions are useful only for certain actor (NachenBlaster or Alien), but they are not set to pure virtual so that each actor does not have to implement their doing-nothing version of the function repeatedly. For NachenBlaster and Alien, it is easy to implement corresponding functions to override this doing-nothing version of the functions.

// Star

Public functions:

    // constructor & destructor

Star(StudentWorld\* world, double x = randInt(0, VIEW\_WIDTH-1));

virtual ~Star();

    // mutator

virtual void doSomething() {

If moving to the left by 1 is not possible (call moveToIfPossible());

Then set the living status of this star to dead;

}

This version of the doSomething() function is declared as virtual but not pure virtual since Star is the final derived class and thus has to be concrete class to be instantiated.

// Nachenblaster

Public functions:

    // constructor & destructor

NachenBlaster(StudentWorld\* world, double x = 0, double y = 128);

virtual ~NachenBlaster();

    // mutator

virtual void doSomething() {

If hitpoint is <= 0,

then set the living status of this ship to false;

return;

Pass in the key that player presses in the current tick;

Switch:

If the player press space,

If # of cabbages is >= 5,

then dynamically allocate a new cabbage in the StudentWorld;

play sound SHOOT;

decrement the # of cabbages by 5;

If the player press tab,

If # of torpedoes is >= 1,

then dynamically allocate a new torpedo in the StudentWorld;

play sound TORPEDO;

decrement the # of torpedoes by 5;

If the player press LEFT, RIGHT, UP, or DOWN,

If moving to the corresponding direction by 6 is possible,

then move the ship by 6 to the corresponding direction;

If the # of cabbages is < 30,

increment the # of cabbages;

}

This version of the doSomething() function is declared as virtual rather than pure virtual since NachenBlaster is the final derived class and thus has to be concrete class to be instantiated.

virtual void sufferDamage(int damageHP);   { decrement HP by damageHP amount }

virtual void increaseHP(int amount);   { increase HP by amount }

virtual void increaseTorpedoes(int amount);   { increase # of torpedoes (Torpedo Goodie) by amount }

These 3 functions are declared as virtual rather than pure virtual since these functions are exclusive to NachenBlaster, and that NachenBlaster is the final derived class and thus has to be concrete class to be instantiated. They are implemented for aliens, projectiles and goodies to mutate the values in case of collison.

    // accessor

virtual int getHP() const { return the HP of the NachenBlaster ship; }

virtual int getCabbages() const { return the # of cabbages of the NachenBlaster ship; }

virtual int getTorpedoes() const { return the # of torpedoes of the NachenBlaster ship; }

These three functions are declared as virtual rather than pure virtual since these functions are exclusive to NachenBlaster, and that NachenBlaster is the final derived class and thus has to be concrete class to be instantiated. They are implemented for the StudentWorld function updateDisplayText() to access.

// Alien

Public functions:

    // constructor & destructor

Alien(StudentWorld\* world, int ID, double x = VIEW\_WIDTH-1, double y = randInt(0, VIEW\_HEIGHT-1));

virtual ~Alien();

    // mutator

virtual void doSomething() = 0;   // pure virtual, can't be instantiated

I chose to define a pure virtual version of the doSomething() function in my derived / base Alien class because all aliens in NachenBlaster should be able to do something, and each type of alien does something in a different way.

virtual void setHP(int HP);

virtual void setTS(double ts);

These two functions allow each alien to set an initial amount of hitpoints and traveling speed for themselves. I define these as virtual but not pure virtual because every alien has hitpoint and traveling speed, and they all process these two values the same way.

virtual bool checkAlive();   { check if HP > 0, if false set "dead"; }

virtual int changeFlightPlan();   // change flight plan if necessary, return flight plan

{

getY();

If y is >= the y upper boundary,

Then set flight plan to 3;

Randomly assign the fligh plan length to a value between 1 and 32;

If y is <= the y lower boundary,

Then set flight plan to 2;

Randomly assign the fligh plan length to a value between 1 and 32;

If the flight plan length is 0,

Randomly assign flight plan to 1, 2, or 3;

Randomly assign the fligh plan length to a value between 1 and 32;

Return the current flight plan (1 for due left, 2 for left-up, and 3 for left-down);

}

virtual void alienMove();   { move according to the flight plan and travel speed of each specific alien; }

virtual bool alienShoot(bool shoot, bool speedUp);   // shoot a projectile due to chance

{

Get the x, y coordinates of the NachenBlaster ship;

Get the x, y coordinates of the current alien;

If the alien x < nachenblaster x and the nachenblaster y is within ±4 of the alien y,

Then if shoot is true,

Return true;

If speedUp is true,

The sets travel direction to due left;

Set flight plan length to the width of world;

Set travel speed to 5;

}

virtual bool alienDie(int boostScore);   // increment player's point, set "dead", play sound effect, introduce explosion

{

If not alive (call checkAlive()),

Then increase score by boostScore;

Play sound DEATH;

Dynamically allocate a new Explosion in the same location where the alien dies;

Increment # of enemy destroyed in StudentWorld for StudentWorld to keep track of (call incrementEnemyDestoryed());

Call dropGoodie();

Return true;

}

virtual bool shipCollision(int shipDamage);   // check if alien ship collides with the nachenblaster

{

If the NachenBlaster ship collides with the alien,

Then decrement the HP of ship by shipDamage (call ship’s sufferDamage());

Set alien’s HP to 0;

Return true;

Return false;

}

virtual bool projectileCollision();   // check if this alien ship collides with any projectile

{

For each actor in StudentWorld,

If the ID is CABBAGE and the cabbage collides with this alien,

Then set the cabbage’s living status to false;

Alien’s HP decrement by 2;

Return true;

If the ID is TORPEDO and the surceID is NACHENBLASTER,

Then set the torpedo’s living status to false;

Alien’s HP decrement by 8;

Return true;

Return false;

}

virtual void sufferDamage(int damageHP);   { decrease HP of alien }

virtual void dropGoodie() = 0;   // pure virtual, drop goodie if necessary

I chose to define a pure virtual version of the dropGoodie() function in my base Alien class because all aliens in NachenBlaster drop or does not drop different goodie due to different chance.

    // Smallgon

    Public functions:

        // constructor & destructor

        Smallgon(StudentWorld\* world);

        virtual ~Smallgon();

        //mutator

        virtual void doSomething() {

Check alive and within boundary (call checkAlive() and checkWithinBoundary()):

Return immediately if either is false;

Check collision and death, and performs necessary tasks (call shipCollision(), projectileCollsion(), and alienDie());

Change flight plan if necessary (call changeFlightPlan());

If shoot due to chance (call alienShoot()),

Play sound SHOOT;

Dynamically allocate a new turnip in the same location where the alien is currently in StudentWorld;

Move according to the flight plan and traveling speed (call alienMove());

Check collision and death again (call shipCollision(), projectileCollsion(), and alienDie());

}

        virtual void dropGoodie() { Nothing; }

    // Smoregon

    Public functions:

        // constructor & destructor

        Smoregon(StudentWorld\* world);

        virtual ~Smoregon();

        // mutator

        virtual void doSomething(){

Check alive and within boundary (call checkAlive() and checkWithinBoundary()):

Return immediately if either is false;

Check collision and death, and performs necessary tasks (call shipCollision(), projectileCollsion(), and alienDie());

Change flight plan if necessary (call changeFlightPlan());

If shoot due to chance (call alienShoot()),

Play sound SHOOT;

Dynamically allocate a new turnip in the same location where the alien is currently in StudentWorld;

Move according to the flight plan and traveling speed (call alienMove());

Check collision and death again (call shipCollision(), projectileCollsion(), and alienDie());

}

        virtual void dropGoodie() {

For a 1/3 chance,

Then for a half chance,

Dynamically allocate a new repair goodie in the same location where the alien dies in StudentWorld;

For the other hald chance,

Dynamically allocate a new torpedo goodie in the same location where the smoregon dies in StudentWorld;

}

    // Snagglegon

    Public functions:

        // constructor & destructor

        Snagglegon(StudentWorld\* world);

        virtual ~Snagglegon();

        // mutator

        virtual void doSomething() {

Check alive and within boundary (call checkAlive() and checkWithinBoundary()):

Return immediately if either is false;

Check collision and death, and performs necessary tasks (call shipCollision(), projectileCollsion(), and alienDie());

Change flight plan if necessary (call changeFlightPlan());

If shoot due to chance (call alienShoot()),

Play sound TORPEDO;

Dynamically allocate a new torpedo in the same location where the alien is currently in StudentWorld;

Move according to the flight plan and traveling speed (call alienMove());

Check collision and death again (call shipCollision(), projectileCollsion(), and alienDie());

}

        virtual void dropGoodie(){

For a 1/6 chance,

Dynamically allocate a new life goodie in the same location where the snagglegon dies in StudentWorld;

}

// Goodie

Public functions:

    // constructor & destructor

    Goodie(StudentWorld\* world, int ID, double x, double y);

    virtual ~Goodie();

    // mutator

    virtual void doSomething() = 0;   // pure virtual, can't be instantiated

    virtual bool pickedUp();   { check if the goodie is picked up (if collide() with the NachenBlaster ship, increment player's point, set "dead", play sound effect); }

    virtual bool checkWithinBoundary();   { overrides Actor's checkWithinBoundary by checking both x and y; }

    // RepairGoodie

    Public functions:

        // constructor & destructor

        RepairGoodie(StudentWorld\* world, double x, double y);

        virtual ~RepairGoodie();

        //mutator

        virtual void doSomething() {

Check alive and within boundary (call checkAlive() and checkWithinBoundary()),

Return immediately if either is false;

Check if the goodie is picked up (call pickedUp()),

Increment the NachenBlaster ship’s HP by 10 (call ship’s increaseHP());

Return;

If possible,

Move left-down by 0.75;

Otherwise (moving off frame),

Set living status of this goodie to false;

Check if the goodie is picked up again (call pickedUp()),

Increment the NachenBlaster ship’s HP by 10 (call ship’s increaseHP());

Return;

}

    // ExtraLifeGoodie

    Public functions:

        // constructor & destructor

        ExtraLifeGoodie(StudentWorld\* world, double x, double y);

        virtual ~ExtraLifeGoodie();

        // mutator

        virtual void doSomething() {

Check alive and within boundary (call checkAlive() and checkWithinBoundary()),

Return immediately if either is false;

Check if the goodie is picked up (call pickedUp()),

Increment the lives of the NachenBlaster ship by 1 (call incLive());

Return;

If possible,

Move left-down by 0.75;

Otherwise (moving off frame),

Set living status of this goodie to false;

Check if the goodie is picked up (call pickedUp()),

Increment the lives of the NachenBlaster ship by 1 (call incLive());

Return;

}

    // FlatulenceTorpedoGoodie

    Public functions:

        // constructor & destructor

        FlatulenceTorpedoGoodie(StudentWorld\* world, double x, double y);

        virtual ~FlatulenceTorpedoGoodie();

        // mutator

        virtual void doSomething() {

Check alive and within boundary (call checkAlive() and checkWithinBoundary()),

Return immediately if either is false;

Check if the goodie is picked up (call pickedUp()),

Increment the # of torpedoes of the NachenBlaster ship by 1 (call incrementTorpedoes());

Return;

If possible,

Move left-down by 0.75;

Otherwise (moving off frame),

Set living status of this goodie to false;

Check if the goodie is picked up (call pickedUp()),

Increment the # of torpedoes of the NachenBlaster ship by 1 (call incrementTorpedoes());

Return;

}

// Projectile

Public functions:

    // constructor & destructor

    Projectile(StudentWorld\* world, int ID, double x, double y, double dir = 0, int sourceID = -1);

    virtual ~Projectile();

    // mutator

    virtual void doSomething() = 0;   // pure virtual, can't be instantiated

    virtual bool hitAlien(int damage);   // collides with alien

{

For each actor in StudentWorld,

If the ID is SMALLGON, SMOREGON, or SNAGGLEGON and that the alien collides with this projectile,

The alien’s HP decrements by damage (call alien’s sufferDamage());

If the ID of the alien is SMALLGON or SMOREGON,

Then call alienDie with 250 boostScores;

If the ID of the alien is SNAGGLEGON,

Then call alienDie with 1000 boostScores;

Set this projectile’s living status to false (call setLife());

Return true;

Otherwise,

Return false;

}

    virtual bool hitShip(int damage);   // collides with nachenblaster

{

Get the x, y coordinates of the NachenBlaster ship;

If the NachenBlaster ship collides with this projectile (call collide()),

The ship’s HP decrements by damage (call nachenblaster’s sufferDamage());

Set this projectile’s living status to false (call setLife());

Return true;

Otherwise,

Return false;

}

    // Cabbage

    Public functions:

        // constructor & destructor

        Cabbage(StudentWorld\* world, double x, double y);

        virtual ~Cabbage();

        //mutator

        virtual void doSomething() {

Check alive and within boundary (call checkAlive() and checkWithinBoundary()),

Return immediately if either is false;

Check if this cabbage hits an alien object (call hitAlien()),

Return if hits;

If not possible to move right by 8 (call moveToIfPossible()),

Then set the living status of the cabbage to false;

Return;

Rotate by 20 degrees (call setDirection());

Check if this cabbage hits an alien object again(call hitAlien()),

Return if hits;

}

    // Turnip

    Public functions:

        // constructor & destructor

        Turnip(StudentWorld\* world, double x, double y);

        virtual ~Turnip();

        // mutator

        virtual void doSomething() {

Check alive and within boundary (call checkAlive() and checkWithinBoundary()),

Return immediately if either is false;

Check if this cabbage hits a ship object (call hitShip()),

Return if hits;

If not possible to move left by 6 (call moveToIfPossible()),

Then set the living status of the turnip to false;

Return;

Rotate by 20 degrees (call setDirection());

Check if this cabbage hits a ship object again (call hitShip()),

Return if hits;

}

    // FlatulenceTorpedoProjectile

    Public functions:

        // constructor & destructor

        FlatulenceTorpedoProjectile(StudentWorld\* world, double x, double y, double dir, int sourceID);

        virtual ~FlatulenceTorpedoProjectile();

        // mutator

        virtual void doSomething() {

Check alive and within boundary (call checkAlive() and checkWithinBoundary()),

Return immediately if either is false;

If the sourceID of this torpedo is NACHENBLASTER,

Then check if this torpedo hits an alien object (call hitAlien()),

Return if hits;

If not possible to move right by 8 (call moveToIfPossible()),

Then set the living status of the torpedo to false;

Return;

Check if this torpedo hits an alien object again (call hitAlien()),

Return if hits;

If the sourceID of this torpedo is SNAGGLEGON,

Then heck if this torpedo hits a ship object (call hitShip()),

Return if hits;

If not possible to move left by 8 (call moveToIfPossible()),

Then set the living status of the torpedo to false;

Return;

Check if this torpedo hits a ship object again (call hitShip()),

Return if hits;

}

Class Goodie and class Projectile, along with their corresponding subclasses are structurally similar to class Alien and the subclasses of Alien in terms of pure virtual and virtual functions, so I didn’t redundantly write the reasons for declaring each function as pure virtual or virtual. The functions declared under the base classes (projectile, goodie, and alien) are shared by all the corresponding subclasses so that each subclass does not have to write their own ones, which also provides consistency.

// Explosion

Public functions:

    // constructor & destructor

Explosion(StudentWorld\* world, double x, double y);

virtual ~Explosion();

    // mutator

virtual void doSomething() {

Set size to be 1.5 the previous size (call GameWorld’s setSize());

Decrement this explosion’s life span;

If life span <= 0,

Then set the living status of this explosion to false;

}

1. There is no known bug nor functionality that I know failed to implement.
2. Design decisions and assumptions:
   1. I decided to use vector as the STL data structure for storing the game objects, although I realized later that list is more suitable data structure for this project.
   2. It may be easier to implement the projectile Cabbage so that when it hits an alien (Smallgon, Smoregon, and Snagglegon), the cabbage merely cause damage but does not check if the alien die or drop goodie correspondingly and then let each alien check these things in the beginning of the alien’s turn. However, I decided to let the aliens die (if possible) and / or drop a goodie (due to chance) right in the tick when they are attacked, rather than one tick after.
   3. For utilization purpose, I decided to declare virtual public functions sufferDamage() and alienDie() in the Actor, both of which are implemented to perform nothing (but NOT pure virtual). Even though some of the actors cannot sufferDamage() and some cannot die in an alien way, this is convenient for collision functions to iterate through the actors vector and perform these two functions if necessary.
   4. Any Actor’s member variables (e.g. a\_hitpoints) must be declared as private, but can be mutated or accessed by corresponding public functions. Besides basic parameters that GraphObject requires, all Actors have a pointer to the StudentWorld in which it is located, and a sourceID defaulted to -1. The sourceID is useless for other actors so it remains -1, but it is convenient for the implementation of projectile Torpedo, which needs to be differentiated by the source it is fired from, so its value can be either IID\_NACHENBLASTER (0) or IID\_SNAGGLEGON (3).
   5. Every function is declared as virtual no matter in the base or derived class, so that they are extendable in future development.
3. Description of testing:
   1. StudentWorld
      1. init(): The loading field part was singled out and tested by comparing the loaded field (30 stars + NachenBlaster ship) directly with the provided in the sample, which is exactly the same. Std::cout statements were also used to ensure that all objects are pushed onto the STL structure.
      2. move(): This part was mainly tested using std::cout statements that writes the original position of the object, their original HP, the new position of the object and their new HP, or if they’re dead. This is to ensure that the STL structure is updated correctly after each actor has performed an action. It is also indirectly tested by comparing with the sample program to see if actions are smooth and continuous.
      3. cleanUp(): This part was tested using the provided g++ sanity check, which my program passed.
      4. Other functions are trivial supporting functions. As long as the major three functions above perform correctly, these does not have to be tested individually.
   2. Actors
      1. Not much public functions to test here, as most methods are virtual or pure virtual to be overrided later, and that most of them are as trivial as returning a value.
         1. collide() is tested by passing in different sets of number and std::cout to see if the judgement is as intended.
         2. moveToIfPossible() and checkWithinBoundary() are mostly tested by freezing the simulation and advancing frame by frame to verify that the actors are moving as expected.
      2. Star
         1. doSomething() is tested by asking StudentWorld’s init() only initialize 30 stars, and monitor if they are moving smoothly, and if stars moving off the boundary behave correctly.
      3. NachenBlaster
         1. doSomething() is tested by asking StudentWorld’s init() only initialize the NachenBlaster ship, and monitor if it is moving as intended (UP, DOWN, RIGHT, and LEFT) and if the correct sound is played plus cabbage and torpedoes decrement and / or increment correctly if space or tab is pressed.
         2. Other methods are as trivial as returning a value or changing a parameter.
      4. Alien
         1. Most functions directly off the Alien class serve to support the doSomething() function of specific aliens, and these functions are tested twice:
            1. Using std::cout to monitor the change of parameters. Print the original position of the alien, their original HP, the new position of the alien and their new HP, the flight plan, or if they’re dead, etc.
            2. Monitoring that the doSomething() function of each alien behaves correctly.
         2. Smallgon
            1. doSomething() is tested by asking the move() function of StudentWorld to only allocate Smallgon aliens, and advancing frame by frame (using breakpoint) to verify that the smallgons are moving as expected. In addition, verify that the NachenBlaster ship decrement the right amount of HP each time it is shot by a smallgon, and that the score increases the right amount each time the NachenBlaster shot down a smallgon.
         3. Smoregon
            1. doSomething() is tested by asking the move() function of StudentWorld to only allocate Smoregon aliens, and advancing frame by frame (using breakpoint) to verify that the smoregons are moving as expected. In addition, verify that the NachenBlaster ship decrement the right amount of HP each time it is shot by a smoregon, and that the score increases the right amount each time the NachenBlaster shot down a smoregon.
            2. dropGoodie() is tested by force each Smoregon to drop a goodie each time it dies, and check if the goodie is of the correct type.
         4. Snagglegon
            1. doSomething() is tested by asking the move() function of StudentWorld to only allocate Snagglegon aliens, and advancing frame by frame (using breakpoint) to verify that the snagglegons are moving as expected. In addition, verify that the NachenBlaster ship decrement the right amount of HP each time it is shot by a snagglegon, and that the score increases the right amount each time the NachenBlaster shot down a snagglegon.
            2. dropGoodie() is tested by force each Snagglegon to drop a goodie each time it dies, and check if the goodie is of the correct type.
      5. Projectile
         1. Not much public functions to test here, as most methods are meant to support the doSomething() function of the subclasses.
         2. Turnip
            1. doSomething() is tested by asking the move() function of StudentWorld to only allocate several Turnip projectile, and advancing frame by frame (using breakpoint) to verify that the turnip is moving as expected. In addition, verify that the NachenBlaster ship is decremented the right amount of HP each time it collides with s turnip.
         3. Cabbage
            1. doSomething() is tested by asking the move() function of StudentWorld to only allocate a Cabbage projectile and a alien in front of it, and advancing frame by frame (using breakpoint) to verify that the turnip is moving as expected. In addition, verify that the alien is decremented the right amount of HP when it collides with the cabbage.
         4. Torpedo
            1. doSomething() is tested by asking the move() function of StudentWorld to only allocate a Torpedo projectile and a alien in front of it, and advancing frame by frame (using breakpoint) to verify that the torpedo is moving as expected. In addition, verify that the alien is decremented the right amount of HP each time it collides with the torpedo.
      6. Goodie
         1. Repair Goodie
            1. doSomething() is tested by forcing the Smoregon to drop repair goodie every time a smoregon is dead. There will then be significantly many repair goodies for the nachenblaster to pick up. Monitor the HP value and score value on the text above to see if the repair goodie behaves correctly, and monitor if the goodie disappear properly.
         2. Life Goodie
            1. doSomething() is tested by forcing the Snagglegon to drop life goodie every time a snagglegon is dead. There will then be significantly many life goodies for the nachenblaster to pick up. Monitor the # of lives and score value on the text above to see if the life goodie behaves correctly, and monitor if the goodie disappear properly.
         3. Torpedo Goodie
            1. doSomething() is tested by forcing the Snagglegon to drop the torpedo goodie every time a snagglegon is dead. There will then be significantly many torpedo goodies for the nachenblaster to pick up. Monitor the # of torpedoes and score value on the text above to see if the torpedo goodie behaves correctly, and monitor if the goodie disappear properly.
      7. Explosion
         1. doSomething() is tested by asking the move() function of StudentWorld to only allocate a Explosion in the middle of the game field, and advancing frame by frame (using breakpoint) to verify that the explosion is moving as expected (enlarge and last only 4 rounds).