Raymond Lin

CS32 - Data Structures and Algorithms

Smallberg

1 March, 2018

Project 3 - Report

1. Class Actor: public GraphObject  
   Actor(int imageID, int startX, int startY, int startDirection, double size, int depth, StudentWorld \*p)

:GraphObject(imageID, startX, startY, startDirection, size, depth)

Construct Actor with these parameters: ID, x-coord, y-coord, starting direction, size, depth and the world in which the object exists. Because it is derived from the GraphObject class, pass in the correct parameters in the initializer list to construct the GraphObject.

virtual void doSomething() = 0;  
 All Actors can do something, but each doSomething function behaves differently.

virtual bool isEnemy() = 0;

All Actors react to the NachenBlaster. If this function returns true, it can damage the NachenBlaster. Serves to distinguish between enemies and friendly objects.

virtual bool isProjectile() = 0;

Used to determine if an Actor is a Projectile.

virtual bool isDamageableObject() = 0;

Used to determine if an Actor is a damageable object, such as a NachenBlaster or an Alien.

virtual bool isNachenBlaster() = 0;

Used to determine if an Actor is the NachenBlaster.

virtual bool isGoodie() = 0;

Used to determine if an Actor is a goodie. This and the above ‘is’ functions are used in the collided function. Because it takes two Actor pointers, these functions must be used to determine how each Actor reacts to the other. (More detailed description in the collided function)

bool getIsDead() const;

Used to determine if an Actor is dead.

void setIsDead(bool in);

Used to set an Actor’s status to dead.

StudentWorld \*getWorld();

Returns the world that the certain Actor is in.

int getID();

Returns the IID of the Actor.  
  
  
Class Star: public Actor

Star(int startX, int startY, double size, StudentWorld\* p)

:Actor(IID\_STAR, startX, startY, 0, size, 3, p)

Construct star with these parameters: x-coord, y-coord, size and world in which it exists. Construct Actor in the initializer list because it is derived from Actor. Some parameters are kept constant such as the id of the star, the direction and the depth, as they are always the same value for stars.

virtual void doSomething();

Moves the star towards the left, and if it goes past the screen, it’s dead.

virtual bool isEnemy()

virtual bool isProjectile()

virtual bool isDamageableObject()

virtual bool isNachenBlaster()

virtual bool isGoodie()

All these functions return false. They must be included, because they are pure virtual functions in the Actor class.

Class Explosion: public Actor

Explosion(int startX, int startY, StudentWorld\* p)

:Actor(IID\_EXPLOSION, startX, startY, 0, 1, 0, p)

Constructs an Explosion using parameters: x-coord, y-coord and world in which it exists. Construct an Actor from those parameters, because it is derived from Actor class. Some are kept constant, such as ID, direction, size and depth, as they are always the same value for explosions.

virtual void doSomething();

Increases the Explosion’s size by a factor of 1.5. Once it gets past a certain size, set the status to dead.

virtual bool isEnemy()

virtual bool isProjectile()

virtual bool isDamageableObject()

virtual bool isNachenBlaster()

virtual bool isGoodie()

All these functions return false. They must be included, because they are pure virtual functions in the Actor class.

Class Goodie: public Actor

Goodies(int id, int startX, int startY, StudentWorld\* p)

:Actor(id, startX, startY, 0, 0.5, 1, p)

Constructs a Goodie using parameters: x-coord, y-coord and world in which it exists. Construct an Actor from those parameters, because it is derived from Actor class. Some are kept constant, such as direction, size and depth, as they are always the same value for Goodies.

virtual void doSomething();

Checks if the goodie is dead or gone past the left of the screen. If not, move it down and left.

virtual bool isEnemy()

virtual bool isProjectile()

virtual bool isDamageableObject()

virtual bool isNachenBlaster()

All these functions return false. They must be included, because they are pure virtual functions in the Actor class.

virtual bool isGoodie()

Returns true because it is a goodie.

Class ExtraLifeGoodie: public Goodie

ExtraLifeGoodie(int startX, int startY, StudentWorld\* p)

:Goodies(IID\_LIFE\_GOODIE, startX, startY, p)

Constructs an ExtraLifeGoodie using parameters: x-coord, y-coord and world in which it exists. Construct a Goodie from those parameters, because it is derived from Goodie class. ID is kept constant because it’s always the same value for ExtraLifeGoodies.

Class RepairGoodie: public Goodie

RepairGoodie(int startX, int startY, StudentWorld\* p)

:Goodies(IID\_REPAIR\_GOODIE, startX, startY, p)

Constructs a RepairGoodie using parameters: x-coord, y-coord and world in which it exists. Construct a Goodie from those parameters, because it is derived from Goodie class. ID is kept constant because it’s always the same value for RepairGoodies.

Class TorpedoGoodie: public Goodie

TorpedoGoodie(int startX, int startY, StudentWorld\* p)

:Goodies(IID\_REPAIR\_GOODIE, startX, startY, p)

Constructs a TorpedoGoodie using parameters: x-coord, y-coord and world in which it exists. Construct a Goodie from those parameters, because it is derived from Goodie class. ID is kept constant because it’s always the same value for TorpedoGoodies.

Class Projectile: public Actor

Projectile(int id, int startX, int startY, StudentWorld\* p)

:Actor(id, startX, startY, 0, 0.5, 1, p)

Constructs a Projectile using parameters: x-coord, y-coord and world in which it exists. Construct an Actor from those parameters, because it is derived from Actor class. Parameters such as direction, size and depth are kept constant because they’re always the same value for Projectiles.

virtual void doSomething() = 0

Pure virtual function because every projectile does something different.

virtual bool isEnemy() = 0

Some projectiles are friendly, such as cabbages and NachenBlaster’s own torpedos. Therefore, the function is pure virtual.

Virtual bool move() = 0

Each projectile move differently. For example, cabbages and friendly torpedos move to the right. Enemy projectiles move to the left.

virtual bool isProjectile()

Returns true because it is a Projectile.

virtual bool isDamageableObject()

virtual bool isNachenBlaster()

virtual bool isGoodie()

All these functions return false. They must be included, because they are pure virtual functions in the Actor class.

Class Cabbage: public Projectile

Cabbage(int startX, int startY, StudentWorld\* p)

:Projectile (IID\_CABBAGE, startX, startY, p)

Constructs a Cabbage using parameters: x-coord, y-coord and world in which it exists. Construct a Projectile from those parameters, because it is derived from Projectile class. ID is kept constant because it’s always the same value for Cabbages.

virtual void doSomething();

If it’s not dead and it hasn’t moved to the left of the screen, move it. Else make it dead.

virtual void move();

Move it 8 spaces to the right and rotate it 20 degrees counter-clockwise.

virtual bool isEnemy()

Returns false. Must be included because it is pure virtual function in the base class Projectile.

Class Turnip: public Projectile

Turnip(int startX, int startY, StudentWorld\* p)

:Projectile(IID\_TURNIP, startX, startY, p)

Constructs a turnip using parameters: x-coord, y-coord and world in which it exists. Construct a Projectile from those parameters, because it is derived from Projectile class. ID is kept constant because it’s always the same value for Turnips.

virtual void doSomething();

Check if Turnip is dead or has gone past the left of the screen. If not, move.

virtual void move();

Move 6 spaces to the left and rotate it 20 degrees counter-clockwise.

virtual bool isEnemy()

Returns true because only enemies fire turnips.

Class Torpedo: public Projectile

Torpedo(int startX, int startY, StudentWorld\* p, bool e)

:Projectile(IID\_TORPEDO, startX, startY, p)

Constructs a torpedo using parameters: x-coord, y-coord, bool e and world in which it exists. Bool e is used to determine whether the torpedo was fired from the NachenBlaster or a Snagglegon. Construct a Projectile from those parameters, because it is derived from Projectile class. ID is kept constant because it’s always the same value for Torpedos.

virtual void doSomething();

Check if it is dead or it as moved past the left or the right side of the screen. If not, move.

virtual void move();

If it’s an enemy’s torpedo, move it 8 spaces to the left. If friendly, move it 8 spaces to the right.

virtual bool isEnemy();

Returns the enemy private member variable. Torpedos fired by the NachenBlaster are friendly. Those fired by Snagglegons are not.

Class DamageableObject: public Actor

DamageableObject(int id, int startX, int startY, StudentWorld \*p, int health, int depth, double size)

:Actor(id, startX, startY, 0, size, depth, p)

Constructs a DamageableObject using parameters: ID, x-coord, y-coord, world in which it exists, health, depth and size. Construct an Actor from those parameters, because it is derived from Actor class. Direction is kept constant because it’s always the same value for DamageableObjects.

int getHealth();

Returns the hp of the object.

virtual void changeHealth(int x);

Adds x number of points to hp.

virtual void doSomething() = 0;

Pure virtual because each Damageable object’s doSomething function is different.

virtual bool isNachenBlaster() = 0;

Pure virtual because the damageable object could be the NachenBlaster.

virtual bool isDamageableObject()

Returns true.

virtual bool isGoodie()

virtual bool isProjectile()

All these functions return false. They must be included, because they are pure virtual functions in the Actor class.

Class NachenBlaster: public DamageableObject

NachenBlaster(StudentWorld\* p)

:DamageableObject(IID\_NACHENBLASTER, 0, 128, p, 50, 0, 1.0)

Constructs a NachenBlaster using parameters: world in which it exists. Construct an DamageableObject from those parameters, because it is derived from DamageableObject class. ID, x-coords, y-coords, health, depth and size are kept constant because they’re always the same value for the NachenBlaster.

virtual void doSomething();

Check if dead. Do certain things based on the key pressed. Move accordingly using the directional keys or WASD. Space fires cabbages. Tab fires a torpedo. Play the appropriate sounds. If cabbage power is less than 30, increment it by one, because it only runs once per tick. If hp is less than 0, set NachenBlaster to dead.

int getCabbagePower();

Returns cabbage power, cp.

virtual bool isEnemy()

Returns false. (You can’t be your own enemy! … or can you?)

virtual bool isNachenBlaster()

Returns true.

Class Alien: public DamageableObject

Alien(int id, int startX, int startY, StudentWorld \*p, int health, double ts, int fpl, int td)

:DamageableObject(id, startX, startY, p, health, 1, 1.5)

Constructs an Alien using parameters: ID, x-coord, y-coord, world in which it exists, health, travel speed, flight path length, and travel direction. Construct an DamageableObject from those parameters, because it is derived from DamageableObject class. Depth and size are kept constant because they’re always the same value for an Alien.

virtual bool isEnemy()

Returns true;

virtual bool isNachenBlaster()

Returns false;

virtual void doSomething();

If its health is less than 0, add an explosion to is coordinates, give it a chance to drop an item, increment the number of ships destroyed in the StudentWorld class, increase the score by the appropriate amount, play a death sound and set it to dead. Return immediately.

virtual void shoot() = 0;

virtual int getScore() = 0;

Pure virtual because each alien does something different when shooting or returning the score when they are killed.

virtual void dropSomething()

Virtual because each alien drops something different.

int getFPL();

Returns the flight path length.

void setFPL(int x);

Sets the flight path length.

double getTravelSpeed();

Returns the travel speed.

void setTravelSpeed(int x);

Sets the travel speed.

int getTravDir();

Returns the direction of travel (down and left, up and left, only left)

void setTravDir(int x);

Sets the travel direction.

virtual ~Alien();

Virtual so that it can be called when an Alien object is destroyed. It decrements the number of ships in the StudentWorld. No need to delete anything because it has no other objects in its private member variables.

Class Smallgon: public alien

Smallgon(int x, int y, StudentWorld \*p)

:Alien(IID\_SMALLGON, x, y, p, (5\*(1+(p->getLevel() - 1) \* 0.1)), 2.0, 0, 3)

Constructs a Smallgon using parameters: x-coord, y-coord, and world in which it exists. Construct an Alien from those parameters, because it is derived from Alien class. ID, health, depth and size are kept constant because they’re always the same value for a Smallgon.

virtual void shoot();

If NachenBlaster is to the left of the Smallgon and within 4 units in the Y direction, there is a chance of shooting at NachenBlaster. When it does, place fire a turnip and play the shoot sound.

virtual int getScore()

Returns 250 because it is the score a player gets for killing a Smallgon.

Class Smoregon: public Alien

Smoregon(int x, int y, StudentWorld \*p)

:Alien(IID\_SMOREGON, x, y, p, (5\*(1+(p->getLevel() - 1) \* 0.1)), 2.0, 0, 3)

Constructs a Smoregon using parameters: x-coord, y-coord, and world in which it exists. Construct an Alien from those parameters, because it is derived from Alien class. ID, health, depth and size are kept constant because they’re always the same value for a Smoregon.

virtual void shoot();

If NachenBlaster is to the left of the Smoregon and within 4 units in the Y direction, there is a chance of shooting at NachenBlaster. When it does, place fire a turnip and play the shoot sound. There is also a chance of Smoregon changing into ramming mode. When it does, set the travel speed to 5 and direction to directly left.

virtual int getScore()

Returns 250 because it is the score a player gets for killing a Smoregon.

virtual void dropSomething()

One in three chance of dropping a goodie. If it does, half chance of dropping torpedos, half change of dropping repair goodie.

Class Snagglegon: public Alien

Snagglegon(int x, int y, StudentWorld \*p)

:Alien(IID\_SMALLGON, x, y, p, (10\*(1+(p->getLevel() - 1) \* 0.1)), 1.75, 0, 1)

Constructs a Snagglegon using parameters: x-coord, y-coord, and world in which it exists. Construct an Alien from those parameters, because it is derived from Alien class. ID, health, depth and size are kept constant because they’re always the same value for a Snagglegon.

virtual void shoot();

If NachenBlaster is to the left of the Snagglegon and within 4 units in the Y direction, there is a chance of shooting at NachenBlaster. When it does, place fire a torpedo and play the torpedo shoot sound.

virtual int getScore()

Returns 1000 because it is the score a player gets for killing a Snagglegon.

virtual void dropSomething()

One in six chance of dropping an extra life goodie.

Class StudentWorld: public GameWorld

StudentWorld(std::string assetDir);

Constructs a StudentWorld. Constructs a GameWorld in the initializer list because it is derived from the GameWorld class.

virtual int init();

Create a NachenBlaster, and all the required stars. Continues the game.

virtual int move();

First make NachenBlaster do something. Then decide if player has finished the level based on the number of aliens killed and the level. If not won, add aliens accordingly. Then loop through the actors vector and make all Actors do something. Another nested while loop determines if actors collide with one another. If anything is dead, free up the memory and erase it from the vector. Determine if a new star needs to be introduced to the game. Determine if NachenBlaster is dead. If it is, return status of player dead. Finally, update the status text and return continue game.

virtual void cleanUp();

Loop through the actors vector and delete all the elements. Delete the NachenBlaster.

virtual ~StudentWorld();

Delete all elements in the actors vector as well as the NachenBlaster.

void addCabbage(int x, int y);

Add a cabbage to the actors vector at the specified location.

void addTorpedo(int x, int y, bool b);

Add a torpedo to the actors vector at the specified location.

void addAlien(int r, int max);

Add an alien to the actors vector at the specified location.

void addLifeGoodie(int x, int y);

Add a life goodie to the actors vector at the specified location.

void addRepairGoodie(int x, int y);

Add a repair goodie to the actors vector at the specified location.

void addTorpedoGoodie(int x, int y);

Add a torpedo goodie to the actors vector at the specified location.

int getNumShips();

Return the number of ships on the screen.

void setNumShips(int x);

Set the number of ships on the screen to x.

void fireTurnip(int x, int y);

Decrement the number of torpedos.

int getNachenX();

Returns the x-coord of NachenBlaster.

int getNachenY();

Returns the y-coord of NachenBlaster.

void updateStatusText();

Updates the status bar.

bool collided(Actor\* a1, Actor\* a2);

Determine if two Actors have collided and how they will react to each other. It takes two Actors because we can pass in anything that is derived from the Actor class. However, because the compiler does not know kind of Actor each class is, we must have methods in the classes that define what object it is, such as isNachenBlaster, or isDamageableObject. First calculate the Euclidean distance between two objects. If they are close enough, figure out what type of objects they are. If not, return false. If the first one is a projectile and the second actor is a DamageableObject, and if their isEnemy functions return different values, that means they must be able to damage each other. Set the projectile to dead. An if-else statement is used to determine the behavior, according to their unique ID’s. If first actor is the NachenBlaster, and their isEnemy functions return different values, the second actor must be a projectile or an Alien. An if-else statement is used to determine the behavior according to their unique ID’s. If isEnemy function returns the same value, the second actor must be a Goodie. An if-else statement is used to determine the behavior according to their unique ID’s. Return true.

void incNumDestroyed();

Increment the numDestroyed - the number of Alien ships destroyed.

int getTorpedos();

Return the numTorpedos.

void setTorpedos(int x);

Set the numTorpedos to x.

void incTorpedos();

Increase numTorpedos by 5.

void fireTorpedo();

Decrement numTorpedos.

void addExplosion(int x, int y);

Add an Explosion object to actors vector at the specified location.

1. Finished all functionalities.
2. It was not specified how to implement the collision function, so I implemented it the StudentWorld class. Because each object reacts with one another different, I distinguished between different objects through methods in their classes. For example, objects that could damage NachenBlaster returned true for the isEnemy function, and vice versa. Another example is simply creating a getID function that identifies every derived class. These were used along with many if-else statements to determine how each object reacted to the other.
3. Class Actor:  
    I tested the Actor class in many ways. First, I made sure that I was not able to instantiate it because it is an abstract class that contains pure virtual functions. Then each of its non-pure-virtual functions were tested by simply calling them. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class Explosion:  
    I tested the explosions class simply by playing the game. I could see the explosions every time an Alien died, and they were not too big.  
     
   Class Star:  
    The star class was tested also by simply playing the game. No strange behavior was seen. It did not collide with other objects in the game. Stars moved towards the left as expected, and new stars appeared on the right side as expected.  
     
   Class Goodie:  
    The Goodie class was tested by making sure that its doSomething function did things that were in common with all goodies, such as check if it is dead or if it has passed the left side of the screen. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class ExtraLifeGoodie:  
    The ExtraLifeGoodie class was tested by making sure that it moved correctly in the game. It moves towards the left and down slowly. I also made sure that it could only collide with the NachenBlaster, and that other Aliens would not collide with it. Additionally, I checked to see if it made the correct sound, increased the user’s score by 100 and incremented the NachenBlaster’s lives by one.  
     
   Class RepairGoodie:  
    The Repair Goodie class was tested in a similar manner to the Extra Life Goodie class. Its movements were tested and its collisions were tested so that it could only collide with the NachenBlaster. Additionally, its behavior is correct as it dies upon collision and increases NachenBlaster’s health by 10, increased the user’s score by 100 and played the correct sound.  
     
   Class TorpedoGoodie:  
    The torpedo goodie class was tested in a similar manner to the Repair and Extra Life Goodies. Its movement was tested and its collisions were tested so that it could only collide with NachenBlaster. Additionally, its behavior is correct as it dies upon collision and increases the torpedo count by 5. It also increases the user’s score by 100 and plays the correct sound.  
     
   Class Projectile:  
    I tested the Projectile class by making sure it could not be instantiated as it is an abstract class. I then tested all of its non-pure-virtual functions. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class Cabbage:  
    The Cabbage class was tested by making sure its functions did the correct thing. Move would move the cabbage towards the right. Then, it was seen in game that it behaved correctly and showed the correct image. It collided with Alien ships and died on impact. It also decreased Aliens’ health upon collision. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class Turnip:  
    The Turnip class was tested by making sure its functions did the correct thing. For example, the move function would move the Turnip towards the left. Then, I saw its behavior in game. It collided with NachenBlaster only, and when it did, it died and NachenBlaster was damaged appropriately. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class Torpedo:  
    The Torpedo class was tested similarly to the Turnip and Cabbage classes as they are all projectiles. For example, the move class had to move the Torpedo either left or right, based on whether the Torpedo was friendly or from the enemy. Its behavior in game was also observed. Friendly torpedos only collided with Aliens, and enemy torpedos only collided with the NachenBlaster. Upon collision, it would die and the object it collided with would be a Damageable object and would be damaged accordingly. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class DamageableObject:  
    I tested the DamageableObject class by making sure its member functions worked correctly. Then I made sure it could not be instantiated because it is an abstract class. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class NachenBlaster:  
    The NachenBlaster class was tested by observing its in game behavior. For example, it behaved appropriately when the direction buttons were pressed as well as tab and space. Its collisions were observed. It could collide with Aliens and Goodies, both of which caused them to die and affect the NachenBlaster in their unique ways. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class Alien:  
    I tested the Alien class by making sure all of its public member functions behaved correctly. I also made sure it could not be instantiated, as it is an abstract class. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class Smallgon:   
    The Smallgon class was tested by observing its in game behavior. Smallgons would move appropriately by changing directions randomly and shooting when in line with the NachenBlaster. It was also tested to make sure its public member functions would behave correctly. For example, its getScore function returns 250. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class Smoregon:  
    The Smoregon class was tested by observing its in game behavior. Smoregons would move appropriately by changing directions randomly. It would also shoot or change into ramming mode when in line with the NachenBlaster. The Smoregon would also drop the correct items at the correct chance. It was also tested to make sure its public member functions would behave correctly. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class Snagglegon:  
    The Snagglegon class was tested by observing its in game behavior. Snagglegons moved appropriately by changing directions randomly. It would also shoot a torpedo when in line with the NachenBlaster. The Snagglegon would also drop the correct items. Finally, I tested its inheritance to see if I could call public functions in its base classes.  
     
   Class StudentWorld:  
    The StudentWorld class was tested simply by playing the game. First the initialize function was tested by observing the game when it first starts. NachenBlaster was constructed at the correct location, and stars were also constructed correctly. The correct return value was returned as the game continued. Next, the move function was tested. The NachenBlaster’s doSomething function was run properly, as it could be appropriately moved by the keys. Then aliens were observed. They were appearing onto the screen at the correct rate, and they moved appropriately, meaning that all of their doSomething functions were called. Collisions occurred at the right time with the correct objects. Stars were seen moving appropriately and appearing at the correct rate. The game continued until NachenBlaster was dead, so it returned the correct values. Finally the cleanup method was tested through the p3 sanity check from the Project 3 spec page. It ensured that there were no memory leaks and that all objects were destroyed. The rest of the helper functions were tested to ensure they behaved correctly.