Brian Tagle

604907076

CS32

1.

Actor::doSomething();

This function is a pure virtual function in the Actor class, for which all of the game objects I created in this project descend. The function is pure virtual because Actor is an abstract class and all of Actor’s subclasses implement the doSomething function in some way in order to move and act in the game world.

Actor::getWorld();

This function gets the studentworld in which the actor resides. This is useful because the studentworld is the only thing that knows about all the game objects and is solely responsible for managing them, so if an Actor wants to know about anything the studentworld or ask the studentworld to do something it must ask the world it is in (by using this function to get its current world) to run one of the functions in the studentworld class.

Actor::setHealth(double healthValue);

This allows the actors to set their own health. This is usually done in the constructor, where an objects health is initialized and set to a certain value. I used this functions so I did not have to pass a hp value as a parameter and pass it through several constructors in initializer lists throughout all of my actors.

Actor::sufferDamage(double hp);

This function is used whenever a projectile hits the player or an alien ships and notifies the game that that player or alien should take damage. I could have used the setHealth function instead of this to deal damage but that is a bit more clumsy and this function is easier to use in most scenarios.

Actor::isAlive();

A function that returns true if the actor is alive and false otherwise. This function is very useful in the doSomething functions in all of the actor subclasses because every object in the game must be alive in order to do something and this function allows for a simply check on the actor’s life status before they do anything else.

Actor::isHostile();

This is a virtual function that returns false by default. Hostile entities such as aliens override this function and make it return true because they are hostile to the player. This function is useful for detecting what should be colliding with what. For example, hostile aliens collide with cabbages but not with turnips because turnips are on the hostile side.

Actor::getDamage();

This function returns the amount of damage that an enemy should suffer, should they get hit by this actor. For example, cabbages and turnips both have damages of 2 because they deal two damage if they hit the enemy(whether it be aliens or the player). Alien ships also have damage numbers because they deal damage when they collide with the NachenBlaster

Actor::getObjectScore();

This function returns the amount of score the player should receive should certain things happen to it. Aliens have score values for if the player destroys them and goodies have values if the player picks them up. These values let the studentworld know when, if, and by how much the player’s score should increase

Actor::GoodieAction();

This is a virtual function that returns false by default and is only overriden in the alien class. In the goodie class this functions rolls the dice on whether or not a goodie should spawn and what kind of goodie. Returns true if a goodie spawns and false otherwise. The studentworld uses this method on object pointers when they die from some sort of collision to see if any goodie should be spawned.

Actor::isGon();

This function is sort of a more specialized version of the isHostile() function. This function returns true if the actor is a “gon” alien. I made this function to help with collision detections because if I used isHostile for collision detections, cabbages were hitting turnips because cabbage were viewing projectiles as hostile entities when cabbages should only be hitting gon aliens.

Actor::isProjectile();

This function returns true if the actor is a projectile and false otherwise. This, along with the previous function, helped me get rid of projectile collisions by making sure if two objects collided and they were projectiles the collision was ignored

NachenBlaster::getCabbagePoints();

Function that returns how much cabbage energy the nachenblaster has, used for updating the top stat text in studentworld

NachenBlaster::getTorpedoCount();

Function that helps update the stat text with how many torpedoes the player has

NachenBlaster::IncreaseTorpedoCount();

Function that increases the nachenblaster torpedo count by 5 when the studentworld notifies it that it should add more torpedoes becauses it has hit a torpedo goodie

Projectile::SpecificMove();

This is a pure virtual function so projectile is an abstract class. The specific move for each of the different torpedoes is moving it by how far that kind of projectile should move and in the right direction. This helped reduce repeat code by not overriding doSomething in any of the projectile subclasses and putting the collision checks and this function all into the doSomething function of the projectile base class.

Goodie::benefit();

This is a pure virtual function so Goodie is an abstract class. Each of the different kinds of goodies bestow different benefits, which are each given by overriding this function in the subclasses of goodie. This cut down on repeat code by doing collisions and alive checks in a single function doSomething in Goodie and calling benefit if needed.

Alien::specialAction()

This is a pure virtual function so Alien is an abstract class. All the aliens move exactly the same minus one tiny exception that my code handles, so most of the alien code is repeated if separately in each alien subclass. Instead, I put the repeat code in doSomething of the alien and called specialAction in it, which will have different effects depending on the alien. Examples of specialAction are shooting torpedoes, turnips, or ramming speed for the smoregon.

2. I believe I finished all features.

3. It wasn’t specified if Aliens should be able to move the tick they are created, so They don’t move.

It was a little unclear if when creating aliens based on formulas given, I should create more until the condition is no longer satisfied using a while loop(creating multiple aliens in one tick) or using an if statement to only create a single alien if the condition is not met. I used an if statement for one alien creation max per tick.

How to make the explosions. Does the tick it is created count as a tick towards its life, I said it did. Also the starting size of explosions in the spec was different from in the sample game, so I just followed the spec.

4. I tested the star class and explosion class mainly through observation because these two classes were basically accessories to the game and testing the other classes basically ended up testing these by accident. Specifically with the star class I made the probably that spawned in 100% during tests to make sure they were spawning in correctly.

I tested the NachenBlaster by moving it around the screen to make sure it moved. I spawned in different types of aliens to make sure the NachenBlaster collided with them correctly and also that the cabbages and torpedos ired by the NachenBlaster would collide with the aliens correctly. These aliens also fired turnips and torpedos which were used to test if they would correctly hit the NachenBlaster. I also spawned different goodies and had the NachenBlaster pick them up to test collision with goodies and if the goodies bestowed the correct benefits upon the player.

I tested the aliens by spawning in multiple aliens several times over to make sure they don’t collide with one another and projectiles that they shoot don’t hit one another. I made sure each of the different aliens were able to interact with one another so there were no problems. I also spawned goodies to make sure the aliens were not able of colliding with them. I verified that they would try to attack the player if the chance in their code to shoot or ram the player were successful.

I tested Projectiles by shooting cabbages and torpedos with my NachenBlaster and creating aliens to fire turnips and torpedos and made sure they were colliding with their targets correctly.

I also spawned goodies to make sure that projectiles were not hitting any goodies.

I tested Goodies by spawning the different types of goodies in and making the player pick them up. This tested collision and also tested if the goodie would give the player the right benefit. I spawned aliens along the path of the goodie to make sure that the goodie was not able to be picked up by the aliens.

I tested the StudentWorld by spawning all the different objects into the world and testing all kinds of collisions over again since the studentworld is responsible for managing the collisions since only it knows the location of all its objects. I also tested the studentworld by calling several functions I made that would spawn different types of objects on demand and all these objects were created successfully.