Bradley Mont

Com Sci 32

Professor Smallberg

Lecture 2, Discussion 2C

UID: 804-993-030

Project 3: NachenBlaster

1.

Actor Class

virtual void doSomething() = 0;

* This function allows each Actor to do an individual action during each tick of the game. I chose to define a pure virtual version of the doSomething() function in my base Actor class because all Actors “do something” in a different way.

bool isAlive() const;

* This function tells whether an Actor is alive or not. It’s in the Actor class because all Actors have an alive/dead state. I made it non-virtual since it works the same for all Actors, and doesn’t need to be overridden.

void kill();

* This function sets an Actor’s state to dead. It’s in the Actor class because all Actors can get killed somehow. I made it non-virtual since it works the same for all Actors, and doesn’t need to be overridden: just set state to dead.

StudentWorld\* getWorld() const;

* This function returns the StudentWorld that an Actor resides in, so the world can then do operations on other objects in the world. It’s in the Actor class because all Actors have a StudentWorld. I made it non-virtual since it works the same for all Actors, and doesn’t need to be overridden: just return the Actor’s world.

virtual bool isAlien() const;

* This function returns whether or not an Actor is an Alien, which is useful in seeing how many Aliens are on the screen. It’s in the Actor class because Alien is a type of Actor, so each Actor is or is not an Alien. I made it virtual since by default most Actors are not Aliens, but in the Alien class, it needs to be overridden to return true.

virtual bool isProjectile() const;

* This function returns whether or not an Actor is a Projectile, which is useful in determing of something collided with a Projectile. It’s in the Actor class because Projectile is a type of Actor, so each Actor is or is not a Projectile. I made it virtual since by default most Actors are not Projectiles, but in the Projectile class, it needs to be overridden to return true.

virtual void ifAlienExecuteCollision(int damageAmt);

* This function executes everything that an Alien must handle during a collision. It’s in the Actor class because any Actor object must be able to call this function since I returned an Actor pointer in the StudentWorld getOneCollidingAlien function. I made it virtual since the Alien class must override the function (which currently does nothing) to execute the Alien’s part of a collision.

DamageableObject Class

double hitPoints() const;

* This function returns the number of hit points a DamageableObject has. It’s in the DamageableObject class since all DamageableObjects have hit points, but not all Actors have hit points. It’s non-virtual since it works the same for all DamageableObjects: just return the numbert of hit points.

void increaseHitPoints(double amt);

* This function increases the number of hit points a DamageableObject has by amt. It’s in the DamageableObject class since multiple types DamageableObjects can possibly increase their hit points. It’s non-virtual since it works the same for all DamageableObjects: just increase the number of hit points.

virtual void sufferDamage(double amt, int cause);

* This function decreases the number of hit points a DamageableObject has by amt. It also has the user indicate the cause. It’s in the DamageableObject class since multiple types DamageableObjects can possibly decrease their hit points. It’s non-virtual since it works the same for all DamageableObjects: just decrease the number of hit points.

Star Class

virtual void doSomething();

* This function checks if a Star is alive, moves it 1 pixel to the left, and kills it if it goes off the grid. It’s in the Star class because all Stars execute the same version of doSomething. It’s virtual since it is overriding the virtual Actor doSomething.

NachenBlaster Class

virtual void doSomething();

* This function checks if the NachenBlaster is alive, responds to any feedback from the player (move or shoot projectile), and adds to cabbage energy points. It’s in the NachenBlaster class because the NachenBlaster executes its own version of doSomething. It’s virtual since it is overriding the virtual Actor doSomething.

void increaseFlatulenceTorpedoes(int amt);

* This function increases the NachenBlaster’s Flatulence Torpedoes by amt. It’s in the NachenBlaster class because only the NachenBlaster can increase its Flatulence Torpedoes. It’s non-virtual since it does not need to be overridden.

int cabbageEnergyPoints() const;

* This function returns the NachenBlaster’s amount of cabbage energy points. It’s in the NachenBlaster class because only the NachenBlaster has cabbage energy points. It’s non-virtual since it does not need to be overridden.

int flatulenceTorpedoCount() const;

* This function returns the NachenBlaster’s amount of flatulence torpedoes. It’s in the NachenBlaster class because only the NachenBlaster needs to access it’s number of flatulence torpedoes left. It’s non-virtual since it does not need to be overridden.

Projectile Class

virtual bool isProjectile() const;

* This function returns true (that the Projectile object is indeed a Projectile). It’s in the Projectile class so the StudentWorld can see if an object is a Projectile. It’s virtual since it’s overriding the Actor version of the function.

Cabbage Class

virtual void doSomething();

* This function checks if the Cabbage is alive and in bounds, checks to see if the cabbage collided with an Alien (giving it the proper health reduction), and then moves the cabbage 2 pixels right if it didn’t collide. It’s in the Cabbage class since all cabbages have a specialized way of moving and delivering damage. It’s virtual since it’s overriding the Projectile version of doSomething.

FlatulenceTorpedo Class

virtual void doSomething();

* This function checks if the FlatulenceTorpedo is alive and in bounds, checks to see if the FlatulenceTorpedo collided with an Alien/NachenBlaster (giving it the proper health reduction), and then moves the FlatulenceTorpedo 8 pixels in it’s direction if it didn’t collide. It’s in the FlatulenceTorpedo class since all FlatulenceTorpedoes have a specialized way of moving and delivering damage. It’s virtual since it’s overriding the Projectile version of doSomething.

Turnip Class

virtual void doSomething();

* This function checks if the Turnip is alive and in bounds, checks to see if the Turnip collided with an Alien (giving it the proper health reduction), and then moves the Turnip left if it didn’t collide. It’s in the Turnip class since all Turnips have a specialized way of moving and delivering damage. It’s virtual since it’s overriding the Projectile version of doSomething.

Alien Class

virtual void doSomething();

* This function checks if the Alien is alive and in bounds, checks for a collision, possibly picks a new flight plan, possibly shoots a projectile, moves the Alien, and then checks for a collision again. It’s in the Alien class since all Alien’s run the same version of doSomething. It’s virtual since it overrides the Actor version of doSomething.

int getFlightPlanLength() const;

* This function returns the flight plan length of the Alien. It’s in the Alien class since all Aliens can have a flight plan length. It’s non-virtual since it runs the same way for all Aliens and doesn’t need to be overridden.

int getTravelDirection() const;

* This function returns the travel direction of the Alien. It’s in the Alien class since all Aliens can have a travel direction. It’s non-virtual since it runs the same way for all Aliens and doesn’t need to be overridden.

double getTravelSpeed() const;

* This function returns the travel speed of the Alien. It’s in the Alien class since all Aliens can have a travel speed. It’s non-virtual since it runs the same way for all Aliens and doesn’t need to be overridden.

void setFlightPlanLength(int length);

* This function sets the flight plan length of the Alien to length. It’s in the Alien class since all Aliens can set their flight plan length. It’s non-virtual since it runs the same way for all Aliens and doesn’t need to be overridden.

void setTravelDirection(int direction);

* This function sets the travel direction of the Alien to direction. It’s in the Alien class since all Aliens can set their travel direction. It’s non-virtual since it runs the same way for all Aliens and doesn’t need to be overridden.

void setTravelSpeed(double speed);

* This function sets the travel speed of the Alien to speed. It’s in the Alien class since all Aliens can set their travel speed. It’s non-virtual since it runs the same way for all Aliens and doesn’t need to be overridden.

virtual bool isAlien() const;

* This function returns true (that the Alien object is indeed an Alien). It’s in the Alien class so the StudentWorld can see if an object is a Alien. It’s virtual since it’s overriding the Actor version of the function.

virtual void ifAlienExecuteCollision(int damageAmt);

* This function has the Alien lose damageAmt hit points, and it kills the Alien and does all proper sounds, score increases, and possible goodie drops if the Alien’s hit points reach zero. It’s virtual since it’s overriding the Actor version of the function.

Smallgon Class

-no public member functions

Smoregon Class

-no public member functions

Snagglegon Class

virtual bool shootProjectile();

* This function checks if the Snagglegon is in position to shoot, and with the right odds, it fires a FlatulenceTorpedo and plays a sound. It’s in the Snagglegon class since Snagglegon’s have their own unique way of shooting projectiles and handling the odds that they do shoot a Projectile. It’s virtual since it’s overriding the Alien version of shootProjectile.

Goodie Class

virtual void doSomething();

* This function checks if the Goodie is alive and in bounds, checks to see if the Goodie collided with the NachenBlaster (giving it a benefit if it collides), moves down and left .75 pixels if it didn’t collide, and then checks for a collision again (giving it a benefit if it collides). It’s in the Goodie class since all Goodies execute the same version of doSomething. It’s virtual since it’s overriding the Actor version of doSomething.

ExtraLifeGoodie Class

-no public member functions

RepairGoodie Class

-no public member functions

FlatulenceTorpedoGoodie Class

-no public member functions

Explosion Class

virtual void doSomething();

* This function checks if the Explosion is alive, increases its size by a factor of 1.5, and kills itself after 4 ticks. It’s in the Explosion class since all Explosions execute a specialized version of doSomething. It’s virtual because it’s overriding the Actor doSomething function.

StudentWorld Class

virtual int init();

* This function adds 30 stars to the grid, adds the NachenBlaster to the grid, and sets the number of destroyed Alien ships to zero. It’s in the StudentWorld class since the StudentWorld is responsible for initializing the grid correctly. It’s virtual since it overrides the GameWorld init function.

virtual int move();

* This function possibly adds new stars and aliens, gives all live actors an opportunity to do something, and if the NachenBlaster didn’t win or die, it removes dead game objects and updates the display text. It’s in the StudentWorld class since the StudentWorld is responsible for giving all Actors a chance to do something and updating the board as necessary. It’s virtual since it overrides the GameWorld move function.

virtual void cleanUp();

* This function removes all actors from the grid and frees any dynamically allocated memory. It’s in the StudentWorld class since the StudentWorld is responsible for freeing all memory at the end of a round. It’s virtual since it overrides the GameWorld cleanUp function.

void addActor(Actor\* newActor);

* This function adds a new Actor to the grid. It’s in the StudentWorld class since the StudentWorld is responsible for adding and removing actors. It’s non-virtual since it does not need to be overridden.

NachenBlaster\* getNachenBlaster() const;

* This function returns the world’s NachenBlaster. It’s in the StudentWorld class since the StudentWorld contains the NachenBlaster as a private member variable. It’s non-virtual since it does not need to be overridden.

NachenBlaster\* getCollidingNachenBlaster(const Actor\* a) const;

* This function returns a pointer to the NachenBlaster if Actor a has collided with the NachenBlaster, and it returns nullptr otherwise. It’s in the StudentWorld class since the StudentWorld can access all of the Actors in the grid. It’s non-virtual since it does not need to be overridden.

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

* This function returns an Actor pointer to an Alien if Actor a has collided with an Alien, and it returns nullptr otherwise. It’s in the StudentWorld class since the StudentWorld can access all of the Actors in the grid. It’s non-virtual since it does not need to be overridden.

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

* This function returns an Actor pointer to a Projectile if Actor a has collided with an Projectile, and it returns nullptr otherwise. It’s in the StudentWorld class since the StudentWorld can access all of the Actors in the grid. It’s non-virtual since it does not need to be overridden.

void recordAlienDestroyed();

* This function adds one to the number of destroyed Alien ships. It’s in the StudentWorld class since the StudentWorld keeps track of how many Alien ships have been destroyed. It’s non-virtual since it does not need to be overridden.

2. I finished all functionalities, and I have no known bugs.

3. I chose to have my NachenBlaster as a separate variable, apart from my list of all other Actors. This allowed me to have an easily-implementible version of a getNachenBlaster() function, where I just returned the m\_NachenBlaster pointer. I used this function multiple times throughout my project. I found it to be much easier than having to traverse through the actors list every time I needed to access the NachenBlaster. Additionally, I chose to have the Projectile handle some parts and have the Alien handle some parts when an Alien collides with a Projectile. I found this to be the most efficient way to handle this.

4.

Actor Class

The key to testing my Actor class was developing it incrementally. This way, if I got an error, it was most likely due to the most recent change that I made. I first made sure that an Actor was constructed and destructed correctly by adding std::cout statements so I knew when Actors were being constructed and destructed. This helped me assure that I had no memory leaks. I then added dummy versions of the functions and made sure that my program still compiled and linked successfully with the dummy functions. Then, one by one, I implemented each function and tested them one at a time, making sure that each function worked before I moved on to the next function.

DamageableObject Class

I couldn’t really test my DamageableObject very thoroughly since it’s an ABC. So I implemented the DamageableObject functions to the best of my ability, and then once I finished NachenBlaster, Alien, etc, I was able to more thoroughly test my DamageableObject functions using its derived classes.

Star Class

I tested the Star class by having a cout statement in the constructor, so I could make sure that the Stars were being properly destroyed once they flew off the edge of the grid. I also froze the game at the beginning to count and make sure that 30 stars were successfully constructed. I also created a sample project to make sure that I was successfully retrieving a random double between .05 and .5 for the size.

NachenBlaster Class

My main method for testing the NachenBlaster class was to play the game myself. I would try going off the grid, colliding with everything, and more to assure that the NachenBlaster handled every situation correctly. I also utilized the display at the top so I could keep track and see if the NachenBlaster was losing the right amount of health, gaining the right amount of cabbage energy points, and more.

Projectile Class

Similar to the DamageableObject class, I really had to wait until I developed my Cabbage, FlatulenceTorpedo, and Turnip classes to fully test the functionality of the Projectile class functions. After that, I was able to see that all the Projectiles were doing what they were supposed to.

Cabbage Class

I mainly tested my Cabbage class by playing the game. I shot a multitude of cabbages with the NachenBlaster and checked to see that they were rotating correctly, going the correct speed, and inflicting the correct amount of damage.

FlatulenceTorpedo Class

I mainly tested my FlatulenceTorpedo class by playing the game. I shot a multitude of FlatulenceTorpedoes with the NachenBlaster and checked to see that they were going the correct speed and inflicting the correct amount of damage. I did the same with Snagglegons shooting FlatulenceTorpedoes at me.

Turnip Class

I mainly tested my Turnip class by playing the game. I had Aliens shoot a multitude of Turnips and then checked to see that they were rotating correctly, going the correct speed, and inflicting the correct amount of damage.

Alien Class

Similar to the Projectile class, I really had to wait until I developed my Smallgon, Smoregon, and Snagglegon classes to fully test the functionality of the Alien class functions. After that, I was able to see that all the Aliens were doing what they were supposed to.

Smallgon Class

I mainly tested my Smallgon class by playing the game. I checked to see that the Smallgons were shooting the correct projectiles when they were supposed to, following correct flight paths, and dying and triggering explosions in the right situations.

Smoregon Class

I mainly tested my Smoregon class by playing the game. I checked to see that the Smoregons were shooting the correct projectiles when they were supposed to, following correct flight paths, and dying and triggering explosions in the right situations.

Snagglegon Class

I mainly tested my Snagglegon class by playing the game. I checked to see that the Snagglegons were shooting the correct projectiles when they were supposed to, following correct flight paths, and dying and triggering explosions in the right situations.

Goodie Class

Similar to the Projectile class, I really had to wait until I developed my ExtraLifeGoodie, RepairGoodie, and FlatulenceTorpedoGoodie classes to fully test the functionality of the Goodie class functions. After that, I was able to see that all the Goodies were doing what they were supposed to.

ExtraLifeGoodie Class

I mainly tested my ExtraLifeGoodie class by playing the game. I checked to see that the ExtraLifeGoodies were properly colliding with the NachenBlaster and giving the correct benefit.

RepairGoodie Class

I mainly tested my RepairGoodie class by playing the game. I checked to see that the RepairGoodies were properly colliding with the NachenBlaster and giving the correct benefit.

FlatulenceTorpedoGoodie Class

I mainly tested my FlatulenceTorpedoGoodie class by playing the game. I checked to see that the FlatulenceTorpedoGoodies were properly colliding with the NachenBlaster and giving the correct benefit.

Explosion Class

I tested my Explosion class by playing my own game and then comparing my explosions to the size of the explosions in the sample game. Also, I played my game to assure that explosions were being triggered at the right time and location.

StudentWorld Class

I tested my StudentWorld class by comparing my game as a whole to the sample game. If I couldn’t find any differences between the two versions, then that means that my StudentWorld is handling everything the same way that the sample version is. Also, I had to use cout statements and the g++ compiler to test for any memory leaks and proper destruction.