Vanessa Lee

504766034

NachenBlaster Report

**A high level description of each of your public member function in each of your classes, and why you chose to define each member function in its host class; also explain why (or why not) you decided to make each function virtual or pure virtual.**

***class StudentWorld:***

***public:***

*StudentWorld::StudentWorld(string assetDir);*

* Initializes base class, GameWorld, with assetDir

*StudentWorld::~StudentWorld();*

* Iterates through vector
* Deletes dynamically allocated Actors
* Clears vector

*int StudentWorld::init();*

* Initialize m\_player to dynamically allocated NachenBlaster
* Pushes m\_player to m\_actor
* Initializes m\_aliensDestroyed = 0 and m\_aliensOnMap = 0
* Creates 30 stars and pushes them onto m\_actor
* Creates stats text on top of screen

*int StudentWorld::move();*

* Iterates through m\_actor to check if Actor is alive
  + If alive, call that Actor’s ActorName::doSomething() function
* Initializes local bool playerAlive = true
* Check if aliens need to be spawned on map
  + If true, call StudentWorld::addAliens() to add aliens
* Iterates through m\_actor to check if Actor is not alive
  + If Actor is NachenBlaster, set playerAlive = false
  + Otherwise, delete dynamically allocated Actor
  + Erase vector slot
* Adds new star onto map (1/15 chance)
* Update stats text
* If playerAlive == false, decrease player lives and end current level
* If m\_aliensDestroyed == total number required to pass level, then advance to next level

*void StudentWorld::cleanUp();*

* Iterates through m\_actor and deletes every single Actor
* Erase every item in vector

*NachenBlaster\* StudentWorld::getPlayer();*

* Returns pointer to m\_player

*void StudentWorld::checkCollision(Actor\* a);*

* Checks each Actor in the vector m\_actor for a Projectile
  + If projectile collides with an Alien ship
    - Set Projectile state to dead so it can be removed
    - Damage alien ship appropriately
* Checks to see if NachenBlaster ship collided with Alien ship
  + If true, set Alien ship state to dead
  + Call StudentWorld::recordAlienDestroyed()
  + Increase score and damage NachenBlaster appropriately
  + Play sound
  + Call StudentWorld::addActor(Actor\* a) to add new Explosion actor

*double StudentWorld::euclidianDist(double x1, double y1, double x2, double y2);*

* Calculates Euclidian distance for collision

*void StudentWorld::recordAlienDestroyed();*

* Increases m\_aliensDestroyed by 1
* Decreases m\_aliensOnMap by 1

*bool StudentWorld::needAliens();*

* Calculates if there needs to be aliens added on to map. Returns true, returns false otherwise.

*void StudentWorld::addActor(Actor\* a);*

* Pushes back Actor onto m\_actor

*void StudentWorld::addAlien();*

* Calculates chance of adding a Smallgon, Smoregon, and Snagglegon
* Calls StudentWorld::addActor(Actor\* a) to add new Alien
* Increases m\_aliensOnMap by 1

*void StudentWorld::decAlien();*

* Decreases m\_aliensOnMap by 1

*double StudentWorld::min(double a, double b);*

* Returns the minimum number of two doubles

*void StudentWorld::updateStats();*

* Updates stats text line

***private:***

*vector<Actor\*> m\_actor* – STL container to hold all Actors

*NachenBlaster\* m\_player* – points to NachenBlaster

*double m\_aliensDestroyed* – keeps track of the number of aliens player has destroyed

*double m\_aliensOnMap* – keeps track of the number of aliens currently on the map

There was no need to make any of the StudentWorld functions virtual or pure virtual (excluding init(), move(), cleanUp()) because I did not redefine any functions in the base class GameWorld. The functions in StudentWorld are specific to StudentWorld.

***class Actor:***

***public:***

*Actor::Actor(StudentWorld\* world, int image, double x, double y, int dir, double size, int depth);*

* Initializes base class GraphObject with image, x, y, dir, size, and depth
* Sets m\_alive = true
* Sets m\_world = world
* Sets m\_image = image

*Actor::~Actor();*

* Does nothing

*virtual void Actor::doSomething() = 0;*

* This function does nothing in Actor
* All derived classes will have their own verison of doSomething(), therefore Actor’s doSomething() will never be called, so it is pure virtual.

*void Actor::setDead();*

* Sets m\_alive = false
* This is not virtual or pure virtual because derived classes can set their own state to dead by calling this function. All Actors will use this function in the same way and will not redefine it in their own class.

*bool Actor::isAlive() const;*

* Returns m\_alive
* This is not virtual or pure virtual because derived classes need to be able to check if they are alive or not. All Actors will use this function in the same way and will not redefine it in their own class.

*StudentWorld\* Actor::getWorld();*

* Returns pointer to StudentWorld
* This is not virtual or pure virtual because derived classes need to be able to access the world they are in.

*bool Actor::didCollide(const Actor\* a1, const Actor\* a2);*

* Calls StudentWorld::euclidianDist(double x1, double y1, double x2, double y2) and returns true if collided.
* This function is not virtual or pure virtual because derived Actors need to be able to call it to see if they collided with a projectile or enemy ship.

*int Actor::getIID() const;*

* Returns m\_image
* This function is not virtual or pure virtual because derived Actors need to be able to check their own image ID. It also allows StudentWorld class to access an Actor’s image ID.

*virtual void Actor::sufferDamage(double amt);*

* Does nothing
* This function is not pure virtual because not every Actor suffers damage, so not every class can redefine their own sufferDamage function, but StudentWorld needs to be able to access and inflict damage on Actors when StudentWorld::checkCollision(Actor\* a) is called. Setting it to virtual will call the most derived version of the function, despite using a base class pointer to access it.

*bool Actor::isProjectile(const Actor\* a);*

* If Actor is a projectile, return true. Return false otherwise.
* This function is not virtual or pure virtual because the derived classes have no need to redefine it anywhere else.

*virtual void Actor::whenDestroyed();*

* Does nothing
* This is function is virtual because it gets redefined in a derived class. This is not pure virtual because not every class needs a whenDestroyed function, but StudentWorld needs to access it.

***private:***

*StudentWorld\* m\_world* – stores a pointer to StudentWorld

*bool m\_alive* – keeps track of whether or not an Actor is alive

*int m\_image* – stores image ID of an Actor

***class DamageableObject : public Actor***

***public:***

*DamageableObject::DamageableObject(StudentWorld\* world, int imageId, double startX,*

*double startY, int startDir, double size, int depth, double hitPoints);*

* Sets m\_hp = hitPoints

*DamageableObject::~ DamageableObject();*

* Does nothing

*virtual void DamageableObject::sufferDamage(double amt);*

* Calls DamageableObject::decHP(amt)

*double DamageableObject::getHP();*

* Returns m\_hp
* This function is not virtual or pure virtual because its derived classes need to call this function and they do not redefine this function because every class needs to get its HP in the same way.

*virtual void DamageableObject::incHP(double amt);*

* Increases m\_hp by amt
* This function is not pure virtual because it is not a dummy function and other classes access this function. This function is virtual because NachenBlaster needs to have their own version of incHP(double amt) so StudentWorld can access it.

*virtual void DamageableObject::decHP(double amt);*

* Decreases m\_hp by amt
* This function is not pure virtual because it is not a dummy function and other classes access this function. This function is virtual because NachenBlaster needs to have their own version of decHP(double amt) so StudentWorld can access it.

***private:***

*double m\_hp –* stores hit points of ships

***class NachenBlaster : public DamageableObject***

***public:***

*NachenBlaster::NachenBlaster(StudentWorld\* world);*

* Sets m\_cabbages = 30
* Sets m\_torpedos = 0

*NachenBlaster::~NachenBlaster()*

* Does nothing

*virtual void NachenBlaster::doSomething();*

* Checks if NachenBlaster is alive
* Gets user input and moves NachenBlaster or shoots projectile depending on input key
* If NachenBlaster has less than 30 cabbages in its arsenal, increase by 1 per tick.
* If HP is less than 0, call Actor::setDead()

*virtual void invHP(double amt);*

* Increase HP by amt

*virtual void decHP(double amt);*

* Decrease HP by amt

*int NachenBlaster::getCabbage();*

* Converts cabbage number to a percentage and returns that percentage
* This function is not virtual or pure virtual because no other class needs to redefine it. It is specific to NachenBlaster class only.

*int NachenBlaster::getTorpedoes();*

* Return m\_torpedoes
* This function is not virtual or pure virtual because no other class needs to redefine it. It is specific to NachenBlaster class only.

*void NachenBlaster::incTorpedoes();*

* Increases m\_torpedos by 5
* This function is not virtual or pure virtual because no other class needs to redefine it. It is specific to NachenBlaster class only.

***private:***

*int m\_cabbage –* stores number of cabbages the NachenBlaster has

*int m\_torpedoes* – stores number of torpedoes the NachenBlaster has

***class Alien : public DamageableObject***

***public:***

*Alien::Alien(StudentWorld\* world, int imageID, double startX, double startY, double hitPoints,*

*double damageAmt, double speed, unsigned int scoreValue);*

* Sets m\_scoreValue = scoreValue
* Sets m\_speed = speed
* Sets m\_dmgAmt = damageAmt
* Sets m\_flight = 0
* Sets m\_dir = 0

*Alien::~Alien()*

* Does nothing

*int Alien::getSpeed();*

* Returns m\_speed
* This function is not virtual or pure virtual because all Aliens need to know their speed

*int Alien::getScoreValue();*

* Returns m\_scoreValue
* This function is not virtual or pure virtual because all Aliens need to know their score value.

*int Alien::getFlight();*

* Returns m\_flight
* This function is not virtual or pure virtual because all Aliens need to know their flight plan.

*int Alien::getTravelDirection();*

* Returns m\_dir
* This function is not virtual or pure virtual because all Aliens need to know their travel direction.

*void Alien::setTravelDirection(int dir);*

* Checks for conditions and sets m\_dir to either DOWN, UP, or LEFT
* Otherwise set m\_dir to dir
* This function is not virtual or pure virtual because the derived Alien classes will need to access this function to set their travel direction.

*void Alien::setFlight(int amt);*

* Set m\_flight = amt
* This function is not virtual or pure virtual because all Aliens need to be able to set their flight plan and it is being managed by the Alien base class.

*void Alien::setSpeed(double amt);*

* Set m\_speed = amt
* This function is not virtual or pure virtual because Aliens need to be able to set their own speed, which is managed by the Alien base class

*virtual void Alien::dropGoodie();*

* Does nothing
* This function is not pure virtual because not all Aliens drop a goodie. Those that do handle it in their own way, so it must be virtual.

*bool Alien::needFlight();*

* Determines if Alien needs a new flight plan based on conditions
* This function is not virtual or pure virtual because every Alien calls this function and does not handle it in their own way. The Alien base class handles their flight plan and travel direction, so the derived classes do not.

*void Alien::move(int dir);*

* Alien will move in the direction given
* Decrease m\_flight by 1
* This function is not virtual or pure virtual because all Aliens move the same

*void Alien::whenDestroyed();*

* Sets state to dead
* Calls StudentWorld::recordAlienDestroyed()
* Increases player score
* Plays sound
* Introduces explosion
* This function is not virtual or pure virtual because all Aliens have the same procedure when destroyed.

***private:***

*int m\_scoreValue –* holds the score value of the Alien when they are defeated by the player

*int m\_speed –* stores the speed of an Alien

*int m\_flight* – stores the flight plan length of an Alien

*double m\_dmgAmt* – stores the damage an Alien does to NachenBlaster when they collide

*int DOWN = 1 –* used to determine how an alien will move

*int UP = 2 –* used to determine how an alien will move

*int LEFT = 3 –* used to determine how an alien will move

*int m\_dir –*  stores directional value (DOWN, UP, LEFT)

***class Smallgon : public Alien***

***public:***

*Smallgon::Smallgon(StudentWorld\* world, double x, double y , double hp);*

* Initializes Alien base class

*Smallgon::~Smallgon();*

* Does nothing

*virtual void Smallgon::doSomething();*

* Check if Smallgon is alive
* If Smallgon is off screen, call StudentWorld::decAlien() and set state to dead. Return.
* If Smallgon has no HP left, call Alien::whenDestroyed()
* Check collision by calling StudentWorld::checkCollision(Actor\* a)
* Shoot turnip if NachenBlaster is in range
* Call Alien::move(int dir) to move Smallgon
* Check collision again

*virtual void Smallgon::sufferDamage(double amt);*

* Decrease HP by amt

***class Smoregon : public Alien***

***public:***

*Smoregon::Smoregon(StudentWorld\* world, double x, double y , double hp);*

* Initializes Alien base class

*Smoregon::~Smoregon();*

* Does nothing

*virtual void Smoregon::doSomething();*

* Check if Smoregon is alive
* If Smoregon is off screen, call StudentWorld::decAlien() and set state to dead. Return.
* If Smoregon has no HP left, call Alien::whenDestroyed()
  + Has a chance to dropGoodie() if dead
* Check collision by calling StudentWorld::checkCollision(Actor\* a)
* Shoot turnip if NachenBlaster is in range
  + Has a chance to change direction to LEFT and increase speed
* Call Alien::move(int dir) to move Smoregon
* Check collision again

*virtual void Smoregon::sufferDamage(double amt);*

* Decrease HP by amt

*virtual void Smoregon::dropGoodie();*

* Chance to drop RepairGoodie or TorpedoGoodie

***class Snagglegon : public Alien***

***public:***

*Snagglegon:: Snagglegon (StudentWorld\* world, double x, double y , double hp);*

* Initializes Alien base class

*Snagglegon::~ Snagglegon ();*

* Does nothing

*virtual void Snagglegon::doSomething();*

* Check if Snagglegon is alive
* If Snagglegon is off screen, call StudentWorld::decAlien() and set state to dead. Return.
* If Snagglegon has no HP left, call Alien::whenDestroyed()
  + Has a chance to dropGoodie() if dead
* Check collision by calling StudentWorld::checkCollision(Actor\* a)
* Shoot torpedo if NachenBlaster is in range
* Call Alien::move(int dir) to move Snagglegon
* Check collision again

*virtual void Snagglegon::sufferDamage(double amt);*

* Decrease HP by amt

*virtual void Snagglegon::dropGoodie();*

* 1/6 chance to drop an ExtraLife goodie

***class Star : public Actor***

***public:***

*Star::Star(StudentWorld\* world, double x, double y, double size);*

* Initalizes Actor base class

*Star::~Star()*

* Does nothing

*virtual void Star::doSomething();*

* Check if alive
* Check if off screen, if so call set to dead
* Move left by 1 pixel

***class Explosion : public Actor***

***public:***

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

* Set m\_ticks = 4

*Explosion::~Explosion();*

* Does nothing

*virtual void Explosion::doSomething();*

* Check if alive
* Increase size by factor of 1.5
* Decrease m\_ticks until 0, then set dead.

***private:***

*int m\_ticks –* counts number of ticks before explosion disappears

***class Projectiles : public Actor***

***public:***

*Projectiles::Projectiles(StudentWorld\* world, int image, double x, double y, int dir, double size,*

*int depth);*

* Initializes Actor base class

*Projectiles::~Projectiles();*

* Does nothing

***class Cabbage : public Projectiles***

***public:***

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

* Initializes Projectile base class

*Cabbage::~Cabbage();*

* Does nothing

*virtual void Cabbage::doSomething();*

* Check if alive
* If cabbage flew off screen, set state to dead
* Move to the right by 8 pixels
* Rotate 20 degrees counter clockwise

***class Turnip : public Projectiles***

***public:***

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

* Initializes Projectile base class

*Turnip::~ Turnip ();*

* Does nothing

*virtual void Turnip::doSomething();*

* Check if alive
* If turnip flew off screen, set state to dead
* Move to the left by 8 pixels
* Rotate 20 degrees counter clockwise

***class Torpedo : public Projectiles***

***public:***

*Torpedo::Torpedo(StudentWorld\* world, double x, double y, int dir);*

* Initializes Projectile base class

*Torpedo::~Torpedo();*

* Does nothing

***class AlienTorpedo : public Torpedo***

***public:***

*AlienTorpedo::AlienTorpedo(StudentWorld\* world double x, double y);*

* Initializes Torpedo base class
* Torpedoes shot by an Alien actor has a direction of 180 degrees

*AlienTorpedo::~AlienTorpedo();*

* Does nothing

*virtual void AlienTorpedo::doSomething();*

* Check if alive
* If flew off screen, set state to dead
* Check for collision with NachenBlaster
* Move 8 pixels to the left
* Check for collision again

***class PlayerTorpedo : public Torpedo***

***public:***

*PlayerTorpedo:: PlayerTorpedo (StudentWorld\* world double x, double y);*

* Initializes Torpedo base class
* Torpedoes shot by an NachenBlaster actor has a direction of 0 degrees

*PlayerTorpedo::~ PlayerTorpedo ();*

* Does nothing

*virtual void PlayerTorpedo::doSomething();*

* Check if alive
* If flew off screen, set state to dead
* Move 8 pixels to the left

***class Turnip : public Projectiles***

***public:***

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

* Initializes Projectile base class

*Turnip::~Turnip();*

* Does nothing

*virtual void Turnip::doSomething();*

* Check if alive
* If turnip flew off screen, set state to dead
* Check for collision with NachenBlaster
* Move 6 pixels to the left
* Rotate 20 degrees counter clockwise
* Check for collision again

***class Goodies : public Actor***

***public:***

*Goodies::Goodies(StudentWorld\* world, int image, double x, double y, int dir, double size, int*

*depth);*

* Initializes Actor base class

*Goodies::~Goodies();*

* Does nothing

*void Goodies::whenCollided();*

* Increase player score by 100 points
* Set state to dead
* Play sound
* This function is virtual because all Goodies do the same thing when a NachenBlaster collides into it

*void Goodies::move();*

* Move down and to the left
* This function is virtual because all Goodies move the same

***class ExtraLife : public Goodies***

***public:***

*ExtraLife::ExtraLife(StudentWorld\* world, double x, double y);*

* Initializes Goodies base class

*ExtraLife::~ExtraLife();*

* Does nothing

*virtual void ExtraLife::doSomething();*

* Check if alive
* If flew off screen, set state to dead
* Check if NachenBlaster collided with Goodie
  + If so, call whenCollided() and increase player lives by 1
* Move

***class RepairGoodie : public Goodies***

***public:***

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

* Initializes Goodies base class

*RepairGoodie::~ RepairGoodie ();*

* Does nothing

*virtual void RepairGoodie::doSomething();*

* Check if alive
* If flew off screen, set state to dead
* Check if NachenBlaster collided with Goodie
  + If so, call whenCollided(0 and increase player HP by 10
* Move

***class TorpedoGoodie : public Goodies***

***public:***

*TorpedoGoodie:: TorpedoGoodie (StudentWorld\* world, double x, double y);*

* Initializes Goodies base class

*TorpedoGoodie::~ TorpedoGoodie ();*

* Does nothing

*virtual void TorpedoGoodie::doSomething();*

* Check if alive
* If flew off screen, set state to dead
* Check if NachenBlaster collided with Goodie
  + If so, call whenCollided(0 and increase player torpedoes by 5
* Move

**A list of all functionality that you failed to finish as well as known bugs in your classes.**

I managed to finish everything in my classes. Sometimes my Aliens shoot out multiple turnips/torpedos at once, instead of shooting just once. My probability for spawning Aliens seem to be a bit off as well.

**A description of how you tested each of your classes.**

I used the debugger in XCode a lot to check my Actor vector to check what was being pushed into my vector. I also used the cerr function to print out things like, how many aliens I destroyed in game (to make sure things were being properly counted).

For the probability based events, I rigged the probability just so I could test whether or not things were even working. For example, I had a problem with one of my Aliens shooting turnips more often than they should, so I fixed the probability to a constant number to figure out whether my function was flawed, or if I had to change my probability calculation. I also rigged the probability of dropping goodies so I could test the functionality of goodies themselves.

I also force spawned Actors in my init() function so I could test them individually.

To test my Alien collision, I force spawned a Smallgon, Snagglegon and Smoregon one above the other and made them stationary. I tested both my cabbages and torpedoes with this method. It was a lot easier to test collision this way when they weren’t moving.