**Project 4**

**1. A high-level description of each of your public member functions 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. For example, “I chose to define a pure virtual version of the blah() function in my base class because all Actors in TunnelMan must have a blah function, and each type of actor defines their own special version of it.”**

| **Actor Class** | |
| --- | --- |
| **Function** | **Description** |
| virtual ~Actor() { } //destructors always virtual | * Destructors are always virtual |
| virtual void **doSomething**() = 0; | * all actors doSomething each tick. * Abstract base class → only subclasses will be instantiated. It is a **pure virtual function:** Each actor has a doSomething function, so they will all be defined separately in their own functions. |
| void **squirtEnemy**(Direction dir, int x, int y); | * fire water at enemy protesters |
| StudentWorld\* **getStudentWorld**() { return m\_studentWorld; } | * Returns pointer to current StudentWorld. Derived classes can access the current StudentWorld() class and functions through this call. This includes TunnelMan, and his coordinates. |
| bool **isLiving**() { return m\_isLiving; } | * All actors are either living or nonliving. For example, the Earth and goodies |
| void **completelyAnnoyed**() { //actor is dead  m\_isLiving = false;  setVisible(false); } | * Dead actors no longer appear. |
| double **distanceAway**(double x1, double y1, double x2, double y2); | * Helper function. All derived functions can access this function (declared in both actor.cpp and WtudentWorld.cpp). * Returns a double of the distance away two objects are from one another. |
| **TunnelMan Class** | |
| **Function** | **Description** |
| virtual ~**TunnelMan**() { } //virtual destructor | * Destructors are always virtual. |
| virtual void **doSomething**(); | * doSomething is virtual because it is redefined for all actors. TunnelMan’s doSomething is different from Earth’s doSomething(). |
| virtual void **annoyed**(int cnt) {  m\_hp-= cnt;  } | * **annoyed**(int cnt) is **NOT** a **virtual function** because most actors never get annoyed (for example, Earth cannot be annoyed!) |
| int **getHP**() {  return m\_hp \* 10;  }    int **getWater**() {  return m\_water;  }    int **getSonar**() {  return m\_sonar;  }    int **getGold**() {  return m\_gold;  } | * These four functions both private return member variables in tunnelman. |
| void **gotGold**() { m\_gold++; }  void **gotWater**() { m\_water++; }  void **gotSonar**() { m\_sonar++; } | * Created so setGameStatText can access tunnelMan’s stats. * These functions modify TunnelMan’s stats when he collects Gold Nuggets, Water, or Sonar Kits. Not virtual because they are never overwritten. Necessary because they modify private member variables (cannot be accessed outside of TunnelMan class by other functions). |
| **Earth Class** | |
| **Function** | **Description** |
| Earth(StudentWorld \*world, int x, int y)  : Actor(world, TID\_EARTH, x, y, right, 0.25, 3) {  //depth 3 (meaning it should always be in the background), size 0.25 (p. 30)  } | * Take in parameters int x and int y, to create Earth at coordinates (x, y). Created using a 2D vector of Earth blocks.   + 2 for-loops to construct Earth, row by row |
| virtual ~**Earth**() { } //destructor | * Destructors always virtual |
| virtual void **doSomething**(){ } | * Does not do anything. |
| virtual void **annoyed**(int cnt) {} | * does not lose lives when annoyed. |
| **Boulder Class** | |
| **Function** | **Description** |
| Boulder(StudentWorld\* world, int x, int y)  : Actor(world, TID\_BOULDER, x, y, down, 1.0, 1) {  //depth 1 (appear in above earth), size 1.0  //m\_state = stable;  setVisible(true);  } | * Private member variables m\_state are initialized as stable * SetVisible to true |
| virtual ~**Boulder**() { } //destructor | * Destructors always virtual |
| virtual void **doSomething**(); | * Virtual, because it overwrites other doSomething() functions for boulder specifically |
| virtual void **annoyed**(int cnt) { }  //boulders cannot be annoyed! | * Boulders cannot be annoyed! |
| int **getTicks**() {  return m\_waitingTicks;  } | * getTicks returns private member variable m\_waitingTicks. m\_waitingTicks cannot be accessed by other functions outside of Boulder class. |
| void **resetTicks**() {  m\_waitingTicks = 0; //resets to 0 ticks  }  void **addTick**() {  m\_waitingTicks++;  } | * resetTicks and addTick modify boulder’s ticks during gameplay. Not virtual because they are never overwritten. Necessary because they modify private member variables (cannot be accessed outside of Boulder class by other functions). |
| // functions to set states of boulder  void **stableBoulder**() { m\_state = stable; }  void **fallingBoulder**() { m\_state = falling; }  void **waitingBoulder**() { m\_state = waiting; } | * These three functions all modify Boulder’s private member variable m\_state in enum State. |
| **Squirt Class** | |
| **Function** | **Description** |
| Squirt(StudentWorld\* world, int x, int y, Direction dir)  : Actor(world, TID\_WATER\_SPURT, x, y, dir, 1.0, 1), m\_travel(4) {  //page 32 manual: start at x, y; size 1.0 and depth 1; initial travel distance of 4 squares  setVisible(true);  } | * Constructor takes parameters from TunnelMan * Initiates private member variables |
| virtual ~**Squirt**() { } | * Destructors always virtual |
| virtual void **doSomething**();  //squirt can only move if space permits (check direction)  void **moveSquirt**(GraphObject::Direction dir); | * Called when the user presses space. * Squirt image appears on screen, moves 4 units in direction that TunnelMan is facing, space permitting * moveSquirt is not virtual because it is never overwritten |
| **Protester Class** | |
| **Function** | **Description** |
| Protester(StudentWorld\* world, int imageID); | * Constructs protester |
| virtual ~**Protester**() { } | * Destructors are always virtual |
| virtual bool **hardMode**() = 0; | * Virtual; all |
| virtual void **hasGold**() = 0; | * Pure virtual → If protester has gold, they are in a “bribed” state   + Abstract base class |
| void **doSomething**(); | * Overwritten doSomething function (not finished) |
| virtual void **annoyed**(int n) = 0; | * Hit by n points to health points |
| void **setProtesterHP**(int n) { m\_protesterHP = n; } //set to different  int getProtesterHP() {  return m\_protesterHP \* 10; } | * Modifies health points (private member variable |
| bool **OOB**() { return m\_OOB; } //if protestor is out of bounds of oil field | * Returns whether protester has left the field |
| **Regular Protester Class** | |
| **Function** | **Description** |
| RegularProtester(StudentWorld\* world)  : Protester(world, TID\_PROTESTER) {  setProtesterHP(5);  } | * Constructs protester while initiating private member variable (m\_protesterHP) |
| virtual bool **hardMode**() { return false; } | * Virtual because hardcore protesters have their own hardMode function that behaves differently from regular protesters in hardMode |
| virtual void **hasGold**(); //protesters are different when they have gold | * I did not complete this function. |
| **Hardcore Protester Class** | |
| **Function** | **Description** |
| HardcoreProtester(StudentWorld\* world)  : Protester(world, TID\_HARD\_CORE\_PROTESTER) {  setProtesterHP(20);  } | * Constructs protester while initiating private member variable (m\_protesterHP) |
| virtual ~**HardcoreProtester**() { } | * Destructors are always virtual |
| virtual bool **hardMode**() { return true; } | * Virtual because hardcore protesters act ddifferently when within 4 blocks of TunnelMan than regular protesters do. |
| virtual void **hasGold**(); | * Virtual. Bribed hardcore protesters act differently from bribed regular protesters with gold. |
| **Goodie Class** | |
| **Function** | **Description** |
| Goodie(StudentWorld\* world, int imageID, int x, int y, Direction dir, unsigned int depth, bool visible, bool TMcanPick, bool pcanPick, bool perma)  : Actor(world, imageID, x, y, dir, 1.0, depth), m\_TMcanPick(TMcanPick), m\_pcanPick(pcanPick), m\_perma(perma) {  setVisible(visible); //different goodies appear and disappear at different times  //all goodies have size 1  } | Initiating private member variables that define:   * Whether TunnelMan can pick object up * Whether Protesters can pick object up * Whether object is permanent * Whether object is visible |
| virtual ~**Goodie**() { } // destructor | * Destructors are always virtual |
| virtual void **doSomething**() = 0; | * Abstract base class. Pure Virtual because goodie object will never be constructed (only derived members are constructed) |
| bool **TMcanPick**() {  return m\_TMcanPick;  }  bool **pcanPick**() {  return m\_pcanPick;  }  bool **isPerma**() {  return m\_perma;  }  void **setPerma**() {  m\_perma = true;  }  void **setNotPerma**() {  m\_perma = false;  } | * Accessor and mutator functions * Returns and modifies private member variables, respectively |
| **Barrel Class** | |
| **Function** | **Description** |
| Barrel(StudentWorld\* world, int x, int y)  : Goodie (world, TID\_BARREL, x, y, right, 2, false, true, false, true){    } | * Constructs barrel * //not visible until shown, TMcanpick, Pcannotpick, perma. |
| virtual ~**Barrel**() {} //virtual destructor | * Destructors are always virtual |
| virtual void **doSomething**(); | * Virtual. Behaves differently from all other actors. Increases score of TunnelMan when oil barrels are picked up. |
| virtual void **annoyed**(int cnt) { } | * Barrels cannot be annoyed |
| **Gold Nugget Class** | |
| Function | Description |
| GoldNugget(StudentWorld\* world, int x, int y, bool visible, bool TMcanPick, bool pcanPick, bool perma)  : Goodie(world, TID\_GOLD, x, y, right, 2, visible, TMcanPick, pcanPick, perma), m\_tick(0) {  //can be picked up by protesters for bribing. start out invisible or visible  //p. 34 -- depth 2, size 1.0  } | * Given parameters x, y, creates gold nugget pool at coordinates (x,y) * Initiates private member variables m\_tick that count down from specified lifetime number (not permanent) |
| **Sonar Kit Class** | |
| **Function** | **Description** |
| SonarKit::SonarKit(StudentWorld\* world, int x, int y)  : Goodie(world, TID\_SONAR, x, y, right, 2, true, true, false, false), m\_tick(0) {  //faces right, visible, only pickupable by TM, temporary  setVisible(true);  m\_tick = std::max(100, 300 - 10 \* (int)getStudentWorld()->getLevel());  } | * Given parameters x, y, creates sonar kit at coordinates (x,y) * Initiates private member variables m\_tick that count down from specified lifetime number (not permanent) |
| virtual ~**SonarKit**() { } | * Destructors are always virtual |
| virtual void **doSomething**(); | * Plays sound when picked up * Reveals whether gold is within close proximity to TunnelMan. If true, TunnelMan gains points. If not picked up, the item should disappear. |
| virtual void **annoyed**(int cnt) { } | * Sonar kits cannot be annoyed |
| **Water Class** | |
| **Function** | **Description** |
| Water::Water(StudentWorld\* world, int x, int y)  : Goodie(world, TID\_WATER\_POOL, x, y, right, 2, true, true, false, false) {  //facing right, start visible, only pickupable by TM, start temporary  m\_tick = std::max(100, 300 - 10 \* (int)getStudentWorld()->getLevel());  } | * Given parameters x, y, creates water pool at coordinates (x,y) * Initiates private member variables m\_tick that count down from specified lifetime number (not permanent) |
| virtual ~**Water**() { } | * Destructors are always virtual |
| virtual void **doSomething**(); | * If water is within radius of TunnelMan, it activates and can be picked up. This gives players more points, and increases m\_water to be squirted at protesters. If not picked up after a certain number of ticks, the water should disappear. |
| virtual void **annoyed**(int cnt) { } | * Water cannot be annoyed |

| **StudentWorld Class** | |
| --- | --- |
| **Function** | **Description** |
| StudentWorld(std::string assetDir)  : GameWorld(assetDir), m\_tunnelMan(NULL), m\_actors(NULL), m\_earth(NULL), m\_nBarrel(0), m\_tick(0)  {  //nBarrel is number of barrels, number of ticks  } | * Constructor that initializes actors to null * Initializes ticks to 0 * This class contains the auxiliary functions from other classes that are either general to multiple classes or are implemented with variables or data structures that are private or otherwise difficult to access |
| virtual int **init**(); | Initializes every |
| virtual int **move**(); | All actors are called to doSomething in this function. |
| virtual void **cleanUp**(); | * Deletes all actors using a vector list * Deletes all pieces of Earth in a loop * Virtual because earth was implemented using private variables |
| virtual ~**StudentWorld**() {  cleanUp();  } | * Destructors always virtual |
| void **removeDeadGameObjects**(); | * Deletes all actors that are dead from screen in following tick |
| void **setDisplayText**();  std::string **formatText**(int level, int lives, int health, int squirts, int gold, int barrelsLeft, int sonar, int score); | * Sets display text. Accesses private member variables (e.g. getLevel(), m\_lives, m\_health, etc.) |
| TunnelMan\* **getTunnelMan**() {  return m\_tunnelMan;  } | * Returns pointer to tunnelman. * Access StudentWorld’s private member variable; cannot be accessed outside this class |
| void **newCharacter**(Actor\* actor) {  m\_actors.push\_back(actor);  } | * New actors are initialized and pushed back onto vector list m\_actors. |
| void **addGoodie**(); | * addsGoodies to the map. Each tick, 1/Gth chance of adding a goodie (Sonar Kit or Water Pool) |
| bool **earthVisible**(int x, int y); | * Returns whether coordinate (x,y) contains earth that has not been dug. |
| Earth\* **getEarth**(int x, int y) { if (y >= 0 && y < 60 && x >= 0 && x < 64) {  return m\_earth[x][y];  }  else {  return NULL;  }  } | * Returns block of earth object at coordinate (x, y) * //get specific block of earth (for when tunnelman digs earth) * Written in StudentWorld because they need to access private member variables in studentWorld (m\_earth) |
| bool **digEarth**(int x, int y); | * Deletes earth at coordinates (x, y) * Accesses private member variables in studentWorld (m\_earth) |
| void **barrelPicked**() {  m\_nBarrel--;  } | * indicates barrel has been picked up by TM * Accesses private member variable in studentWorld (m\_nBarrel) |
| bool **inMap**(double x, double y); //if coordinate is within the map bool | * Returns whether coordinate (x, y) is within map borders AND is not a boulder. * Accesses private member variable in studentWorld (m\_earth) |
| bool **canMove**(GraphObject::Direction dir, int x, int y); | * Returns if object can move in direction at coordinate (x, y) * Accesses directions in graphObject (can only be accessed through StudentWorld) |
| void **randomPos**(int& x, int& y);  void **randomPosNoEarth**(int& x, int& y); | * Passes by random position either in or outside of earth (as name suggests) * Accesses private member variable in studentWorld (m\_earth) |
| double **distanceAway**(double x1, double y1, double x2, double y2); //  bool **withinRadiusofProtester**(int x, int y, int hpDamage);  Protester\* **closestProtestor**(int x, int y); | * calculate distance between two objects * //whether protester was in radius of 3, and was damaged (returns true if protester is hit) * Returns pointer to closest protester to coordinates * Accesses private member variable in studentWorld |

**2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. “I wasn’t able to implement the Squirt class.” or “My Hardcore Protester doesn’t work correctly yet so I just treat it like a Regular Protester right now.”**

I was unable to finish the protestor functions.

They are defined in the actor.h files, and given properties (annoyed, constructors, etc.), but do not have doSomething() functions defined and do not show up on screen.

Protesters cannot move on their own, and do not currently have algorithms that define how they search for TunnelMan. Protesters cannot give up.

When boulders fall, there have been cases where TunnelMan loses a life, even though he is not under the boulder when it falls.

I have also not implemented TunnelMan’s ability to use the Sonar Kit. Pressing tab results in a break of the switch (does nothing).

When TunnelMan completes a level, the level does not increment right now. I treat it currently like the default level (when the player completes the level, the level restarts).

**3. A list of other design decisions and assumptions you made, e.g.: i. It was ambiguous what to do in situation X, and this is what I decided to do.**

It was ambiguous how often to create Sonar Kits and Water Pools. I assumed that the probability of sonar kits and water pools being created were dependent. This means that 1/5th of the time a goodie was created, Sonar Kits were created, while the remaining times a goodie was created, it was a Water Pool.

There were no other assumptions made.

**Design Choices**

* I used an enum to list out the different states of Boulders. This seemed like a more efficient way to change states of the boulders, and also less mistake-prone because I wouldn’t have to set the boolean functions of all three states separately.
* Initially, I checked whether TunnelMan overlapped with earth in the move() function in StudentWorld.cpp. If this was true, I would destroy the earth pieces that TunnelMan was standing over. I later moved this to a separate function, so that I could clean up the move() function and also reuse this piece of code easily in other functions that needed to call it.

**4. A description of how you tested each of your classes (1-2 paragraphs per class)**

**TunnelMan**

Movement

I tested movement by pressing arrow keys and WASD on my keyboard, and observing how the character changed positions around the screen. If the character changed direction according to the corresponding arrow key/WASD key, I concluded that the movement was implemented correctly.

Illegal Bounds

I tested illegal bounds by walking along the borders of the map and observing that TunnelMan could still be seen. I also tried to escape the boundaries of the map by pressing arrow keys until I was at the edges, and observed that TunnelMan’s avatar did not leave the map.

Interactions with objects

I tested object interaction for each type of object separately:

* For boulders, I moved TunnelMan underneath them to observe that his movement and his standing underneath them would trigger the falling boulders
* For oil barrels, I moved TunnelMan over the object so that he would pick it up. I encountered some issues with this because I accidentally redefined a setVisibile() function that never set m\_barrels to decrement. This resulted in oils never being picked up. I debugged this by using cout to print/output locations of where oil barrels were created so that I could quickly test and check that this capability was correctly implemented. In my final submitted version, TunnelMan is able to pick up oil barrels
* For goodies (Water Pools, Sonar Kits), I tested them in a similar way as oil barrels by moving TunnelMan over the location of the object. I also used a similar strategy as testing oil barrels (cout and print) to quickly test that this was implemented correctly.
* I tested water squirts by triggering them with the space bar and observing that water was displayed in each of the directions TunnelMan was facing. (e.g. water squirted north when TunnelMan faced north, etc.)
* I tested Game functions (escape key pressed by user) by pressing the escape key and observing that the right actions were performed (e.g. restart level, lose a life, etc.)

**Earth**

I tested TunnelMan’s digging functionality by digging in a given direction (moving in a direction by pressing the corresponding arrow key) and observing that earth pieces are destroyed and created in the expected locations.

I also tested that Earth pieces were destroyed and recreated properly after pressing ESCAPE to restart the level (since pressing ESCAPE calls cleanUp()). Before finding my errors in cleanUp(), restarting the level resulted in TunnelMan being able to walk through Earth.

**Boulder**

I tested the implementation of boulders by ensuring that they would fall as expected when TunnelMan moved underneath them. To do this, I navigated TunnelMan underneath boulders and observed that they would fall onto him. Then, I moved TunnelMan under a boulder and moved him away quickly to observe that they would not fall onto TunnelMan in order to test that the delay was implemented correctly.

I tested that TunnelMan could not walk over boulders by trying to navigate him to walk over the boulders and observing that he was blocked by them.

Finally, I tested that pressing escape to restart the level correctly called cleanUp() so that existing boulders on that level would be deleted and then recreated by observing the output of my test script (cerr to display the location and number of boulders on a level).

**Squirt**

I tested the implementation of Squirt by observing that water squirted as expected in all directions, when TunnelMan was facing the corresponding direction (e.g water squirts north when TunnelMan faces north). Initially, my code had a bug where water would not squirt in the North direction due to a typo in my for-loop. This has been fixed in the submitted version.

Additionally I tested that water squirts would not be able to pass through earth or boulder blocks, but would be able to be shot over other goodies. I tested this by positioning TunnelMan close to earth/boulder blocks and observing that these blocks were resistant to water. Similarly, I tested the squirts’ interaction with other goodies by moving TunnelMan close to those goodies and observing that water could be shot over them.

**Barrel**

I tested the implementation of barrels by using cerr to output the locations of both the barrels and of TunnelMan, which allowed me to navigate TunnelMan close to the barrels and observe that they would correctly change their status to “visible” when TunnelMan was nearby. I also printed the number of barrels that were generated on each level (using the formula provided in the spec document) to verify that the correct number of barrels were being created.

Finally, I moved TunnelMan over the barrels to verify that the right sound would be played when TunnelMan picked up the barrel. This was challenging because my initial implementation had a typo in a member function name (my submitted version of the code performs as expected).

**Gold Nugget**

I tested the implementation of gold nuggets by observing that they would appear (i.e. change their status to visible) when TunnelMan’s avatar was close to their generated location. I did this by walking TunnelMan over known gold nugget locations (I outputted their coordinates using a similar cout script as I used to test oil barrels above) and saw that the icons appeared on screen.

Then, I tested that TunnelMan was able to pick up gold nuggets by walking over visible nuggets and observing that the status line updated correctly. Simultaneously, I listened to confirm that a sound was played when TunnelMan picked up his gold nugget.

**Sonar Kit**

I tested the implementation of Sonar Kits by checking that they were first created only a fraction of the time (1/5th probability when goodies appear). I did this visually, by observing that Sonar Kits appeared less often than Water Pools (See below).

I confirmed that Sonar Kits would always appear at the top left corner of the map by outputting their coordinates using the cout/print function in my debugging window. I then moved TunnelMan to the Sonar Kits, and observed that he was able to pick up sonar kits. I checked to ensure that there would be a corresponding sound effect for when he picked up a Sonar Kit. Finally, I observed that the status line would update as expected after TunnelMan picked up a Sonar Kit.

**Water Pool**

Similar to how I tested Sonar Kits, I checked that Water Pools are created a fraction of the time (4/5th probability when goodies appear).

Visually, I checked that Water Pools appeared more often than Sonar Kits. I also confirmed that Water Pools would always appear in spaces without Earth by outputting the coordinates of where they were created.

Then, I walked over visible Water Pools with TunnelMan, and confirmed that they were able to be picked up. When this happened, I made sure a sound would play by listening to the audio. Finally, I checked that the status line would update with more water.