NOTE: IF THE DESCRIPTIONS OF THE IMPLEMENTATIONS OF THE PUBLIC FUNCTIONS HERE ARE UNREADABLE FOR WHATEVER REASON, I HAVE COMMENTED MY MEMBER FUNCTIONS IN THE CPP FILES AS WELL

class StudentWorld : public GameWorld

{

public:

StudentWorld();

~StudentWorld();

virtual int init();

virtual int move();

virtual void cleanUp();

Player\* getUser() const;

bool hasBrick(int x, int y);

bool hasPermaBrick(int x, int y);

bool hasDestructableBrick(int x, int y);

void setDisplayText();

Level getLevelObject() const;

int Zumis() const;

void decreaseZumis();

void insert(Object\* actor);

bool isBrickDestroyed(int x, int y);

bool hasBugSprayer(int x, int y);

bool searchAndDestroyZumi(int x, int y);

bool searchAndDestroyBrick(int x, int y);

void insertDeadBricks(int x, int y);

void exterminate(int x, int y);

void setDone();

bool searchMaze(Zumi\*, int x, int y, int direction);

void executeSearch(Zumi\* bug);

bool contains(int x, int y);

}

StudentWorld::StudentWorld() initializes all member variables of my StudentWorld class. I needed this function to make sure none of my member variables initially pointed to garbage values or invalid memory. This is a constructor, so it has to be in StudentWorld

void StudentWorld::insertDeadBricks(int x, int y)

{

For the list of Coordinates of deadBricks

If x and y are not coordinates in the list

Insert Coordinates of (x,y) into the list

}

This function stores the Coordinates (a class I created in Actor.cpp) which holds the X and Y values of Bricks which have been destroyed. I need this function to store the location of destroyable bricks which have been destroyed, so I can notify the actors that they can move onto the square which was previously blocked. I put this in the student world class because it was not related to any of the actor classes

void StudentWorld::insert(Object\* actor) Push backs a new actor into the list of actors. I need this function because the list of actors is contained in the StudentWorld class, but is private, and the actor class needs to insert some items. Therefore, it also has to be in the studentWorld class, to get access to the private list

void StudentWorld::setDone() Set member variable finished to true. I need this function because my Exit::doSomething needs to indicate that the level has finished, which it does by calling this function. However, the member variable m\_finished is a private member of the StudentWorld class.

virtual int init()

{

Initializes all member variables

Loads level data

If level files are unable to load, print out appropriate error message and terminate program

Take contents of level file and print them to the screen

Insert actors into list of actors

Continue the game

} This function was required by the spec. It loads the level. I need it to display the graphics to the screen and notify

~StudentWorld::StudentWorld() removes all dynamically allocated data, and clears all STL’s. This function was required by the spec. It is needed to remove all dynamically allocated data in the actors list that may be left if the player has won the game, or the player has lost the game

void StudentWorld::cleanUp() Deletes the dead dynamically allocated data and clears all STL’s. This function was required by the spec. It is needed to remove all dynamically allocated data when a player loses a life.

bool StudentWorld::hasBrick(int x, int y) If the coordinate has a destroyable brick or permanent brick, return true. It is in the StudentWorld class because it is more related to the environment of the level. It is needed to help tell if the actors can move to a specific player.

bool StudentWorld::hasDestructableBrick(int x, int y) If the coordinate has a destroyable brick return true. It is in the StudentWorld class because it is more related to the environment of the level. It is needed to be a more specific indicator of whether or not an actor can move to a specific location.

bool StudentWorld::hasPermaBrick(int x, int y) If the coordinate has a permanent brick return true. It is in the StudentWorld class because it is more related to the environment of the level. It is needed to be a more specific indicator of whether or not an actor can move to a specific location.

bool StudentWorld::searchMaze(Zumi\* bug, int x, int y, int direction)

{

Clear the list of visited coordinates and queue of coordinates

Push the current coordinates (x, y) onto the queue, store these values and the direction into local variables (xc, yc, dir)

While queue is not empty

{

If the coordinates are the same as the player’s coordinates, return true

Pop the front off the queue

If the coordinates are not visited and can be reached by the zumi

If this is the first step, indicate the direction as up

Else indicate the direction as the previous direction

Insert the coordinates into the list of visited coordinates

If the coordinates are not visited and can be reached by the zumi

If this is the first step

indicate the direction as up

Else

indicate the direction as the previous direction

Insert the coordinates into the list of visited coordinates

If the coordinates are not visited and can be reached by the zumi

If this is the first step

indicate the direction as down

Else

indicate the direction as the previous direction

Insert the coordinates into the list of visited coordinates

If the coordinates are not visited and can be reached by the zumi

If this is the first step

indicate the direction as left

Else

indicate the direction as the previous direction

Insert the coordinates into the list of visited coordinates

}

Return false

}

This function is needed to search the maze and see if a complex Zumi can reach a player. If he can, he takes the shortest route. This is put in the StudentWorld class because it needs access to the queues and lists which hold the coordinates of locations, which are private variables.

bool StudentWorld::contains(int x, int y)

{

For the list of coordinates

If the coordinates match x and y, return true

} This function is needed to search the list of Coordinates already visited, to make sure the searchMaze function does not run infinitely. It is put in the StudentWorld class because it is a helper function to searchMaze.

void StudentWorld::executeSearch(Zumi\* bug)

{

If the direction in the queue is down, tell bug to move down

If the direction in the queue is up, tell bug to move up

If the direction in the queue is right, tell bug to move right

If the direction in the queue is left, tell bug to move left

Pop the top of the queue

} This function is needed to execute the move found in searchMaze. It is put in studentWorld because it needs access to the queue, which is a private member.

void StudentWorld::setDisplayText()

{

Store values of score, level, lives, and bonus into stringstream

Set a string into this stringstream

Use this string as a parameter to set the game stat text

Decrement the value of the bonus

} This function is needed to set the string which is called by the setGameStatText function. It is in the StudentWorld class because it is related to the environment.

Level StudentWorld::getLevelObject() const returns the level variable “l”. This function is needed because it is more convenient to use this in the actor class to get the level options (such as the probability of a goodie, or the number of ticks needed for a Zumi to move), rather than create functions for all these different variables in the StudentWorld class. Putting it in the StudentWorld class is necessary because l is a private member variable of the StudentWorld class.

Player\* StudentWorld::getUser() const returns a pointer to the player. It is needed because some actors need to know the player’s location, and the player’s pointer is a private member variable of the studentworld class.

int StudentWorld::Zumis() const Returns the number of Zumi’s. It is in the StudentWorld class because it accesses a private member variable in the Studentworld class, but needs to be accessed by some of the actors

void StudentWorld::exterminate(int x, int y)

{

For the list of actors if the coordinates are the same as the bugspray

{

If the actor is a destroyable brick that hasn’t been destroyed, destroy it

If the actor is an alive zumi, kill it

If the actor is an alive bugsprayer, kill it

}

} This is needed to let the bugspray know what to kill. It is in the studentWorld function because it needs access to the list of actors, which is private

bool StudentWorld::searchAndDestroyZumi(int x, int y)

{

For the list of actors

If the actor is a zumi

If the coordinates of the zumi are the same as the coordinates of the player

Return true;

} This function is needed to alert the player whether or not it is killed by a zumi. This Is in the studentWorld class because it needs to access the list of actors, which is private

bool StudentWorld::searchAndDestroyBrick(int x, int y)

{

For the list of actors

If the actor is a Destroyable brick

If the brick is not destroyed

If the bricks coordinates are the same as the coordinates of the player

Return true

} This function is needed to alert the player whether or not it is in the same space as a destroyable brick. Is in the studentWorld class because it needs to access the list of actors, which is private

void StudentWorld::decreaseZumis() decreases the number of zumis by 1. This function is needed because the actors need to decrease the Zumis sometimes. Is in the studentWorld class because it needs to access a private member variable

bool StudentWorld::isBrickDestroyed(int x, int y)

{

For the list of Coordinates of dead bricks

If the coordinates of this brick are the same as any of the coordinates in the list

Return true

} This function is needed to check if a destroyable brick has been destroyed, and thus alert actors if they can move onto that space. It is in the studentWorld class because it needs to access a private member variable

bool StudentWorld::hasBugSprayer(int x, int y)

{

For the list of actors

If the actor is a bugsprayer

If the bugsprayer’s coordinates are the same as the player’s

Return true;

} It is in the studentWorld class because it needs to access the list of actors, which is private. I need this function to make sure that the player is alerted when he walks onto bugspray

int StudentWorld::move()

{

Set the Display text above the arena

Get the player to move

If the game is finished

Increase the score by the appropriate amount

Return level finished

For the list of actors

If the actor is alive, move

If the player is dead

Return the player died

For the list of actors

If the actor is dead, deallocate its memory

Continue the game

} This function was required by the spec. I needed it to move all the actors, and deallocate the memory for all the actors who have died

class Object: public GraphObject

{

public:

Object(int id, int x, int y, StudentWorld\* s);

virtual ~Object();

virtual void doSomething() = 0; I set it purely virtual because every object has a different purpose and method to moving. I put it in this class because every object (except for permanent Bricks) do something interesting. \*\*\*ALL CLASSES WILL HAVE THIS FUNCTION NOW IMPLEMENTED BECAUSE NON-ABC’S HAVE TO DEFINE THIS FUNCTION

StudentWorld\* getWorld(); I put in this class because many derived classes need access to Studentworld at some point

virtual void kill(); I put this in the Object class because every object, save for permanent bricks, may be killed through the course of the game.

bool isDead() const; . I put this in this class because every object, save for permanent bricks, will need to determine whether or not they are dead

};

Object::Object(int id, int x, int y, StudentWorld\* s)

{

Initializes GraphObject with id, and coordinates x and y

Sets all objects to visible

Sets the studentworld pointer to s

Sets all objects life status to dead

}

Object::~Object() does nothing. It is virtual because it is a destructor.

StudentWorld\* Object::getWorld() returns the StudentWorld private member pointer s.

void Object::kill() Sets variable m\_dead to true.

bool Object::isDead() const returns variable m\_dead value

class Brick : public Object

{

public:

Brick(int id, int x, int y, StudentWorld\* s);

virtual void doSomething();

};

Brick::Brick(int id, int x, int y, StudentWorld\* s) Initializes Object class

void Brick::doSomething() does nothing

class destroyableBrick: public Brick

{

public:

destroyableBrick(int x, int y, StudentWorld\* s);

virtual void kill(); This function is in this class because it does dies (unlike a permanent brick or other objects, so it needs its own virtual function)

};

destroyableBrick::destroyableBrick(int x, int y, StudentWorld\* s)

:Brick(IID\_DESTROYABLE\_BRICK, x, y, s) Initializes Brick class

void destroyableBrick::kill()

Inserts coordinates of new dead brick into list of coordinates in studentWorld, sets this brick to dead

class Exit : public Object

{

public:

Exit(int x, int y, StudentWorld\* s);

virtual void doSomething();

};

Exit::Exit(int x, int y, StudentWorld\* s)

:Object(IID\_EXIT, x, y, s)

Initializes Object class, sets visible to false, initializes member variable of visibility to false

void Exit::doSomething()

{

If number of zumis <= 0 and the exit is visible

play the sound that reveals the exit

set member variable of visibility to true

If coordinates of this and player are the same

play the sound that says the player finished the level

indicate to the Studentworld that the level is finished

}

class BugSprayer : public Object

{

public:

BugSprayer(int x, int y, StudentWorld\* s);

virtual void doSomething();

virtual void kill(); This function is here because it does not die like all other objects, so it needs to have its own virtual function

private:

int m\_lifetime;

};

BugSprayer::BugSprayer(int x, int y, StudentWorld\* s)

:Object(IID\_BUGSPRAYER, x, y, s)

Initialize the Object class, and set the member variable about its life to 40 ticks

void BugSprayer::doSomething()

{

If its dead, return

Decrement its lifetime by 1

If its lifetime is <= 0, kill it

}

void BugSprayer::kill()

{

Insert a new bugspray item into this space

For all 4 directions for up to two spaces

If the map has a permanent brick on this space, break out of loop

Else if the map has a destructible brick that hasn’t been destroyed, place a new bugspray onto space, break out of loop

Else insert bugspray onto space, go to next space

Play sound that the bugspray has exploded

Set lifetime to 0, kill it, decrement sprayers by 1

}

class Zumi : public Object

{

public:

Zumi(int id, int x, int y, StudentWorld\* s, int ticksPerMove);

virtual void doSomething();

int randInt(int lowest, int highest); this function was defined in this class because only zumi’s have use for finding a random int

virtual void kill(); This function was defined in this class because Zumi’s die in different ways than other objects

bool canZumiMove(int x, int y); This function was defined in this class because Zumi’s can move differently than other objects

bool turnToMove(); This function was defined in this class because only Zumi’s can move on certain turns, and both Complex Zumi’s and Simple Zumi’s behave in this way

void simpleMove(); This function was defined in this class because only Zumi’s move in this pattern, and both Complex Zumi’s and Simple Zumi’s can exhibit this behaviour

void setDirection(int direction); This function was defined here because it needs access to m\_direction

private:

int m\_ticksPerMove;

int m\_direction;

int m\_ticks;

};

Zumi::Zumi(int id, int x, int y, StudentWorld\* s, int ticksvalue)

:Object(id, x, y, s)

Initialize object class, set ticks per Zumi move to ticksvalue, set initial direction to a random direction, set number of ticks passed to 0

bool Zumi::canZumiMove(int x, int y)

{

If (the x, y coordinate did not a brick initially OR the x, y coordinate had a destructible brick that was destroyed) AND the x, y coordinate doesn’t have a bugsprayer

Return true

else

return false

}

bool Zumi::turnToMove() If the number of ticks is not 0 and is divisible by the number of ticks per zumi move ,return true

otherwise return false

Either way, increment the number of ticks

void Zumi::simpleMove()

{

If the zumi can move in the current direction (using the canZumiMove function)

Move it

Otherwise, set ticks to 0 and pick a new direction for it to move in

}

void Zumi::doSomething()

{

If its dead, return

If its coordinates are the same as the player’s, kill the player

If it’s the Zumi’s turn to move (via the turnToMove function)

Move the zumi (via the simpleMove function)

}

void Zumi::kill()

{

Decrease the number of zumis by 1, play the sound indicating a zumi has died, and increase the score by 100

Generate a random number between 0 and 99

Get probability that the Zumi drops a goodie via getOptionValue function

If probability greater than random number

{

Generate a new random number

Create a number line from 0 to 99, where each number is assigned to a WalkThroughWalls goodie, extraLife goodie, or MoreSprayers goodie. Whichever Goodie the random number is associated with, that Goodie is created

}

Set the zumi to dead

}

int Zumi::randInt(int lowest, int highest)

Generates a random number between the lowest and the highest, inclusive

void Zumi::setDirection(int direction)

If direction is valid, set the Zumi’s current direction to that

Class ComplexZumi: public Zumi

{

public:

ComplexZumi(int id, int x, int y, StudentWorld\* sw, int ticksPerMove);

virtual void doSomething(); This function was defined in this class because it moves differently than regular Zumi’s

virtual void kill(); This function was defined in this class because it needs 500 extra points when it dies, as opposed to 100 for regular zumis

};

ComplexZumi::ComplexZumi(int id, int x, int y, StudentWorld\* s, int ticksPerMove)

:Zumi(id, x, y, s, ticksPerMove) Initializes the Zumi class

void ComplexZumi::doSomething()

{

If its dead, return

If its coordinates are the same as the player’s coordinates, kill the player

If it’s the zumi’s turn to move (via the turnToMove function)

{

If the smell distance (found via the getLevelOption function) is at least equal to both the vertical distance AND the horizontal distance between the player

{

If there is a path between the zumi and the bug (determined using the searchMaze function)

Move the zumi one step in that shortest path (using the executeSearch function)

Otherwise, move like a simple zumi

}

Else, move like a simple zumi

}

}

void ComplexZumi::kill()

Increase the score by 400, and kill it just like a simple zumi

class Player : public Object

{

Player(int x, int y, StudentWorld\* sw);

virtual void doSomething(); This function was defined in this class because it moves differently than all other objects

virtual void kill(); This function was defined in this class because it dies differently than all other objects

void setJesus(); This function is defined in this class because it is the only one that can walk through destructible bricks

void increaseSprayers(); This function is defined in this class because it is the only one that can drop sprayers

int getMaxSprayers() const; This function is defined in this class because it is the only one that can walk through destructible bricks

void decrementSprayers();This function is defined in this class because it is the only one that can walk through destructible bricks

bool canPlayerMove(int x, int y); This function is defined in this class because players have different requirements to whether or not they can move

int getSprayers() const; This function is defined in this class because it is the only one that can walk through destructible bricks

private:

bool m\_ethereal;

int max\_sprayers;

int m\_sprayers;

int m\_SprayerLife;

int m\_WallLife;

};

Player::Player(int x, int y, StudentWorld\* s)

:Object(IID\_PLAYER, x, y, s)

Initializes the object, sets the amount of sprayers to 0, player cannot walk through walls, sets max amount of sprayers to 2

bool Player::canPlayerMove(int x, int y)

{

(If the coordinate did not have a brick there OR the coordinate had a destroyable brick that has been destroyed) OR (the player can walk through destructible bricks AND this coordinate has a destructible brick)

Return true

Otherwise return false

}

void Player::doSomething()

{

If the player is dead, return

If the coordinates of this player are the same as any zumi’s (via the searchAndDestroyZumi function), kill this player

If the player can walk through destructible bricks, decrement the time it can do that by 1, and if it equals zero, the player can’t walk through them anymore

If the player can drop more than 2 sprayers, decrement the time it can do that by 1, and if it equals zero, the player can only drop 2 sprayers now

If the player can no longer walk through destructible bricks and is in one right now, kill the player

Take the user keyboard input and move in that direction

If the user presses space

{

If the sprayers on the board are less than the maximum allowed AND the space doesn’t have a bugsprayer AND (the board didn’t have a brick OR the board had a brick which has been destroyed)

Drop a bugsprayer, else, ignore

}

}

void Player::setJesus()

The player can now walk through destructible bricks, for a time determined by the getLevelOption function

void Player::decrementSprayers() If the amount of sprayers is greater than 0, decrement by 1

void Player::kill() Kill the player, and play the sound that indicates it is dead

void Player::increaseSprayers() The max amount of sprayers is determined by the getOptionValue function, for a time determined by the getOptionValue function

int Player::getMaxSprayers() const return the max number of sprayers allowed currently

int Player::getSprayers() const return the numbe of sprayers on the board currently

class BugSpray : public Object

{

public:

BugSpray(int x, int y, StudentWorld\* sw);

virtual void doSomething();

private:

int m\_ticks;

};

BugSpray::BugSpray(int x, int y, StudentWorld\* sw)

:Object(IID\_BUGSPRAY, x, y, sw) Initializes the object

void BugSpray::doSomething()

{

If its dead, return

Decrement its time to be alive by 1

If the new time is less than 0, kill it

Else

{

If its coordinates are the same as the player’s kill the player

If its coordinates are the same as a destructable brick, a Zumi, or a BugSprayer, kill any of them (using exterminate function)

}

}

class Goodie : public Object

{

public:

Goodie(int id, int x, int y, StudentWorld\* sw);

virtual void doSomething() = 0; This function is pure virtual because there is no Goodie by itself (it is an ABC)

int getLife() const; This function is defined here because all goodies need to know their life

void decrementLife(); This function is defined here because all goodies need to decrement their life

private:

int m\_lifetime;

};

Goodie::Goodie(int id, int x, int y, StudentWorld\* sw)

:Object(id, x, y, sw)

Initialize the object, and set its life on the board for the time specified by get getOptionValue function

int Goodie::getLife() const return its life left

void Goodie::decrementLife() decrement its life by 1

class ExtraLife : public Goodie

{

public:

ExtraLife(int x, int y, StudentWorld\* sw);

virtual void doSomething(); This function is defined here because it has different behaviour than the other goodies

};

ExtraLife::ExtraLife(int x, int y, StudentWorld\* sw)

:Goodie(IID\_EXTRA\_LIFE\_GOODIE, x, y, sw) Initialize the goodie object

void ExtraLife::doSomething()

{

If its dead, return

Decrement its life by 1

If its life is at most 0, kill it

Else

If its coordinates are the same as the players, increase the player’s life by 1, increase score by 1000, indicate it got a goodie

}

class WalkWalls : public Goodie

{

public:

WalkWalls(int x, int y, StudentWorld\* sw);

virtual void doSomething(); This function is defined here because it has different behaviour than the other goodies

};

WalkWalls::WalkWalls(int x, int y, StudentWorld\* sw)

:Goodie(IID\_WALK\_THRU\_GOODIE, x, y, sw) Initialize the goodie object

void WalkWalls::doSomething()

{

If its dead, return

Decrement its life by 1

If its life is at most 0, kill it

Else

If its coordinates are the same as the players, the player can now walk through destroyable bricks, increase score by 1000, indicate it got a goodie

}

class MoreSprayers : public Goodie

{

public:

MoreSprayers(int x, int y, StudentWorld\* sw);

virtual void doSomething(); This function is defined here because it has different behaviour than the other goodies

};

MoreSprayers::MoreSprayers(int x, int y, StudentWorld\* sw)

:Goodie(IID\_INCREASE\_SIMULTANEOUS\_SPRAYER\_GOODIE, x, y, sw) Initialize the goodie object

void MoreSprayers::doSomething()

{

If its dead, return

Decrement its life by 1

If its life is at most 0, kill it

Else

If its coordinates are the same as the players, the player can now drop more sprayers, increase score by 1000, indicate it got a goodie

}

2: As far as I know, my project has no bugs

3: The spec was unclear about what to put in to StudentWorld, and what to put in Actor, so I put the STL in StudentWorld, which meant I had to put a lot of my functions in StudentWorld.

Also, if a Complex Zumi is within smelling distance, and moves towards the player, but ends up out of smelling distance, the spec didn’t specify whether to update the current direction to the direction it is moving in that instance. I thought it seemed reasonable to update the current direction, since it is the same zumi, and may move in a new direction when chasing the player

4: To test my player class, I modified various maps with various types of actors, such as permanent bricks, simple zumis, complex zumis, destructible bricks, and exits. I made sure the player couldn’t walk through any of the bricks, died everytime it touched a zumi, and advanced to the next level if it went through a visible exit, ignored it otherwise. If a brick was destroyed, it should be able to walk through it. If the player picked up the walk through walls goodie, it should be able to walk through it. I made the player stay in a destructible brick with the power up, the player should die when the power up runs out.

To test my Object class, I just made sure I couldn’t create an Object (eg. Couldn’t say new Object x);

To test my Zumi class (which is treated as a Simple Zumi), I just used the second level to see whether they would move in random directions no matter where the player was. I also destroyed bricks around it to make sure that it could move through destroyed bricks. I also made sure if I dropped a bugSprayer, it couldn’t move through it. If the bugspray exploded on its square, it died. Also, the score increased by 100

To test my Complex Zumi class, I used level 002 but I removed all the simple zumis. Initially, they should behave like Simple Zumis, since they can’t reach me (they’re blocked by bricks). However, if I blow up the bricks so that they can reach me, and I’m within their smell distance (both horizontally and vertically), they should chase me until I get out of their smell distance. They also should die when in contact with bugspray, and shouldn’t move through the bricks, and should be able to move through bricks which have been destroyed. Also, the score increased by 500

To test my BugSprayer class, I would walk around level 1 and keep pressing the space bar, but the bugsprayer would only drop if there were less than 2 currently on the field. If I got the walk through goodie, I would make sure that I couldn’t drop bugsprayers on the brick. I modified the level for a short while to guarantee an extra goodie sprayer, and I made sure that I was able to drop the maximum amount of bugsprayers.

To test my BugSpray class, I made a level filled with all different types of actors. I made sure that if I dropped a bugsprayer, it detonated after the specified amount of time, and that if there were any other bugsprayers in the squares they detonated immediately, and that players, destroyable bricks, or zumis around would be killed as well.

To test my Brick class (which is treated as PermanentBricks), I made sure that they did not move, and that there would be no bugsprayer on the square ever. Also, Zumis or players could never walk through it, even if the player had a WalkThroughWalls goodie.

To test my destructableBrick class, I made sure they did not move, and that if a BugSprayer detonated in its vicinity, the BugSpray would evaporate the Brick, but not the contents past it. Also, on the next turn of the Zumi or the players, they should be able to walk through that space.

To test my Goodie class, I made sure I could not make one of that type (eg. Say Goodie\* x = new Goodie;)

To test my WalkThroughWalls goodie, I temporarily edited a level file so the probability of getting one was 100 percent, and then seeing if my player could walk through walls for the correct amount of time. Also, I would have him stand on a destructible brick, until the power ran out, and then make sure he died immediately. He also was unable to walk through Permanent Bricks. I also made sure the score increased by 1000

To test my ExtraLife goodie, I temporarily edited a level file so the probability of getting one was 100 percent, and then seeing if the lives incremented to 4. I then died 4 times to make sure all those lives were valid (not just a number on the screen). The score also increased by 1000.

To test my increasedSprayers goodie, I temporarily edited a level file so the probability of getting one was 100 percent, and then seeing if I could drop the amount of sprayers specified in the data file. The sprayers behaved exactly like they would if they were no goodie. The score increased by 1000.