Project 3
NachenBlaster

For questions about this project, first consult your TA.
If your TA can't help, ask Professor Nachenberg.

Time due:

Part 1: 11 PM, Thursday, February 22
Part 2: 11 PM, Thursday, March 1

WHEN IN DOUBT ABOUT A REQUIREMENT, YOU WILL NEVER LOSE CREDIT IF YOUR SOLUTION WORKS THE SAME AS OUR POSTED SOLUTION. SO PLEASE DO NOT ASK ABOUT ITEMS WHERE YOU CAN DETERMINE THE PROPER BEHAVIOR ON YOUR OWN FROM OUR SOLUTION!

BACK UP YOUR SOLUTION EVERY 30 MINUTES TO THE CLOUD OR A THUMB DRIVE. WE WILL NOT ACCEPT "MY COMPUTER CRASHED" EXCUSES FOR LATE WORK.

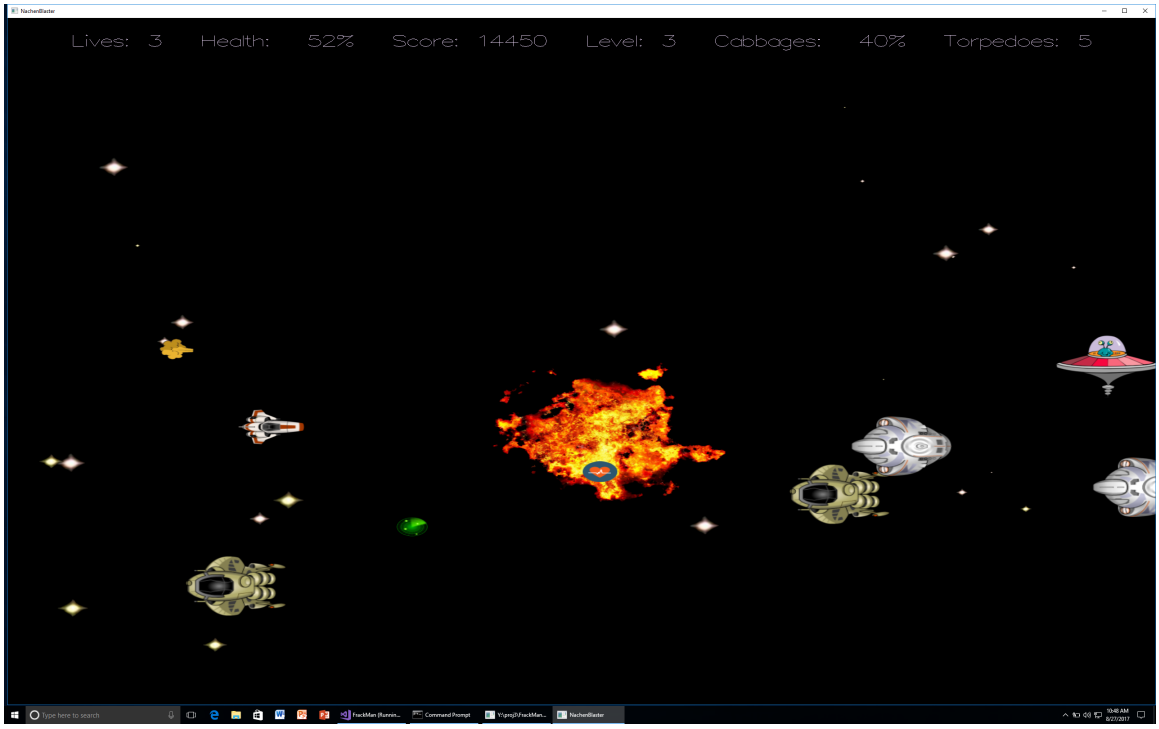
PLEASE THROTTLE THE RATE YOU ASK QUESTIONS TO 1 EMAIL PER DAY! IF YOU'RE SOMEONE WITH LOTS OF QUESTIONS, SAVE THEM UP AND ASK ONCE.

Table of Contents

Introduction

NachenGames corporate spies have learned that SmallSoft is planning to release a new game called NachenBlaster, and would like you to program an exact copy so NachenGames can beat SmallSoft to the market. To help you, NachenGames corporate spies have managed to steal a prototype NachenBlaster executable file and several source files from the SmallSoft headquarters, so you can see exactly how your version of the game must work (see posted executable file) and even get a head start on the programming. Of course, such behavior would never be appropriate in real life, but for this project, you'll be a programming villain.

In NachenBlaster, your job is to fly your spaceship through outer space, shooting down alien spacecraft and gathering the goodies that they drop. The higher the score you receive, the more respected you'll be in Star Fleet. Here's a screen-shot of NachenBlaster:



On the middle-left of the screen, you'll see your intrepid ship, the NachenBlaster – she's not the newest spaceship on the planet, but she's trusty as a 1997 Chevy Impala. The NachenBlaster comes equipped with a near-endless supply of cabbages that it can fire at the evil space aliens.

On the screen, you'll also see five enemy ships. The lower-left enemy ship shown here is a Smoregon ship. Smoregon are evil green broccoli-like aliens with weepy green ooze flowing from the eyelids on their feet. And of course, being aliens they love shooting

Turnips at your NachenBlaster ship. Smoregon are especially useful to shoot down, because when you do, they often drop goodies that you can snag and use (like repair kits and flatulence torpedoes).

The two silver-colored enemy ships are Smallgon ships. Smallgons are evil aliens that closely resemble string cheese. They are much less wealthy than the Smorgons, so they never drop goodies when they're shot down. But like Smorgons, they will often shoot Turnips at your ship if given the chance to do so.

The final enemy, in the upper-right corner of the screen, is the Snagglegon flying saucer. Snagglegons are a nasty race of aliens who resemble three-day-old bananas stolen from B-Plate, covered in fuzzy black mold (don't act innocent, we know you know what those look like). Snagglegons are extremely aggressive and have much newer ships than the Smallgons or Smorgons. They also happen to possess a magic life-saving kit which they sometimes drop when their ships are destroyed. Snagglegons also have deadlier weapons, and shoot Flatulence Torpedoes at your ship instead of the usually nasty turnips.

At the start of each game, your ship is placed in outer space and must save humanity by destroying the aliens. You start with 3 total ships (your active ship and two spares should your craft be tragically destroyed). Once you've destroyed enough of the aliens on the current level, the level will be over. Assuming you succeed in destroying enough of the aliens, you'll then proceed to the next level where even more alien ships will come at you. And of course, in each new level, the enemy ships become more difficult to destroy (having more health points, also known as hit points). On and on the game goes until all of your ships are destroyed and you die. If your ship gets destroyed before the end of a level (and you still have a spare ship), then you have to replay the entire level over from scratch. Such a depressing game.

Your NachenBlaster ship can move up, down, left and right. Your ship can and will often collide with enemy ships; this won't necessarily destroy your ship outright, but will do a significant amount of damage, so it should be avoided. On the upside, if your ship collides with an enemy ship, it will always destroy the shoddily-built enemy ship. Your ship can fire cabbages, and, if you've been lucky enough to snag some off of a Smorgon, it can also fire Flatulence Torpedoes – which do much more damage than your slimy cabbages.

You control the direction of your ship with the arrow keys, or for lefties and others for whom the arrow key placement is awkward, WASD or the numeric keypad: up is *w* or *8*, left is *a* or *4*, down is *s* or *2*, right is *d* or *6*. Use the space key to fire cabbages, and the tab key to fire Flatulence Torpedoes, if your ship has any. To quit the game at any time, press the 'q' key.

Game Details

In NachenBlaster, the player starts out a new game with three lives and continues to play until all of his/her lives have been exhausted. There are multiple levels in NachenBlaster, beginning with level 1 (NOT zero), and during each level the player must destroy an increasingly large number of alien ships. Specifically, to complete level N, the player must destroy T = 6+(4*N) enemy ships by killing them with cabbages, flatulence torpedoes, or colliding with them (which is a bad way to destroy enemy ships, by the way, since it damages your ship). So, for example, the NachenBlaster must destroy (shoot or collide with) at least 10 (6 + 4 * 1 = 10) alien ships to complete level 1.

The NachenBlaster screen is exactly 256 pixels wide by 256 pixels high. The bottom-leftmost pixel has coordinates x=0,y=0, while the upper-rightmost pixel has coordinate x=255,y=255, where x increases to the right and y increases upward toward the top of the screen. The *GameConstants.h* file we provide defines constants that represent the maze's width and height (VIEW_WIDTH and VIEW_HEIGHT), which you should use in your code instead of hard-coding the integers. Every flying object in the game will have an x coordinate in the half-open range [0,VIEW_WIDTH), and a y coordinate in the half-open range [0, VIEW_HEIGHT).

At the beginning of each level (and when the player restarts a level because their ship was destroyed), you must place 30 stars randomly across the display. Their x and y coordinates must be in the ranges [0, VIEW_WIDTH), and [0, VIEW_HEIGHT), respectively. The size of each star must be a random value between .05 to .5. Note: Each of your game objects like stars and alien ships has a size which can range from .05 to 2 units or more, and which determines the size of the graphic shown on the screen.

Once a player completes a level, they advance to the next level. At the beginning of each level (or when the player restarts a level because their ship was destroyed), the NachenBlaster ship gets its full 50 hit points, and has its cabbage inventory reset to its full amount. However, the NachenBlaster does not maintain any flatulence torpedoes that it had during the previous level; it always starts each new round with zero flatulence torpedoes.

Once a level begins, it is divided into small time periods called *ticks*. During a given tick, every flying object in the game (including the NachenBlaster, each of the alien ships, stars, all projectiles, and all goodies) has an opportunity to do something (e.g., move/fire a projectile/fall/etc.).

During each tick of the game, your program must do the following:

- You must give each flying object – including the NachenBlaster/player, aliens, stars, projectiles, goodies, etc. - a chance to do something – e.g., fire, move, die, drop a goodie, etc.

- You must check to see if the NachenBlaster has died (lost all of its energy) and therefore has lost its current life. If so, you must indicate this to our game framework (we'll tell you how later).
- You must delete all dead flying objects from the game – this includes alien ships that have been destroyed by projectiles or collisions (i.e., had their hit points drop to, or below, zero), projectiles that have collided with an enemy ship, flying objects that have flown off the edges of the screen and are no longer visible, goodies that have been picked up, etc.
- You must update the game statistics line at the top of the screen, including the number of remaining lives the NachenBlaster has, its health percentage, the player's current score, the current level number, the percent of cabbage power the player has left to fire at the current moment, and how many Flatulence Torpedoes the player currently has to fire.. (The health percentage is the percentage of the maximum 50 hit points the NachenBlaster currently has.)
- Check to see if the player has completed the current level, and if so, end the current level so the player may advance to the next level.

During each tick, the game may also need to introduce one or more new flying objects into the game – for instance, a new star or an alien ship. You must use the following rules to decide whether to introduce one or more new flying objects into the game:

- There is a one in fifteen chance that you will introduce a new star into the game on the far right side of the screen (at x=VIEW_WIDTH-1). Each such star will have a random y coordinate between [0, VIEW_HEIGHT). The size of each new star must also be determined randomly, and must be between .05 and .5 units in size.
- You must use the following formula to decide whether to add a new alien ship during a given tick:
    - First, determine how many alien ships have been destroyed during the current level (not including any ships destroyed before the player last lost a ship). Call this number D.
    - Second, compute the number of Remaining alien ships that must be destroyed before the level is completed, R. Remember that the total number of ships that must be destroyed per level is T = 6+(4*n), so the number of remaining ships R is equal to T minus D (the number of destroyed ships thus far).
    - Third, compute the maximum number of alien ships that should be on the screen at a time: M = 4 + (.5 * current_level_number)
    - If the current number of alien ships on the screen is less than *min(M, R)*, then you must introduce one new alien ship at the far right side of the screen during the current tick.
- Should you need to add a new alien ship to the screen during a tick, you must use the following formula to decide which type of ship to add:
    - S1 = 60
    - S2 = 20 + n * 5, where n is the current level number.
    - S3 = 5 + n * 10, where n is the current level number.

- S = S1 + S2 + S3
- With probability S1/S, the new ship you add must be a Smallgon.
- With probability S2/S, it must be a Smoregon.
- With probability S3/S, it must be a Snagglegon.
- All new alien ships must start at x=VIEW_WIDTH-1, and a random y between 0 and VIEW_HEIGHT-1, inclusive.

The status line at the top of the screen must have the following components:

```
Lives: 3  Health: 100%  Score: 24530  Level: 3  Cabbages: 80%  Torpedoes: 4
```

Each labeled value of the status line must be separated from the next by exactly two spaces. For example, the 3 between "Lives: " and "Health:" must have two spaces after it. You may find the Stringstreams writeup on the class web site to be helpful.

There are three major types of aliens in NachenBlaster: Smallgons, Smoregons, and Snagglegons. Their respective behaviors are described in the sections on alien behaviors below.

There are three major types of goodies in NachenBlaster: Extra Life Goodies, Repair Goodies, and Flatulence Torpedo Goodies. Their respective behaviors are described in the sections on goodies below.

There are three major types of projectiles in NachenBlaster: cabbages, turnips and Flatulence Torpedoes. Their respective behaviors are described in the sections on projectile behaviors below.

If the NachenBlaster's hit points reaches zero (the NachenBlaster loses hit points when it collides with a projectile or with an alien ship), the ship dies and loses one "life." If, after losing a life, the NachenBlaster has one or more remaining lives left, it is placed back on the current level and they must again complete the entire level from scratch (with the level starting as it was at the beginning of the first time it was attempted). If the NachenBlaster dies and has no lives left, then the game is over.

The player may fire the NachenBlaster's cabbage cannon by pressing the spacebar. A cabbage fired by the NachenBlaster will do 2 points of damage to the first alien ship it comes into contact with, and then the cabbage will disappear from the impact. The player may launch a Flatulence Torpedo, if the ship has any, by pressing the tab key. A Flatulence Torpedo fired by the NachenBlaster will do 8 points of damage to the first alien ship it hits, and then the torpedo will disappear from the impact. Projectiles fired by the NachenBlaster can never damage the NachenBlaster and if the projectile and the NachenBlaster overlap in space, the two will just fly past each other.

Alien ships may also fire projectiles at the NachenBlaster. A turnip fired by a Smallgon or Smoregon will do 2 points of damage to the NachenBlaster. A flatulence torpedo fired by a Snagglegon will do 8 points of damage to the NachenBlaster (but no damage to other alien ships. A projectile fired by an alien ship can never damage other alien ships (it

will pass right through them), and if the projectile and an alien ship overlap in space, they will ignore each other.

If an alien ship dies either from being hit by a projectile fired by the NachenBlaster or through a collision with the NachenBlaster, the player receives points:

For destroying a Smallgon or Smoregon:	250 points
For destroying a Snagglegon:	1000 points

The player does not earn points for ships that have flown off of the screen and therefore didn't die due to a collision with the NachenBlaster or a NachenBlaster-fired projectile. The NachenBlaster also earns 100 points and special benefits by picking up a goodie of any type.

When a NachenBlaster dies because its hit points goes to zero or below, the player's number of remaining lives is decremented by 1. If the player still has at least one life left, then the user is prompted to continue and given another chance by restarting the current level from scratch.

Your game implementation must play various sounds when certain events occur, using the playSound method provided by our GameWorld class, e.g.:

```
pointerToGameWorld->playSound(SOUND_TORPEDO); // player fired a torpedo
```

- You must play a SOUND_TORPEDO sound any time the player successfully fires a torpedo.
- You must play a SOUND_PLAYER_SHOOT sound any time the player successfully fires a cabbage.
- You must play a SOUND_BLAST sound any time an enemy projectile collides with the NachenBlaster.
- You must play a SOUND_ALIEN_SHOOT sound any time a Smallgon or Smoregon fires a turnip.
- You must play a SOUND_TORPEDO sound any time a Snagglegon fires a torpedo.
- You must play a SOUND_GOODIE sound any time the player successfully picks up a goodie.
- You must play a SOUND_DEATH sound any time an alien ship dies due to being hit by a projectile or colliding with the NachenBlaster.
- You must play a SOUND_BLAST sound any time an alien ship is hit by a projectile fired by the NachenBlaster, but only if it does not die from the blast – if it died from the blast, you'd have to play a SOUND_DEATH sound.
- You must play a SOUND_FINISHED_LEVEL sound every time the player successfully completes a level.

Constants for each specific sound, e.g., SOUND_TORPEDO, may be found in our *GameConstants.h* file.

Two flying objects will collide (e.g., an alien ship and the NachenBlaster, the NachenBlaster and a goodie, an alien ship and a NachenBlaster-fired projectile, or an alien-fired projectile and the NachenBlaster) if the Euclidean distance between the two flying objects is less than the value in this formula:

X1,Y1 are the coordinates of flying object 1; R1 is flying object 1's radius.
X2,Y2 are the coordinates of flying object 2; R2 is flying object 2's radius.

```
IF (euclidean_dist(x1, y1, x2, y2) < .75 * (R1 + R2))
        THEN the two flying objects collided;
```

The x,y values may be obtained with the getX() and getY() methods in GraphObject, and the radius values may be obtained with the getRadius() method in GraphObject. Your game objects will all be derived from our GraphObject, so these functions will be easy to call.

There are additional game details that you must address in your implementation – these will be described in the sections below.

So how does a video game work?

Fundamentally, a video game is composed of a bunch of game objects; in NachenBlaster, those objects include the NachenBlaster ship, aliens like Smallgons, Smoregons, Snagglegons, goodies (e.g., Extra Life Goodies, Repair Goodies, Flatulence Torpedo Goodies), projectiles (e.g., turnips, cabbages and Flatulence Torpedoes), stars, and explosions. Let's call these objects "actors," since each object is an actor in our video game. Each actor has its own x,y location in space, its own internal state (e.g., a Snagglegon knows its location, what direction it's moving, etc.) and its own special algorithms that control its actions in the game based on its own state and the state of the other objects in the world. In the case of the NachenBlaster ship, the algorithm that controls the ship object is the user's own brain and hand, and the keyboard! In the case of other actors (e.g., Smallgon), each object has an internal autonomous algorithm and state that dictates how the object behaves in the game world.

Once a game begins, gameplay is divided into *ticks*. A tick is a unit of time, for example, 50 milliseconds (that's 20 ticks per second).

During a given tick, the game calls upon each object's behavioral algorithm and asks the object to perform its behavior. When asked to perform its behavior, each object's behavioral algorithm must decide what to do and then make a change to the object's state (e.g., move the object 1 pixel to the left), or change other objects' states (e.g., when a Smoregon's algorithm is called by the game, it may determine that the NachenBlaster ship has moved into its line of fire, and it may fire a turnip at it). Typically the behavior exhibited by an object during a single tick is limited in order to ensure that the gameplay

is smooth and that things don't move too quickly and confuse the player. For example, a Smallgon will move just a couple of pixels left/right/up/down, rather than moving ten or more pixels per tick; a Smallgon moving, say, 20 pixels in a single tick would confuse the user, because humans are used to seeing smooth movement in video games, not jerky shifts.

After the current tick is over and all actors have had a chance to adjust their state (and possibly adjust other actors' states), the graphical framework that we provide animates the actors onto the screen in their new configuration. So if a Smallgon changed its location from 10,5 to 9,5 (moved one pixel left), then our game framework would erase the graphic of the Smallgon from location 10,5 on the screen and draw the Smallgon's graphic at 9,5 instead. Since this process (asking actors to do something, then animating them to the screen) happens 20 times per second, the user will see somewhat smooth animation.

Then, the next tick occurs, and each actor's algorithm is again allowed to do something, our framework displays the updated actors on-screen, etc.

Assuming the ticks are quick enough (a fraction of a second), and the actions performed by the objects are subtle enough (i.e., a Smallgon doesn't move 3 inches away from where it was during the last tick, but instead moves 1 millimeter away), when you display each of the objects on the screen after each tick, it looks as if each object is performing a continuous series of fluid motions.

A video game can be broken into three different phases:

Initialization: The Game World is initialized and prepared for play. This involves allocating one or more actors (which are C++ objects) and placing them in the game world so that they will appear in the maze.

Game play: Game play is broken down into a bunch of ticks. During each tick, all of the actors in the game have a chance to do something, and perhaps die. During a tick, new actors may be added to the game and actors who die must be removed from the game world and deleted.

Cleanup: The player has lost a life (but has more lives left), the player has completed the current level, or the player has lost all of their lives and the game is over. This phase frees all of the objects in the World (e.g., NachenBlaster, all aliens, all stars, all explosions, all goodies, all projectiles, etc.) since the level has ended. If the game is not over (i.e., the player has more lives), then the game proceeds back to the *Initialization* step, where the level is repopulated with new occupants, and game play starts for the level.

Here is what the main logic of a video game looks like, in pseudocode (The *GameController.cpp* we provide for you has some similar code):

```
while (The NachenBlaster has lives left)
{
   Prompt_the_user_to_start_playing(); // "press a key to start"
   Initialize_the_game_world();        // you're going to write this

   while (the NachenBlaster is still alive)
   {
          // each pass through this loop is a tick (1/20th of a sec)

          // you're going to write code to do the following
         Tell_each_actor_to_do_something();
         Delete_any_dead_actors_from_the_world();

          // we write this code to handle the animation for you
         Animate_each_actor_to_the_screen();
         Sleep_for_50ms_to_give_the_user_time_to_react();
   }
     // the NachenBlaster died - you're going to write this code
   Cleanup_all_game_world_objects();   // you're going to write this
   if (the NachenBlaster is still alive)
      Prompt_the_NachenBlaster_to_continue();
}

Tell_the_user_the_game_is_over();      // we provide this
```

And here is what the `Ask_all_actors_to_do_something()` function might look like:

```
void Tell_each_actor_to_do_something()
{
   for each actor on the level:
       if (the actor is still alive)
           tell the actor to doSomething();
}
```

You will typically use a container (an array, vector, or list) to hold pointers to each of your live actors. Each actor (a C++ object) has a *doSomething( )* member function in which the actor decides what to do. For example, here is some pseudocode showing what a (simplified) Smallgon might decide to do each time it gets asked to do something:

```
class Smallgon: public SomeOtherClass
{
   public:
       virtual void doSomething()
       {
           If the player is in my line of sight, then
                  Fire a turnip in the direction of the player
           Else if I still want to continue moving in the current direction
                  Move one pixel in my current direction
                  Decrement the number of remaining ticks to move in this direction
           Else if I want to choose a new direction
                  Pick a new direction to move, and pick how many ticks to move
                  in that direction.
       }
       ...
};
```

And here's what the NachenBlaster's *doSomething( )* member function might look like:

```
class NachenBlaster: public …
{
   public:
        virtual void doSomething()
        {
             Try to get user input (if any is available)
               If the user pressed the UP key then
                   Increase my y location by one
               If the user pressed the DOWN key then
                   Decrease my y location by one
               ...
               If the user pressed the space bar to fire and the player has
                         cabbage power left, then
                  Introduce a new cabbage object into the game
               ...
        }
        ...
};
```

What Do You Have to Do?

You must create a number of different classes to implement the NachenBlaster game. Your classes must work properly with our provided classes, and **you must not modify our classes or our source files in any way to get your classes to work properly (doing so will result in a score of zero on the entire project!).** Here are the specific classes that you must create:

1. You must create a class called *StudentWorld* that is responsible for keeping track of your game world and all of the actors/objects (your NachenBlaster, aliens, projectiles, goodies, stars, explosions, etc.) that are inside the game.
2. You must create a class to represent the NachenBlaster ship in the game.
3. You must create classes for Smallgons, Smoregons, Snagglegons, cabbages, turnips, Flatulence Torpedoes, Extra Life Goodies, Repair Goodies, Flatulence Torpedo Goodies, Stars, explosions, etc., as well as any additional base classes (e.g., an Alien base class if you find it convenient) that help you implement your actors.

You Have to Create the StudentWorld Class

Your *StudentWorld* class is responsible for orchestrating virtually all game play – it keeps track of the whole game world (the space field and all of its inhabitants such as Smallgons, Snagglegons, the NachenBlaster, stars, goodies, projectiles, etc.). It is responsible for initializing the game world at the start of the game, asking all the actors to do something during each tick of the game, destroying an actor when it disappears (e.g., an alien dies, an alien, projectile or star flies off the screen, an explosion finishes

exploding, a projectile impacts a ship, etc.), and destroying *all* of the actors in the game world when the user loses a life.

Your StudentWorld class **must** be derived from our GameWorld class (found in *GameWorld.h*) and **must** implement at least these three methods (which are defined as pure virtual in our GameWorld class):

```
virtual int init() = 0;
virtual int move() = 0;
virtual void cleanUp() = 0;
```

The code that you write must *never* call any of these three functions (except that *StudentWorld*'s destructor may call *cleanUp()*). Instead, our provided game framework will call these functions for you. So you have to implement them correctly, but you won't ever call them yourself in your code (except in the one place noted above).

Each time a level starts, our game framework will call the *init()* method that you defined in your *StudentWorld* class. You don't call this function; instead, our provided framework code calls it for you.

The *init()* method is responsible for constructing a representation of the current level in your StudentWorld object and populating it with initial objects (e.g., stars), using one or more data structures that you come up with.

The *init()* method is automatically called by our provided code either (a) when the game first starts, (b) when the player completes the current level and advances to a new level (that needs to be initialized), or (c) when the user loses a life (but has more lives left) and the game is ready to restart at the current level.

After the *init()* method finishes initializing your data structures/objects for the current leve, it must return `GWSTATUS_CONTINUE_GAME`.

Once a level has been prepared with a call to the *init()* method, our game framework will repeatedly call the *StudentWorld's move()* method, at a rate of roughly 20 times per second. Each time the *move()* method is called, it must run a single tick of the game. This means that it is responsible for asking each of the game actors (e.g., the NachenBlaster, each Smallgon, goodie, projectile, star, explosion, etc.) to try to do something: e.g., move themselves and/or perform their specified behavior. Finally, this method is responsible for disposing of (i.e., deleting) actors that need to disappear during a given tick (e.g., a cabbage that flies off the screen or collides with an enemy, a dead Smallgon, etc.). For example, if a Smallgon is shot by the player and its hit points drains to zero, then its state should be set to dead, and then after all of the alive actors in the game get a chance to do something during the tick, the *move()* method should remove that Smallgon from the game world (by deleting its object and removing any reference to the object from the *StudentWorld's* data structures). The *move()* method will automatically be called once during each tick of the game by our provided game framework. You will never call the *move()* method yourself.

The *cleanup()* method is called by our framework when the NachenBlaster completes the current level or loses a life (i.e., its hit points reach zero due to being shot by or colliding with the aliens). The *cleanup()* method is responsible for freeing all actors (e.g., all Smallgon objects, all star objects, all explosion objects, all projectiles, all goodie objects, the NachenBlaster object, etc.) that are currently in the game. This includes all actors created during either the *init()* method or introduced during subsequent game play by the actors in the game (e.g., a turnip that was added to the screen by a Smoregon) that have not yet been removed from the game.

You may add as many other public/private member functions or private data members to your *StudentWorld* class as you like (in addition to the above three member functions, which you *must* implement).

Your *StudentWorld* class must be derived from our *GameWorld* class. Our *GameWorld* class provides the following methods for your use:

```
unsigned int getLevel() const;
unsigned int getLives() const;
void decLives();
void incLives();
unsigned int getScore() const;
void increaseScore(unsigned int howMuch);
void setGameStatText(string text);
bool getKey(int& value);
void playSound(int soundID);
```

getLevel() can be used to determine the current level number.

getLives() can be used to determine how many lives the NachenBlaster has left.

decLives() reduces the number of NachenBlaster lives by one.

incLives() increases the number of NachenBlaster lives by one.

getScore() can be used to determine the NachenBlaster's current score.

increaseScore() is used by a *StudentWorld* object (or your other classes) to increase the user's score upon successfully destroying an alien or picking up a Goodie of some sort. When your code calls this method, you must specify how many points the user gets (e.g., 250 points for destroying a Smallgon). This means that the game score is controlled by our *GameWorld* object – you *must not* maintain your own score data member in your own classes.

The *setGameStatText()* method is used to specify what text is displayed at the top of the game screen, e.g.:

```
Lives: 3  Health: 100%  Score: 24530  Level: 3  Cabbages: 80%  Torpedoes: 4
```

getKey() can be used to determine if the user has hit a key on the keyboard to move the NachenBlaster or to fire a projectile. This method returns true if the user hit a key during the current tick, and false otherwise (if the user did not hit any key during this tick). The only argument to this method is a variable that will be set to the key that was pressed by the user (if any key was pressed). If the function returns true, the argument will be set to one of the following values (defined in *GameConstants.h*):

```
KEY_PRESS_LEFT
KEY_PRESS_RIGHT
KEY_PRESS_UP
KEY_PRESS_DOWN
KEY_PRESS_SPACE
KEY_PRESS_TAB
```

The *playSound()* method can be used to play a sound effect when an important event happens during the game (e.g., an alien dies or the NachenBlaster picks up a goodie). You can find constants (e.g., SOUND_PLAYER_SHOOT) that describe what noise to make in the *GameConstants.h* file. The *playSound()* method is defined in our GameWorld class, which you will use as the base class for your StudentWorld class. Here's how this method might be used:

```
// if a Smallgon reaches zero hit points and dies, make a dying sound

if (theAlienHasZeroHitPoints())
    studentWorldObject->playSound(SOUND_DEATH);
```

init() Details

Your StudentWorld's *init()* member function must:

1. Initialize the data structures used to keep track of your game's world.
2. Allocate and insert a valid NachenBlaster object into the game world.
3. Allocate and insert star objects into the game world.

Your *init()* method must construct a representation of your world and store this in a *StudentWorld* object. It is **required** that you keep track of all of the actors (e.g., aliens like Snagglegons, stars, explosions, projectiles like cabbages, goodies, etc.) in a **single** STL collection such as a *list* or *vector*. (To do so, we recommend using a container of pointers to the actors). If you like, your *StudentWorld* object may keep a separate pointer to the NachenBlaster object rather than keeping a pointer to that object in the container with the other actor pointers; the NachenBlaster is the **only** actor allowed to not be stored in the single actor container. The *init()* method may also initialize any other *StudentWorld* member variables it needs, such as the number of alien ships destroyed so far on this level, or how many more alien ships need to be destroyed before the current level is over.

You must not call the *init()* method yourself. Instead, this method will be called by our framework code when it's time for a new game to start (or when the player completes a level or needs to restart a level).

move() Details

The *move()* method must perform the following activities:

1. It must ask all of the actors that are currently active in the game world to do something (e.g., ask a Smallgon to move itself, ask a goodie to check if it has come into contact with the NachenBlaster, and if so, grant it a special power, give the NachenBlaster a chance to move up, down, left or right, etc.).
    a. If an actor does something that causes the NachenBlaster to die (e.g., a projectile or alien ship collides with the NachenBlaster), then the *move()* method should immediately return `GWSTATUS_PLAYER_DIED`.
    b. Otherwise, if the required number of alien ships have been destroyed in the current level to advance to the next level, then the *move()* method must return a value of `GWSTATUS_FINISHED_LEVEL`.
2. It must then delete any actors that have died during this tick (e.g., a Smallgon that was killed by a cabbage and so should be removed from the game world, or a goodie that disappeared because the NachenBlaster picked it up).
3. It must update the status text on the top of the screen with the latest information (e.g., the user's current score, the number of torpedoes the NachenBlaster has, the current level, etc.).

The *move()* method must return one of three different values when it returns at the end of each tick (all are defined in *GameConstants.h*):

```
GWSTATUS_PLAYER_DIED
GWSTATUS_CONTINUE_GAME
GWSTATUS_FINISHED_LEVEL
```

The first return value indicates that the NachenBlaster died during the current tick, and instructs our provided framework code to tell the user the bad news and restart the level if the user has more lives left. If your *move()* method returns this value and the NachenBlaster has more lives left, then our framework will prompt the player to continue the game, call your *cleanup()* method to destroy the level, call your *init()* method to re-initialize the level from scratch, and then begin calling your *move()* method over and over, once per tick, to let the user play the level again.

The second return value indicates that the tick completed without the NachenBlaster dying BUT the NachenBlaster has not yet completed the current level. Therefore the game play should continue normally for the time being. In this case, the framework will advance to the next tick and call your *move()* method again.

The final return value indicates that the NachenBlaster has completed the current level (that is, it successfully destroyed or collided with the appropriate number of alien ships to

complete the level). If your *move()* method returns this value, then the current level is over, and our framework will call your *cleanup()* method to destroy the level, our framework will then advance to the next level, then call your *init()* method to prepare that level for play, etc…

IMPORTANT NOTE: The skeleton code that we provide to you is hard-coded to return a `GWSTATUS_PLAYER_DIED` status value from our dummy version of the *move()* method. Unless you implement something that returns `GWSTATUS_CONTINUE_GAME` your game will not display any objects on the screen! So if the screen just immediately tells you that you lost a life once you start playing, you'll know why!

Here's pseudocode for how the move() method might be implemented:

```
int StudentWorld::move()
{
      // The term "actors" refers to all aliens, the NachenBlaster, goodies,
      // stars, explosions, projectiles, stars, etc.

      // Give each actor a chance to do something, incl. the NachenBlaster
    for each of the actors in the game world
    {
        if (actor[i] is still active/alive)
        {
            // tell each actor to do something (e.g. move)
          actor[i]->doSomething();

          if (theNachenBlasterDiedDuringThisTick())
                return GWSTATUS_PLAYER_DIED;

          if (theNachenBlasterCompletedTheCurrentLevel())
          {
                increaseScoreAppropriately();
                return GWSTATUS_FINISHED_LEVEL;
          }
        }
    }

      // Remove newly-dead actors after each tick
    removeDeadGameObjects(); // delete dead game objects

      // Update the Game Status Line
    updateDisplayText();   // update the score/lives/level text at screen top

      // the player hasn't completed the current level and hasn't died, so
      // continue playing the current level
    return GWSTATUS_CONTINUE_GAME;
}
```

Give Each Actor a Chance to Do Something

During each tick of the game each active actor must have an opportunity to do something (e.g., move around, shoot, etc.). Actors include the NachenBlaster, aliens Smallgons, Smoregons, Snagglegons, explosions (which are added to the game when an alien ship dies and must linger on the screen for a second or two, then disappear), projectiles like cabbages, turnips and torpedoes, goodies like Extra Life Goodies, and stars.

Your *move()* method must iterate over active actor that's active in the game (i.e., held by your *StudentWorld* object) and ask it to do something by calling a member function in the actor's object named *doSomething()*. In each actor's *doSomething()* method, the object will have a chance to perform some activity based on the nature of the actor and its current state: e.g., a Smallgon might move one pixel left, the NachenBlaster might shoot a cabbage, an explosion may increase in size, etc.

It is possible that one actor (e.g., a cabbage projectile) may destroy another actor (e.g., a Smallgon) during the current tick. If an actor has died earlier in the current tick, then the dead actor must not have a chance to do something during the current tick (since it's dead).

To help you with testing, if you press the `f` key during the course of the game, our game controller will stop calling move() every tick; it will call move() only when you hit a key (except the `r` key). Freezing the activity this way gives you time to examine the screen, and stepping one move at a time when you're ready helps you see if your actors are moving properly. To resume regular game play, press the `r` key.

Remove Dead Actors after Each Tick

At the end of each tick, your *move()* method must determine which of your actors are no longer alive, remove them from your container of active actors, and use a C++ delete expression to free their objects (so you don't have a memory leak). So if, for example, a Smallgon's hit points go to zero (due to it being shot) and it dies, then it should be noted as dead, and at the end of the tick, its *pointer* should be removed from the StudentWorld's container of active objects, and the Smallgon object should be deleted (using a C++ delete expression) to free up memory for future actors that will be introduced later in the game. Or, for example, when a star, goodie, projectile or alien ship flies off the screen, it needs to be deleted as well. (Hint: Each of your actors could maintain a dead/alive status.)

cleanUp() Details

When your *cleanUp()* method is called by our game framework, it means that the NachenBlaster lost a life (e.g., its hit points reached zero due to being shot) or has completed the current level. In this case, every actor in the entire game (the NachenBlaster and every alien, goodie, projectile, star, explosion object, etc.) must be deleted and removed from the StudentWorld's container of active objects, resulting in an empty level. If the user has more lives left, our provided code will subsequently call your *init()* method to reload and repopulate the space field with a new set of stars, and the level will then continue from scratch with a brand new set of aliens.

You must not call the *cleanUp()* method yourself when the NachenBlaster dies. Instead, this method will be called by our code when *init()* returns an appropriate status.

You Have to Create the Classes for All Actors

The NachenBlaster game has a number of different game objects, including:

- The NachenBlaster
- Aliens: Smallgons, Smoregons, Snagglegons
- Stars
- Projectiles: Cabbages, Turnips, Flatulence Torpedoes
- Explosions
- Goodies: Repair Goodies, Extra Life Goodies, Flatulence Torpedo Goodies

Each of these game objects can occupy a level and interact with other game objects within the visible screen view.

Now of course, many of your game objects will share things in common – for instance, every one of the objects in the game (Smallgons, the NachenBlaster, Stars, Explosions, etc.) has x,y coordinates. Many game objects have the ability to perform an action (e.g., move or shoot) during each tick of the game. Many of them can potentially be attacked (e.g., the NachenBlaster and alien ships) and could "die" during a tick. Certain objects like projectiles and goodies "activate" when they come into contact with a proper target (e.g., goodies activate when they overlap with the NachenBlaster ship, projectiles activate and damage their victim when they overlap with an enemy ship). Every game object also needs some attribute that indicates whether or not they are still alive or they died during the current tick, etc.

It is therefore your job to determine the commonalities between your different game objects and make sure to factor out common behaviors and traits and move these into appropriate base classes, rather than duplicate these items across your derived classes – this is in fact one of the tenets of object oriented programming.

Your grade on this project will depend upon your ability to intelligently create a set of classes that follow good object-oriented design principles. Your classes must never duplicate code or a data member – if you find yourself writing the same (or largely similar) code across multiple classes, then this is an indication that you should define a common base class and migrate this common functionality/data to the base class. Duplication of code is a so-called *code smell*, a weakness in a design that often leads to bugs, inconsistencies, code bloat, etc.

Hint: When you notice this specification repeating the same text nearly identically in the following sections (e.g., in the Extra Life Goodie section and the Repair Goodie section, or in the Smallgon and Smoregon sections) you must make sure to identify common

behaviors and move these into proper base classes. NEVER duplicate behaviors across classes that can be moved into a base class!

You MUST derive all of your game objects directly or indirectly from a base class that we provide called *GraphObject*, e.g.:

```
class Actor: public GraphObject
{
public:
	…
};

class Smoregon: public Actor
{
public:
	…
};

class Projectile: public Actor
{
public:
	…
};
```

GraphObject is a class that we have defined that helps hide the ugly logic required to graphically display your actors on the screen. If you don't derive your classes from our *GraphObject* base class, then you won't see anything displayed on the screen! ☺

The *GraphObject* class provides the following methods that you may use:

```
GraphObject(int imageID, double startX, double startY,
            int startDirection = 0, double size = 1.0, int depth = 0);
double getX() const;                    // in pixels (0-255)
double getY() const;                    // in pixels (0-255)
virtual void moveTo(double x, double y); // in pixels (0-255)
int getDirection() const;               // in degrees (0-359)
void setDirection(int d);               // in degrees (0-359)
void setSize(double size);
double getSize() const;
double getRadius() const;               // in pixels (0-255)
```

You may use any of these member functions in your derived classes, but you **must not** use any other member functions found inside of *GraphObject* in your other classes (even if they are public in our class). You must not redefine any of these methods in your derived classes since they are not defined as virtual in our base class.

GraphObject(int imageID,
int startX,
int startY,
int startDirection,
double size,
int depth)

is the constructor for a new *GraphObject*. When you construct a new *GraphObject*, you must specify an image ID that indicates how the *GraphObject* should be displayed on screen (e.g., as a Smallgon, a NachenBlaster, a star, an explosion, etc.). You must also specify the initial x,y location of the object. The x value may range from 0 to VIEW_WIDTH-1 inclusive, and the y value may range from 0 to VIEW_HEIGHT-1 inclusive. Notice that you pass the coordinates as x, y (i.e., column, row starting from bottom left, and ***not*** row, column). You may also specify the initial direction an object is facing – usually 0 degrees is just fine for this parameter. (Right is 0, up is 90, left is 180, and down is 270.) You may also optionally specify the initial size of an object, with 1.0 being the default size. Finally, you must specify the depth of the object. An object of depth 0 is in the foreground, whereas objects with increasing depths are drawn further in the background. Thus an object of depth zero always covers object with a depth greater than zero, and an object with a depth of one always covers objects of depth two or greater.

One of the following IDs, found in *GameConstants.h*, must be passed in for the imageID value:

```
IID_NACHENBLASTER
IID_SMALLGON
IID_SMOREGON
IID_SNAGGLEGON
IID_REPAIR_GOODIE
IID_LIFE_GOODIE
IID_TORPEDO_GOODIE
IID_TORPEDO
IID_TURNIP
IID_CABBAGE
IID_STAR
IID_EXPLOSION
```

If you derive your game objects from our *GraphObject* class, they will be displayed on screen automatically by our framework (e.g., a Smallgon image will be drawn to the screen at the *GraphObject's* specified x,y coordinates if the object's Image ID is IID_SMALLGON).

getX() and *getY()* are used to determine a *GraphObject's* current location in the space field. Since each *GraphObject* maintains its own x,y location, this means that your derived classes MUST NOT also have x,y member variables, but instead use these functions and *moveTo()* from the *GraphObject* base class.

moveTo(double x, double y) is used to update the location of a GraphObject within the space field. For example, if a Smallgon's movement logic dictates that it should move one pixel to the left, you could do the following:

```
moveTo(getX()-1, y);       // move one pixel to the left
```

You ***must*** use the *moveTo()* method to adjust the location of a game object if you want that object to be properly animated. As with the *GraphObject* constructor, note that the order of the parameters to *moveTo* is x,y (col,row) and NOT y,x (row,col).

getDirection() is used to determine the direction a GraphObject is facing, and returns a value between 0 and 359 degrees.

setDirection(Direction d) is used to change the direction a GraphObject is facing. For example, you could use this method and *getDirection()* to adjust the direction a turnip or cabbage is facing as it flies through the air.

getSize() is used to determine the size of a GraphObject, and returns a positive value.

setSize(double size) is used to change the size of a GraphObject to the specified size.

getRadius() is used to determine the radius of a GraphObject in pixels on the screen. For example, a GraphObject with a size of 1 will have a radius of 8 pixels. You can use this function, along with getX() and getY() to determine if two GraphObjects have collided in space.

The NachenBlaster

Here are the requirements you must meet when implementing the NachenBlaster class.

What the NachenBlaster Must Do When It Is Created

When it is first created:

1. A NachenBlaster object must have an image ID of IID_NACHENBLASTER.
2. A NachenBlaster must always start at location x=0, y=128.
3. A NachenBlaster has a direction of 0.
4. A NachenBlaster has a size of 1.0.
5. A NachenBlaster has a depth of 0.
6. A NachenBlaster, in its initial state:
    a. Has 50 hit points.
    b. Has 30 cabbage energy points.

What the NachenBlaster Must Do During a Tick

The NachenBlaster must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something, the NachenBlaster must do the following:

1. The NachenBlaster must check to see if it is currently alive. If not, then the NachenBlaster's *doSomething()* method must return immediately – none of the following steps should be performed.
2. Otherwise, the *doSomething()* method must check to see if the user pressed a key (the section below shows how to check this). If the user pressed a key:
    a. If the user pressed the space bar and if the NachenBlaster has at least five cabbage energy points, the NachenBlaster will fire a cabbage, which reduces their cabbage energy count by 5 cabbage energy points. To fire a cabbage, a new cabbage object must be added exactly 12 pixels to the right of the NachenBlaster ship, with a starting direction of zero degrees. For example, if the NachenBlaster is at x=10,y=7, then the cabbage would be created at location x=22, y=7. When the NachenBlaster fires a cabbage, it must play the SOUND_PLAYER_SHOOT sound effect (see the *StudentWorld* section of this document for details on how to play a sound). Hint: When you create a new cabbage object in the proper location, give it to the StudentWorld to manage (e.g., animate) along with the other game objects.
    b. If the user pressed the tab key and if the NachenBlaster has any Flatulence Torpedoes in its inventory, the NachenBlaster will fire a Flatulence Torpedo, and will have its torpedo count decremented by one. To fire a torpedo, a new torpedo object must be added exactly 12 pixels to the right of the NachenBlaster ship, with a starting direction of zero degrees. When the NachenBlaster fires a torpedo, it must play the SOUND_TORPEDO sound effect (see the *StudentWorld* section of this document for details on how to play a sound).
    c. If the user asks to move up, down, left or right by pressing a directional key AND the movement would not cause the NachenBlaster to move off the screen (i.e., have an x value < 0 or >= VIEW_WIDTH or a y value < 0 or >= VIEW_HEIGHT), then the NachenBlaster must update its location by six pixels in the specified direction with the *GraphObject* class's *moveTo()* method. Of course, any movement may cause a collision with an alien ship or projectile, so you must check for this. See the section below.
    d. Regardless of what the NachenBlaster does, it will receive one cabbage energy point per tick of the game up to a maximum of 30 cabbage energy points, so even if the NachenBlaster ship shoots all of its cabbages, within five ticks it will have the energy to fire one more.

<u>What the NachenBlaster Must Do When It Collides With a Projectile or Alien Ship</u>

When a projectile like a turnip or a Flatulence Torpedo collides with the NachenBlaster, or vice versa, it will incur damage, decreasing the NachenBlaster's hit points by an amount dictated by the projectile (different projectiles do different amounts of damage).

Similarly, if an alien ship collides with the NachenBlaster or vice versa, the NachenBlaster incurs damage and the alien ship will be destroyed.

To detect if a collision occurs, you must use the Euclidean distance approach described on page 10.

If the NachenBlaster's hit points reach zero or below, it must set its state to dead so the current round of play can end.

If the NachenBlaster collides with a projectile, the game must play an impact sound effect: SOUND_BLAST. If the NachenBlaster collides with an alien ship (which will thus be destroyed), play the SOUND_DEATH effect.

Getting Input From the User

Since NachenBlaster is a *real-time* game, you can't use the typical *getline* or *cin* approach to get a user's key press within the NachenBlaster's *doSomething()* method—that would stop your program and wait for the user to type something and then hit the Enter key. This would make the game awkward to play, requiring the user to hit a directional key then hit Enter, then hit a directional key, then hit Enter, etc. Instead of this approach, you will use a function called *getKey()* that we provide in our *GameWorld* class (from which your StudentWorld class is derived) to get input from the user[1]. This function rapidly checks to see if the user has hit a key. If so, the function returns true and the int variable passed to it is set to the code for the key. Otherwise, the function immediately returns false, meaning that no key was hit. This function could be used as follows:

```
void NachenBlaster::doSomething()
{
    ...
    int ch;
    if (getWorld()->getKey(ch))
    {
        // user hit a key during this tick!
        switch (ch)
        {
          case KEY_PRESS_LEFT:
            ... move NachenBlaster ship to the left ...;
            break;
          case KEY_PRESS_RIGHT:
            ... move NachenBlaster ship to the right ...;
            break;
          case KEY_PRESS_SPACE:
            ... add a cabbage in front of the NachenBlaster...;
            break;

          // etc…
        }
    }
    ...
}
```

[1] Hint: Since your NachenBlaster class will need to access the *getKey()* method in the *GameWorld* class (which is the base class for your *StudentWorld* class), your NachenBlaster class (or more likely, one of its base classes) will need a way to obtain a pointer to the *StudentWorld* object it's playing in. If you look at our code example, you'll see how the NachenBlaster's *doSomething()* method first gets a pointer to its world via a call to *getWorld()* (a method in one of its base classes that returns a pointer to a *StudentWorld*), and then uses this pointer to call the *getKey()* method.

Star

Stars don't really do much. They just keep moving toward the left of the screen. Here are the requirements you must meet when implementing a star class.

What a Star Must Do When It Is Created

When it is first created:

1. A star object must have an image ID of IID_STAR.
2. A star must always start at a location as specified by the code that constructs it.
3. A star has a direction of 0.
4. A star has a randomly chosen size in the range .05 and .50.
5. A star has a depth of 3.

What a Star Must Do During a Tick

A star must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the star must move exactly 1 pixel to the left. Once the star reaches the left side of the screen (goes to x=-1 or less), it must set its state to dead so it can be destructed and removed from the level by your *StudentWorld* class.

What a Star Must Do When It Is Attacked or Collides with Another Space Object

Stars cannot be attacked or collide with other actors.

Explosion

When an alien ship is destroyed, you must introduce an explosion object into the game at the same position as the alien ship that was destroyed. Yes, in our game framework, an explosion is implemented as an object just like an alien ship, a projectile, or a goodie.

What an Explosion Must Do When It Is Created

When it is first created:

1. An explosion object must have an image ID of IID_EXPLOSION.
2. An explosion must always start at a location as specified by the code that constructs it.
3. An explosion has a direction of 0.
4. An explosion has an initial size of 1.
5. Explosions have a depth of 0.

What an Explosion Must Do During a Tick

An explosion must be given an opportunity to do something during every tick (in its *doSomething()* method). When given an opportunity to do something during a tick, the explosion must increase its size by a factor of 1.5 over its current size.

After exactly four ticks from the creation of the explosion, the explosion must set its state to dead so it can be destroyed and removed from the level by your *StudentWorld* class.

What an Explosion Must Do When It Is Attacked or Collides with Another Space Object

Explosions cannot be attacked or collide with other actors.

Cabbage

You must create a class to represent a cabbage (or perhaps some abstract type of projectile, of which a cabbage is one type). Here are the requirements you must meet when implementing the cabbage class.

What a Cabbage Must Do When It Is Created

When it is first created:

1. A cabbage object must have an image ID of IID_CABBAGE.
2. A cabbage must have its x,y location specified for it – the NachenBlaster that fires the cabbage must pass in this x,y location when constructing a cabbage object.
3. A cabbage has an initial direction of 0.
4. A cabbage has a size of .5.
5. A cabbage has a depth of 1.

What a Cabbage Must Do During a Tick

Each time a cabbage object is asked to do something (during a tick):

1. The cabbage must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The cabbage must determine if it's flown off of the right side of the screen (its x coordinate is greater than or equal to VIEW_WIDTH), and if so, mark itself as dead so that the StudentWorld object can remove it from the space field. It must do nothing more during the current tick.
3. Otherwise, the cabbage must check to see if it has collided with an alien (see the Euclidean distance check on page 10 of this document). If the cabbage overlaps with an alien (Smoregon, Smallgon or Snagglegon) ship:

- Damage the victim ship appropriately (by causing the object to lose 2 hit points); the damaged object can then deal with this damage in its own unique way (this may result in the damaged object dying/disappearing, a sound effect being played, the user possibly getting points, etc.) Hint: The cabbage can tell the alien object that it has been damaged by calling a method the alien object has (presumably named *sufferDamage* or something similar).
- Set the cabbage's own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
- Do nothing else during the current tick.

4. The cabbage must move itself 8 pixels to the right.
5. The cabbage must rotate its direction by 20 degrees counter-clockwise.
6. Finally, after the cabbage has moved itself, the cabbage must AGAIN check to see if has collided with an alien ship, using the same algorithm described in step #3 above. If so, it must perform the same behavior as described in step #3 (e.g., damage the object, etc.), but does not move any further during this tick.

What a Cabbage Must Do When It Is Attacked

Cabbages can't be attacked, silly. All projectiles will pass right through other projectiles.

Turnip

You must create a class to represent a turnip (or perhaps some abstract type of projectile, of which a turnip is one type). Here are the requirements you must meet when implementing the turnip class.

What a Turnip Must Do When It Is Created

When it is first created:

1. A turnip object must have an image ID of IID_TURNIP.
2. A turnip must have its x,y location specified for it – the alien ship that fires the turnip must pass in this x,y location when constructing a turnip object.
3. A turnip has an initial direction of 0.
4. A turnip has a size of .5.
5. A turnip has a depth of 1.

What a Turnip Must Do During a Tick

Each time a turnip object is asked to do something (during a tick):

1. The turnip must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.

2. The turnip must determine if it's flown off of the left side of the screen (its x coordinate is less than 0), and if so, mark itself as dead so that the StudentWorld object can remove it from the space field. It must do nothing more during the current tick.
3. Otherwise, the turnip must check to see if it has collided with the NachenBlaster (see the Euclidean distance check on page 10 of this document). If the turnip overlaps with the NachenBlaster ship:
   - Damage the NachenBlaster appropriately (by causing the object to lose 2 hit points); the damaged object can then deal with this damage in its own unique way (this may result in the damaged object dying/disappearing, a sound effect being played, the user possibly getting points, etc.) Hint: The turnip can tell the NachenBlaster that it has been damaged by calling a method the NachenBlaster has (presumably named *sufferDamage* or something similar).
   - Set the turnip's own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   - Do nothing else during the current tick.
4. The turnip must move itself 6 pixels to the left.
5. The turnip must rotate its direction by 20 degrees counter-clockwise.
6. Finally, after the turnip has moved itself, the turnip must AGAIN check to see if has collided with the NachenBlaster, using the same algorithm described in step #3 above. If so, it must perform the same behavior as described in step #3 (e.g., damage the object, etc.), but does not move any further during this tick.

What a Turnip Must Do When It Is Attacked

Turnips can't be attacked, silly. All projectiles will pass right through other projectiles.

Flatulence Torpedo

You must create a class to represent a Flatulence Torpedo (or perhaps some abstract type of projectile, of which a Flatulence Torpedo is one type). Here are the requirements you must meet when implementing the Flatulence Torpedo class.

What a Flatulence Torpedo Must Do When It Is Created

When it is first created:

1. A Flatulence Torpedo object must have an image ID of IID_TORPEDO.
2. A Flatulence Torpedo must have its x,y location specified for it – the alien ship or NachenBlaster that fires the Flatulence Torpedo must pass in this x,y location when constructing a Flatulence Torpedo object.
3. A Flatulence Torpedo has a direction of either 0 degrees (if it was fired by the NachenBlaster) or 180 degrees (if it was fired by a Snagglegon).
4. A Flatulence Torpedo has a size of .5.
5. A Flatulence Torpedo has a depth of 1.

What a Flatulence Torpedo Must Do During a Tick

Each time a Flatulence Torpedo object is asked to do something (during a tick):

1. The Flatulence Torpedo must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The Flatulence Torpedo must determine if it's flown off of the left or right sides of the screen (its x coordinate is less than 0 or >= VIEW_WIDTH), and if so, mark itself as dead so that the StudentWorld object can remove it from the space field. It must do nothing more during the current tick.
3. Otherwise, the Flatulence Torpedo must check to see if it has collided with an enemy. If the Flatulence Torpedo was fired by a Snagglegon ship, then its only enemy is the NachenBlaster. Otherwise, if the Flatulence Torpedo was fired by the NachenBlaster then its enemy is any alien ship. See the Euclidean distance check on page 10 of this document to determine how to check for collisions. If the Flatulence Torpedo collides with an enemy ship:
   - Damage the enemy appropriately (by causing the object to lose 8 hit points); the damaged object can then deal with this damage in its own unique way (this may result in the damaged object dying/disappearing, a sound effect being played, the user possibly getting points, etc.) Hint: The Flatulence Torpedo can tell the enemy object that it has been damaged by calling a method the enemy object has (presumably named *sufferDamage* or something similar).
   - Set the Flatulence Torpedo's own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   - Do nothing else during the current tick.
4. The Flatulence Torpedo must move itself 8 pixels to the left (if it was fired by a Snagglegon) or 8 pixels to the right (if it was fired by the NachenBlaster).
5. NOTE: Flatulence Torpedoes do NOT rotate like cabbages or turnips.
6. Finally, after the Flatulence Torpedo has moved itself, the Flatulence Torpedo must AGAIN check to see if has collided with an enemy, using the same algorithm described in step #3 above. If so, it must perform the same behavior as described in step #3 (e.g., damage the object, etc.), but does not move any further during this tick.

What a Flatulence Torpedo Must Do When It Is Attacked

Flatulence Torpedoes can't be attacked, silly. All projectiles will pass right through other projectiles.

Extra Life Goodie

You must create a class to represent an Extra Life Goodie. When the NachenBlaster collides with (picks up) this goodie (see page 10 for how to check for collisions between two objects) it gives the NachenBlaster an extra life! Here are the requirements you must meet when implementing the Extra Life Goodie class.

What an Extra Life Goodie Must Do When It Is Created

When it is first created:

1. An Extra Life Goodie object must have an image ID of IID_LIFE_GOODIE.
2. An Extra Life Goodie must always start at the proper location as specified by the alien ship that drops it.
3. An Extra Life Goodie has a direction of 0.
4. An Extra Life Goodie has a size of .5.
5. An Extra Life Goodie has a depth of 1.

What an Extra Life Goodie Must Do During a Tick

Each time an Extra Life Goodie is asked to do something (during a tick):

1. The Extra Life Goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The Extra Life Goodie must determine if it's flown off the screen, and if so, mark itself as dead so that the StudentWorld object can remove it from the space field. It must do nothing more during the current tick.
3. Otherwise, if the NachenBlaster collides with an Extra Life Goodie, then the Extra Life Goodie must:
    a. Inform the *StudentWorld* object that the user is to receive 100 more points.
    b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    c. Play a sound effect to indicate that the NachenBlaster picked up the goodie: SOUND_GOODIE.
    d. Inform the *StudentWorld* object that the NachenBlaster is to gain one extra life.
    e. Immediately return.
4. The Extra Life Goodie must move itself down AND left by .75 units in each direction, using the *moveTo()* method.
5. The Extra Life Goodie must then again check to see if it collided with the NachenBlaster, and if so, perform the steps outlined in item #3 above.

What an Extra Life Goodie Must Do When It Is Attacked

Extra Life Goodies can't be attacked. Projectiles will pass right over them.

Repair Goodie

You must create a class to represent a Repair Goodie. When the NachenBlaster collides with (picks up) this goodie (see page 10 for how to check for collisions between two objects) it gives the NachenBlaster extra 10 hit points, up to its maximum of 50 hit points! Here are the requirements you must meet when implementing the Repair Goodie class.

What a Repair Goodie Must Do When It Is Created

When it is first created:

1. A Repair Goodie object must have an image ID of IID_REPAIR_GOODIE.
2. A Repair Goodie must always start at the proper location as specified by the alien ship that drops it.
3. A Repair Goodie has a direction of 0.
4. A Repair Goodie has a size of .5.
5. A Repair Goodie has a depth of 1.

What a Repair Goodie Must Do During a Tick

Each time a Repair Goodie is asked to do something (during a tick):

1. The Repair Goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The Repair Goodie must determine if it's flown off the screen, and if so, mark itself as dead so that the StudentWorld object can remove it from the space field. It must do nothing more during the current tick.
3. Otherwise, if the NachenBlaster collides with a Repair Goodie, then the Repair Goodie must:
    a. Inform the *StudentWorld* object that the user is to receive 100 more points.
    b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    c. Play a sound effect to indicate that the NachenBlaster picked up the goodie: SOUND_GOODIE.
    d. Inform the NachenBlaster object that it just got 10 additional hit points (any additional hit points must NOT cause the NachenBlaster to exceed 50 hit points).
    e. Immediately return.
4. The Repair Goodie must move itself down AND left by .75 units in each direction, using the *moveTo()* method.
5. The Repair Goodie must then again check to see if it collided with the NachenBlaster, and if so, perform the steps outlined in item #3 above.

What a Repair Goodie Must Do When It Is Attacked

Repair Goodies can't be attacked. Projectiles will pass right over them.

Flatulence Torpedo Goodie

You must create a class to represent a Flatulence Torpedo Goodie. When the NachenBlaster collides with (picks up) this goodie (see page 10 for how to check for collisions between two objects) it gives the NachenBlaster additional 5 Flatulence Torpedoes! Here are the requirements you must meet when implementing the Flatulence Torpedo Goodie class.

What a Flatulence Torpedo Goodie Must Do When It Is Created

When it is first created:

1. A Flatulence Torpedo Goodie object must have an image ID of IID_TORPEDO_GOODIE.
2. A Flatulence Torpedo Goodie must always start at the proper location as specified by the alien ship that drops it.
3. A Flatulence Torpedo Goodie has a direction of 0.
4. A Flatulence Torpedo Goodie has a size of .5.
5. A Flatulence Torpedo Goodie has a depth of 1.

What a Flatulence Torpedo Goodie Must Do During a Tick

Each time a Flatulence Torpedo Goodie is asked to do something (during a tick):

1. The Flatulence Torpedo Goodie must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The Flatulence Torpedo Goodie must determine if it's flown off the screen, and if so, mark itself as dead so that the StudentWorld object can remove it from the space field. It must do nothing more during the current tick.
3. Otherwise, if the NachenBlaster collides with a Flatulence Torpedo Goodie, then the Flatulence Torpedo Goodie must:
   a. Inform the *StudentWorld* object that the user is to receive 100 more points.
   b. Set its state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
   c. Play a sound effect to indicate that the NachenBlaster picked up the goodie: SOUND_GOODIE.
   d. Inform the NachenBlaster object that it just received 5 Flatulence Torpedoes. There is no maximum number of torpedoes that the NachenBlaster may have.
   e. Immediately return.

4. The Flatulence Torpedo Goodie must move itself down AND left by .75 units in each direction, using the *moveTo()* method.
5. The Flatulence Torpedo Goodie must then again check to see if it collided with the NachenBlaster, and if so, perform the steps outlined in item #3 above.

What a Flatulence Torpedo Goodie Must Do When It Is Attacked

Flatulence Torpedo Goodies can't be attacked. Projectiles will pass right over them.

Smallgon

You must create a class to represent a Smallgon alien ship. Here are the requirements you must meet when implementing the Smallgon class.

What a Smallgon Must Do When It Is Created

When it is first created:

1. A Smallgon object must have an image ID of IID_SMALLGON.
2. A Smallgon must always start at the proper location as passed into its constructor.
3. A Smallgon has a direction of 0.
4. A Smallgon has a size of 1.5.
5. A Smallgon has a depth of 1.
6. A Smallgon starts with exactly 5 * (1 + (current_level_number – 1) * .1) hit points.
7. A Smallgon starts with a flight plan of length 0.
8. A Smallgon starts with a travel speed of 2.0.

What a Smallgon Must Do During a Tick

Each time a Smallgon is asked to do something (during a tick):

1. The Smallgon must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The Smallgon must determine if it's flown off of the left side of the screen (its x coordinate is less than 0), and if so, mark itself as dead so that the StudentWorld class can remove it from the space field. If so, it must do nothing more during the current tick.
3. The Smallgon must check to see if it has collided with a NachenBlaster-fired projectile or the NachenBlaster ship itself (see the Euclidean collision detection approach on page 10). If so, it must follow the steps outlined in the Collision section below.
4. The Smallgon will check to see if it needs a new flight plan because the current flight plan length has reached zero OR the Smallgon has reached the top or

bottom of the screen due to its current flight plan. If either condition is true, the Smallgon will use the following approach to select a new flight plan:

   a. If the Smallgon's y coordinate is greater than or equal to VIEW_HEIGHT-1, then the Smallgon will set its travel direction to down and left.
   b. Otherwise, if the Smallgon's y coordinate is less than or equal to 0, then the Smallgon will set its travel direction to up and left.
   c. Otherwise if the Smallgon's flight plan length is 0, the Smallgon will set its travel direction by randomly selecting a new one from these three choices: due left, up and left, or down and left.
   d. The Smallgon will pick a random new flight plan length in the range [1, 32].

5. If the NachenBlaster is to the left of the Smallgon AND the Smallgon has a y coordinate that is within [-4,4] pixels of the NachenBlaster's y coordinate, then:
   a. There is a 1 in ((20/CurrentLevelNumber)+5) chance during this tick that the Smallgon will:
      i. Fire a turnip toward the NachenBlaster, adding the new turnip 14 pixels to the left of the center of the Smallgon ship and at the Smallgon's y coordinate. Hint: When you create a new turnip object in the proper location, give it to your StudentWorld to manage (e.g., animate) along with the other game objects.
      ii. Play a SOUND_ALIEN_SHOOT sound effect.
      iii. Do nothing else during the current tick.
6. The Smallgon will try to move in its current travel direction (as determined by the last flight plan) and then reduce its flight plan length by 1. If its travel direction is:
   a. Up and left: It must move N pixels up and N pixels left using the *moveTo()* method, where N is the current travel speed.
   b. Down and left: It must move N pixels down and N pixels left using the *moveTo()* method, where N is the current travel speed.
   c. Due left: It must move N pixels left using the *moveTo()* method, where N is the current travel speed.
7. Finally, after the Smallgon ship has moved itself, it must AGAIN check to see if has collided with the NachenBlaster or a NachenBlaster-fired projectile, using the same algorithm described in step #3 above. If so, it must perform the same behavior as described in step #3 (e.g., damage the object, etc.).

<u>What a Smallgon Must Do When It Collides with a Projectile or the NachenBlaster</u>

If a Smallgon collides with a NachenBlaster-fired projectile (or vice versa):

1. The Smallgon ship must be damaged by the projectile as indicated by the projectile (e.g., 2 points of damage from a cabbage, etc.).
2. If the collision results in the Smallgon's hit points reaching zero or below, then the Smallgon must:
   a. Increase the player's score by 250 points.

    b. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    c. Play a SOUND_DEATH sound effect.
    d. Introduce a new explosion object into the space field at the same x,y location as the Smallgon.
3. Otherwise, if the projectile injured the Smallgon ship but didn't destroy the ship, it must play a SOUND_BLAST sound effect.

If a Smallgon collides with the NachenBlaster ship or vice versa, it must:

1. Damage the NachenBlaster appropriately (by causing it to lose 5 hit points) – the NachenBlaster object can then deal with this damage in its own unique way. Hint: The Smallgon object can tell the NachenBlaster that it has been damaged by calling a method the NachenBlaster has (presumably named *sufferDamage* or something similar).
2. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
3. Increase the player's score by 250 points.
4. Play a SOUND_DEATH sound effect.
5. Introduce a new explosion object into the space field at the same x,y coordinates as the Smallgon (see the Explosion section for more details)

Smoregon

You must create a class to represent a Smoregon alien ship. Here are the requirements you must meet when implementing the Smoregon class.

What a Smoregon Must Do When It Is Created

When it is first created:

1. A Smoregon object must have an image ID of IID_SMOREGON.
2. A Smoregon must always start at the proper location as passed into its constructor.
3. A Smoregon has a direction of 0.
4. A Smoregon has a size of 1.5.
5. A Smoregon has a depth of 1.
6. A Smoregon starts with exactly 5 * (1 + (current_level_number – 1) * .1) hit points.
7. A Smoregon starts with a flight plan of length 0.
8. A Smoregon starts with an initial travel speed of 2.0.

What a Smoregon Must Do During a Tick

Each time a Smoregon is asked to do something (during a tick):

1. The Smoregon must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The Smoregon must determine if it's flown off of the left side of the screen (its x coordinate is less than 0), and if so, mark itself as dead so that the StudentWorld class can remove it from the space field. If so, it must do nothing more during the current tick.
3. The Smoregon must check to see if it has collided with a NachenBlaster-fired projectile or the NachenBlaster ship itself (see the Euclidean collision detection approach on page 10). If so, it must follow the steps outlined in the Collision section below.
4. The Smoregon must check to see if it needs a new flight plan because the current flight plan length has reached zero OR the Smoregon has reached the top or bottom of the screen due to its current flight plan. If either condition is true, the Smoregon will use the following approach to select a new flight plan:
    a. If the Smoregon's y coordinate is greater than or equal to VIEW_HEIGHT-1, then the Smoregon will set its travel direction to down and left.
    b. Otherwise, if the Smoregon's y coordinate is less than or equal to 0, then the Smoregon will set its travel direction to up and left.
    c. Otherwise if the Smoregon's flight plan length is 0, the Smoregon will set its travel direction by randomly selecting a new one from these three choices: due left, up and left, or down and left.
    d. The Smoregon will pick a random new flight plan length in the range [1, 32].
5. If the NachenBlaster is to the left of the Smoregon AND the Smoregon has a y coordinate that is within [-4,4] pixels of the NachenBlaster's y coordinate, then:
    a. There is a 1 in ((20/CurrentLevelNumber)+5) chance during this tick that the Smoregon will:
        i. Fire a turnip toward the NachenBlaster, adding the new turnip 14 pixels to the left of the center of the Smoregon ship and at the Smoregon's y coordinate. Hint: When you create a new turnip object in the proper location, give it to your StudentWorld to manage (e.g., animate) along with the other game objects.
        ii. Play a SOUND_ALIEN_SHOOT sound effect.
        iii. Do nothing else during the current tick.
    b. There is a 1 in ((20/CurrentLevelNumber)+5) chance during this tick that the Smoregon will change its flight plan:
        i. It will set its travel direction to due left.
        ii. It will set the length of its flight plan to VIEW_WIDTH steps.
        iii. It will set its travel speed to 5 pixels per tick: Ramming speed!.
        iv. Continue on with the next step.
6. The Smoregon will try to move in its current travel direction (as determined by the last flight plan) and then reduce its flight plan length by 1. If its travel direction is:

    a. Up and left: It must move N pixels up and N pixels left using the *moveTo()* method, where N is the current travel speed.
    b. Down and left: It must move N pixels down and N pixels left using the *moveTo()* method, where N is the current travel speed.
    c. Due left: It must move N pixels left using the *moveTo()* method, where N is the current travel speed.
7. Finally, after the Smoregon ship has moved itself, it must AGAIN check to see if has collided with the NachenBlaster or a NachenBlaster-fired projectile, using the same algorithm described in step #3 above. If so, it must perform the same behavior as described in step #3 (e.g., damage the object, etc.).

<u>What a Smoregon Must Do When It Collides with a Projectile or the NachenBlaster</u>

If a Smoregon collides with a NachenBlaster-fired projectile (or vice versa):

1. The Smoregon ship must be damaged by the projectile as indicated by the projectile (e.g., 2 points of damage from a cabbage, etc.).
2. If the collision results in the Smoregon's hit points reaching zero or below, then the Smoregon must:
    a. Increase the player's score by 250 points.
    b. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    c. Play a SOUND_DEATH sound effect.
    d. Introduce a new explosion object into the space field at the same x,y location as the Smoregon.
    e. There is a 1/3 chance that the destroyed Smoregon ship will drop a goodie. If it decides to drop a goodie, there is a 50% chance that it will be a Repair Goodie, and a 50% chance that it will be a Flatulence Torpedo Goodie. The goodie must be added to the space field at the same x,y coordinates as the destroyed ship.
3. Otherwise, if the projectile injured the Smoregon ship but didn't destroy the ship, it must play a SOUND_BLAST sound effect.

If a Smoregon collides with the NachenBlaster ship or vice versa, it must:

1. Damage the NachenBlaster appropriately (by causing it to lose 5 hit points) – the NachenBlaster object can then deal with this damage in its own unique way. Hint: The Smoregon object can tell the NachenBlaster that it has been damaged by calling a method the NachenBlaster has (presumably named *sufferDamage* or something similar).
2. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
3. Increase the player's score by 250 points.
4. Play a SOUND_DEATH sound effect.
5. Introduce a new explosion object into the space field at the same x,y coordinates as the Smoregon (see the Explosion section for more details)

6. There is a 1/3 chance that the destroyed Smoregon ship will drop a goodie. If it decides to drop a goodie, there is a 50% chance that it will be a Repair Goodie, and a 50% chance that it'll be a Flatulence Torpedo Goodie. The goodie must be added to the space field at the same x,y coordinates as the destroyed ship.

Snagglegon

You must create a class to represent a Snagglegon alien ship. Here are the requirements you must meet when implementing the Snagglegon class.

What a Snagglegon Must Do When It Is Created

When it is first created:

1. A Snagglegon object must have an image ID of IID_SNAGGLEGON.
2. A Snagglegon must always start at the proper location as passed into its constructor.
3. A Snagglegon has a direction of 0.
4. A Snagglegon has a size of 1.5.
5. A Snagglegon has a depth of 1.
6. A Snagglegon starts with exactly 10 * (1 + (current_level_number – 1) * .1) hit points.
7. A Snagglegon starts with an initial travel direction of down and left.
8. A Snagglegon starts with a travel speed of 1.75.

What a Snagglegon Must Do During a Tick

Each time a Snagglegon is asked to do something (during a tick):

1. The Snagglegon must check to see if it is currently alive. If not, then its *doSomething()* method must return immediately – none of the following steps should be performed.
2. The Snagglegon must determine if it's flown off of the left side of the screen (its x coordinate is less than 0), and if so, mark itself as dead so that the StudentWorld class can remove it from the space field. If so, it must do nothing more during the current tick.
3. The Snagglegon must check to see if it has collided with a NachenBlaster-fired projectile or the NachenBlaster ship itself (see the Euclidean collision detection approach on page 10). If so, it must follow the steps outlined in the Collision section below.
4. The Snagglegon will check to see if it needs a new flight plan because it has reached the top or bottom of the screen due to its current flight plan. If either condition is true, the Snagglegon will use the following approach to select a new flight plan:

    a. If the Snagglegon's y coordinate is greater than or equal to VIEW_HEIGHT-1, then the Snagglegon will set its travel direction to down and left.
    b. Otherwise, if the Snagglegon's y coordinate is less than or equal to 0, then the Snagglegon will set its travel direction to up and left.
5. If the NachenBlaster is to the left of the Snagglegon AND the Snagglegon has a y coordinate that is within [-4,4] pixels of the NachenBlaster's y coordinate, then:
    a. There is a 1 in ((15/CurrentLevelNumber)+10) chance during this tick that the Snagglegon will:
        i. Fire a Flatulence Torpedo toward the NachenBlaster, adding the new torpedo 14 pixels to the left of the center of the Snagglegon ship and at the Snagglegon's y coordinate. Hint: When you create a new Flatulence Torpedo object in the proper location, give it to your StudentWorld to manage (e.g., animate) along with the other game objects.
        ii. Play a SOUND_TORPEDO sound effect.
        iii. Do nothing else during the current tick.
6. The Snagglegon will try to move in its current travel direction. If its travel direction is:
    a. Up and left: It must move N pixels up and N pixels left using the *moveTo()* method, where N is the current travel speed.
    b. Down and left: It must move N pixels down and N pixels left using the *moveTo()* method, where N is the current travel speed.
7. Finally, after the Snagglegon ship has moved itself, it must AGAIN check to see if has collided with the NachenBlaster or a NachenBlaster-fired projectile, using the same algorithm described in step #3 above. If so, it must perform the same behavior as described in step #3 (e.g., damage the object, etc.).

<u>What a Snagglegon Must Do When It Collides with a Projectile or the NachenBlaster</u>

If a Snagglegon collides with a NachenBlaster-fired projectile (or vice versa):

1. The Snagglegon ship must be damaged by the projectile as indicated by the projectile (e.g., 2 points of damage from a cabbage, etc.).
2. If the collision results in the Snagglegon's hit points reaching zero or below, then the Snagglegon must:
    a. Increase the player's score by 1000 points.
    b. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
    c. Play a SOUND_DEATH sound effect.
    d. Introduce a new explosion object into the space field at the same x,y location as the Snagglegon.
    e. There is a 1/6 chance that the destroyed Snagglegon ship will drop an Extra Life goodie. The goodie must be added to the space field at the same x,y coordinates as the destroyed ship.

3. Otherwise, if the projectile injured the Snagglegon ship but didn't destroy the ship, it must play a SOUND_BLAST sound effect.

If a Snagglegon collides with the NachenBlaster ship or vice versa, it must:

1. Damage the NachenBlaster appropriately (by causing it to lose 15 hit points) – the NachenBlaster object can then deal with this damage in its own unique way. Hint: The Snagglegon object can tell the NachenBlaster that it has been damaged by calling a method the NachenBlaster has (presumably named *sufferDamage* or something similar).
2. Set its own state to dead (so that it will be removed from the game by the *StudentWorld* object at the end of the current tick).
3. Increase the player's score by 1000 points.
4. Play a SOUND_DEATH sound effect.
5. Introduce a new explosion object into the space field at the same x,y coordinates as the Snagglegon (see the Explosion section for more details)
6. There is a 1/6 chance that the destroyed Snagglegon ship will drop an Extra Life goodie. The goodie must be added to the space field at the same x,y coordinates as the destroyed ship.

Object Oriented Programming Tips

Before designing your base and derived classes for Project 3 (or for that matter, any other school or work project), make sure to consider the following best practices. These tips will help you not only write a better object oriented program, but also help you get a better grade on P3!

Try your best to leverage the following best practices in your program, but don't be overly obsessive – it's rarely possible to make a set of perfect classes. That's often a waste of time. Remember, the best is the enemy of the good (enough).

Here we go!

1. **Avoid using dynamic cast to identify common types of objects. Instead add methods to check for various classes of behaviors:**

   Don't do this:

```
void decideWhetherToAddOil(Actor *p)
{
    if (dynamic_cast<BadRobot *>(p) != nullptr ||
         dynamic_cast<GoodRobot *>(p) != nullptr ||
         dynamic_cast<ReallyBadRobot *>(p) != nullptr ||
         dynamic_cast<StinkyRobot *>(p) != nullptr)
         p->addOil();
}
```

Do this instead:

```
void decideWhetherToAddOil (Actor *p)
{
      // define a common method, have all Robots return true, all
      // biological organisms return false
   if (p->requiresOilToOperate())
        p->addOil();
}
```

2. Always avoid defining specific isParticularClass() methods for each type of object. Instead add methods to check for various common behaviors that span multiple classes:

Don't do this:

```
void decideWhetherToAddOil (Actor *p)
{
       if (p->isGoodRobot() || p->isBadRobot() || p->isStinkyRobot())
              p->addOil();
}
```

Do this instead:

```
void decideWhetherToAddOil (Actor *p)
{
      // define a common method, have all Robots return true, all
      // biological organisms return false
    if (p->requiresOilToOperate())
      p->addOil();
}
```

3. If two related subclasses (e.g., SmellyRobot and GoofyRobot) each directly define a member variable that serves the same purpose in both classes (e.g., m_amountOfOil), then move that member variable to the common base class and add accessor and mutator methods for it to the base class. So the Robot base class should have the m_amountOfOil member variable defined once, with getOil() and addOil()functions, rather than defining this variable directly in both SmellyRobot and GoofyRobot.

Don't do this:

```
class SmellyRobot: public Robot
{
       …
  private:
    int m_oilLeft;
};

class GoofyRobot: public Robot
{
       …
  private:
    int m_oilLeft;
```

```
};
```

Do this instead:

```
class Robot
{
  public:
    void addOil(int oil) { m_oilLeft += oil; }
    int getOil() const { return m_oilLeft; }
  private:
    int m_oilLeft;
};
```

4. **Never make any class's data members public or protected. You may make class constants public, protected or private.**

5. **Never make a method public if it is only used directly by other methods within the same class that holds it. Make it private or protected instead.**

6. **Your StudentWorld methods should never return a vector, list or iterator to StudentWorld's private game objects or pointers to those objects. Only StudentWorld should know about all of its game objects and where they are. Instead StudentWorld should do all of the processing itself if an action needs to be taken on one or more game objects that it tracks.**

Don't do this:

```
class StudentWorld
{
  public:
    vector<Actor*> getActorsThatCanBeZapped(int x, int y)
    {
       …      // create a vector with a actor pointers and return it
    }
};

class NastyRobot
{
  public:
    virtual void doSomething()
    {
             …
      vector<Actor*> v;
      vector<Actor*>::iterator p;

      v = studentWorldPtr->getActorsThatCanBeZapped(getX(), getY());
      for (p = actors.begin(); p != actors.end(); p++)
        p->zap();

       }
};
```

Do this instead:

```
class StudentWorld
{
```

```
  public:
    void zapAllZappableActors(int x, int y)
    {
      for (p = actors.begin(); p != actors.end(); p++)
        if (p->isAt(x,y) && p->isZappable())
          p->zap();
    }
};

class NastyRobot
{
  public:
    virtual void doSomething()
    {
              …
      studentWorldPtr->zapAllZappableActors(getX(), getY());
    }
};
```

7. If two subclasses have a method that shares some common functionality, but also has some differing functionality, use an auxiliary method to factor out the differences:

Don't do this:

```
class StinkyRobot: public Robot
{
       …
  public:
    virtual void doDifferentiatedStuff()
    {
      doCommonThingA();
      passStinkyGas();
      pickNose();
      doCommonThingB();
    }
};

class ShinyRobot: public Robot
{
       …
  public:
    virtual void doDifferentiatedStuff()
    {
      doCommonThingA();
      polishMyChrome();
      wipeMyDisplayPanel();
      doCommonThingB();
    }
};
```

Do this instead:

```
class Robot
{
  public:
    virtual void doSomething()
    {
        // first do the common thing that all robots do
      doCommonThingA();
```

```
        // then call a virtual function to do the differentiated stuff
      doDifferentiatedStuff();

        // then do the common final thing that all robots do
      doCommonThingB();
    }

  private:
    virtual void doDifferentiatedStuff() = 0;
};

class StinkyRobot: public Robot
{
      …
  private:
       // define StinkyRobot's version of the differentiated function
    virtual void doDifferentiatedStuff()
    {
        // only Stinky robots do these things
      passStinkyGas();
      pickNose();
    }
};

class ShinyRobot: public Robot
{
      …
  private:
      // define ShinyRobot's version of the differentiated function
    virtual void doDifferentiatedStuff()
    {
        // only Shiny robots do these things
      polishMyChrome();
      wipeMyDisplayPanel();
    }
};
```

Yes, it is legal for a derived class to override a virtual function that was declared private in the base class. (It's not trying to *use* the private member function; it's just defining a new function.)

Don't know how or where to start? Read this!

When working on your first large object oriented program, you're likely to feel overwhelmed and have no idea where to start; in fact, it's likely that many students won't be able to finish their entire program. Therefore, it's important to attack your program piece by piece rather than trying to program everything at once.

Students who try to program everything at once rather than program incrementally almost always fail to solve CS32's project 3, so don't do it!

Instead, try to get one thing working at a time. Here are some hints:

1. When you define a new class, try to figure out what public member functions it should have. Then write dummy "stub" code for each of the functions that you'll fix later:

```
class Foo
{
  public:
       int chooseACourseOfAction() { return 0; }      // dummy version
};
```

   Try to get your project compiling with these dummy functions first, then you can worry about filling in the real code later.
2. Once you've got your program compiling with dummy functions, then start by replacing one dummy function at a time. Update the function, rebuild your program, test your new function, and once you've got it working, proceed to the next function.
3. **Make backups of your working code frequently. Any time you get a new feature working, make a backup of all your .cpp and .h files just in case you screw something up later.**

BACK UP YOUR .CPP AND .H FILES TO A REMOVABLE DEVICE OR TO ONLINE STORAGE JUST IN CASE YOUR COMPUTER CRASHES!

If you use this approach, you'll always have something working that you can test and improve upon. If you write everything at once, you'll end up with hundreds of errors and just get frustrated! So don't do it.

Building the Game

The game assets (i.e., image and sound files) are in a folder named *Assets*. The way we've written the main routine, your program will look for this folder in a standard place (described below for Windows and macOS). A few students may find that their environment is set up in a way that prevents the program from finding the folder. If that happens to you, change the string literal `"Assets"` in *main.cpp* to the full path name of wherever you choose to put the folder (e.g., `"Z:/NachenBlaster/Assets"` or `"/Users/fred/NachenBlaster/Assets"`).

To build the game, follow these steps:

For Windows

Unzip the NachenBlaster-skeleton-windows.zip archive into a folder on your hard drive. Double-click on NachenBlaster.sln to start Visual Studio.

If you build and run your program from within Visual Studio, the Assets folder should be in the same folder as your *.cpp* and *.h* files. On the other hand, if you launch the program

by double-clicking on the executable file, the Assets folder should be in the same folder as the executable.

For macOS

Unzip the NachenBlaster-skeleton-mac.zip archive into a folder on your hard drive. Double-click on our provided NachenBlaster.xcodeproj to start Xcode.

If you build and run your program from within Xcode, the Assets directory should be in the directory yourProjectDir*/DerivedData/*yourProjectName*/Build/Products/Debug* (e.g., */Users/fred/NachenBlaster/DerivedData/NachenBlaster/Build/Products/Debug*). On the other hand, if you launch the program by double-clicking on the executable file, the Assets directory should be in your home directory (e.g., */Users/fred*).

What to Turn In

Part #1 (20%)

Ok, so we know you're scared to death about this project and don't know where to start. So, we're going to incentivize you to work incrementally rather than try to do everything all at once. For the first part of Project 3, your job is to build a really simple version of the NachenBlaster game that implements maybe 15% of the overall project. You must program:

1. A class that can serve as the base class for all of your game's actors (e.g., the NachenBlaster, all types of aliens, goodies, stars, projectiles, etc.):
    i. It must have a simple constructor.
    ii. It must be derived from our GraphObject class.
    iii. It must have a member function named *doSomething()* that can be called to cause the actor to do something.
    iv. You may add other public/private member functions and private data members to this base class, as you see fit.
2. A Star class, derived in some way from the base class described in 1 above:
    i. It must implement the specifications described in the Star section above (e.g., causing each star to move left during each tick of the game, destroying the star when it flies off the left side of the screen).
    ii. You may add any public/private member functions and private data members to your Star class as you see fit, so long as you use good object oriented programming style (e.g., you must not duplicate functionality across classes).
3. A limited version of your NachenBlaster class, derived in some way from the base class described in 1 above (either directly derived from the base class, or derived from some other class that is somehow derived from the base class):

i. It must have a constructor that initializes the NachenBlaster – see the NachenBlaster section for more details on where to initialize the NachenBlaster.
ii. It must have an Image ID of IID_NACHENBLASTER.
iii. It must have a limited version of a *doSomething()* method that lets the user pick a direction by hitting a directional key. If the NachenBlaster hits a directional key during the current tick and this will not cause the NachenBlaster to move off of the space field, it updates the NachenBlaster's location appropriately. All this *doSomething()* method has to do is properly adjust the NachenBlaster's x,y coordinates, and our graphics system will automatically animate its movement it around the space field!
iv. You may add other public/private member functions and private data members to your NachenBlaster class as you see fit, so long as you use good object oriented programming style (e.g., you must not duplicate functionality across classes).

4. A limited version of the *StudentWorld* class.
    i. Add any private data members to this class required to keep track of Stars as well as the NachenBlaster object. You may ignore all other items in the game such as Smallgons, projectiles, goodies, etc. for Part #1.
    ii. Implement a constructor for this class that initializes your data members.
    iii. Implement a destructor for this class that frees any remaining dynamically allocated data, if any, that has not yet been freed at the time the *StudentWorld* object is about to destroyed.
    iv. Implement the *init()* method in this class. It must create the NachenBlaster and insert it into the space field at the proper starting location, as detailed in the spec above. It must also create all of the Stars and add them to the space field as specified in the spec above.
    v. Implement the *move()* method in your *StudentWorld* class. During each tick, it must ask your NachenBlaster and Stars to do something. Your *move()* method need not check to see if the NachenBlaster has died or not; you may assume for Part #1 that the NachenBlaster cannot die. Your *move()* method does not have to deal with any actors other than the NachenBlaster and the Stars. During each tick, the move() method may also need to introduce new stars onto the far right side of the screen, as detailed in the spec above.
    vi. Implement a *cleanup()* method that frees any dynamically allocated data that was allocated during calls to the *init()* method or the *move()* method (i.e., it should delete all your allocated Stars and the NachenBlaster). Note: Your *StudentWorld* class must have both a destructor and the *cleanUp()* method even though they likely do the same thing (in which case the destructor could just call *cleanup()*).

As you implement these classes, repeatedly build your program – you'll probably start out with lots of errors… Relax and try to remove them and get your program to run. (Historical note: A UCLA student taking CS 131 once got 1,800 compilation errors when compiling a 900-line class project written in the Ada programming language. His name was Carey Nachenberg.)

You'll know you're done with Part #1 when your program builds and does the following: When it runs and the user hits Enter to begin playing, it displays a space field with the NachenBlaster in its proper starting position and a bunch of stars in the background. If your classes work properly, you should be able to move the NachenBlaster around the space field using the directional keys, with the NachenBlaster displayed over the foreground of any stars it passes by; the stars drift to the left.

Your Part #1 solution may actually do more than what is specified above; for example, if you are further along in the project, and what you have builds and has at least as much functionality as what's described above, then you may turn that in instead.

Note, the Part #1 specification above doesn't require you to implement any Smallgons, Smoregons, Snagglegons, goodies, projectiles, etc. (unless you want to). You may do these unmentioned items if you like but they're not required for Part #1. **However, if you add additonal functionality, make sure that your NachenBlaster, Star, and StudentWorld classes still work properly and that your program still builds and meets the requirements stated above for Part #1!**

If you can get this simple version working, you'll have done a bunch of the hard design work. You'll probably still have to change your classes a lot to implement the full project, but you'll have done most of the hard thinking.

What to Turn In For Part #1

You must turn in your source code for the simple version of your game, which **must build without errors** under either Visual Studio or Xcode. We may also devise a simple test framework that runs under g32; if we do, your code **must build without errors** in that framework. If it does not also run without errors, that indicates some fundamental problem that will probably cost you a lot of points. You will turn in a zip file containing nothing more than these four files:

```
Actor.h             // contains base, NachenBlaster, and Star class declarations
                    //    as well as constants required by these classes
Actor.cpp           // contains the implementation of these classes
StudentWorld.h      // contains your StudentWorld class declaration
StudentWorld.cpp    // contains your StudentWorld class implementation
```

You will not be turning in any other files – we'll test your code with our versions of the the other .cpp and .h files. **Therefore, your solution must NOT modify any of our files**

or you will receive zero credit! (Exception: You may modify the string literal `"Assets"` in *main.cpp*.) You will not turn in a report for Part #1; we will not be evaluating Part #1 for program comments, documentation, or test cases; all that matters for Part #1 is correct behavior for the specified subset of the requirements.

Part #2 (80%)

After you have turned in your work for Part #1 of Project 3, we will discuss one possible design for this assignment. For the rest of this project, you are welcome to continue to improve the design that you came up with for Part #1, **or you can use the design we provide**.

In Part #2, your goal is to implement a fully working version of the NachenBlaster game, which adheres exactly to the functional specification provided in this document.

What to Turn In For Part #2

You must turn in your source code for your game, which **must build without errors** under either Visual Studio or Xcode. We may also devise a simple test framework that runs under g32; if we do, your code **must build without errors** in that framework. If it does not also run without errors, that indicates some fundamental problem that will probably cost you a lot of points. You will turn in a zip file containing nothing more than these five files:

```
Actor.h             // contains declarations of your actor classes
                    //     as well as constants required by these classes
Actor.cpp           // contains the implementation of these classes
StudentWorld.h      // contains your StudentWorld class declaration
StudentWorld.cpp    // contains your StudentWorld class implementation

report.docx, report.doc, or report.txt  // your report (10% of your grade)
```

You will not be turning in any other files – we'll test your code with our versions of the the other .cpp and .h files. **Therefore, your solution must NOT modify any of our files or you will receive zero credit!** (Exception: You may modify the string literal `"Assets"` in *main.cpp*.)

You must turn in a report that contains the following:

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 sneeze() function in my base Actor class because all actors in

NachenBlaster are able to sneeze, and each type of actor sneezes in a different way."

2. A list of all functionality that you failed to finish as well as known bugs in your classes, e.g. "I didn't implement the Turnip class." or "My Snagglegon doesn't work correctly yet so I treat it like a Smoregon right now."
3. A list of other design decisions and assumptions you made; e.g., "It was not specified what to do in situation X, so this is what I decided to do."
4. A description of how you tested each of your classes (1-2 paragraphs per class).

FAQ

Q: The specification is silent about what to do in a certain situation. What should I do?
A: Play with our sample program and do what it does. Use our program as a reference. If neither the specification nor our program makes it clear what to do, do whatever seems reasonable and document it in your report. **If the specification is unclear, but your program behaves like our demonstration program, YOU WILL NOT LOSE POINTS!**

Q: What should I do if I can't finish the project?!
A: Do as much as you can, and whatever you do, make sure your code builds! If we can sort of play your game, but it's not complete or perfect, that's better than it not even building!

Q: Where can I go for help?
A: Try TBP/HKN/UPE – they provide free tutoring and can help your with your project!

Q: Can I work with my classmates on this?
A: You can discuss general ideas about the project, but don't share source code with your classmates. Also don't help them write their source code.

GOOD LUCK!