Assignment #3: Animation, Collision Detection, and Sound

For this assignment you are to extend your game from Assignment #2 (A2) to support additional features including collision detection and sound.

Animation

For project 3 you must animate your helicopters and birds. You may add other animation as desired, however, your project must implement spinning rotors, your helicopters must point in the direction of their heading and your birds must implement flapping wings.

The Helicopter should be composed of two separate images. The helicopter body or fuselage, and the helicopter blade.

- The blade will appear to rotate proportionately with respect to the current speed of the helicopter.
- At the start of the game the Helicopter should be stationary on the “home” helipad and the blade should be motionless. Note that this is slightly different from the last project. The “home” pad is in addition to the first pad that must be reached. The blade should then ramp up to a minimum rotational speed. This minimum rotational speed corresponds to the helicopters speed of zero.
    - The helicopter cannot move until the minimum rotational speed is met.
    - The rotational speed must increment linearly from zero to minimum rotational speed over a period of at least a second. The time is not critical, make it look good.
- The rotational speed of the blade is then proportional to the speed of the helicopter with the minimum rotational speed correlating with a helicopter speed of zero and the maximum rotational speed corresponding to maximum helicopter speed.
    - The simplest thing to do here is a linear ramp, however, you may use whatever scaling that you think looks good.

The main body of the helicopter must also rotate and always face in the direction of the helicopter’s heading.

- While the helicopter can still rotate while at a speed of zero, the helicopter cannot rotate until the blade reaches minimum rotational speed.

The Bird must have at least three images used in a simple sprite that makes it appear that the wings are flapping. The wings should flap fast enough to appear smooth. This is typically at least three full rotations through all images a second, however, it can vary based on your images and the effect that you’re going for.

Collision Detection and Response

After invoking `move()` for all movable objects, your Tick method must determine if there are any collisions between objects, and if so to perform the appropriate "collision response".

Start your process of handling collision detection and response is by having `GameObject` class implement a new interface called "`ICollider`" which declares two methods:

1. `boolean collidesWith(GameObject otherObject)`
2. `void handleCollision(GameObject otherObject)`

These are intended for performing collision detection and response, respectively. In the previous assignment, collisions were caused by pressing one of the "pretend collision buttons" (i.e., "Collide With NPH", "Collide With Base", "Collide With Bird", "Collide with Refueling blimp") and the objects involved in the collision were chosen arbitrarily. Now, the type of collision will be detected automatically during collision detection, so the pretend collision buttons are no longer needed and should be removed. Collision detection will require objects to check to see if they have collided with other objects, so the actual collisions will no longer be arbitrary, but will correspond to actual collisions in the game world. There are more hints regarding collision detection in the notes below.

Collision response (that is, the specific action taken by an object when it collides with another object) will be similar as before. Hence**,** `handleCollision()` method of a game object should call the appropriate collision handling method in `GameWorld` from the previous assignment. Collisions also generate a *sound* (see below) and thus, collision handling methods in `GameWorld` should be updated accordingly.
In addition to the player Helicopter, NPHs can also collide with other NPHs, skyscrapers, birds, and refueling blimps. The effects of these collisions on an NPH would be the same as the effects caused by these collisions on the player Helicopter (i.e., collision with other NPH would increase damage level and fade color of both NPHs, collision with base would increase the *lastSkyScreaperReached* value of the NPH given that the base number is one more than the current *lastSkyScreaperReached* value, collision with a bird would increase the damage level of the NPH, collision with an refueling blimp would increase NPH's energy level, reduce capacity of the refueling blimp to zero, fade the color of the refueling blimp, and add a new refueling blimp).

Since we now automatically detect collisions, we do not need to (and thus, we should not) assign the next *lastSkyScreaperReached* value to NPHs each time "Change Strategies" button is hit. This value of the NPH will now be updated as a result of collisions with the bases. In addition, although we still assume that NPHs never run out of energy, collision between a NPH and an refueling blimp would increase the energy level of the NPH.

Visitor Pattern (Extra Credit)

Once you have this working you may want to *refactor your code to use the Visitor Pattern* for collision detection. With the visitor pattern the need to determine they type of the argument within **handleCollsion** and each object as you will invoke a visitor with the specific argument types of the two objects that are colliding.

Sound

You may add as many sounds into your game as you wish. However, you must implement particular, clearly different sounds for *at least* the following situations:

1. when Helicopters collides, e.g., a player Helicopter-NPH or NPH-NPH collision happens (such as a crash sound),
2. when a Helicopter (NPH/player Helicopter) collides with an refueling blimp (such as an electric charge sound),
3. any collision which results in the player Helicopter losing a life (such as an explosion or dying sound),
4. some sort of appropriate background sound that loops continuously during animation.

Optionally

5. You may add the sound of the helicopter flying if you wish, some thoughts on this
   a. The speed of the helicopter could change the pitch or timbre of the sound
   b. NPHs should also make a sound that may get louder as the NPH gets closer to the player.
6. Blimps may also make a sound, perhaps a low droning sound that gets louder as you get closer
7. Birds could make sounds
   a. These could be played somewhat randomly in time, again, getting louder as the birds get closer.
8. Music could change as the game becomes more tense, perhaps when a number of NPHs get closer than they should.

Sounds should only be played if the "Sound" attribute is "On". Note that except for the "background" and helicopter rotor/engine sounds, sounds are played as a result of executing a collision. Hence, you must play these sounds in collision methods in **`GameWorld`**.

You may use any sounds you like, as long as I can show the game to the Dean and your mother (in other words, they are not disgusting or obscene). Short, unique sounds tend to improve game playability by avoiding too much confusing sound overlap. Do not use copyrighted sounds. You may search the web to find these non-copyrighted sounds (e.g., **`www.findsounds.com`**).

You must copy the sound files directly under the *src* directory of your project for CN1 to locate them. You should add **`Sound`** and **`BGSound`** classes to your project to add a capability for playing regular and looping (e.g., background) sounds, respectively, as discussed in the lecture notes. These classes encapsulate given sound files by making use of **`InputStream`**, **`MediaManager, and Media`** build-in CN1 classes. In addition to these built-in classes, **`BGSound`** also utilizes **`Runnable`** build-in CN1 interface. You should create a single sound object in **`GameWorld`** for each audio file and when you need to play the same sound file, you should use this single instance (e.g., all Helicopter and refueling blimp collisions should use the same sound object).

Additional Notes

1. All requirements of the previous labs that can be followed for this lab are still applicable. E.g., you may not use the gui editor.
2. Because the sound can be turned on and off by the user (using the menu), and also turns on/off automatically with the pause/play button, you will need to test the various combinations. For example, turning off sound, then pressing pause, then pressing play, should result in the sound ***not*** coming back on. There are several sequences to test.
3. When two objects collide handling the collision should be done only once. For instance, when two helicopters collide (i.e., helicopter1 and helicopter2) the helicopters damage level should be increased only once, not twice. In the two nested loops of collision detection, **helicopter1.collidesWith(helicopter2)** and **helicopter2.collidesWith(helicopter1)** will both return true. However, if we handle the collision with **helicopter1.handleCollision(helicopter2)** we should not handle it again by calling **helicopter2.handleCollision(helicopter1)**. This problem is complicated by the fact that in most cases the same collision will be detected repeatedly as one object passes through the other object. Another complication is that more than two objects can collide simultaneously. One straight-forward way of solving this complicated problem, is to have each collidable object (that may involve in such a problem) keep a list of the objects that it is already colliding with. An object can then skip the collision handling for objects it is already colliding with. Of course, you'll also have to remove objects from the list when they are no longer colliding.

   Implementation of this solution would require you to have a **`Vector`** (or **`Arraylist`**) for each collidable object (i.e., all game objects in Skymail3k game) which we will call as "collision vector". When collidable object obj1 collides with obj2, right after handling the collision, you need to add obj2 to collision vector of obj1. If obj2 is also a collidable object, you also need to add obj1 to collision vector of obj2. Each time you check for possible collisions (in each clock tick, after the moving the objects) you need to update the collision vectors. If the obj1 and obj2 are no longer colliding, you need to remove them from each other's collision vectors. You can use **`remove()`** method of **`Vector`** to remove an object from a collision vector. If two objects are still colliding (passes through each other), you should not add them again to the collision vectors. You can use **`contains()`** method of **`Vector`** to check if the object is already in the collision vector or not. The **`contains()`** method is also useful for deciding whether to handle collision or not. For instance, if the collision vector of obj2 already contains obj1 (or collision vector of obj1 already contains obj2), it means that collision between obj1 and obj2 is already handled and should not be handled again.

Deliverables

See the Canvas assignment for submission details. *<u>All submitted work must be strictly your own</u>!*