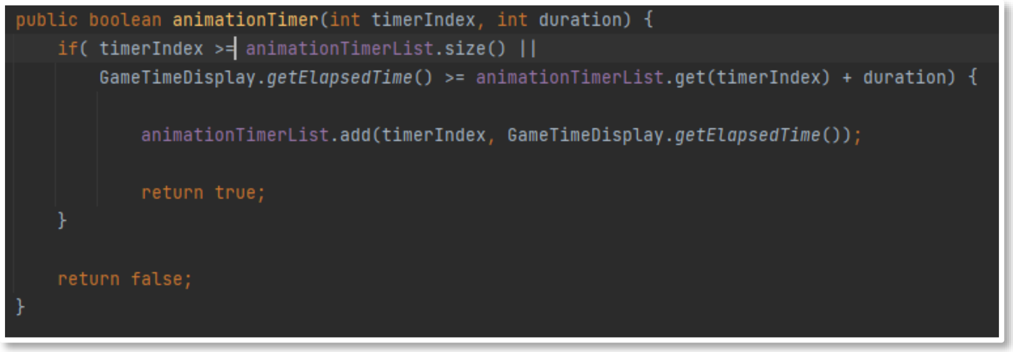
Kendall Helms
AS2 – Project Writeup
CSC 133 – 04
5/14/21

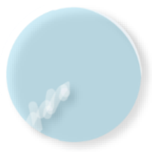
SkyMail3000 v 0.3.0
Assignment 3 Project Writeup

Animation – Animation in this iteration of the project is a refinement of that of the last iteration. The general improvement was a move from tick-based to time-based triggering, which allows consistency even when the tick rate varies as a result of processor load, as well as for more intuitive programming. To assist with this, a new method, *animationTimer()* was written into the *GameObject* class, which keeps track of animation times in a list, and determines whether a certain duration has passed, returning a boolean.



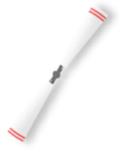
```
public boolean animationTimer(int timerIndex, int duration) {
    if( timerIndex >= animationTimerList.size() ||
        GameTimeDisplay.getElapsedTime() >= animationTimerList.get(timerIndex) + duration) {

            animationTimerList.add(timerIndex, GameTimeDisplay.getElapsedTime());

            return true;
    }

    return false;
}
```

- ***Helicopter –*** The *Helicopter*'s rotor, previously represented by a low-alpha circle, is now drawn as a two-bladed rotor which spins according to the spec. The rotor is represented by two images – one for slow rotation prior to spinup, and another for after, which represents motion blur using alpha and an augmented shape.



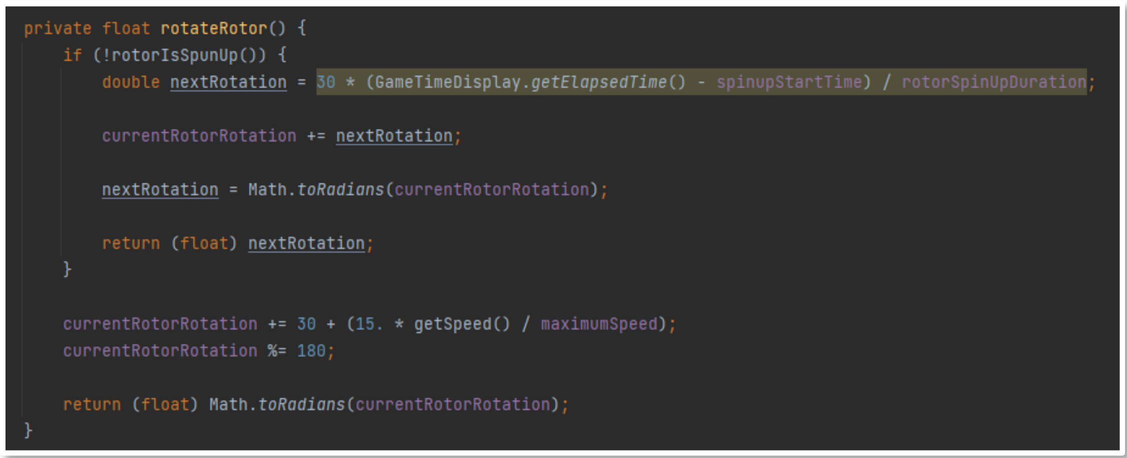
Old rotor design

New rotor design - 1



New rotor design - 2

The method of spinning up the rotor before the start of play is fairly simple. Spinup occurs over a set period of time – currently 1.5 seconds. Each tick of the game calls a method which checks to see if this time has elapsed yet. If it hasn't, the method adds a certain number of degrees to the rotation of the rotor, in proportion to how much of the spinup period has elapsed. The closer to the end of spinup, the faster the rotor will rotate. This gives a smooth-looking animation and it represents a real helicopter rotor pretty convincingly. Once this period is passed, the draw method moves from the first to second of the new rotor images. From this point forward, the speed of rotation is determined to be the rotation rate at the end of spinup plus a certain amount in proportion to the forward speed of the helicopter. The rotor rotation method is shown below.

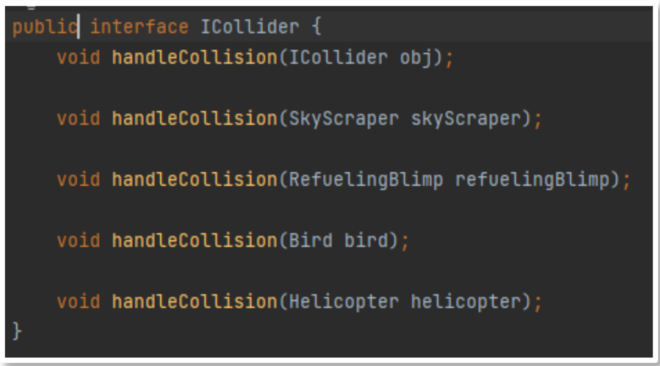


```
private float rotateRotor() {
    if (!rotorIsSpunUp()) {
        double nextRotation = 30 * (GameTimeDisplay.getElapsedTime() - spinupStartTime) / rotorSpinUpDuration;

        currentRotorRotation += nextRotation;

        nextRotation = Math.toRadians(currentRotorRotation);

        return (float) nextRotation;
    }

    currentRotorRotation += 30 + (15. * getSpeed() / maximumSpeed);
    currentRotorRotation %= 180;

    return (float) Math.toRadians(currentRotorRotation);
}
```

Animation of the helicopter's body has not changed from the last iteration.

- ***Birds* –** For the most part, *Birds* are unchanged from the previous assignment. The greatest difference is that the filmstrip animation used previously is now time-driven, rather than tick-driven as referenced earlier.
- ***RefuelingBlimps* –** These now fade away before despawning, using the *Image.modifyAlpha()* method in CN1.

***Collisions* –** Collision was an interesting problem in this assignment, and I believe I came up with an interesting solution. *GameObjects* now implement the *ICollider* interface as per the spec. My implementation of *ICollider* includes only an overloaded *handleCollision()* method; one for each concrete *GameObject* type taken as an argument, and a final one which takes any *ICollider*. *GameObject* implements each of those that takes a concrete type as an empty method,



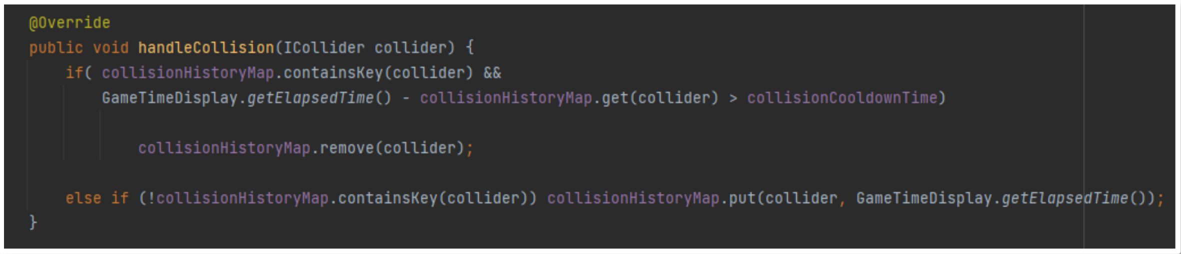
```
public interface ICollider {
    void handleCollision(ICollider obj);

    void handleCollision(SkyScraper skyScraper);

    void handleCollision(RefuelingBlimp refuelingBlimp);

    void handleCollision(Bird bird);

    void handleCollision(Helicopter helicopter);
}
```

allowing its extensions to only incorporate a *handleCollision()* method for a type if it requires a specific behavior. All concrete objects are required to override the *handleCollision(ICollider obj)* method. When a collision is detected, the detection method calls the *handleCollision(ICollider obj)* methods of both colliding objects with the other as its argument. Then, each of these methods calls *obj.handleCollision(this)* to finally get to the desired collision handling methods in each object. I think this may be a slightly unorthodox approach – it's something like a modified visitor pattern in which objects play the role of both the visitor and the visited. This design allows collision behavior specific to each object to be implemented only within that object. In other words, things like sounds that only a helicopter makes when it hits another object are only accessed from within the helicopter.

For collision detection, I implemented a class of static methods called *CollisionDetector*. The methods contained in this class are utilities for detecting collisions of different shapes. To assist with this, each concrete type implements either *IRectangle* or *ICircle* – extenstions of *ICollider*. These interfaces include methods to assist in calculating if these objects overlap. *CollisionDetector.collidesWith()* takes two *IColliders* and determines whether they implement one shape interface or the other, then sends them to the proper private method to determine if they have collided.

The *CollisionDetector* is used in a method called by *GameWorld.tick()*, aptly named *detectCollisions()*. This method checks all objects against each other, storing the consumed objects in a collection which is referenced in each iteration to prevent double collisions.

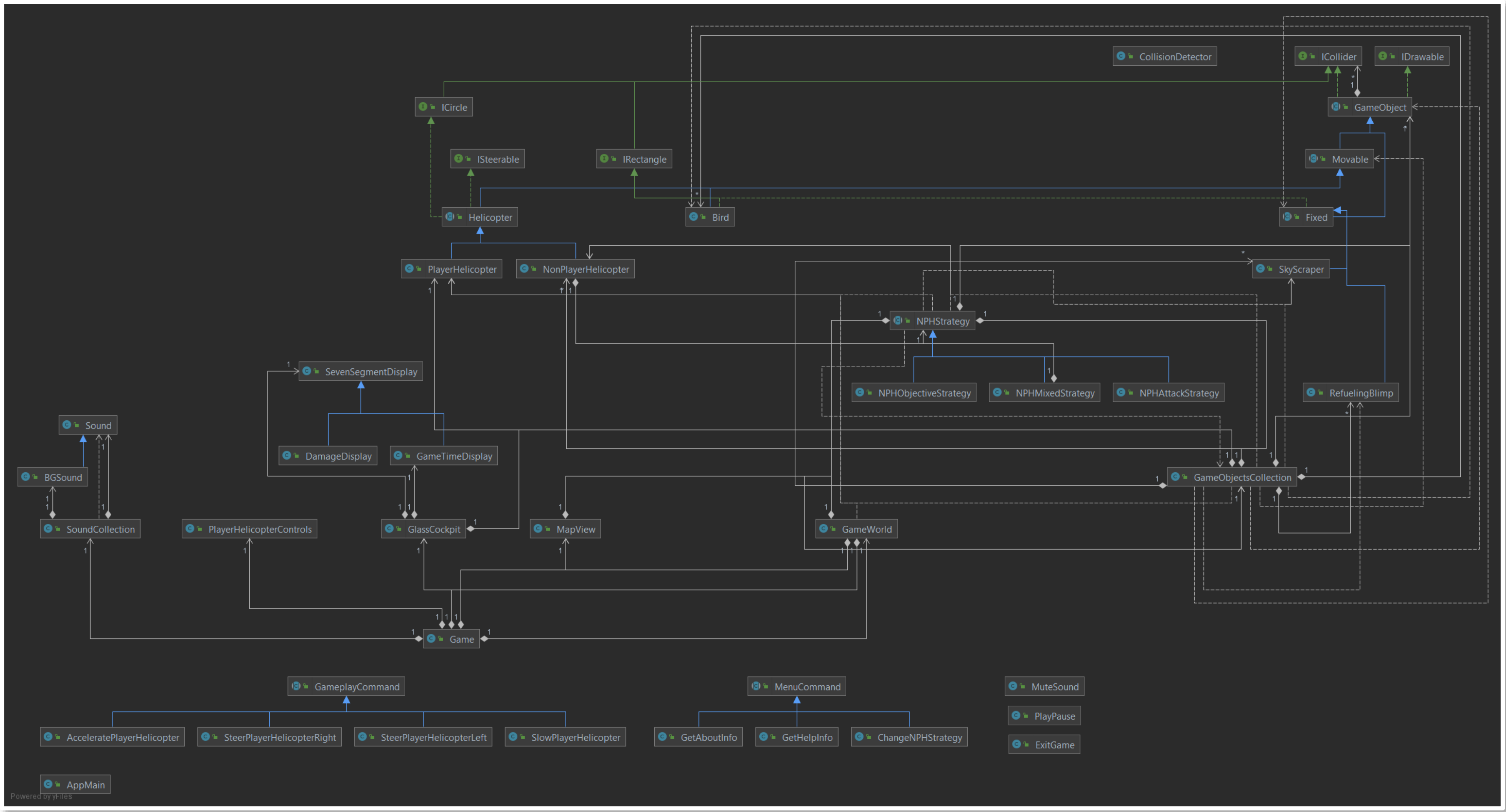
Finally, to prevent repeated collisions, the *GameObject.handleCollision(ICollider)* method places colliders into a *HashMap* with the collider as the key, and the current game time as its value. Then, if the *handleCollision()* method finds the object in the *HashMap* before a sufficient amount of time has passed, the collision behavior won't be executed.



```
@Override
public void handleCollision(ICollider collider) {
    if( collisionHistoryMap.containsKey(collider) &&
        GameTimeDisplay.getElapsedTime() - collisionHistoryMap.get(collider) > collisionCooldownTime)

            collisionHistoryMap.remove(collider);

    else if (!collisionHistoryMap.containsKey(collider)) collisionHistoryMap.put(collider, GameTimeDisplay.getElapsedTime());
}
```

Sound – Sound was a somewhat frustrating and difficult problem to solve. While I had aspirations of creating some fun object-oriented sound design, the limitations of CN1 made that seem less realistic without a lot more time and research involved. In the end, I landed on creating another class of static methods containing all of the necessary game sounds called *SoundCollection*. This class is instantiated at the construction of *Game*, which seems to alleviate most of the timing issues related to media creation in CN1. From there, all of the sounds are accessed through static getter methods within whatever scope they're needed. *BGSound* extends *Sound* and implements *Runnable*, looping with the use of a *UITimer* set to the duration of the sound. All sounds contain a boolean *mute*, which when set to true immediately pauses the *Media* contained within *Sound*, and prevents *Media.play()* from being called. This is in unison through another static method found within *SoundCollection*, which is called by a new keybound *Command* called *MuteSound*.

UML Diagram



CollisionDetector
ICollider
IDrawable
ICircle
GameObject
ISteerable
IRectangle
Movable
Helicopter
Bird
Fixed
PlayerHelicopter
NonPlayerHelicopter
SkyScraper
NPHStrategy
SevenSegmentDisplay
NPHObjectiveStrategy
NPHMixedStrategy
NPHAttackStrategy
RefuelingBlimp
Sound
DamageDisplay
GameTimeDisplay
BGSound
GameObjectsCollection
SoundCollection
PlayerHelicopterControls
GlassCockpit
MapView
GameWorld
Game
GameplayCommand
MenuCommand
MuteSound
PlayPause
AcceleratePlayerHelicopter
SteerPlayerHelicopterRight
SteerPlayerHelicopterLeft
SlowPlayerHelicopter
GetAboutInfo
GetHelpInfo
ChangeNPHStrategy
ExitGame
AppMain
Powered by yFiles