Computer Science E214 (2020) Project: Cosmic Conquistadors

NOTE:

The project will eventually be exported in a structure suitable for command line compiling and running. At the moment the structure is tailored to development in VS Code.

Thus, **if you want to test in on the command line now**. You need to rearrange the folders into this structure:

```
src/
  geom/
    BoundingShape.java
    ...
  resources/
    audio/
      ambientmain_0.wav
      ...
    images/
      blueExplosion00000.png
      ...
    highscores.txt
    ...
  AnimatedImage.java
  Bunker.java
  ...
```

NB TO DO's:

Code:

- Shield stuff:
  - include in tutorial
  - make enemies shoot more often so that you need to use shield more often
  - shield collision radius
- Update updateable spelling to updatable... :/
- Enemy visual damage based on hitpoints
- Different enemy types --- quick fix ???

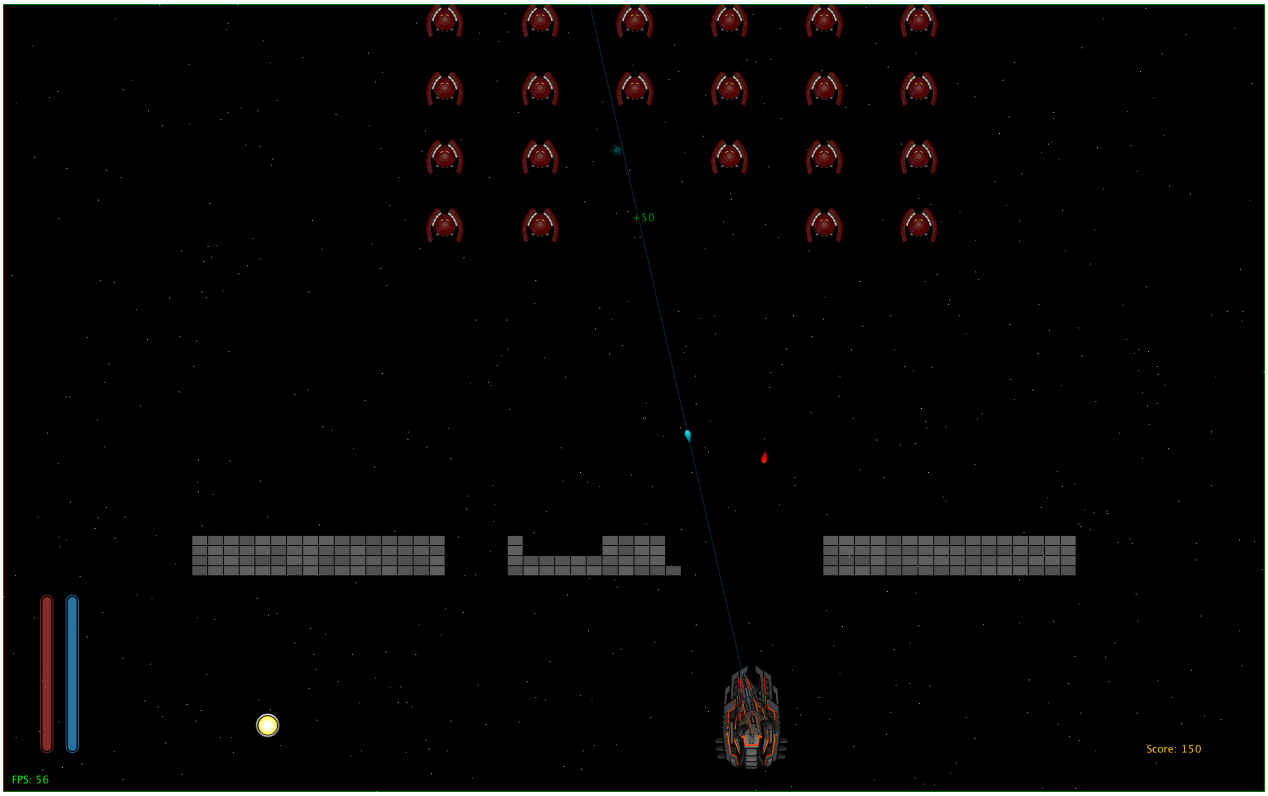
Documentation:

- Write Class inheritance section

- Complete additional work section
- Complete sources section
    - Get media sources from old repository and Google Chrome bookmarks
- Add table of contents

Admin:

- export and zip files
- peer rating
- submission



Group Members

	Name	Student Number
#1	Emi Dreckmeyr	21604754
#2	Michael Knight	23263962
#3	Nicol Visser	16986431

Execution Details

The main class is `MainGame.java` and is located under `./src/` directory.

```
> cd src
```

Compiling:

1. Compile all java files in the `./src/` directory.

```
> javac *.java
```

2. Compile all java files in the `./src/geom` directory.

```
> javac geom/*.java
```

Running:

1. Run in Windowed Mode:

```
> java MainGame
```

2. Run in Fullscreen Exclusive Mode (recommended):

Add the argument `-f` or `-fullscreen`.

```
> java MainGame -f
```

*** Visual Debugging Mode:**

Visual debugging mode shows the shapes used for collision. This is helpful to understand how collision detection works and to find bugs.

To enable visual debugging mode, add the argument `-d` or `-debug` when running the game.

e.g.

```
> java MainGame -f -d
```

Interface Inheritance

Interfaces in the `geom` package:

The `geom` package contains objects that can be associated with 2D geometry.

It has two interfaces, the `Shape` interface and an extension of it, the `BoundingShape` interface.

`LineSegment` and `Ray` implement the `Shape` interface, whereas `Rectangle` and `Circle` implement the `BoundingShape` interface.

`Shape` Interface:

`Rectangle`, `Circle`, `LineSegment` and `Ray`, by (directly or indirectly) implementing the `Shape` interface, have methods for:

- drawing the shape on a `Graphics2D` object
    - `public boolean intersects(Shape shape);`
- checking whether or not the shape intersects another shape.
    - `public void draw(Graphics2D g);`

`BoundingShape` Interface:

`Rectangle` and `Circle`, by implementing the `BoundingShape` interface, also have methods for

- checking whether or not the bounding shape contains a point
- checking whether or not the bounding shape completely contains another shapeÏ
- returning a random point inside the bounding shape

```
public boolean contains(double x, double y);
public boolean contains(Vector2D point);
public boolean contains(Shape shape);
public Vector2D getRandomPositionInside();
```

Example:

Interfaces in the game (default package)

`Collidable` Interface

`Collidable` aids with collision detection and handling.

Classes that implement `Collidable` have methods for:

- returning the `BoundaryShape` used for collision
- checking whether or not the object is colliding with another `Collidable` object
- handling the collision with another `Collidable` object, assuming they do collide

```
public Shape getCollisionShape();
public boolean isCollidingWith(Collidable otherCollidable);
public void handleCollisionWith(Collidable otherCollidable);
```

In our game, the game objects are often stored in `ArrayLists` of the same type. Instead of creating an N body simulation where all `Collidables` are checked against eachother, we rather only check certain groups of `Collidables` with other groups of `Collidables` or an individual `Collidable`. For this approach, two static methods are available to check and handle collisions.

```
public static void checkAndHandleCollisions(ArrayList<? extends Collidable>
group1, ArrayList<? extends Collidable> group2) {
  ...
}

public static void checkAndHandleCollisions(Collidable collidable1, ArrayList<?
extends Collidable> group2) {
  ...
}
```

`Disposable` Interface

We mentioned that our game objects are often stored in `ArrayLists`. At some point after a `Missile` has exploded or an `Enemy` has died, we want to remove that object from the list so that it does not have to be rendered and can be garbage collected by Java. Therefore we made the `Disposable` interface.

A class that implements this interface has a method `public boolean mayBeDisposed()` to check whether or not the item is ready to be disposed.

For convenience the `Disposable` interface contains a method that will iterate through an `ArrayList` of `Disposables`, checks whether any item is ready to be disposed and then removes that item from the list.

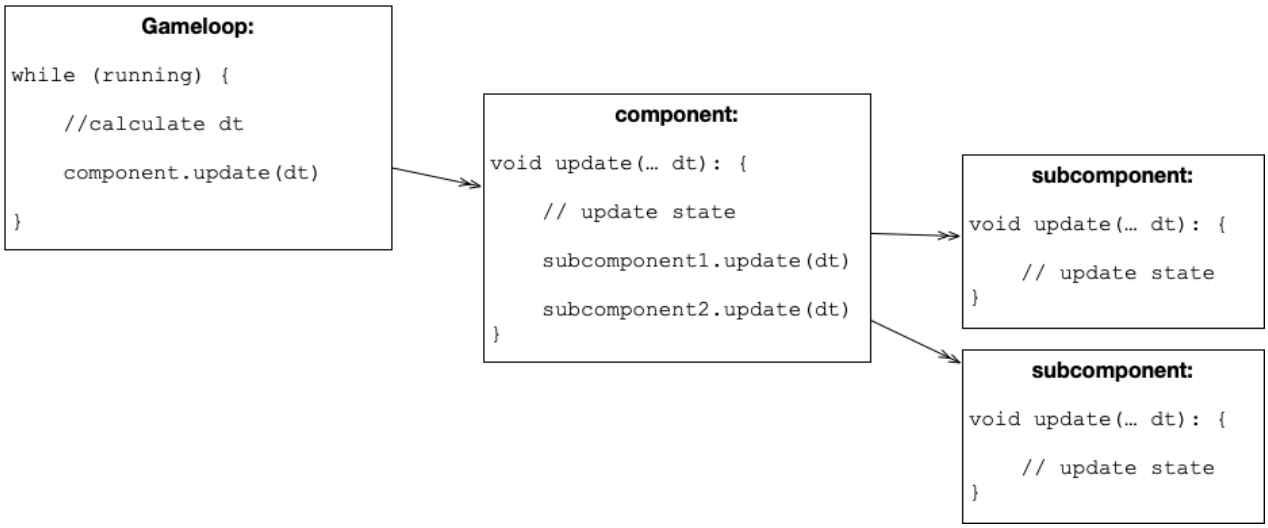
`Updateable` Interface

Classes that implement the `Updateable` interface have an `update` method as follows:

```
> public void update(int dt);
```

An updateable class 'wants' its `update` method to be called in whenever the game updates its state in the game loop. The argument `dt` is the targeted amount of time in milliseconds between frames.

The idea is that in the game loop, once we know the timestep we call the `update` method of a component which is `Updateable`. Then in that component we have subcomponents which are also `Updateable` and we call them, untill the entire tree of updateable are drawn. This call tree can be seen in the following diagram:



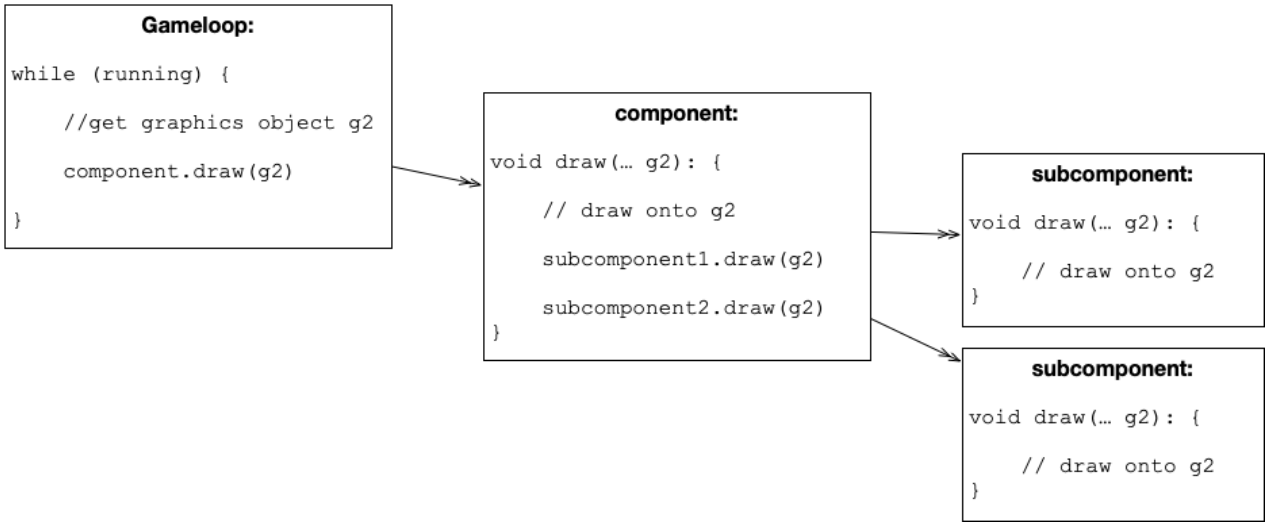
`Drawable` Interface

Classes that implement the `Drawable` interface have a `draw` method as follows:

```
> public void draw(Graphics2D g2);
```

A drawable class 'wants' its `draw` method to be called in whenever the game renders a frame. The argument `g2` is the `Graphics2D` object onto which the object will draw its contents.

The idea is that in the game loop, once we have the `Graphics2D` object to draw to, we call the `draw` method of a componet which is `Drawable`. Then in that component we have subcomponents which are also `Drawable` and we call them, untill the entire tree of drawables are drawn. This call tree can be seen in the following diagram:



`Shakeable` (an experimental) Interface

NOTE: This interface is experimental and is probably not be the best way to implement the functionality. The reason for choosing a functional interface is purely educational.

Shakeable is a functional interface that can be used together with lambda expressions to 'pass a method' to another class via its constructor. The idea was to pass a method that 'shakes' the screen from the InvaderGameState class to the Shooter class such that when a missile hits the shooter, we can call the shake method of the InvadersGameState class from within the Shooter class.

Idea gained from https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Class Inheritance

Swing Objects

Gameplay Objects

Class Structure Diagrams

FIGURE 1 shows the class and interface inheritance of objects in the `geom` package. These objects form an important base for the 2D programming in other classes.

FIGURE 1: Inheritance Diagram: Geometry: `geom` package

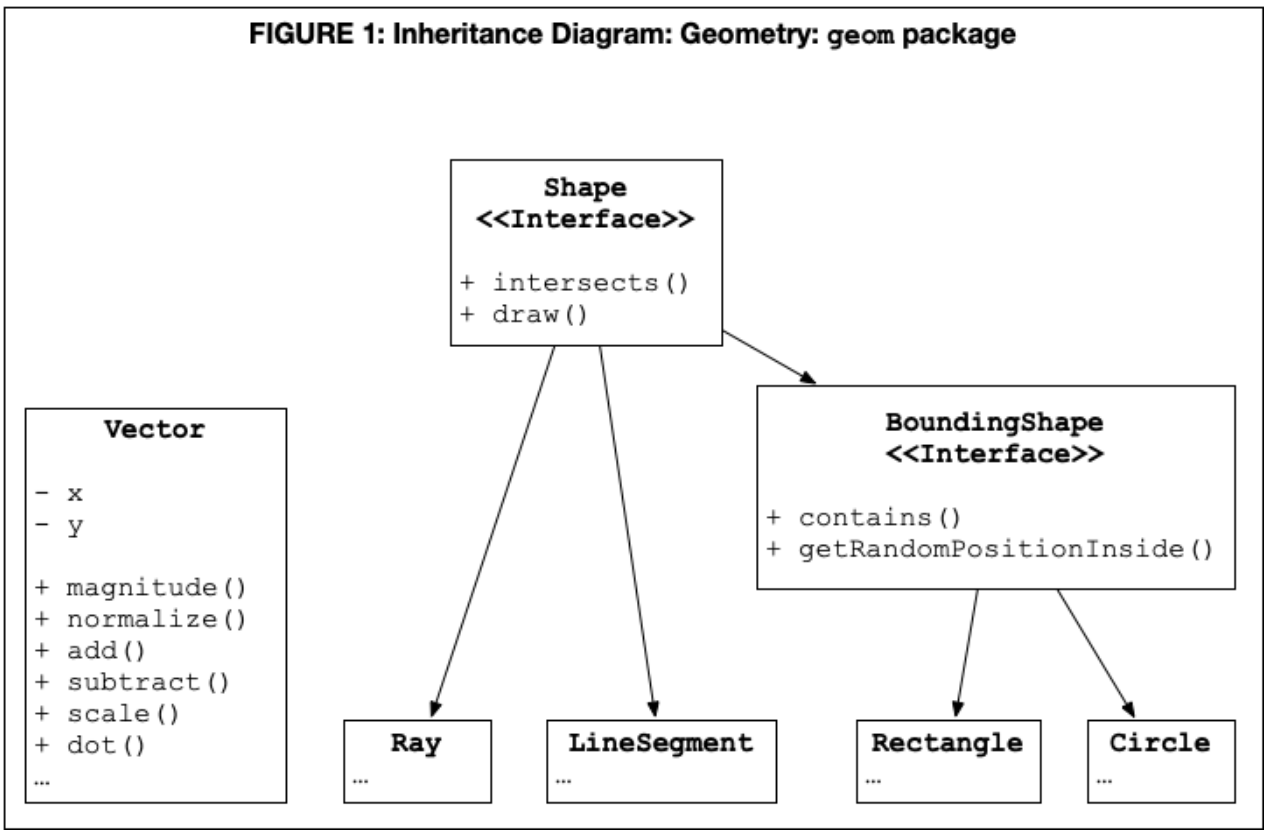


FIGURE 2 shows the class and interface inheritance of objects that directly or indirectly inherit from the `javax.swing` framework.

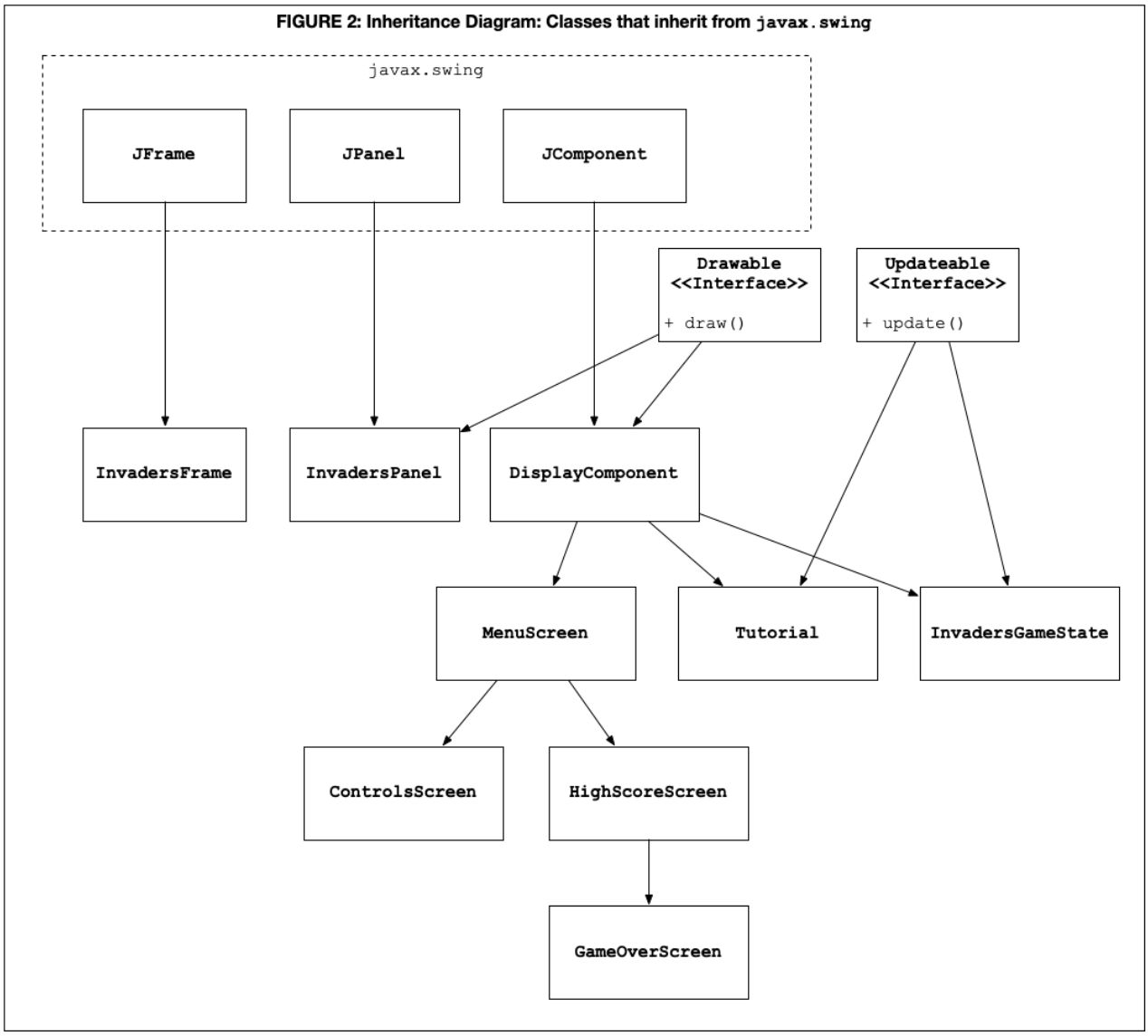
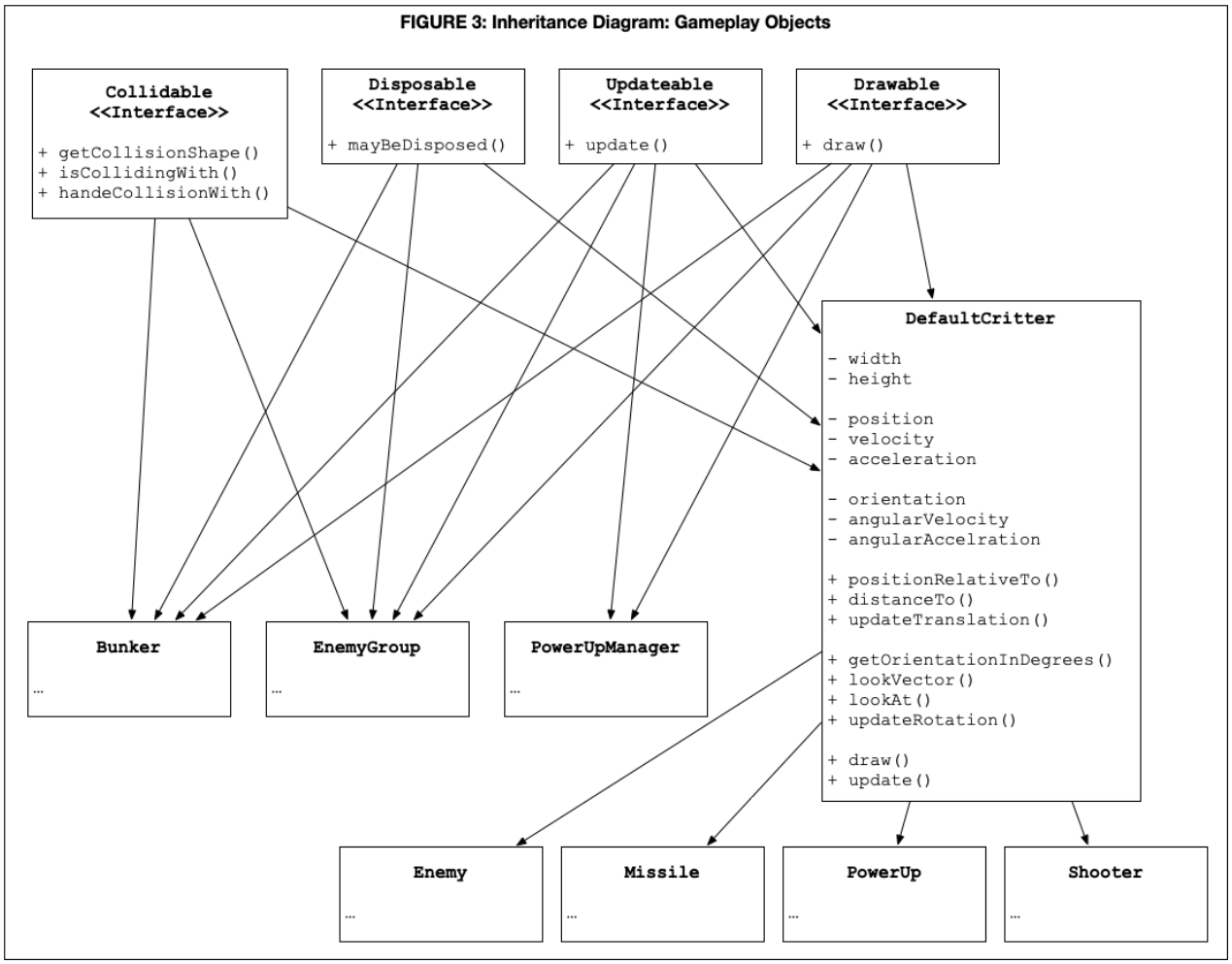
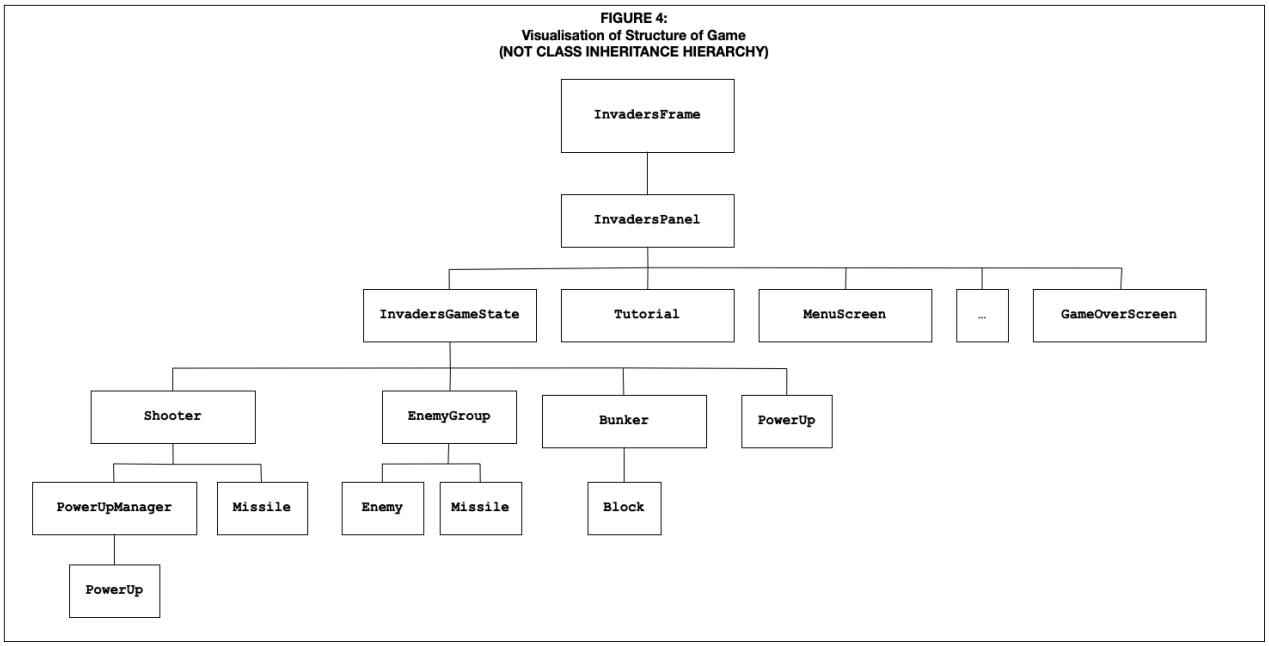


FIGURE 3 shows the class and interface inheritance of objects that relate to gameplay, such as missiles, shooters etc:



The following diagram, **FIGURE 4**, should give a rough idea of the "parent-child" relationships that the classes might have. Note, this is not exact or exhaustive, but is helpful to understand the game and where to find implementations of the classes. You may read the diagram as follows. If `Missile` is a child of `EnemyGroup` it translates to 'there is a `Missile` instance declared somewhere in the class `EnemyGroup` .



Summary of Additional Work

From recommendations in Marking sheet

Improved graphics

- Starfield background with parallax effect depending on shooter velocity.
- Various menu screens and functionality to switch between them.

Add sounds

- Volume control for programmer to change volume of sound without editing sound file.

Add music

- Two tracks, one for menus and one for gameplay. Menu music fades out befor game music starts.

Leaderboard/high score screen

- Save, view and reset high scores. Swop active keylisteners to listen for text input while entering name.

Progressively harder levels

Extra lives

- Powerups for health as well as hitpoints made extra lives feel redundant. These alternatives especially with their visuals are at least as complicated as extra lives.

Additional shooter

- Competitive mode where two shooters can see who get the most points

Enemies counterattack

Bunkers Power-ups (1 or 2 ticks)

Hit-points

Different weapons

Laser Gun or Regular Missiles. Missiles fired at bunkers burst and remove bunkers in a certain radius.

Different enemy types

-

Other:

Customizable controls for both players

Custom `geom` package that serves as API for 2D geometry. Includes intersection and containment methods.

Better colission detection

- Effective use of shapes (circles, rays, rectangles) instead of just a point and distance
- Use of minimum bounding boxes to reduce amount of checks

Animated text that fade away

Fullscreen exclusive mode

Allowing various window sizes by dynamically positioning in viewport.

Tutorial

- Helps player get to know the controls
- Shows how careful programming lead to components being reuseable in a different environment with different objectives.

Screen shake animation

External Libraries

The `In` and `Out` classes are used from the booksite's standard library, https://introcs.cs.princeton.edu/java/stdlib/. This was to simplify the process of reading and writing to text files due to some time constraints.

The class `GameAudio` is a trimmed down version of the booksite's `StdAudio` library. With customizations such as:

- volume control of sound
- looping background music that can fade out and play a different track

External Sources

Creating game loop in full screen exclusive mode:

https://docs.oracle.com/javase/tutorial/extra/fullscreen/index.html

"Setting Up Full-Screen Exclusive Mode" in https://www.oreilly.com/library/view/killer-game-programming/0596007302/ch04.htmlÏ