OGP Assignment 2012-2013: Asteroids Part III – draft

This text describes the third part of the assignment for the course 'Object-oriented Programming'. The groups that were formed for the first and second part should preferably work together on the third part. Students that have worked alone on the second part must also complete the third part on their own. If problems arise within a group, the group can split and each team member is required to complete the assignment on its own. If a group splits, this must be reported to ogp-project@cs.kuleuven.be.

In this third part, we extend the program developed in parts I and II focussing particularly on inheritance. Modifications of the assignment with respect to part II are highlighted in blue.

1 Assignment

Asteroids is an arcade game where the player controls a space craft in an asteroid field. The goal of the game is to evade and destroy objects such as enemy vessels and asteroids in a two-dimensional, rectangular space. In this assignment, we will create a game loosely based on the original arcade game released in 1979 by Atari.

The third part of this assignment extends the game created in parts I and II with computer-controlled opponents. Each such opponent executes a program which determines its ship's actions (turning, firing, etc.) throughout the game.

Note that if the assignment does not specify how to work out a certain aspect of the game, select the option you prefer.

The documentation of the class `Ship` must be worked out both formally and informally, while the documentation of the classes `World`, `Asteroid`, `Bullet` and of any helper classes you define must be worked out formally. It is not necessary to provide documentation for the classes for representing programs.

1.1 Game World

A game world is a two-dimensional, rectangular area containing ships, asteroids and bullets. Each world has a particular size, described by a width and height expressed in kilometres (km). The size of a world cannot change after construction. Both the width and height must lie in the range 0 to `Double.MAX_VALUE` (both inclusive) for all worlds. In the future, the upper bound on the width and height may decrease. However, all worlds will share the same upper bound.

A world contains ships, asteroids and bullets. At all times, each ship, asteroid or bullet is located in at most one world. No world contains the same ship, asteroid, or bullet twice. Ships, asteroids and bullets are circular entities. If such an entity is located in a world, then the circle must lie fully within the bounds of that world and it shall not overlap with other ships, asteroids or bullets within that world (only exception: a ship cannot be hit by its own bullets - see later). Note that the aforementioned properties may take a margin of error ϵ into account.The class `World` shall provide methods for adding and removing ships, asteroids and bullets. Those methods must be worked out defensively. It should be possible to ask a world what asteroids, ships and bullets it contains.

The documentation for the class `World` must be worked out only in a formal way. In fact, this applies to all classes in the project, except for the class `Ship`, whose documentation must be worked out both formally and informally.

1.2 Ship

Each spaceship is located at a certain position (x, y) (even if it is not associated with a world). Both x and y are expressed in kilometres (km). All aspects related to the position of a ship shall be worked out defensively.

Each spaceship has velocities v_x and v_y (even if it is not associated with a world) that determine the vessel's movement per time unit in the x and y direction, respectively. Both v_x and v_y are expressed in kilometres per second (km/s). The speed of a ship, computed as $\sqrt{v_x^2 + v_y^2}$, shall never exceed the speed of light c, $300000 km/s$. In the future, this limit need not remain the same for each ship, but it will always be less than or equal to c. All aspects related to velocity must be worked out in a total manner.

The shape of each ship is a circle with finite radius σ (expressed in kilometres) centred on the ship's position. The radius of a ship must be larger than zero and never changes during the program's execution. All aspects related to the radius must be worked out defensively.

Each ship has a mass expressed in kilograms (kg). At all times, the mass of a ship is strictly positive. The mass of ship determines its resistance to acceleration and the effect of collisions with other ships.

Each ship faces a certain direction expressed as an angle in radians (even if it is not associated with a world). For example, the angle of a ship facing right is 0, a ship facing up is at angle $\pi/2$, a ship facing left is at angle π and a ship facing down is at angle $3\pi/2$. The class `Ship` must provide a method to turn the ship by adding a given angle to the current direction. All aspects related to the direction must be worked out nominally.

Each ship has a thruster. The thruster is either enabled or disabled. When the thruster is enabled, the space ship accelerates in the direction it is facing. For now, each ship is equipped with the same kind of thruster. An active thruster exerts $1.1 \cdot 10^{21}$ Newton on its ship. In future iterations, the power output of thrusters need not remain the same for each ship. The acceleration generated by the thruster can be derived from Newton's second law of motion ($F = ma$).

The class `Ship` shall provide methods to inspect the position, velocity, radius, mass and direction.

1.3 Asteroid

Each asteroid is located at a certain position (x, y) (even if it is not associated with a world). Both x and y are expressed in kilometres (km). All aspects related to the position of an asteroid shall be worked out defensively.

Each asteroid has velocities v_x and v_y (even if it is not associated with a world) that determine the asteroid's movement per time unit in the x and y direction, respectively. Both v_x and v_y are expressed in kilometres per second (km/s). The speed of an asteroid, computed as $\sqrt{v_x^2 + v_y^2}$, shall never exceed the speed of light c, $300000 km/s$. In the future, this limit need not remain the same for each asteroid, but it will always be less than or equal to c. All aspects related to velocity must be worked out in a total manner.

The shape of each asteroid is a circle with finite radius σ (expressed in kilometres) centred on the asteroid's position. The radius of an asteroid must be larger than zero and never changes during the program's execution. All aspects related to the radius must be worked out defensively.

Each asteroid has a mass expressed in kilograms (kg). For simplicity, we assume the mass density ρ of each asteroid is the same, namely $2.65 \cdot 10^{12} kg/km^3$. The mass m of an asteroid can hence be computed as follows:

$$m = \frac{4}{3}\pi r^3 \rho$$

where r is the radius of the asteroid.

The class `Asteroid` shall provide methods to inspect the position, velocity, radius and mass.

1.4 Bullet

Each bullet is located at a certain position (x, y) (even if it is not associated with a world). Both x and y are expressed in kilometres (km). All aspects related to the position of a bullet shall be worked out defensively.

Each bullet has velocities v_x and v_y (even if it is not associated with a world) that determine the bullet's movement per time unit in the x and y direction, respectively. Both v_x and v_y are expressed in kilometres per second (km/s). The speed of a bullet, computed as $\sqrt{v_x^2 + v_y^2}$, shall never exceed the speed of light c, $300000 km/s$. In the future, this limit need not remain the same for each bullet, but it will always be less than or equal to c. All aspects related to velocity must be worked out in a total manner.

The shape of each bullet is a circle with finite radius σ (expressed in kilometres) centred on the bullet's position. The radius of a bullet must be larger than zero and never changes during the program's execution. All aspects related to the radius must be worked out defensively.

Each bullet has a mass expressed in kilograms (kg). For simplicity, we assume the mass density ρ of each bullet is the same, namely $7.8 \cdot 10^{12} kg/km^3$. The mass m of an bullet can hence be computed as follows:

$$m = \frac{4}{3} \pi r^3 \rho$$

where r is the radius of the bullet.

The class `Bullet` shall provide methods to inspect the position, velocity, radius and mass. In addition, the class should provide a method that returns its source (the ship that fired the bullet). The source of a bullet cannot be `null` and does not change after construction.

If a ship is located within a world and that world contains less than 3 bullets fired by that ship, it can fire bullets. The initial speed of such a bullet is $250 km/s$ and the direction of its velocity is equal to the direction the ship is facing[1]. For now, the radius of a bullet fired by a ship is $3km$. It is initially placed right next to the ship (at the angle the ship is facing). However, if the bullet's initial position is already (partially) occupied by another entity, then the bullet immediately collides with that entity. If the bullet would be

[1]Note that the velocity of the ship itself has no influence on the initial velocity of its bullets. Even though this is not physically correct, it is fine for our game.

located (partially) outside of the world, then firing has no effect. Firing also has no effect if there are three or more bullets of that ship within the world.

1.5 Collisions

Ships, asteroids and bullets may collide with each other and with the boundaries of the world they are in (if any). Your solution should include methods to predict and resolve collisions.

1.5.1 Predicting Collisions

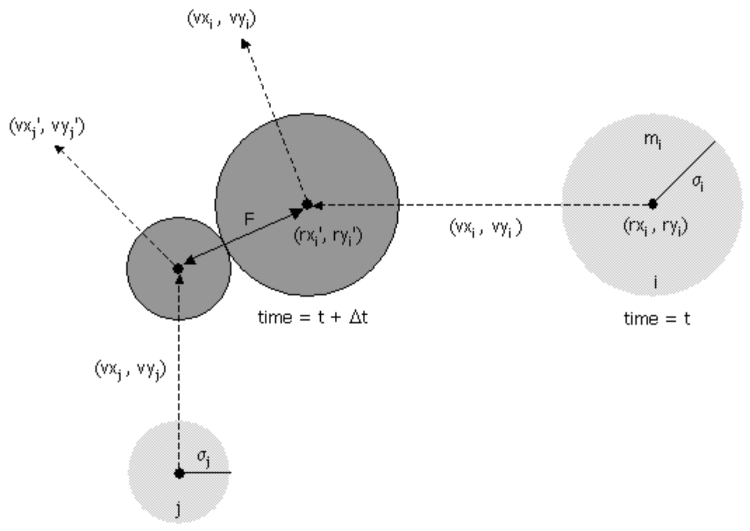
Predicting when (i.e. in how many seconds) two entities will collide (if ever) can be computed as follows:

- Given the positions and velocities of two entities i and j at time t, we wish to determine if and when they will collide with each other.
- Let (rx'_i, ry'_i) and (rx'_j, ry'_j) denote the positions of the entities i and j at the moment of contact, say $t + \Delta t$. When the entities collide, their centres are separated by a distance of $\sigma = \sigma_i + \sigma_j$. In other words: $\sigma^2 = (rx'_i - rx'_j)^2 + (ry'_i - ry'_j)^2$
- During the time prior to the collision, the entities move in straight-line trajectories. Thus,
  $rx'_i = rx_i + \Delta t \cdot vx_i$, $ry'_i = ry_i + \Delta t \cdot vy_i$
  $rx'_j = rx_j + \Delta t \cdot vx_j$, $ry'_j = ry_j + \Delta t \cdot vy_j$
- Substituting these four equations into the previous one, solving the resulting quadratic equation for Δt, selecting the physically relevant root, and simplifying, we obtain an expression for Δt in terms of the known positions, velocities, and radii:

$$\Delta t = \begin{cases} \infty & \text{if } \Delta v \cdot \Delta r \geq 0, \\ \infty & \text{if } d \leq 0, \\ -\frac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise,} \end{cases}$$

  where $d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v)(\Delta r \cdot \Delta r - \sigma^2)$ and
  $\Delta r = (\Delta x, \Delta y) = (rx_j - rx_i, ry_j - ry_i)$
  $\Delta v = (\Delta vx, \Delta vy) = (vx_j - vx_i, vy_j - vy_i)$
  $\Delta r \cdot \Delta r = (\Delta x)^2 + (\Delta y)^2$
  $\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2$
  $\Delta v \cdot \Delta r = (\Delta vx)(\Delta x) + (\Delta vy)(\Delta y)$.

A ship cannot be hit by its own bullet (i.e. they never collide). A bullet can therefore overlap with its source.



Ships, asteroids and bullets collide not only with each other but also with the boundaries of the space they are in. Your solution should include one or more methods to determine when (i.e. in how many seconds) a ship, asteroid or bullet collides with a boundary (if ever).

1.5.2 Resolving Collisions

Resolve collisions as follows:

- When two ships collide, they bounce off each other. That is, the velocity of both ships is updated to reflect the collision.
- When two asteroids collide, they bounce off each other.That is, the velocity of both asteroids is updated to reflect the collision.
- When a bullet hits another ship, asteroid or bullet (other than its source), both the bullet and the entity die.
- When a ship collides with an asteroid, the ship dies. The asteroid is not affected.
- Ships, asteroids and bullets bounce off boundaries. More specifically, when an entity collides with a horizontal boundary (e.g. the top of the world), negate the y component of its velocity. For example, when a ship with velocity $(5, 8)$ collides with the top boundary, change its

velocity to $(5, -8)$. When an entity collides with a vertical boundary, negate the x component of its velocity.

- When a bullet collides with a boundary for the second time, the bullet dies. That is, a bullet bounces off boundaries only once.

When two ships or asteroids collide, they bounce off each other. The new velocity of the entities i and j can be computed as follows:

$$(v_x'^i, v_y'^i) = (v_x^i + J_x/m_i, v_y^i + Jy/m_i)$$

$$(v_x'^j, v_y'^j) = (v_x^j - J_x/m_j, v_y^j - Jy/m_j)$$

where

$$J_x = \frac{J\Delta x}{\sigma}$$

$$J_y = \frac{J\Delta y}{\sigma}$$

$$J = \frac{2m_i m_j(\Delta v \cdot \Delta r)}{\sigma(m_i + m_j)}$$

Note that the mass of an object influences collisions resolution.

If an intermediate computation in an algorithm implemented in one of the aforementioned methods overflows (i.e. yields $-\infty$ or ∞), then the method's return value does not have to match the answer which is mathematically (i.e. when reasoning without overflow) expected.

1.6 Advancing Time

The state of a world evolves as time passes. For example, the position of an entity changes over time if the object is flying through space at a non-zero velocity. Similarly, an entity's velocity can change because of collisions.

The class `World` should provide a method `evolve`, which advances the state of the world a certain number of seconds Δt. Implement this method as follows:

1. Predict the first collision C (i.e. the collision that happens before all other collisions).

2. Suppose C happens in t_C seconds. If t_C is larger than Δt, go to step 5. Otherwise, advance all ships, asteroids and bullets t_C seconds (to the time right before the first collision).

3. Resolve C.

4. Subtract t_C from Δt and go to step 1.

5. Advance all ships, asteroids and bullets Δt seconds.

Advancing a ship, asteroid or bullet entails updating its position based on its current velocity. If a ship's thruster is enabled, additionally modify its velocity (*after* updating its position) based on its acceleration. The new velocity of the ship (v'_x, v'_y) is derived as follows:

$$\begin{aligned} v'_x &= v_x + a_x \Delta t \\ v'_y &= v_y + a_y \Delta t \end{aligned}$$

Here a_x and a_y respectively represent the acceleration along the X and Y axis. The acceleration can be derived from the ship's direction, mass and the amount of force generated by the thruster by Newton's second law of motion as described in Section 1.2. Note that it is not necessary to take collisions into account when advancing a ship, asteroid or bullet. Finally, the ship's program (if any) executes every 0.2 seconds and can order the ship to perform an action as described in Section 1.8.

Extra: Efficient Collision Prediction

The first step of `evolve` consists of predicting the next collision. A simple way to implement this step is to compute the set of all collisions and to return the minimum of this set. However, if a world contains many entities, recomputing all collisions over and over can slow down the game.

Provide a more efficient implementation of step 1 by storing all potential future collisions in a data structure. Note that certain collisions can become invalid when the velocity of an entity changes (e.g. collision occurs or thruster active). Ideally, a single iteration of evolve (steps one to three) takes at most $O(N \log(N^2))$ time if all thrusters are disabled, where N is the number of entities.

1.7 Death

Ships, asteroids and bullet can die when they collide with each other or with the boundaries of their world. When an entity dies, it is removed from its world (if any). In addition, when an asteroid dies, it spawns two smaller asteroids, provided the asteroid is located within a world and provided the asteroid's radius is larger than or equal to 30 kilometres. The radius of each child is equal to half of the radius of the parent asteroid. The direction of the velocity of the first child is determined at random. The other child moves in the opposite direction. The speed of their velocities is 1.5 times the speed of their parent. Finally, both children are placed at a distance $r/2$ (where r is the radius of the parent) from the center of the parent asteroid. The centres of the parent and of both children should lie on a single line.

A world never contains dead ships, asteroids or bullets.

1.8 Programs

A program consists of a number of global variables and a main statement. For example, consider the program shown below.

```
double d;
bool b;

d := 0.2;
b := true;
while(b) {
  turn d;
  d := d + 0.1;
  fire;
}
```

This program declares two global variables named `d` and `b`. The former variable has type `double` while the latter has type `bool`. The main statement is a sequence consisting of an assignment and a while loop. The body of this loop is a sequence consisting of three statements: a turn statement, an assignment and a fire statement.

A ship can store a program that determines its actions. The program associated with a ship executes as time advances. A program can order its ship to perform an action every 0.2 seconds (i.e. it executes until it encounters an action statement every 0.2 seconds). When the program associated with a ship runs, it executes until either there are no statements left or until it encounters an action statement (turning, enabling the thruster, disabling the thruster, firing a bullet or doing nothing). The next time the program runs, it continues right after this action statement. As an example, consider the program shown above. The first time this program runs, it executes until it encounters the turn statement, making the ship turn `d` radians. The second execution of the program continues right after the turn command: the variable `d` is incremented by 0.1 and the program encounters the fire statement, making the ship fire a bullet. The third time the program runs, it continues right after the fire statement (even if the fire statement had no effect). As the condition of the loop still holds, it again executes the body and stops right after the turn statement. Note that the values of the global variables must be remembered across invocations.

Illegal operations performed by a program should be handled in a total manner. That is, if a program performs an illegal operation (e.g. it tries to evaluate `getx null` or `true + true`) before it encounters an action statement, execution of the program stops. Subsequent runs of the program immediately return.

By default, a ship is not associated with a program.

1.8.1 Statements

The syntax of statements `s` in Backus Normal form (BNF) notation is as follows:

```
s ::=
  x := e;
| while(e) { s }
| foreach(kind, x) { s }
| if (e) { s } else { s }
| print e;
| s*
| action

action ::=
  turn e;
| fire;
| thrust;
| thrust_off;
| skip

kind ::=
  ship
| asteroid
| bullet
| any
```

That is, a statement is either an assignment, a while loop, a for-each loop, an if-then-else, a print statement, a sequence of zero or more statements or an action statement. There are five different kinds of action statements: turning, shooting, enabling the thruster, disabling the thruster and doing nothing (skip). For-each loops can range over all ships, all asteroids, all bullets or all entities.

A for-each loop iterates over all entities of a given kind. As an example, consider the program shown below:

```
double x;
double y;
double r;
double nearestAsteroid;
entity a;
double distance;
double distanceToNearestAsteroid;

x := getx self;
y := gety self;
r := getradius self;
nearestAsteroid := null;
foreach(asteroid, a) do {
  ax := getx a;
  ay := gety a;
  ar := getradius a;
  distance := sqrt((((x - ax) * (x - ax)) + ((y - ay) * (y - ay))));
  if(nearestAsteroid == null) then {
    nearestAsteroid := a;
    distanceToNearestAsteroid := distance;
  } else {
    if(distance < distanceToNearestAsteroid) then {
      nearestAsteroid := a;
      distanceToNearestAsteroid := distance;
    }
  }
}
if(nearestAsteroid != null) then {
  print distanceToNearestAsteroid;
}
```

This program iterates over all asteroids to find the one that is closest to the ship executing the program. The variable `a` in the for-each loop gets assigned a different value in each iteration. The total number of iterations is equal to the number of asteroids in the world of the ship executing the program. Note that the variable used in a for-each loop (here `a`) must be an *existing* global variable. The body of a for-each loop should not contain action statements.

Students working alone do not have to support for-each loops.

1.8.2 Expressions

The syntax of expressions `e` in BNF notation is as follows:

```
e ::=
  x
| c
| true
| false
| null
| self
| e + e
| e - e
| e * e
| e / e
| sqrt(e)
| sin(e)
| cos(e)
| e && e
| e || e
| ! e
| getx e
| gety e
| getvx e
| getvy e
| getradius e
| e < e
| e <= e
| e > e
| e >= e
| e == e
| e != e
```

An expression is either a variable, a double constant, true, false, null, self (i.e. the ship that executes the program), an addition, a subtraction, a multiplication, a division, a square root, a sine, a cosine, a conjunction, a disjunction, a negation, getx, a gety, a getvx, a getvy, a getradius or a comparison (less than, less than or equal to, greater than, greater than or equal to, equal to or different from).

The expressions `getx e`, `gety e`, `getvx e`, `getvy e`, `getradius e` respectively compute the x-coordinate, y-coordinate, velocity along the X-axis, velocity along the Y-axis and radius of the entity `e`.

Students working alone do not have to support conjunction, disjunction, negation, getvx, getvy, sine, cosine and modulo expressions.

1.8.3 Types

Expressions, global variables and values can have three possible types: `double`, `bool` and `entity`. A variable of type `entity` is either `null` or a reference to a ship, asteroid or bullet.

All global variables are initialized to default values (based on their type) before the first excution of the program.

Extra: Type Checking

If you wish to obtain a score of 17 or more, your solution should include a method for typechecking a program. Typechecking means that the type correctness of the program is checked before it is executed, thereby eliminating the possibility of certain run-time errors. Informally, our programs are type correct if they satisfy the following properties:

- If the main statement mentions a global variable, then that variable is declared as a global variable.
- Operations are applied only to expressions of the correct type. For example, an addition `e1 + e2` is type correct only if both `e1` and `e2` have type `double`.
- The type of the expression `e` in each assignment `x := e;` in the program is the same as the type of the global variable `x`.
- The body of a for-each loop does not contain any action statements.

If you support type checking, the program associated with a ship must be type-correct at all times.

1.8.4 Parsing

The assignment comes with a number of example programs stored in text files. Reading a text file containing a program and converting it from its textual representation into a number of objects that represent the program in-memory is called *parsing*.

The assignment includes a parser. To parse a `String` object, instantiate the class `ProgramParser` and call its `parse` method. This method constructs an in-memory representation of the program by calling methods in the `ProgramFactory` interface. You should provide a class that implements this interface.

The parser was generated using the ANTLR parser generator based on the file `AsteroidsParser.g4`. It not necessary to understand or modify this file.

2 Testing

Write a JUnit test suite for the classes `Ship` and `World` that tests each public method. Testing other classes is optional. Include this test suite in your submission.

3 User Interface

We provide a graphical user interface (GUI) to visualise the effects of various operations on space crafts. The user interface is included in the assignment.

To connect your implementation to the GUI, write a class `Facade` that implements `IFacade`. `IFacade.java` contains additional instructions on how to implement the required methods. To start the program, run the `main` method in the class `Asteroids`. After starting the program, you can press keys to modify the state of the program. `Esc` terminates the program (or returns to the main menu if a game is running). In a single player game, the ship is controlled by the arrow keys and space bar is used for firing bullets. In a multi player game, the second player is controlled by the WASD keys[2] and `Control` is used for firing.

By default, the game runs in full screen mode. To start the game in a window, pass `-window` on the command line. To disable sound, pass `-nosound` on the command line. To change the default program executed by the computer-controlled opponent, use the `-ai` command line option.

You can freely modify the GUI as you see fit. However, the main focus of this assignment are the classes `Ship`, `World`, `Asteroid`, `Bullet`, etc. No additional grades will be awarded for changing the GUI.

You can freely modify the default AI program.

We will test that your implementation works properly by running a number of JUnit tests against your implementation of `IFacade`. As described in the documentation of `IFacade`, the methods of your `IFacade` implementation shall only throw `ModelException`.

[2]These controls are tuned for Qwerty keyboards; change them to ZQSD in `WorldView.java` on Azerty keyboards.

4 Submitting

The solution must be submitted via Toledo as a jar file individually by all team members before the 26th of May 2013 at 11:59 pm. You can generate a JAR file on the command line or using eclipse (via export). Include all source files (including tests) and the generated class files. Include your name, studies and a link to your code repository in the comments in your solution. When submitting via Toledo, make sure to press OK until your solution is submitted!