Aix*Marseille université
Socialement engagée

Année 2021-22

Domaine Sciences et Technologies
Portail René Descartes

Programmation 1 : Projet
Code UE : SPO2U07L

Projet : astéroïdes

1 Description du sujet

Ce sujet est un sujet à faire si vous avez déjà fini de travailler sur tous les sujets de TP. Il est consacré au développement d'un jeu en temps réel. Vous travaillerez à partir d'une version inaboutie du jeu, pour y ajouter des fonctionnalités.

1.1 Récupérer le dépôt

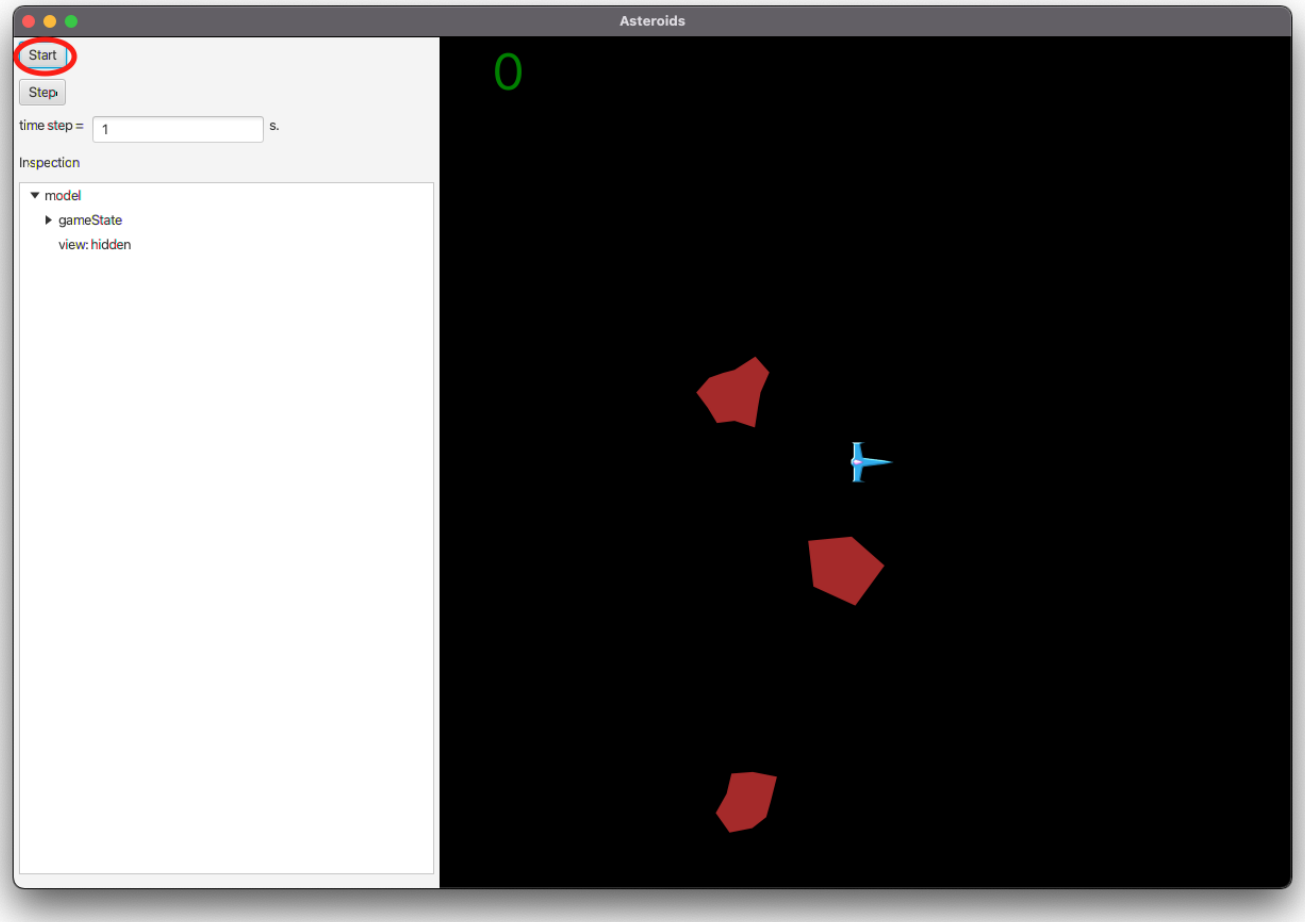
Comme pour le TP 6, on va utiliser git pour la gestion de versions. Il vous faut donc vous reporter aux consignes du TP 6. Le lien vers le projet à forker est le suivant : lien.

1.2 Exécuter le projet du dépôt

Pour compiler et exécuter votre programme, il faut passer par l'onglet gradle à droite.

— pour les tests il faut cliquer deux fois sur `AsteroidsGradle` -> `Tasks` -> `verification` -> `test`.

— pour exécuter l'application, il faut cliquer deux fois sur `AsteroidsGradle` -> `application` -> `run`. Vous devriez obtenir l'affichage suivant :



Tâche 1 : Vérifier que le programme s'exécute correctement.

1.3 Consignes pour le début du TP

Tâche 2 : Modifiez le fichier `README.md`. Mettez votre nom, votre **numéro de groupe** ainsi que le nom et le **numéro de groupe** de votre éventuel co-équipier. Faites un `commit` avec pour message "inscription d'un membre de l'équipe", puis un `push`.

1.4 Structure du projet

Le projet est constituée de nombreuses classes. Vous aurez à modifier les classes des packages `game` principalement, ainsi que `viewModel` et `view` parfois. Vous ne devez pas touchez aux fichiers des autres packages, ni aux classes dont un commentaire indique qu'il ne faut pas les modifier.

2 Documentation et modifications des paramètres

2.1 Génération de la documentation

Le programme est partiellement documenté, via des commentaires au format `javadoc`.

Tâche 3 : Générez la documentation, en suivant ces instructions :

- Créer un répertoire `doc` dans le répertoire du projet. Vous pouvez le faire depuis IntelliJ avec un clic droit sur la racine du projet.
- Ouvrir le menu `Tools > Generate Javadoc...`
- Positionner le champ `Output directory` sur le répertoire `doc` que vous venez de créer.
- Ajouter `-html5` dans le champ `Other command line arguments`.
- cliquez sur `Valider`.

Si tout se passe bien, le répertoire `doc` se remplit de fichiers, et un navigateur s'ouvre avec la documentation du projet. Vous pouvez aussi accéder au projet en ouvrant le fichier `index.html` du répertoire `doc` dans un navigateur.

2.2 Modifier les paramètres du programme

Tâche 4 : Trouver comment modifier le nombre d'astéroïdes, pour le faire passer à 10. (lire les documentations de `Space`)

Tâche 5 : Trouver comment modifier la taille des astéroïdes. (lire les documentations de `Asteroids`)

Déplacement du vaisseau spatial

Le vaisseau spatial se contrôle avec les touches flèches du clavier. Dans cette section, on se concentre sur la classe `Spaceship`.

Que se passe-t-il actuellement lorsqu'on tente de déplacer le vaisseau spatial ?

On souhaite suivre les lois physiques un peu plus fidèlement. On s'intéresse dans un premier temps uniquement au déplacement en ligne droite. D'après nos connaissances en physique, le déplacement d'un objet ponctuel est régi par sa masse m et la somme des forces s'y appliquant $\vec{F}$. Si on note $\vec{p}$, $\vec{v}$ et $\vec{a}$ les vecteurs position, vitesse (la *vélocité*) et accélération respectivement, on a :

$$\begin{array}{lccc} & m\vec{a} = \vec{F}, & \vec{a} = \frac{\mathrm{d}\vec{v}}{\mathrm{d}t}, & \vec{v} = \frac{\mathrm{d}\vec{p}}{\mathrm{d}t} \\ \text{c'est-à-dire} & m\vec{a} = \vec{F}, & \mathrm{d}\vec{v} = \mathrm{d}t \times \vec{a}, & \mathrm{d}\vec{p} = \mathrm{d}t \times \vec{v} \end{array}$$

Dans ces notations, $\mathrm{d}t$ est la différentielle de temps, $\mathrm{d}\vec{p}$ la différentielle de position et $\mathrm{d}\vec{v}$ la différentielle de vitesse. Ce qui veut dire que lorsque $\mathrm{d}t$ secondes passent (pour $\mathrm{d}t$ suffisamment petit),

— le temps passe de t à $t + \mathrm{d}t$,
— la position du vaisseau passe de $\vec{p}$ à $\vec{p} + \mathrm{d}\vec{p} = \vec{p} + \mathrm{d}t \times \vec{v}$,
— la vélocité du vaisseau passe de $\vec{v}$ à $\vec{v} + \mathrm{d}\vec{v} = \vec{v} + \mathrm{d}t \times \vec{a}$.

Les équations physiques nous donnent donc les instructions nécessaires pour mettre à jour, à chaque pas de temps, les caractéristiques déterminant l'état du vaisseau.

Tâche 6 : Actuellement, le vaisseau n'a qu'une seule caractéristique représentée dans la classe `Spaceship` : sa position. Ajouter une propriété pour sa vélocité $\vec{v}$.

Tâche 7 : Assurez-vous que la vélocité est initialement nulle.

Le vaisseau est soumis à une accélération nulle par défaut, ou par l'accélération constante procurée par son moteur lorsque celui-ci est actif.

Tâche 8 : Ajouter un attribut contenant un nombre correspondant à la puissance $|\vec{F}/m|$ du moteur :

```
private static final double MAIN_ENGINE_POWER = 50;
```

L'accélération $\vec{a}$ du moteur allumé est donc donnée par :

```
`direction.multiply(MAIN_ENGINE_POWER);
```

Tâche 9 : Écrire une méthode `public Vector getAcceleration()` qui retourne le vecteur d'accélération du vaisseau, selon l'état du moteur (allumé ou éteint).

Tâche 10 : Modifier la fonction `update`, en utilisant les lois physiques décrites ci-dessus, pour que la trajectoire (la position et la vitesse) du vaisseau soit correctement calculée. Utilisez les méthodes des objets de la classe `Vector` pour réaliser les opérations vectorielles. Vous pouvez aussi vous inspirer de la méthode `Asteroid.update`.

Tâche 11 : Testez le comportement du vaisseau maintenant. En quoi diffère-t-il du comportement initial ?

Tâche 12 : Modifiez la valeur du paramètre `MAIN_ENGINE_POWER` pour obtenir une accélération qui vous convienne : ni trop faible, ni trop forte.

On s'intéresse maintenant à faire tourner le vaisseau. Pour cela, le vaisseau dispose de moteurs latéraux qui, lorsqu'ils sont allumés, provoquent une rotation du vaisseau sur lui-même autour de son centre, soit dans le sens horaire, soit dans le sens anti-horaire.

Tâche 13 : Dans un premier temps, ajoutez des attributs au vaisseau, déterminant si les moteurs latéraux gauches ou droits sont allumés.

Tâche 14 : Ajoutez des méthodes permettant d'allumer ou d'éteindre les moteurs latéraux.

Lorsqu'un moteur latéral est allumé, le vaisseau tourne sur lui-même au rythme d'un tour complet par seconde (sa vitesse angulaire), dans un sens ou dans l'autre.

Tâche 15 : Écrire une méthode `updateDirection`, paramétrée par le délai `dt`, modifiant le vecteur direction par une rotation d'un angle proportionnel au temps écoulé.

Tâche 16 : Faire en sorte que lorsque la position du vaisseau est mise à jour, sa direction soit elle aussi mise à jour.

Ouvrir la classe `ViewModel` du package `viewModel`. Cette classe assure la communication entre la fenêtre graphique et le modèle de notre jeu.

Tâche 17 : En vous inspirant des méthodes présentes, ajouter dans cette classe des méthodes pour démarrez et stoppez les moteurs latéraux du vaisseau du jeu.

Dans la classe `View` du package `view`, repérez les deux méthodes `handleKeyPressed` et `handleKeyReleased`. Ces deux méthodes choisissent les actions à effectuer en fonction de la touche pressée ou relâchée par le joueur. `handleKeyPressed` est partiellement complétée en commentaire. La structure de contrôle `switch` permet de choisir une action selon la nature de la touche pressée (`UP`, `LEFT`, ...).

Tâche 18 : Modifier les cas `LEFT` et `RIGHT` en choisissant dans chaque cas l'action sur `viewModel` correspondant à la touche pressée. Faire de même pour `handleKeyReleased`.

Tâche 19 : Lancez le jeu et vérifiez que vos modifications fonctionnent correctement. Par exemple, est-ce que le vaisseau tourne-t-il correctement ? Au besoin, faites les corrections nécessaires.

Tâche 20 : Modifier la vitesse angulaire pour obtenir une vitesse de rotation qui vous convienne.

Tâche 21 : Comme pour l'accélération du moteur, il est bon d'avoir une constante définissant cette vitesse angulaire en dehors des méthodes de calcul. Créez donc cette constante.

Tâche 22 : Ajoutez une action supplémentaire, permettant de freiner le vaisseau.

Le freinage fonctionne comme l'accélération, mais en sens inverse et avec une acceleration plus faible. Ainsi, il faut modifier le calcul de l'accélération, ajouter des propriétés représentant l'état du moteur de recul, ajouter des méthodes pour accéder ou modifier l'état du moteur de recul, puis liés ces méthodes à la touche `DOWN` (en créant les méthodes nécessaire dans `ViewModel`). La puissance du moteur de recul devra être déterminée par une constante de la classe `Spaceship`.

3 Consommation

Les moteurs consomment de l'énergie. Ajoutez des propriétés au vaisseau pour l'état de son stock d'énergie. Modifiez les méthodes de sorte que le vaisseau consomme de l'énergie selon les actions effectuées. Comme il fonctionne à l'énergie solaire, l'énergie disponible augmente lentement avec le temps (mais pas assez pour que les moteurs restent allumés en permanence), sans dépasser les capacités de stockage du vaisseau. En absence d'énergie, les moteurs ne s'allument plus.

On se place dans la classe `Spaceship`.

Tâche 23 : Ajouter un attribut `fuel` indiquant la quantité de carburant dans le réservoir. On mesurera cette quantité en secondes de consommation du moteur principal.

Tâche 24 : Ajouter une constante définissant la capacité du réservoir (par exemple 5 secondes de consommation).

Tâche 25 : Ajouter une constante définissant la quantité de carburant rechargée par seconde (par exemple 0.2 secondes de consommation par seconde). Puis des constantes définissant la consommation des différents moteurs (1 pour le moteur principal, 0.3 pour les moteurs latéraux, 0.5 pour le moteur de recul).

Tâche 26 : Ajouter une méthode privée `getCurrentConsumption()` retournant la consommation net par seconde du vaisseau. Il faudra prendre en compte l'état de chaque moteur, ainsi que le taux de recharge du carburant.

Tâche 27 : Ajouter une méthode privée `getAutonomy(double dt)`, calculant combien la réserve de carburant du vaisseau peut tenir dans l'état actuel des moteurs. Si ce temps excède `dt`, on retourne `dt`.

Par exemple, si les moteurs principaux et latéral gauche sont allumés, le vaisseau consomme $1 + 0.3 - 0.2$ net par seconde, et épuise sa réserve actuelle r de carburant en $t = r/(1 + 0.3 + 0.2)$ secondes. On retourne le minimum de t et dt. Si tous les moteurs sont éteints, le vaisseau produit 0.2 par seconde jusqu'à remplissage du réservoir, et donc a une réserve illimitée à ce rythme, on retourne alors `dt`.

Tâche 28 : Modifier la méthode `updateVelocity`, en prenant en compte le temps d'allumage des moteurs calculé par `getAutonomy`.

L'accélération doit être multiplié non pas par `dt` comme avant, mais par l'autonomie au régime actuel. Ainsi, s'il n'y a pas assez de carburant pour tenir les moteurs allumés pendant un temps `dt`, l'accélération sera d'autant moins forte.

Tâche 29 : Modifier de façon similaire la méthode `updateDirection`. La vitesse de rotation est ajustée selon l'autonomie.

Tâche 30 : Dans la méthode `update`, ajouter après la mise-à-jour de la vélocité, de la direction et de la position, une mise-à-jour du niveau de carburant.

Utiliser de nouveau l'autonomie et la consommation actuelle. Assurez-vous que le niveau de carburant est positif et inférieur à la capacité du réservoir.

Tâche 31 : Ajouter une méthode publique retournant le pourcentage de carburant dans le réservoir.

Tâche 32 : Dans la classe `ViewModel`, ajouter aussi une méthode fournissant le pourcentage de carburant dans le réservoir du vaisseau spatial.

Tâche 33 : Dans la classe `CanvasView`, ajouter une méthode `renderFuel` prenant le pourcentage de carburant du vaisseau, et l'affichant sur le canvas sous la forme d'une barre rectangulaire sur un côté de l'écran.

La longueur de la barre est proportionnelle au taux de remplissage du réservoir : la barre doit faire la longueur ou la hauteur de l'écran quand le réservoir est plein (avec si vous le souhaitez une petite marge), et disparaître lorsqu'il est vide. Voici quelques informations pour vour aider :

— La propriété `context` gère le dessin : pinceau et feuille.
— sa méthode `setFill` permet de changer la couleur de remplissage. Les couleurs sont `Color.WHITE`, `Color.BLUE`, ... utilisez l'auto-complétion.
— sa méthode `fillRect(x,y,width,height)` permet de peindre un rectangle.
— le canvas fait 800x800 pixels, l'origine est en haut à gauche.

Tâche 34 : Dans cette même classe, modifier `render()` pour ajouter le rendu du niveau de carburant.

Vérifier que tout fonctionne comme prévu.

4 Animation des moteurs

Ce serait mieux si on visualisait quand les moteurs sont allumés. Pour cela, il faut modifier l'affichage du vaisseau. Le répertoire `resources` dans `src/main/ressources` contient les images nécessaires pour cela.

Tâche 35 : Dans la classe `ViewModel`, ajoutez des méthodes permettant de savoir pour chaque moteur du vaisseau s'il est allumé.

Tâche 36 : Dans la classe `CanvasView`, dans la méthode `render(Spaceship)`, ajoutez des instructions, avec des conditions appropriées, permettant d'afficher les images des différents moteurs en action.

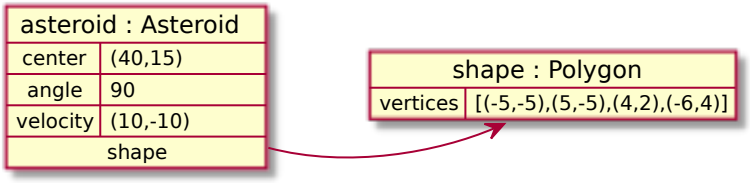
Les images des différents moteurs sont déjà définies par des chaînes de caractères en fin de fichier.
Vérifier le bon fonctionnement de cet ajout.

5 Collision point - astéroïde

Important : pour cette partie, munissez-vous d'une feuille de papier et du crayon. Certains questions vous demanderont de faire des calculs à la main.

La collision d'un vaisseau spatial avec un astéroïde est en général fatale. Nous souhaitons ajouter des fonctionnalités pour que le jeu se termine lorsque le vaisseau percute un astéroïde.

Tâche 37 : Consulter la classe `Asteroid`. Quels attributs d'un astéroïde permettent de déterminer son occupation de l'espace ?



L'astéroïde décrit par ce diagramme d'objets a ses sommets aux coordonnées $[(45, 10), (45, 20), (38, 19), (36, 9)]$. En effet, les coordonnées contenues dans l'attribut `vertices` sont relatives au centre du polygone, et en supposant un angle de 0. Pour obtenir les coordonnées absolues des sommets, il faut pour chaque coordonnée relative, d'abord faire une rotation par l'angle actuel de l'astéroïde, puis une translation selon la position du centre.

Tâche 38 : Vérifier que ce calcul produit bien les coordonnées indiquées.

Tâche 39 : Ajouter une méthode `public boolean contains(Vector point)` à la classe `Asteroid`, qui pour l'instant retourne toujours `false`.

Tâche 40 : Consulter les méthodes de la classe `Polygon`. Lesquelles de ses méthodes seront utiles pour implémenter la méthode `Asteroid.contains` ?

Dans un premier temps, supposons que l'astéroïde soit en position $(0, 0)$ et n'ait subi aucun mouvement ou rotation.

Tâche 41 : Écrire la méthode `contains` sous ces restrictions.

En réalité, l'astéroïde a tourné et est dans une position plus générale. Il faut donc d'abord transformer sa forme en lui appliquant une rotation puis une translation.

Tâche 42 : Modifier `Asteroid.contains` pour obtenir la version définitive de cette méthode.

Afin de s'assurer que l'implémentation est correcte, nous souhaitons tester la méthode `contains`, en utilisant les outils de Java. Le code qu'on vous a fourni contient déjà une classe de test `PolygonTest` dans le répertoire `test/java/tools/`.

Pour tester la méthode `Asteroid.contains`, vous allez ajouter les instructions suivantes dans la méthode `testContains` :

— la création d'un polygone :

```
Polygon shape = new Polygon(
  List.of(new Vector(-5,-5),
          new Vector(5,-5),
          new Vector(4,2),
          new Vector(-6,4)
        ));
```

— la création d'un astéroïde `asteroid`, de forme `shape`, de centre $(40, 15)$, de vélocité nulle, de vélocité angulaire $90°s^{-1}$, et de taille 1.
— la mise à jour de l'astéroïde après 1 seconde (ce qui a pour effet de le tourner de 90°).
— Deux tests utilisant la méthode `contains` :

```
assertThat(asteroid.contains(new Vector(40,15)).isTrue();
assertFalse(asteroid.contains(new Vector(20,45)).isFalse();
```

Le premier test vérifie que le point $(40, 15)$ est dans l'astéroïde, le deuxième test vérifie que le point $(20, 45)$ n'est pas dans l'astéroïde.

> **Tâche 43 :** Rajoutez ces instructions et lancez le test, en cliquant sur la flèche verte dans la marge du test.

Un rapport s'affichera rapidement : si c'est vert c'est bon, si c'est rouge des erreurs sont détectées et il vous faut alors corriger votre programme.

> **Tâche 44 :** Ajouter d'autres points au test, dans et en dehors de l'astéroïde, pour bien s'assurer que les calculs soient corrects.

Ne pas choisir de points sur le bord de l'astéroïde : les imprécisions de calcul rendent le résultat de `contains` incertain pour ces points.

> **Tâche 45 :** Ajouter une deuxième méthode de test, utilisant un autre polygone et d'autres points.

6 Collision vaisseau spatial - astéroïde

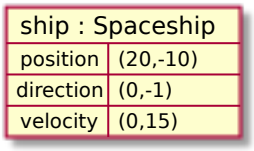
On se place maintenant dans la classe `Spaceship`.

> **Tâche 46 :** Ajouter une méthode publique `collides` dans la classe `Spaceship` dont le but est de tester si le vaisseau percute un astéroïde.

Quels sont les paramètres de cette méthode ? Quel est le type de retour de cette méthode ? Pour l'instant, faites que cette méthode retourne toujours la même valeur.

Tâche 46 : Implémenter la méthode, de sorte qu'il y ait collision si le centre du vaisseau (représenté par l'attribut `position`) est contenu dans l'astéroïde.

Considérer seulement le centre n'est pas très pertinent. La classe `Spaceship` contient une liste de points de contact : ce sont des points sur le bord du vaisseau spatial. Nous supposerons que le vaisseau entre en contact avec un astéroïde si un de ces points de contact est contenu dans l'astéroïde. Les coordonnées de ces points de contacts sont données en supposant le centre du vaisseau au point $(0, 0)$ et sa direction étant $(1, 0)$.



Tâche 47 : Lister les coordonnées absolues des points de contacts pour le vaisseau de ce diagramme.

Pour avoir une coordonnée absolue, il faut d'abord faire une rotation adaptée selon la direction du vaisseau, puis une translation par la position du vaisseau.

Tâche 48 : Compléter la méthode `Spaceship.collides`.

Utilisez une boucle `for` sur les points de contact du vaisseau. Pour chaque point calculez sa position absolue et testez si elle est dans l'astéroïde.

7 Collision vaisseau spatial - champ d'astéroïdes

Pour cette section, on se place dans la classe `Space`.

Tâche 49 : Ajouter une méthode `hasCollision`, déterminant si le vaisseau spatial est en collision avec au moins un astéroïde.

Tâche 50 : Modifier la méthode `isGameOver` pour qu'elle retourne `true` si le vaisseau spatial est en collision.

Tâche 51 : Vérifier que le jeu se fige lors d'une collision entre le vaisseau spatial et un astéroïde.

8 Vies et invulnérabilité

Pour cette section, on se place dans la classe `Spaceship`.

On souhaite rendre le vaisseau spatial invulnérable aux collisions en début de partie, et lui donner un petit nombre de vies : ainsi ce ne sera qu'à la troisième collision que la partie se termine. Après chaque collision, le vaisseau devient invulnérable pendant 3 secondes.

Tâche 52 : Ajoutez un attribut au vaisseau spatial, représentant le temps restant d'invulnérabilité, en secondes.

Si cette propriété est négative, le vaisseau est vulnérable.

Tâche 52 : Ajoutez au vaisseau une méthode `isInvulnerable()`.

Tâche 53 : Ajouter une méthode permettant de rendre le vaisseau invulnérable pour un nombre donné de secondes.

Si le vaisseau était déjà invulnérable, son nouveau temps restant d'invulnérabilité est le maximum (pas la somme1) du temps qu'il lui restait et du nombre de secondes reçu en paramètre.

Tâche 54 : Modifier la méthode `update` du vaisseau, de sorte que le temps d'invulnérabilité soit respecté.

Autrement dit, le temps restant d'invulnérabilité, s'il est positif, doit diminué avec le temps.

Tâche 55 : Modifier la détection de collision du vaisseau : si le vaisseau est invulnérable, il n'y a pas de collision.

Tâche 56 : Faire en sorte que le vaisseau soit invulnérable pendant les 5 premières secondes du jeu.

Tâche 57 : Toujours dans la classe `SpaceShip`, ajouter une méthode `getInvulnerabilityTime()`.

Tâche 58 : Dans la classe `viewModel`, ajouter des méthodes `isSpaceshipInvulnerable` et `getSpaceshipInvulnerabilityTime`.

On souhaite pouvoir visualiser le fait que le vaisseau est invulnérable. On propose plusieurs façons :

— faire clignoter le vaisseau,
— afficher une ellipse autour ou en-dessous du vaisseau,
— changer la couleur du vaisseau,
— afficher une barre de progression représentant le temps d'invulnérabilité restant.

Tâche 59 : Choisir un de ces effets, ou bien un autre selon votre imagination, et le réaliser, en modifiant la méthode `render(Spaceship spaceship)` de la classe `CanvasView` du package `views`.

Après vérification que votre programme fonctionne comme attendu, retournez dans la classe `Spaceship`.

Tâche 60 : Ajouter des *vies* au vaisseau spatial.

Pour cela, il vous faudra ajouter un attribut pour le nombre de vies avec un accesseur.
Si le vaisseau entre en collision avec un astéroïde, il perd une vie. La partie se termine lorsque le vaisseau a perdu toutes ses vies. Après chaque collision, le vaisseau bénéficie d'une période d'invulnérabilité.

Tâche 61 : Faire les modifications nécessaires dans les méthodes de `Spaceship` et `Space`.

Tâche 62 : Ajouter une méthode dans la classe `ViewModel` pour obtenir le nombre restant de vies du vaisseau.

Tâche 63 : Modifier `CanvasView`, en ajoutant une méthode `renderLives`, de sorte que pour chaque vie restante, une image réduite du vaisseau soit affichée dans un coin du canvas.

Utilisez la méthode `getImage` pour obtenir l'image du vaisseau, et consultez la méthode `renderSpaceShipImage` pour voir comment l'afficher.
Vérifier que le programme fonctionne comme attendu.

9 Classe projectile minimale

On souhaite ajouter au vaisseau la possibilité de tirer des projectiles capables de détruire les astéroïdes. Les projectiles seront des objets de la classe `Projectile`. Ils se déplacent en ligne droite à vitesse constante. Lorsqu'ils entrent en collision avec un astéroïde, le projectile et l'astéroïde disparaissent. Un projectile lancé depuis plus de 10 secondes devient inoffensif (il disparaît).

Tâche 64 : Créer dans le package `game` une nouvelle classe `Projectile`.

Un projectile est caractérisé par :

— sa position,
— sa vélocité,
— sa durée de vie restante.

Tâche 65 : Ajouter les trois attributs correspondants dans la classe `Projectile`.

Tâche 66 : Ajouter un constructeur, initialisant la durée de vie à 10 secondes, et les autres propriétés selon les paramètres reçus.

La position d'un `Projectile` devra pouvoir être consultée par le reste du jeu ainsi que par les méthodes d'affichage.

Tâche 67 : Ajouter donc un accesseur `getPosition`.

Tâche 68 : Ajouter une méthode `public void update`, qui met à jour la durée de vie et la position du projectile.

Notez que le projectile ne subit pas d'accélération, sa vélocité est donc constante.

Tâche 69 : Ajouter un prédicat `isAlive` permettant de vérifier si la durée de vie du projectile n'est pas écoulée.

Tâche 70 : Dans la classe `Spaceship`, ajouter une méthode `public Projectile fire()`.

Le nouveau projectile doit être placé à une distance de 30 (pixels) en avant du vaisseau. Il se déplace à 100 pixels par seconde droit devant le vaisseau. Puisque le vaisseau est possiblement en mouvement, il hérite aussi de la vélocité du vaisseau. Sa vélocité initiale doit donc être la somme de la vélocité du vaisseau d'une part, et d'un vecteur de même direction que le vaisseau et de norme 100 d'autre part.

10 Ajout des projectiles au modèle

Le vaisseau peut tirer de multiples projectiles. Il faut enregistrer leur position dans l'état du jeu, qui est représenté par la classe `Space`.

Tâche 71 : Dans la classe `Space`, ajoutez un attribut `projectiles` qui contiendra les projectiles actifs.

Tâche 72 : Assurez-vous que l'attribut `projectiles` soit correctement initialisé sachant qu'au début, l'espace ne contient aucun projectile.

Tâche 73 : Dans le méthode `update` de `Space`, faire en sorte que la position de chaque projectile soit mise à jour Pour cela, créer une méthode `updateProjectiles(double dt)`, qui sera appelée dans `update`, et dont le rôle est de mettre à jour la position et la durée de vie de chaque projectile.

Tâche 74 : Ajouter une méthode publique `getProjectiles` permettant d'obtenir la liste des projectiles actifs.

Tâche 75 : Ajouter une méthode `public void addProjectile`, prenant un projectile en paramètre, et ajoutant ce projectile à la liste des projectiles.

11 Afficher les projectiles

Nous avons modifié le modèle pour qu'il contienne des projectiles. Il nous faut maintenant les afficher à l'écran.

Tâche 76 : Dans la classe `viewModel`, ajouter une méthode `public List<Projectile> getProjectiles()` retournant la liste des projectiles du modèle.

Tâche 77 : Dans la même classe, ajouter une méthode `public void fireSpaceshipGun()`, qui fait tirer un projectile au vaisseau et ajoute le projectile obtenu à l'espace.

On se place dorénavant dans la classe `CanvasView`.

Tâche 78 : Ajouter une méthode `private void render(Projectile bullet)` prenant un projectile en argument. Cette méthode doit afficher le projectile dans la zone de dessin.

Pour rappel, la classe `CanvasView` contient une propriété `GraphicsContext context` qui représente la zone de dessin et les opérations de dessin que nous pouvons faire. Il nous faut :

— une première instruction pour changer la couleur du pinceau (de la même façon que dans la méthode `renderAsteroid`). Choisissez la couleur qui vous plait.

— Une deuxième instruction pour dessiner un disque de diamètre 4 : `fillOval`. Consultez la documentation (`Ctrl+Q` ou documentation en ligne, pour connaître la signification des paramètres de cette méthode. Attention à bien positionner le centre du projectile.

Tâche 79 : Modifier la méthode `render` pour qu'elle appelle la méthode `render` pour tous les projectiles.

Tâche 80 : Finalement, aller dans la classe `View`. Modifier la méthode `handleKeyPressed` pour que lorsque la touche `SPACE` est appuyée, le vaisseau lance un projectile.

Tâche 81 : Démarrer le programme. Vérifier que tout fonctionne correctement, ou faites les ajustements nécessaires.}

12 Gérer la fin de vie des projectiles

Pour l'instant les projectiles restent actifs indéfiniment. On veut les faire disparaître dès que leur temps de vie est écoulé. Les modifications suivantes ont toutes lieu dans la classe `Space`.

Tâche 82 : Ajouter une méthode `private List<Projectile> getDeadProjectiles()`.

Dans cette méthode, il faut :
— commencer par créer une liste vide,
— ensuite ajouter à la liste chaque projectile s'il est mort,
— puis retourner la liste.

Tâche 83 : Ajouter une méthode `private void removeDeadProjectiles()`.

Dans cette méthode, pour tout projectile mort, retirez le projectile mort de la liste des projectiles actifs, en utilisant la méthode de liste adéquate.

Tâche 84 : Modifier la méthode `update` : à la fin de cette méthode, retirer les projectiles morts de la liste des projectiles actifs à l'aide de la méthode écrite juste avant.

Tâche 85 : Relancer le programme et vérifier le bon fonctionnement de cette modification.

13 Destructions des astéroïdes

Nous souhaitons faire en sorte que l'impact d'un projectile sur un astéroïde fragmente cet astéroïde en plusieurs astéroïdes plus petits. Si un projectile touché est déjà tout petit, alors il sera simplement détruit. Pour commencer, faisons en sorte que l'impact d'un projectile détruise l'astéroïde (sans création de fragments).

Tâche 86 : Dans la classe `Projectile`, ajouter un prédicat prenant un astéroïde en paramètre, et déterminant si le projectile est contenu dans la zone de cet astéroïde.}

Le modèle à chaque mise-à-jour devra gérer les projectiles de la manière suivant :

1. Déplacer les projectiles selon le pas de temps `dt` et mettre à jour leur durée de vie,
2. Chercher tous les projectiles et tous les astéroïdes en contact,
3. Supprimer tout projectile en contact avec un astéroïde,
4. Remplacer tous les astéroïdes en contact avec un projectile par des astéroïdes plus petits.
5. Supprimer tous les projectiles en fin de vie.

Tâche 87 : Créer une méthode `processProjectiles(double dt)` dans `Space`. Réaliser dans cette méthode les points 1 et 5. Modifier la méthode `update` pour quelle utilise `processProjectile`.

On va rajouter les points 2, 3 et 4 dans les questions suivantes.

Tâche 88 : Créer deux variables contenant des ensembles vides dans la méthode `processProjectiles` :
— La première contiendra les projectiles en contact avec un astéroïde,
— La seconde contiendra les astéroïdes touchés par un projectile.

Tâche 89 : Toujours dans la méthode `processProjectiles`, à la suite des instructions, ajouter des instructions testant pour chaque projectile et chaque astéroïde, si le projectile est contenu dans l'astéroïde. Dans ce cas, on ajoute le projectile et l'astéroïde aux ensembles de projectiles et d'astéroïdes en contact.

Tâche 90 : À la suite, ajouter des instructions retirant chaque projectile touchant un astéroïde (les projectiles de l'ensemble construit juste avant). Vous pouvez utiliser une boucle `for` sur un ensemble.

Tâche 91 : Encore à la suite, ajouter de la même façon des instructions retirant chaque astéroïde touché par au moins un projectile.

On s'occupera de fragmenter les astéroïdes touchés plus tard, pour l'instant on se contente de les faire disparaître.

Vous obtenez ainsi une méthode `processProjectiles` complète mais longue et complexe. Nous souhaitons modifier notre nouvelle méthode `processProjectiles` pour la rendre plus lisible. Spécifiquement, nous voulons que ses instructions soient :

```
private void processProjectiles(double dt) {
  updateProjectiles(dt);
  Set<Bullet> hittingProjectiles = new HashSet<>();
  Set<Asteroid> hitAsteroids = new HashSet<>();
  findProjectileHits(hittingProjectiles, hitAsteroids);
  remove(hittingProjectiles);
  fragment(hitAsteroids);
```

On peut utiliser l'outil d'extraction de méthode (clic droit puis `Refactor > Extract > Method...`) pour prendre des instructions et ordonner à IntelliJ de les transformer en une méthode indépendante.

> **Tâche 92 :** En utilisant l'extraction de méthode et le renommage (raccourci `Shift + F6`), refactoriser la méthode `processProjectiles` jusqu'à ce qu'elle soit telle que ci-dessus.

> **Tâche 93 :** Vérifier que le programme fonctionne correctement, avec la nouvelle fonctionnalité.

14 Fragmentation des astéroïdes

On ajoute maintenant la fonctionnalité de fragmentation des astéroïdes : un astéroïde détruit est remplacé par des astéroïdes plus petits. En dessous d'une certaine taille, un astéroïde détruit disparaît sans se fragmenter (ce qui permet au jeu de pouvoir se terminer).

> **Tâche 94 :** Dans la classe `Asteroid`, ajouter une méthode `fragments` qui retourne une liste d'astéroïdes.

Pour l'instant, la méthode retourne une liste vide, nous la compléterons plus tard, pour qu'elle retourne une liste d'astéroïdes plus petits correspondant au fragment de l'astéroïde une fois celui-ci détruit.

L'attribut `size` permet de connaître la taille approximative de l'astéroïde. La classe `Space` contient déjà la définition de la taille initiale d'un astéroïde, sous la forme d'une constante.

> **Tâche 95 :** Ajouter les constantes suivantes dans `Space` :
> — la taille en-dessous de laquelle un algorithme détruit ne génère pas de fragments,
> — le nombre de fragments lorsqu'un astéroïde de taille suffisante est détruit,
> — le ratio de la taille d'un fragment sur la taille de l'astéroïde détruit (qui doit être inférieur à 1).

Revenir à la méthode `fragments` de la classe `Asteroid`.

> **Tâche 96 :** Ajouter des instructions pour créer une liste et la remplir d'astéroïdes correspondant aux fragments créés par la destruction.

Pour créer un fragment, on utilise le générateur aléatoire `Space.generator`. Les fragments créés doivent être dans la position de l'astéroïde dont ils proviennent, et avoir une taille égale à la taille de l'astéroïde dont ils proviennent fois le ratio de taille défini ci-dessus.

Tâche 97 : Dans la classe `Space`, trouver la méthode chargée de gérer la destruction de l'astéroïde. Ajouter à cette méthode les instructions pour ajouter les fragments de l'astéroïde détruit à l'espace.

Tâche 98 : Vérifiez le fonctionnement du programme et corrigez le si nécessaire.

Tâche 99 : Dans la classe `Asteroid`, ajouter les instructions pour que lorsque l'astéroïde détruit est suffisamment petit, il ne se fragmente pas, mais disparaisse simplement sans trace.

Tâche 100 : Vérifier le fonctionnement du programme.}

15 Score

On améliore maintenant la façon dont le score est calculé. On veut favoriser les tireurs les plus efficaces et réguliers, capables de détruire rapidement tous les astéroïdes. Voici le système que nous voulons implémenter.

Le score est augmenté à chaque fois qu'un astéroïde est détruit. Pour chaque astéroïde détruit, le joueur marque 10 points de base. Pour récompenser la régularité du joueur, ces 10 points sont multipliés par un entier strictement positif, le *multiplicateur*. Le multiplicateur vaut initialement 1. Il est augmenté de 1 chaque fois qu'un astéroïde est détruit (après avoir compté son score). Il redescend de 1 si pendant 3 secondes, aucun astéroïde n'est touché. Il ne peut pas descendre en-dessous de 1.

Tâche 101 : Créer dans le package `game` une nouvelle classe `Score`.

Tâche 102 : Ajoutez dans la classe `Score` un attribut `value` pour stocker la valeur du score (un `double`) initialement nul avec un accesseur.

Tâche 103 : Ajouter aussi une méthode `public void update(double dt)`. Pour l'instant cette méthode ne fait rien.

Tâche 104 : Dans la classe `Space`, modifier l'attribut `score`, pour que son type devienne `Score`. Supprimer la méthode `updateScore`, et modifier la méthode `update` en remplaçant l'appel `updateScore(score)`.

Tâche 105 : Mettez à jour les méthodes `getScore` de `Space` et de `ViewModel`, pour que celle de `Space` retourne un objet de type `Score` et que celle de `viewModel` retourne toujours un `double`.

Tâche 106 : Vérifier que le programme fonctionne. Le score devrait être 0 et ne jamais changer.

Tâche 107 : Dans la classe `Score`, ajouter une méthode privée `addPoints(double points)`, permettant d'augmenter le score d'un nombre donné de points.

Tâche 108 : Dans la classe `Score`, ajouter une méthode `public notifyAsteroidHit`. Cette méthode utilise `addPoints` pour augmenter le score de 10 points.

Tâche 109 : Dans la classe `Space`, faites en sorte que lorsqu'un astéroïde est fragmenté ou détruit, le score augmente de 10 points.

Tâche 109 : Vérifiez le bon fonctionnement du programme.

Tâche 110 : Ajouter une propriété `int multiplier` à la classe `Score`, initialement 1. Ajoutez un accesseur pour cet attribut.

Tâche 111 : Modifier la méthode `addPoints` de `Score` pour que le nombre de points ajoutés soit multiplié par le multiplicateur.

Tâche 112 : Créez une méthode `addToMultiplier` pour modifier le multiplicateur.

Elle prend en argument un entier déterminant de combien le multiplicateur change (négatif si le multiplicateur décroît). Attention, le multiplicateur doit rester strictement positif.

Tâche 113 : Modifier `Score` de sorte que le multiplicateur augmente de 1 lorsqu'un astéroïde est touché.

Tâche 114 : Dans la classe `ViewModel`, ajouter une méthode `getScoreMultiplier` retournant le multiplicateur du score.

Tâche 115 : Dans la classe `CanvasView`, consultez les instructions permettant d'afficher le score, dans la méthode `renderScore`.

Tâche 116 : Ajouter des instructions pour que la valeur du multiplicateur s'affiche aussi, en dessous du score, précédée du symbole $\times$.

Tâche 117 : Vérifiez le bon comportement du programme.

On veut faire en sorte que le multiplicateur diminue de 1 au bout de 3 secondes consécutivement sans événement. Pour cela, on ajoute une sorte de compte à rebours.

Tâche 118 : Ajoutez une propriété `multiplierTimer` permettant de mesurer le temps restant avant de devoir diminuer le multiplicateur.

Tâche 119 : Faire en sorte que lorsque le multiplicateur est modifié, le compte à rebours `multiplierTimer` est remis à 3.

Tâche 120 : Modifier `update` pour que `multiplierTimer` décroisse lorsque le temps passe. Lorsqu'il atteint 0, diminuez de 1 le multiplicateur.

Tâche 121 : Vérifier le bon comportement du programme.

Tâche 122 : Vous assurer que les instructions des méthodes ne fassent jamais référence à des littéraux numériques.

Pour cela, pour chaque littéral numérique, créez une constante (`private final static double CONSTANT_NAME = ...;`). Vous pouvez utilisez le menu `Refactor > Extract > Constant...` en sélectionnant d'abord le littéral à transformer en constante, pour qu'IntelliJ crée la constante correspondante.