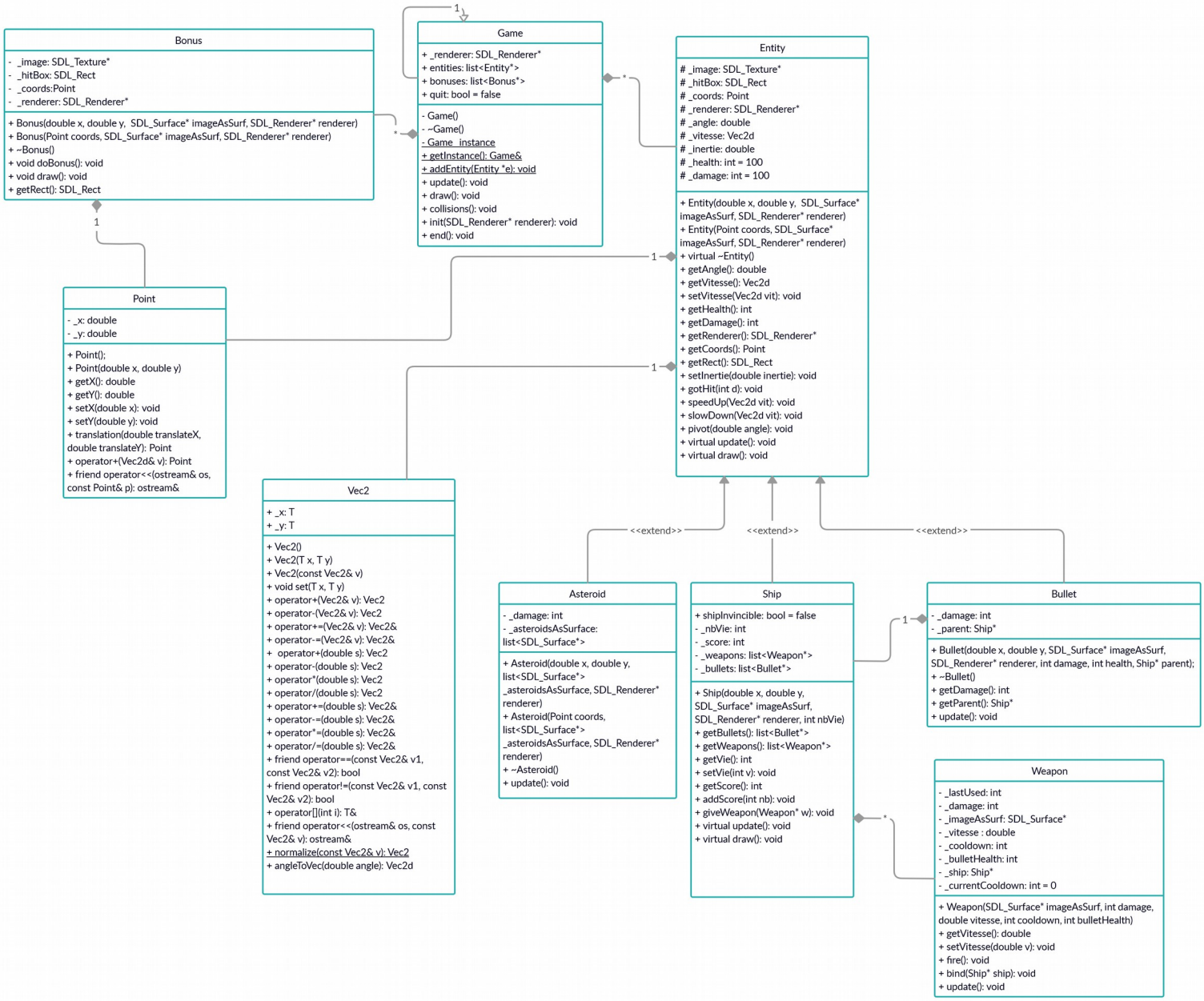
Projet Asteroids
Ezriel Steinberg (binôme : Nicholas Faucheux).

Projet Astéroïds

Étant donné que nous ne sommes que deux sur le projet, nous avons décider de ne pas se répartir les tâches à l'avance. Nous avons donc chacun travaillé plus ou moins sur chaque partie, en fonction de notre temps disponible et des nécessités. Nous avons commencé par les classes Point et Vec2 (vecteur de 2 dimensions). Sur ceux-ci, les plusieurs opérateurs ont étés surchargés, comme : l'addition entre un vecteur et un point, l'addition entre deux vecteurs, la multiplication d'un vecteur par un réel, et [] pour accéder aux éléments x et y. On a aussi ajouté la normalisation de d'un vecteur, et une fonction pour obtenir un vecteur normalisé à partir d'un angle. Vec2 est une classe à template, et peut donc être définie pour tous les vecteurs de nombres (int, char, long etc). Cependant, on utilise un typedef pour définir vec2d, vecteur de deux double, qui est le plus utilisé (pour les vitesses, en particulier).



Nous avons alors travaillé sur l'objet « Entity ». Conceptuellement, c'est un objet qui se déplace sur l'écran, en tout cas qui peux se déplacer. Ainsi, un vaisseau (Ship), qui est un joueur, ainsi que les astéroïdes, ou les balles (Bullets), sont tous des Entities. En tant qu'objet physique, une entité peut se déplacer, i.e a une vitesse ; a une force de friction, en fait une inertie qui peut être inférieur à 0, afin de la ralentir au cours du temps : c'est typiquement le cas d'un Ship mais pas d'un Asteroid. Il peut aussi pivoter. Bien sur, on a des getters et des setters pour ces attributs. Au niveau du jeu, une entité a un nombre de vie (ou santé), possède une fonction draw() pour se dessiner à son emplacement, et update() pour se mettre à jour. Pour une entité « pure », c'est juste ajouter la vitesse à la position et se téléporter d'une extrémité de

la carte à l'autre. La position étant un point, et la vitesse un vecteur, on utilise une simple addition grâce à la surcharge des opérateurs ; de même, la modification de vitesse par rapport à l'inertie est une simple multiplication. L'implémentation a été assez compliqué, car il a fallu jongler avec pas mal de notion SDL pour la rotation ; en effet, sur internet, on trouve plutôt l'utilisation de rotozoom, qui est ici absent. On a donc choisi de stocker dans chaque entité, son image non modifiée, en tant que sdl_surface ; et à chaque affichage, on génère une (sdl_)texture à partir de la surface et de son angle. La dernière fonction importante est getRect(), qui renvoie le rectangle correspondant à la hitbox de l'entité. Cela est utile en interne pour l'afficher au bon endroit, et en externe pour vérifier les collisions.

Astéroïde est à priori l'objet hérité d'Entity le plus simple. En fait, on a même débattu pour savoir s'il était nécessaire de l'ajouter, ou si on pouvait utiliser directement Entity ; on a fait le choix d'avoir une classe abstraite. Et de fait, cela c'est révélé judicieux, car l'astéroïde est en fait très particulier. En effet, là ou la destruction d'une entité classique ne fait qu'appeler son destructeur et le supprimer de la liste des entités, ici, on peut créer des astéroïdes « enfants ». Pour cela, on donne au constructeur une liste d'images, c'est à dire une image pour les astéroides du premier split, puis celle pour ceux du deuxième, et ainsi de suite. On pourrait imaginer des modifications mineures : indiquer un répertoire d'où prendre une image aléatoire, donner plusieurs images différents par split, etc. Cependant, on a préféré se concentrer sur l'ajout de fonctionnalités.

Ship est aussi une entité, doté d'un peu d'informations supplémentaires : le score, le nombre de vies restantes, ainsi qu'une liste d'armes.

Une arme n'est pas une entité, c'est un objet auquel on fournit les informations sur les balles (objet Bullets) qu'il va créer, comme leur vitesse, les dommages qu'elles infligent,… ainsi que son cooldown. Une fois une arme créé, on la « bind » à un vaisseau. Alors, on peut « tirer » l'arme : elle va alors trouver sa localisation dans l'espace grâce au vaisseau auquel elle est liée, puis va créer et ajouter une Bullet, héritant de Entity, à la liste des entités en jeu. Cette dernière a un certain nombre de point de vie, qui diminue d'un à chaque update(), faisant disparaître les anciennes balles. A noter que chaque balle possède un lien vers le vaisseau qui l'a tiré. Pour l'instant, cela sert à augmenter le score de ce joueur lorsque cette balle touche un astéroïde, mais a été pensé pour permettre un jeu en multi-joueur, en coopération en en player versus player.

Un bonus n'est pas non plus une entité ; en effet, il ne se déplace pas. Cependant, il est bien affiché à l'écran, et devrait avoir donc dans l'héritage, une place au dessus d'Entity, comme InGameElement. Pour l'instant, il a plutôt été fait comme « proof of concept » et est donc une classe à part entière. De même, nous n'en avons fait qu'un seul, mais en théorie cela devrait être une classe abstraite, avec différents types de bonus héritant de celle ci, en surchargeant une méthode pour définir l'effet du bonus. Pour l'instant, c'est plus simple : le bonus ne fait qu'accélérer la vitesse des balles d'une arme portée. Il en apparaît de manière aléatoire au cours du temps.

Game est une classe implémentant le design pattern « singleton ». Il ne peut en effet n'exister qu'un objet Game. Le principal attribut de cet objet est la liste entities, qui contient toutes les entités en jeu. Ainsi, un appel à Game::update itère sur celle-ci, en appelant update() sur chacune ; et de même pour draw(). On utilise donc le polymorphisme, ces objets pouvant en effet être assez différents. On appel aussi Game::collisions pour faire le calcul des collisions entre les entités, et faire les actions ad hoc. Game contient aussi la liste des bonus en jeu.

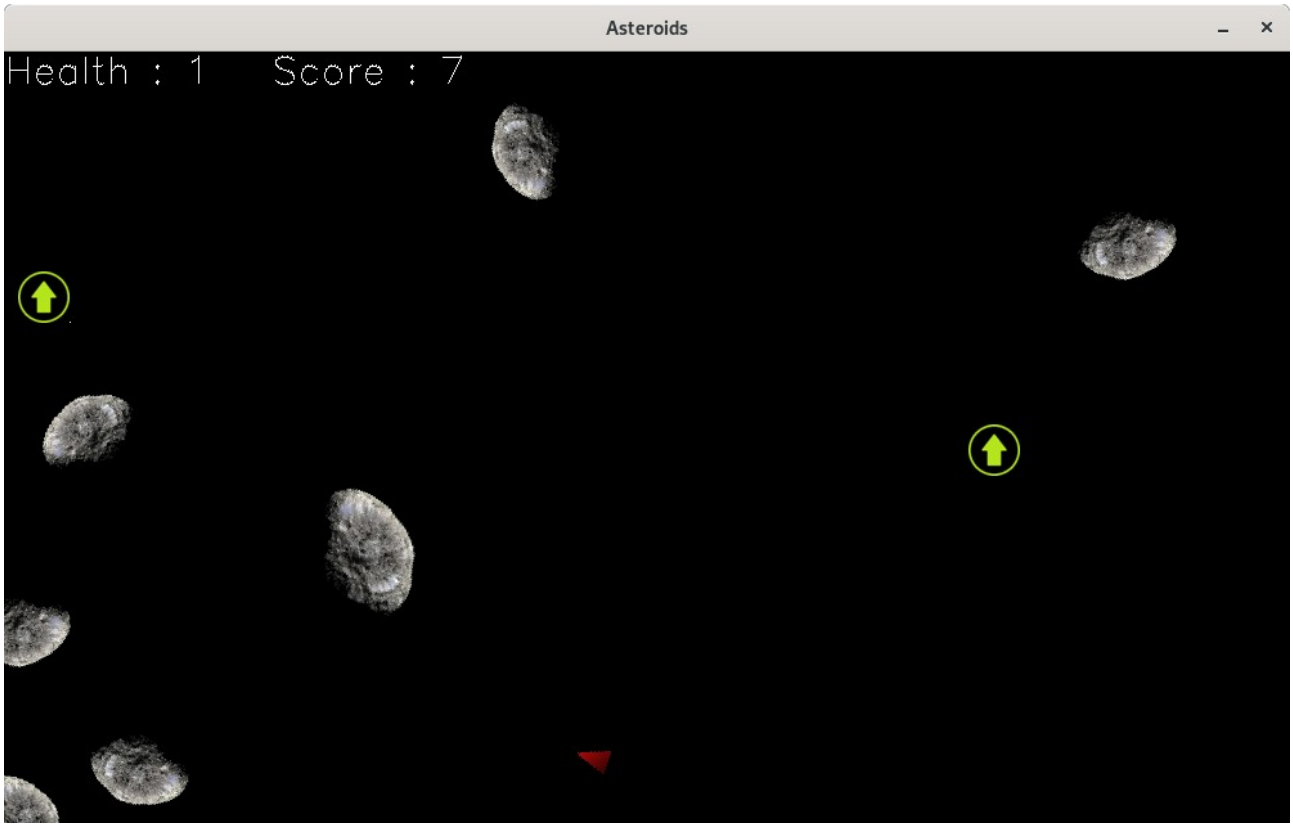
Pour l'affichage du texte, nous avons utilisé ce qui a été donné dans la consigne, bien que l'utilisation de SDL_ttf aurait à mon avis donné un résultat plus agréable. Une fonction print_char encapsule le dessin des arêtes, et print_string encapsule print_char.

Un point intéressant à souligner est l'utilisation d'un thread pour l'invincibilité des vaisseau. En effet, lorsqu'un astéroïde touche un joueur, il faut le rendre invincible pour ne pas qu'au prochain *tick* il reprenne un dégât ; on le rends donc invincible pendant 2 secondes. C'est un thread lancé en parallèle qui se charge, après les deux secondes, d'enlever cette invincibilité.

Le jeu est jouable en solo ou en multijoueur (coopération). Cependant, la gestion des deux possibilités a été faite rapidement, grâce à un code assez modulable, mais n'a pas été vraiment optimisé. En particulier, on a du code répété. Le mode multijoueur a en effet été ajouté pour montrer la possibilité d'ajouter rapidement des fonctionnalités. A terme, il faudrait rendre la classe Game virtuel, et qu'en hérite GameSolo et GameMulti (ou éventuellement GamePVP etc). ; c'est en effet la principale différence entre les différents types de jeux, à part l'initialisation qui est pour l'instant dans le main, le Game n'étant pour l'instant pas assez spécialisé.

Actuellement, le score du joueur est augmenté d'un à chaque fois qu'il touche un astéroïde ; d'autres choix auraient été possibles (quand il en détruit un, après chaque seconde de survie, etc).

Pour le design, n'étant pas designers, nous avons préféré utiliser des images trouvés, pour les astéroïdes. Cela rends certes le visuel un peu redondant, mais est à mon avis plus joli qu'un simple cercle. A noter également que si l'on a simplement utilisé la même image à des tailles différentes, le programme est fait pour utiliser n'importe quelle image (en .bmp) pour des tailles d'astéroïde différentes, ou bien même entre astéroïdes.


Asteroids
Health : 1 Score : 7


Asteroids
Health : 9 Score : 1
Health : 8 Score : 1