INFO-H-301 Programmation orientée objet
Projet : Asteroids

Dasnoy Damien, De Groote Nicolas, Delhaye Quentin

Année académique 2012–2013

1 Description de l'architecture orientée objet

Nous utilisons, comme conseillé pour ce type de programme, un design pattern de type Model-View-Controller. La vue s'occupe de gérer les entrées et sorties. Les entrées sont les touches du clavier et les boutons utilisables via la souris. Tandis que la sortie est l'affichage du jeu à l'écran. (Les différents panels et boutons ainsi que la fenêtre `Space` dans laquelle se déroule le jeu.)

Le modèle contient toutes les données et méthodes relatives au jeu en tant que tel comme les mouvements des `SpaceObjects`, la détection des collisions, la création de bonus, etc.

Le contrôleur s'occupe de faire le lien entre le modèle et la vue quand c'est nécessaire par l'intermédiaire d'interfaces. Il agit comme un « chef d'orchestre ». Il reçoit des informations de la vue ou du modèle et réagit ensuite en envoyant au modèle ou à la vue la prochaine tâche à effectuer en fonction de ces informations. Il ne stocke aucune donnée lui même mais distribue celles qu'il reçoit.

Exemple : Un bouton est actionné par l'utilisateur, la vue envoie cette information au contrôleur qui va dire au modèle ce qu'il doit faire (lancer une nouvelle partie dans `Game` si c'est le bouton `Start` par exemple) ainsi qu'à la vue (comme afficher de nouveaux éléments).

Puisque chaque élément du jeu est indépendant des autres, l'architecture orientée objet s'impose d'elle-même (en plus de l'être par l'intitulé du cours). Dans le modèle, chaque élément se déplaçant à l'écran est une instance d'un `SpaceObject` dont héritent les quatre classes particularisant ces éléments. Il en va de même dans la vue où ce sont les design qui sont instanciés. Dans chacun de ces blocs, la classe gérant la partie, `Game` dans le modèle et `Space` dans la vue, contient une ou plusieurs listes (hashtable dont la clé est l'ID de l'objet déterminé par `Game`) de ces objets. Lorsqu'un objet est créé, il est ajouté à la liste et sa destruction se fait simplement en le retirant de cette même liste à l'index spécifié par son ID.

2 Principales fonctionnalités du programme

2.1 Apparition des astéroïdes en cours de jeu

Nous utilisons pour notre jeu un système de difficulté croissante en fonction du temps. Pour cela, il faut que des astéroïdes apparaissent au cours de la partie avec une fréquence croissante.

Thread et timer : C'est dans le `GameThread`, la classe dans laquelle tourne le `run()` du modèle, que se fait l'appel de la fonction `addTimedRock` de la classe `Game`. Cette méthode ajoute des astéroïdes si nécessaire. Pour cela, la fonction récupère une valeur du timer et la compare avec le moment ou elle doit ajouter un Rock. Si c'est le cas, la classe `Game` appelle sur elle-même la méthode `addRocks` qui ajoute un Rock à la liste de rocks du modèle et en notifie la vue pour l'affichage.

La figure 1 représente le diagramme de séquence de l'ajout des `Rocks` en fonction du temps.

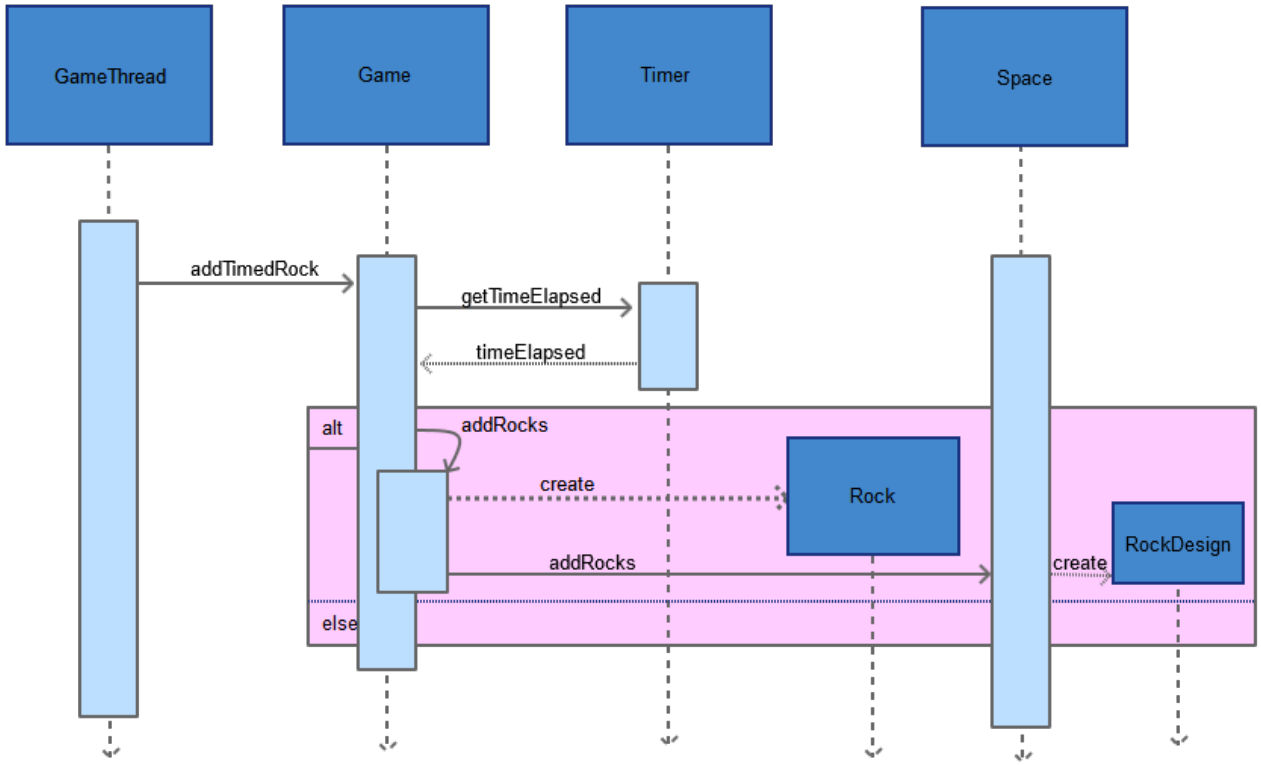

Figure 1: Diagramme de séquence de l'ajout des `Rocks` au cours de la partie.

2.2 Mise à jour des éléments

Les objets de type `SpaceObjects` ont tous au moins deux attributs : leur ID, qui permet des les identifier dans la liste des `SpaceObjects`, et leur position.

Dans `GameThread` se trouvent également les appels des fonctions de `Game` qui réalisent les mises à jour de tous les `SpaceObjects` (le `Ship`, les `Rocks`, les `Bonus` et les `Shots`). Pour les différents types de `SpaceObjects`, les méthodes sont assez semblables. Dans un premier temps, `Game` signale aux `SpaceObjects` du modèle qu'ils doivent se mettre à jour (petites différences selon l'objet). Ensuite il récupère les positions et ID des objets pour signaler à la vue (`Space`) de mettre les design à jour avec ces nouvelles positions (`RockDesign`, `ShotDesign`, `ShipDesign` et `BonusDesign`).

La figure 2 montre l'exemple pour les `Rocks`, représenté sur un diagramme de séquence.

2.3 Gestion du clavier

Le déplacement du vaisseau est contrôlé par le joueur au moyen des touches directionnelles du clavier. Il peut également tirer des projectiles avec la touche espace, la touche `P` quant à elle met le jeu en pause et affiche les meilleurs scores. La classe `CommandLstnr` permet de réaliser ces fonctionnalités. En effet, cette classe appartenant à la vue implémente `KeyListener` pour détecter si une touche du clavier est enfoncée puis relâchée. `CommandLstnr` implémente également `IMoveShip` qui permet de faire le lien avec le contrôleur qui implémente aussi cette interface. Cette interface contient notamment la méthode nécessaire pour actualiser la position du vaisseau en fonction des touches directionnelles activées `updateMoveShip()`. Elle contient aussi la méthode pour changer le nombre de tirs `updateShots()` et celle pour changer l'état du jeu (pause ou en jeu) : `updatePauseState()`. `CommandLstnr` héritant de `TimerTask`, un timer programme le lancement de ces fonctions toutes les 20ms.

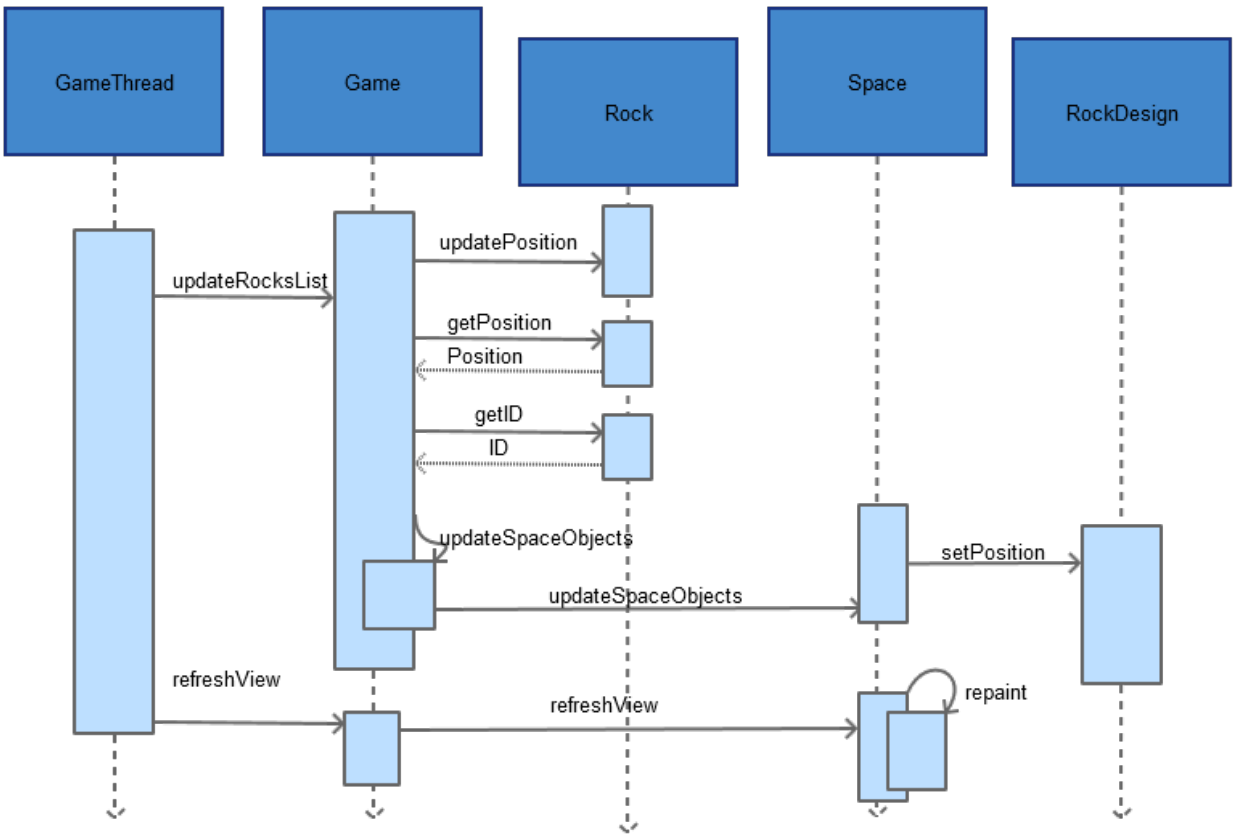

Figure 2: Diagramme de séquence de la mise à jour d'un `Rock`.

2.4 GameThread

Tout au long d'une partie en cours (pas en pause) le thread contenu dans `GameThread` est lancé par un timer implémenté dans `Game`. Dans sa méthode `run()`, plusieurs méthodes sont exécutées à tour de rôle. Il s'agit de `detectCollision()`, `addTimedRock()`, `addTimedBonus()`, `updateRocksList()`, `updateShotsList()`, `updateBonusList()`, `updateShip()` et `refreshView()` qui sont appelées sur l'objet `game`, une instance de la classe `Game` représentant la partie en cours. Ces méthodes servent respectivement à

1. Détecter les collision entre `SpaceObjects`.
2. Ajouter des rochers en fonction du temps.
3. Ajouter des bonus en fonction du temps.
4. Mettre à jour la liste des rochers : enlever les rochers explosés, ajouter les nouveaux et mettre à jour les positions.
5. Mettre à jour la liste des tirs : enlever ceux ayant expirés ou rencontré un rocher, ajouter les nouveaux tirs et mettre à jour leur position.
6. Mettre à jour la liste des tirs : enlever ceux ayant expirés ou rencontré un rocher, ajouter les nouveaux tirs et mettre à jour leur position.

7. Mettre à jour la position du vaisseau.

8. Demander à la vue de se rafraichir.

2.5 Détection des collisions

La fonction `detectCollision()` de la classe `Game` appelée par la méthode `run()` de `GameThread` est chargée de vérifier les différentes collisions possibles entre `SpaceObjects`. Pour commencer, celle-ci parcourt la liste des Rocks et compare chaque position à celle des éléments de la liste des Shots et à celle du vaisseau. Enfin, en parcourant la liste des bonus, elle compare les positions de ceux-ci avec la position du vaisseau. Grâce à la fonction `isOn()` commun à tous les `SpaceObjects`, elle peut aisément vérifier si deux objets se superposent et donc entrent en collision. Plusieurs cas sont possibles en fonction des objets qui se superposent :

- Tir sur un rocher : invocation de la méthode `rockCollision` qui vérifie si le rocher n'en est pas encore à sa taille minimale. Le cas échéant, deux nouveaux astéroïdes sont créés et le plus gros est détruit ainsi que le tir à la base de cette destruction.

- Rocher sur un vaisseau : le vaisseau perd un point de vie. Si ce compte tombe à zéro, le vaisseau est détruit et la partie est terminée.

- Vaisseau sur un bonus : le bonus est appliqué et supprimer des listes de la vue et du modèle.

2.6 Gestion des bonus

Pour aider le joueur dans sa quête vers le meilleur score, le jeu fera apparaître régulièrement des bonus. En effet, à chaque exécution du thread principal de la partie (`GameThread`), la méthode `addTimedBonus()` est invoquée. Cette dernière est fort similaire à `addTimedRock()`, sauf en un point essentiel : la probabilité d'apparition d'un bonus décroît en fonction du nombre de bonus déjà apparu : `if(probabilité < 0.95-Nombre de bonus déjà apparus/100)`. Si l'utilisateur a de la chance, le jeu créera alors un bonus (dans le modèle et son alter ego dans la vue) ainsi qu'un `TimerBonus` qui est un compte à rebours avant que l'objet ne disparaisse.

Il y a alors deux possibilité :

- l'utilisateur ne ramasse pas le bonus et le timer arrive à son terme. Le jeu ordonne alors à la vue de supprimer ce bonus que l'utilisateur n'a daigné prendre et le supprime de la liste des bonus présents.

- L'utilisateur ramasse le bonus. Le modèle annule donc le timer de disparition, supprime le bonus de sa liste et demande à la vue d'en faire autant. Le bonus est ensuite appliqué et si ce dernier a un effet temporaire, un nouveau `BonusTimer` est créé pour le surveiller. Au terme du temps imparti, le timer demande au modèle d'annuler le bonus via `unApplyBonus()` et tout rentre dans l'ordre.

Ce méchanisme est présenté à la figure 3.

2.7 Pause

Un jeu sans temps mort peut être éprouvant, c'est pourquoi une pause est souvent la bienvenue, permettant par exemple de se prendre un petit café..

Au cours d'une partie, l'utilisateur peut activer la pause à tout moment en appuyant sur la touche P. La classe gérant le clavier, `CommandLstnr`, change alors son booléen pause et l'envoie au contrôleur

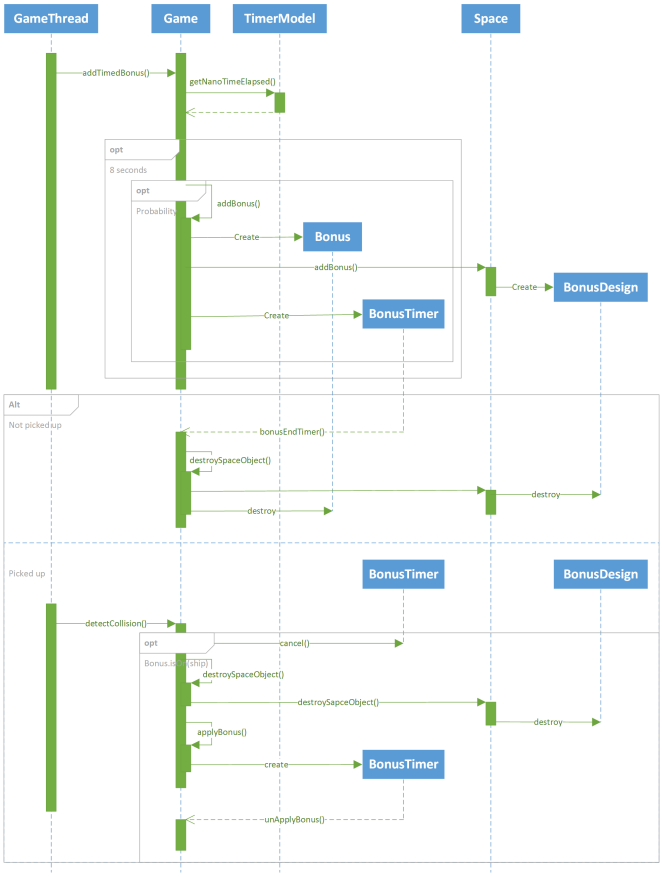

Figure 3: Diagramme de séquence du cycle de vie d'un bonus.

via `updatePauseState()`. Le contrôleur analyse ensuite ce booléen en le comparant à son propre état de pause. Si l'état à changé, l'appel à `changePauseState()` est déclenché.

Cette méthode fait plusieurs choses : elle récupère les highscores auprès du modèle pour les afficher sur le panel de pause, change l'état de la pause dans le modèle et enfin, informe la vue qu'elle doit échanger le panel de jeu, `Space`, avec celui de pause, `PausePanel`.

Lorsque le modèle est informé qu'il doit changer son état de pause, il peut réagir de deux façons différentes selon que l'on active ou désactive la pause :

- si la pause est activée, il arrête la programmation du lancement du thread principal, `GameThread`, ainsi que l'incrémentation du timer tenant à jour le temps écoulé depuis le début de la partie, en sauvegardant en mémoire ce temps écoulé. Néanmoins, le timer des différents bonus, lui, n'est pas arrêté. La pause ne peut avoir que des avantages, en laissant tourner ces timers, on désencourage le joueur à y faire appel.
- Si la pause est désactivée, les timers précédemment annulés sont relancés au même rythme et celui comptant le temps écoulé, `TimerModel`, est relancé avec le temps d'arrêt mis en mémoire.

Cette implémentation est représentée à la figure 4.

2.8 Options

À tout moment l'utilisateur peut appuyer sur le bouton `Settings` qui affichera une nouvelle fenêtre comme le montre la figure 5. Cette fenêtre permet de choisir parmi sept niveaux de difficulté, du plus

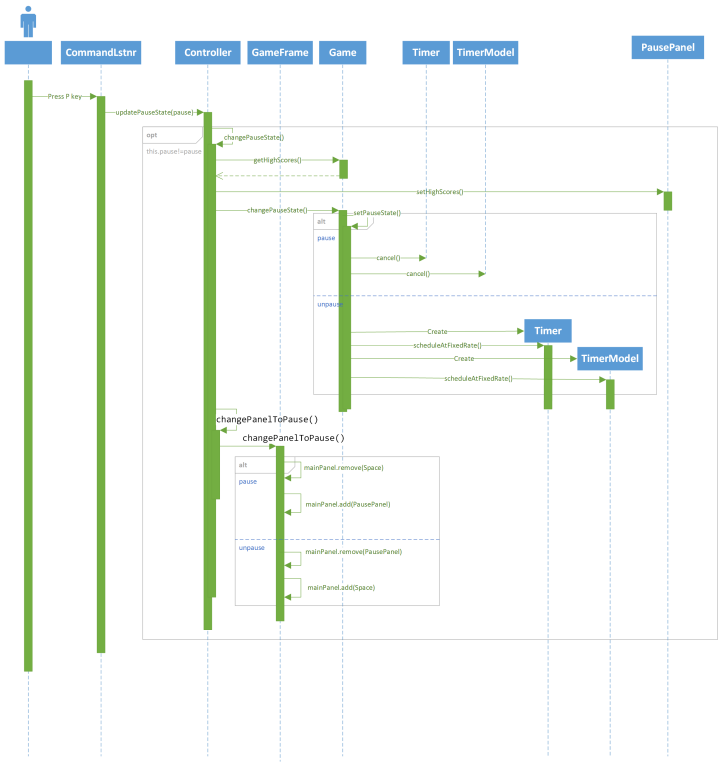

Figure 4: Diagramme de séquence de l'activation de la pause par l'utilisateur.

simple au plus ardu.

Une fois le choix fait, appuyer sur le bouton `Apply` (invisible à l'image puisqu'il se trouve sous le menu déroulant) ferme cette fenêtre et envoie un message au contrôleur en l'informant de la nouvelle difficulté. Cette dernière n'est pas appliquée à la partie en cours, il est nécessaire de redémarrer la partie puisque le niveau de difficulté est envoyé au modèle via la méthode `newGame()` appelée à tout début de partie.

2.9 High Scores

La partie se termine quand le vaisseau rencontre un rocher et que ça vie passe à 0 ou moins. À ce moment-là, la fonction `detectCollision()` de `Game` vérifie si le score atteint à la fin de cette partie est un nouveau record grâce à la méthode `checkHighScore()` de la classe `Score`. Cette méthode ouvre un fichier texte où sont enregistrés les meilleurs scores précédents et les place dans un vecteur avec la méthode `getHighScores()` (qui est également utilisée pour récupérer les high scores quand on met le jeu en pause). Ce vecteur de scores est ensuite trié par ordre décroissant par la méthode `sortHighScores()`. Si le score actuel de la dernière partie est supérieur à un score sauvegardé, on remplace le plus petit d'entre eux par le nouveau record et on retrie le vecteur. Enfin, ce nouveau vecteur est sauvegardé dans le fichier grâce à la méthode `saveScore()`.

3 Captures d'écran

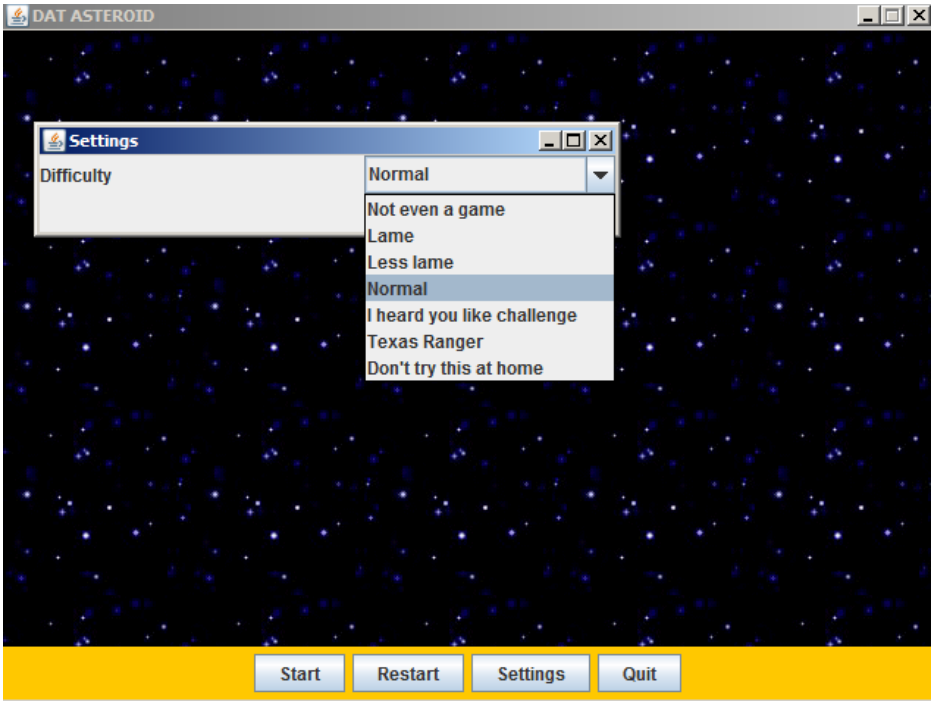

Figure 5: Options

Figure 6: Fin de partie

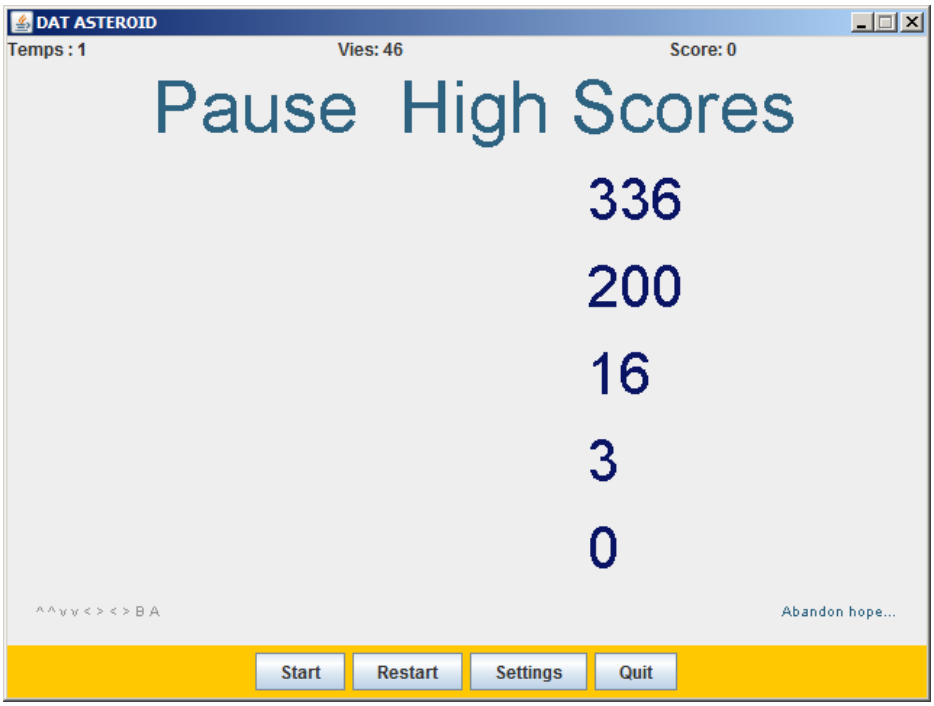

Figure 7: Pause

Figure 8: In game