Thomas Vadebout - Léa Lortholarie - Dianne Raad

Billard

2019-2020

☆ Cahier des charges du programme

Description

Jeu de billard sur ordinateur en vue du dessus de la table. Il faut 2 joueurs humains pour jouer. L'IA ne sera pas implémentée dans un premier temps. On propose un billard qui suit les règle du pool anglais. [7 billes rouges, 7 billes oranges, 1 bille noire et 1 bille blanche].

Le jeu doit pouvoir se jouer à la souris. La queue se positionne automatiquement autour de la bille blanche. Le joueur choisi dans un premier temps l'angle de tir. Puis il choisi la force du tir.

Une interface graphique est utilisée pour montrer les élements de jeu. De plus, le score est affiché en haut de la fenêtre.

Les sciences

Les sciences sont présentes dans ce jeu par le biais de la mécanique. En effet, la seconde loi de Newton est utilisée pour déteminer l'accélération des différentes billes en fonction des forces qu'elles vont recevoir.

De plus, nous allons gérer les collisions des balles entre elles. Ainsi que les collisions avec les murs et les trous.

Améliorations possibles

- Faire un fonctionnement par équipe avec potentiellement plusieurs joueurs.
- Respecter l'intégralité des règles d'un vrai jeu de billard.
- Faire une fenêtre à la fin pour indiquer quelle équipe a gagné.
- Faire une fenêtre principale permettant de changer les options de la partie.

☆ Principe de l'algorithme pour répondre aux problèmes

Gestion des contacts entre les billes durant un tour

L'une des principales problématiques de notre projet était la gestion des collisions entre les billes lors de leur déplacement.

C'est la méthode executerTour qui va permettre d'effectuer tous les mouvements liées à un tir de la part du jour. Pour la gestion des contacts, nous avons de choisi de travailler en fractionnant le temps où les billes bougent.

Pour chaque intervalle de temps successif, chaque bille qui possède une vitesse va effectuer un mouvement en fonction de sa vitesse. Elle va alors se trouver à une nouvelle position et son accélération sera mise à jour. Cette nouvelle accélération va permettre de calculer la nouvelle vitesse. On effectue des actions seulement sur les billes qui sont considérées en mouvement pour éviter de surcharger le programme en tâches inutiles. Dans la méthode actualisation, on sélectionne les billes en mouvement grâce à estEnMouvement de Bille puis on appelle actualiser sur chacun d'entre elle.

Ensuite, pour chaque bille qui a bougé, on va chercher toutes les interactions qu'elle aurait pu avoir avec tous les autres éléments du plateau. A chaque fois q'une interaction a eu lieu, on crée un contact. On va répertorier tous les contacts qui ont eu lieu pendant cet intervalle de temps et pour chacun, on va changer la vitesse des billes qui ont eu ce contact. Pour gérer les contact, il a été choisi de créer une classe Contact. C'est la fonction déterminerCollision qui va créer une liste de tous les contacts : pour les billes en mouvement, elle va créer un contact avec tous les éléments du plateau grâce à la méthode recoitContact. On ne garde dans la liste que les contacts qui ont eu lieu, donc les contacts non nuls. Puis pour chaque contact, la méthode traiterCollision va modifier la vitesse des objets concernés en fonction du contact. On recommence ces intervalles de temps jusqu'à ce que plus aucune bille ne soit en mouvement. C'est la méthode tourEstTerminé qui va verifier que plus aucune bille ne bouge.

La vitesse étant un double, il arrivait souvent qu'elle ne soit jamais précisément égale à zéro, la fonction estEnMouvement va donc considérer qu'a partir d'une certaine vitesse, la bille est statique. La méthode va donc forcer sa vitesse à zéro et retourner que la bille est immobile à partir de cette vitesse prédéfinie.

Gestion des évènements d'un tour en fonction des règles

Lorsqu'un tour se termine, il est nécessaire d'appliquer les pénalités en fonctions des règles et du coup qui vient d'être joué.

Un tour de billard se résume à une action faite par le joueur au début du tour : déterminer un angle et une force de tir. Une fois ces actions effectuées, la classe MoteurPhysique prend le relais pour executer la physique d'un tour. Cette physique peut amener à des conséquence sur le score, les fautes, la validité d'un tour. Nous avons donc ajouté la classe DescriptionTour qui se charge de déterminer les conséquences importantes d'un tour. Pour cela, MoteurPhysique appelle des méthodes pour mettre à jour les informations importantes : par exemple, quelle est la première bille entrée en contact avec la bille blanche ou la liste des billes tombées dans les poches pendant le tour. Au fur et à mesure des mises à jour, la classe DescriptionTour prend en comptes les différentes fautes de jeu. À la fin du tour, l'instance de DescriptionTour liée au tour est récupérée par Plateau qui se chargera

d'appliquer les pénalités, la possibilité de rejouer si le joueur a empoché une bille de son équipe et il actualisera le score.

Choix du tir par l'utilisateur

L'interaction principale de l'utilisateur est le fait d'effectuer un tir sur la bille blanche, nous avons décidé d'utiliser l'interface graphique pour récupérer les informations que l'utilisateur va devoir transmettre.

Dans un premier temps, nous avons décidé que l'utilisateur pourrait seulement tirer dans le centre d la bille qui était considérée comme un point. Il pouvait alors faire naviguer sa souris dans tout le terrain afin de choisir le vecteur force du tir qui était le le vecteur reliant sa souris au centre de la bille. On utilisait un MouseMotionListener pour mémoriser la position de la souris à tout moment. Lorsqu'un nouveau tour est lancé, l'attribut tirJoue de PanelJeu est null et on attend que l'utilisateur le définisse dans la fonction attendreTir. Dès lors ou l'utilisateur clique sur le terrain, on définit tirJoue dans la fonction MousePressed grâce au vecteur reliant la position de la souris au centre de la bille blanche. Lorsque tirJoue est défini, donc que l'on est dans la phase de déplacement des billes, l'affichage lié au choix du tir n'est pas affiché par paint.

Le problème qui se posait avec l'utilisation de cet interface était le suivant : lorsque la bille blanche était près d'un bord, il était impossible d'effectuer un grand tir pour l'éloigner du bord puisque la souris ne pouvais faire un vecteur que sur le plateau.

Dans un second temps, la solution retenue à été de garder le suivi et le clic de la souris pour définir un vecteur unitaire qui caractériserait la direction de la force appliquée. Puis grâce à un curseur, on peut faire varier la force et la valider grâce à un bouton. Pour l'affichage, la méthode paint va afficher une queue qui va tourner avec la souris autour de la bille blanche et qui va s'immobiliser lorsque la direction est validée par un clique. Ensuite, quand le curseur va varier, on va afficher une estimation de la position de la bille après ce tir si elle ne rencontrait aucun obstacle, jusqu'au moment ou le joueur valide la norme, dès lors, la motion des billes se lance et les affichages liés au tir disparaissent.

PanelJeu permet donc à la fois de choisir la norme mais affiche aussi les éléments du plateau en temps réel.

☆ Problèmes rencontrés et solutions trouvées

Implémentation des règles du jeu

Normalement toutes les règles du jeu sont implémentées. Nous n'avons pas vu de bugs liés à une mauvaise interprétation des règles du jeu. Les règles du jeu que nous utilisons sont décrites sur la fenêtre règle du jeu qui est affichée en appuyant sur le bouton "Règles du jeu ».

Bugs liés aux contacts

Les contacts sont une partie centrale du programme. Ils doivent être traités avec la plus grande rigueur et précision. Malgré les efforts que nous avons fait pour s'occuper correctement les contacts, il existe des bugs. La classe intermédiaire contact n'était peut-être pas la meilleure idée, mais elle permet déjà d'enlever une bonne partie des problèmes. De plus, cela rend le code plus facilement maintenable. En effet, toutes les actions suivant un contact sont décrites dans la classe Contact.

~ Des billes qui passent à travers les elements !

Un bug majeur que nous avons résolu permettait aux billes de traverser les éléments. C'est à cause du temps informatique qui est discret. Le contact n'est pas vu exactement au bon moment, donc deux objets se chevauchent avant que le contact soit considéré. Étant donné que les objets pouvaient être en partie superposés, ils étaient considérés comme tout le temps en contact. Cependant, avec notre façon de gérer les contacts, les deux objets ne pouvaient plus se séparer.

Pour régler ce bug, on applique une correction de décalage proportionnelle au chevauchement (profondeurContact dans notre code). Ce décalage est opéré selon la normale au contact (c'est la direction selon laquelle le décalage à faire un minimal).

La correction n'est pas physiquement parfaite, mais au regard de la vitesse d'actualisation, elle est suffisante.

~ Les contacts ne se font pas dans le bon ordre

Comme avant, du fait que le temps est discret, plusieurs contacts étaient détectés en même temps. Cependant, les contacts doivent tout de même être traités dans le bon ordre pour ne pas nuire à l'expérience de jeu.

Pour régler ce bug, nous trions la liste des contacts détectés lors d'une actualisation. Ce tri est basé sur la profondeur des contacts. Un contact profond doit être traité avant un contact peu profond. Cette solution n'est pas parfaite, mais elle permet déjà de supprimer une bonne partie des incohérences physiques lorsque de nombreux contacts ont lieu en même temps.

☆ Bibliographie

- Tracé d'un trait plus épais : https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics2D.html

- Utilisation du JSlider et de la méthode ChangeEvent: https://www.developpez.net/forums/d1742950/java/interfaces-graphiques-java/awt-swing/agents-placement-fenetres/jslider-changelistener/
- https://stackoverflow.com/
- https://docs.oracle.com/javase/7/docs/api/

☆ Diagramme UML commenté

Cf Annexe

☆ Suggestions d'amélioration

Code spaghetti

Bien que nous avons fait l'effort de bien séparer nos classes en limitant leur champ d'action, certains objets sont modifiés dans des classes qui ne devraient pas modifier ces objets au vu des permissions que nous voulions donner à ces classes. Par exemple, listeElements qui est une liste gérée par Plateau est passée à MoteurPhysique pour qu'il puisse faire ses traitements. Il va même jusqu'à modifier la liste en question en supprimant les billes qui sont tombées. Le programme marche bien, mais il est plus difficile à maintenir. Nous avons décidé de ne pas corriger ce mauvais design car cela aurait complexifié le code.

Une solution aurait été de passer la liste d'éléments avec la méthode unmodifiableList(..). La liste n'aurait pas pu être modifiée directement par MoteurPhysique. À côté de ça, pour tout de même gérer la suppression des billes tombées, il aurait fallu créer une interface BilleTombeListener implémentée par Plateau afin d'avoir une méthode onBilleTombée(Bille b) appelée au moment où une bille tombe par MoteurPhysique. Cela complexifie largement le code, mais le rend plus propre et maintenable.

Il y a plusieurs endroits où le code suit un design mal pensé. Cependant, pour un petit projet comme le notre, c'est suffisant pour ne pas trop se prendre la tête tout en ayant un jeu fonctionnel.

Fenêtre principale qui ne marche pas

Lorsque l'on ajoute la FenetrePrincipale, il n'est plus possible de lancer la partie. En effet, quand on appuie sur le bouton "Lancer Partie", la Fenêtre de jeu s'ouvre, mais les éléments ne s'affichent pas. Le constructeur de Plateau s'execute correctement. Nous avons tenté des debugs à l'aide de quelques System.out.println( ) placées à des endroits stratégiques.

Nous n'avons pas trouvé pourquoi ce bug est présent. Nous avons donc désactivé la Fenêtre Principale. La partie se lance automatiquement avec des paramètres prédéfinis.

Améliorations possibles

Possibilité de mette le jeu en réseau
Possibilité de changer la taille du terrain
Possibilité de rentrer les différents joueurs

☆ Echéancier

Cf Annexe

☆ Pourcentage d'implication des membres du groupe

Thomas : 43%
Léa : 37%
Dianne : 20%