Alexis WAMSTER, Djabrail KHAPIZOV, Mikhail ANANI

Mars- Avril 2024

Rapport SAE41_2023

Morpion Solitaire

https://grond.iut-fbleau.fr/wamster/SAE41_2023

IUT Sénart Fontainebleau

Sommaire

Introduction

Dans ce rapport, nous explorons le développement d'une application Android dédiée au Morpion Solitaire, un jeu de réflexion classique à un joueur. Ce jeu, traditionnellement pratiqué sur une feuille à carreaux, commence par dessiner 36 intersections en forme de croix grecque. Les joueurs y tracent des lignes, horizontales, verticales, ou diagonales, reliant quatre intersections déjà marquées à une cinquième, initialement non marquée, étendant ainsi la croix à chaque coup. La partie se conclut lorsque plus aucun coup n'est possible, le score étant déterminé par le nombre total de coups effectués, le record actuel étant de 178 coups.

Notre objectif est de transposer cette expérience ludique dans un environnement numérique, en proposant une application qui non seulement simule fidèlement le jeu sur papier mais enrichit également l'expérience utilisateur grâce à des fonctionnalités interactives et une interface intuitive. Développée en Java, cette application se veut un projet collaboratif, réalisé en binôme ou trinôme, qui mettra à l'épreuve nos compétences en programmation acquises au cours de notre formation.

L'application propose un menu principal permettant de démarrer une nouvelle partie ou d'accéder aux options, une activité de jeu où la feuille de jeu, potentiellement infinie, peut être déplacée dans toutes les directions. En outre, le système reconnaît automatiquement la fin de la partie, affiche le score final et offre la possibilité de naviguer sur la feuille pour contempler le travail accompli. Des règles optionnelles peuvent également être activées pour varier l'expérience de jeu.

Fonctionnalités

Notre programme permet de jouer au morpion solitaire. Pour cela, il faut cliquer sur le bouton jouer.

Lorsqu'on lance une partie, nous arrivons sur une grille avec une croix constituée de croix plus petites. à partir d'ici, il est possible de:
- Se déplacer en utilisant deux doigts;
- Tracer un trait faisant glisser son doigt du point de départ du trait jusqu'à son point d'arrivée;
- Il est également possible de recentrer la grille sur la croix de départ en double cliquant.

Lorsqu'un trait est correctement placé, il devient plus clair et fait apparaître une nouvelle croix. Il fait également augmenter le score de 1.
Lorsque l'utilisateur n'a plus la possibilité de tracer de trait, un popup avec le message "Félicitations !" apparaît à l'écran. Celui-ci réapparaît à chaque fois que l'utilisateur essaie de tracer un nouveau trait alors qu'il ne peut plus.

Lorsque l'utilisateur tourne son écran, les données de la partie ne sont pas perdues.

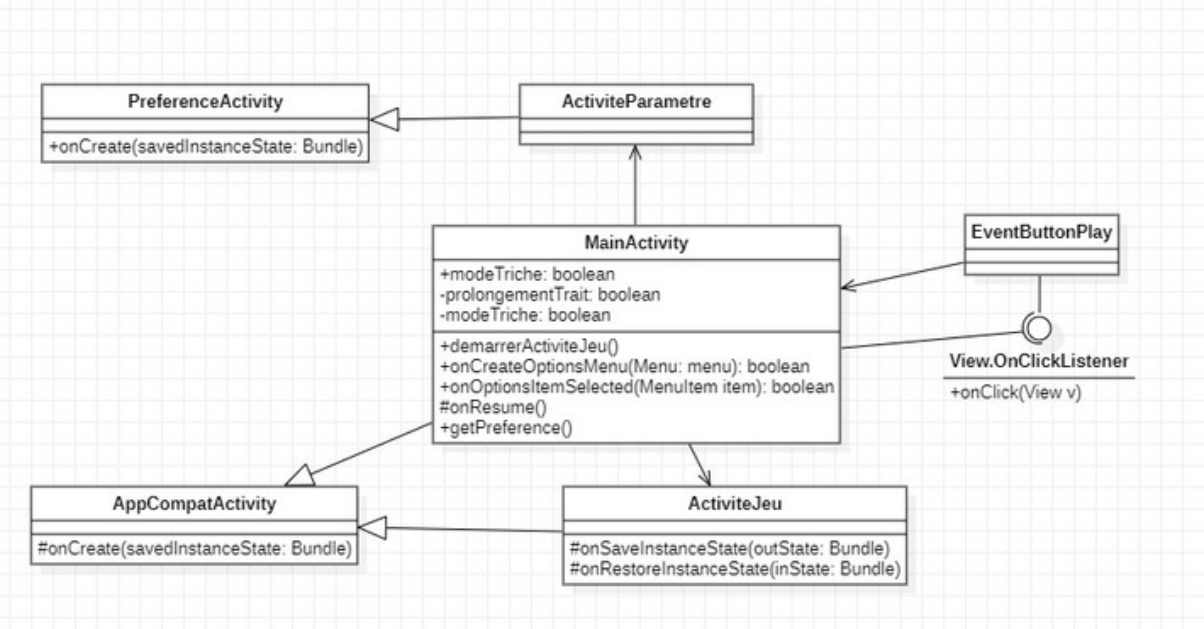
L'utilisateur peut, quand il le souhaite, revenir sur le menu principal en utilisant la fonctionnalité retour de son smartphone.

Le menu principal, en plus d'offrir la possibilité de jouer, permet aussi de régler plusieurs options:
- Commencer le jeu à partir d'une croix petite ou grande;
- Activer ou non l'option qui permet de prolonger un trait;
- Activer ou non l'option triche (qui permet de montrer au joueur un emplacement de croix qu'il à la possibilité de placer en traçant le trait
correspondant).
Ces options sont conservées même si l'utilisateur quitte l'application pendant plusieurs années sans la relancer.

Encore une fois, l'utilisateur peut quand il le souhaite revenir sur le menu principale en utilisant la fonctionnalité retour de son smartphone.

Structures

Activité principale

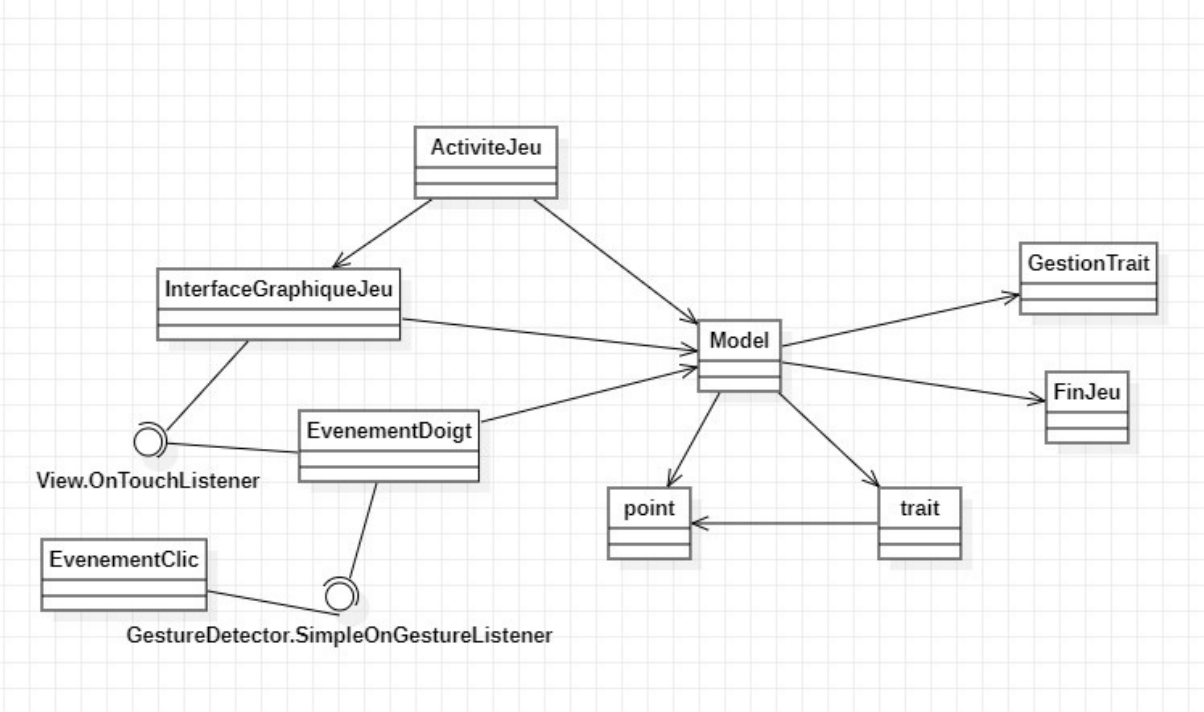


Lorsqu'on lance le jeu, l'accueil de l'application se lance. C'est grâce à la classe MainActivity et à sa méthode onCreate(). A chaque fois que l'utilisateur arrive sur cette activité, la classe récupère les préférences du joueur avec la méthode getPreference() appelée dans le onCreate() et le onResume().

Pour changer ces préférences, le joueur accède à l'activité des paramètres en appuyant successivement sur deux boutons gérer par le système via les méthode onCreateOptionMenu() onOptionItemSelected() (cf. Activité paramètre).

Pour lancer le jeu, l'accueil contient un bouton "jouer". L'événement lié à ce bouton est géré par la classe EventButtonPlay qui utilise l'interface View.OnClickListener. Lorsqu'on appuie sur le bouton, on appelle la méthode demarrerActiviteJeu() de la classe MainActivity pour lancer le jeu en envoyant toutes les préférences stockées dans les attributs dans un bundle.

Activité Jeu



Le démarrage du jeu commence avec la méthode onCreate() de la classe ActiviteJeu. Cette classe:

- récupère la classe Interface Graphique Jeu du fichier xml.
- récupère les préférences via le bundle;
- créer le model Model avec les préférences associé;
- associe ce model à l'interface graphique pour que l'interface puisse récupérer et afficher les informations de la parties situé dans Model;
- créer un écouteur d'événement EvenementDoigt qui détecte les différents gestes tactiles effectuer par le joueur, et agit sur le modèle en conséquence.

Cette écouteur implémente l'interface View.OnTouchListener et cache également un autre écouteur d'événement EvenementClic qui lui va plutôt détecter les différents types de clic avec l'interface GestureDetector.SimpleOnGestureListener. Ces deux écouteurs d'événement associés agissent tous les deux sur le modèle et permettent de couvrir tous
les gestes tactiles standard.

- L'écouteur d'événement est ensuite ajouté à l'interface graphique pour être fonctionnel.

ActiviteJeu permet aussi de sauvegarder les donnée du jeu lors d'une rotation d'écran par exemple en les sérialisant ou en les désérialisant dans un bundle avec les méthode OnSaveInstanceState() et OnRestoreInstanceState().

Les croix et les traits du jeu sont stockés dans modèle sous forme d'un dictionnaire de ce type:

Clef : position d’une croix
Valeur: liste des traits qui passe par cette croix

Pendant le jeu, les écouteurs d'événements agissent sur Model, et Interface Graphique accède au donnée du modèle pour afficher correctement le jeu.

Model utilisé pour faire ces calculs d’autres classes utiles comme Point et trait pour faire des opérations sur les points et les traits. D’autres classes ont été créées comme FinJeu et GestionTrait pour vérifier si le jeu est terminé et si un trait est valide. Ces deux classes permettent d’alléger en méthodes les classes Model et Trait (qui sont déjà bien remplies).

Les classes Trait et Point sont en vérité utilisées un peu partout dans le code mais pour simplifier le diagramme, ces classes ne sont reliées qu’au model.

Activité paramètre

Le diagrammes de l’activité paramètre tient une une classe: ActiviteParametrre. Lorsque celle- ci est appelée, elle se contente d’afficher le fichier xml correspondant. Nous n’avons pas jugé utile de mettre un écouteur d'événement qui ajuste les préférences du joueur dès qu’il coche une case, puisque les préférences ne concernent que le cœur du jeu. Elles peuvent donc être récupérées seulement quand le joueur lance une partie.

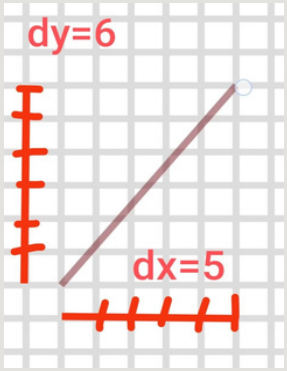
C’est donc MainActivity qui récupère les préférences au moment opportun.(cf. Activité principale)

Algorithme qui décide si un coup est légal

L'algorithme qui décide si un coup est légal connaît la position de départ et d'arrivée du trait que l'utilisateur essaie de tracer.

A partir de ces deux positions un premier filtre s'applique pour éliminer les traits qui ne font pas la bonne taille et qui n'ont pas la bonne orientation.

On regarde:
- la taille de la longueur en abscissed x
- la taille de la longueur en ordonnéd y

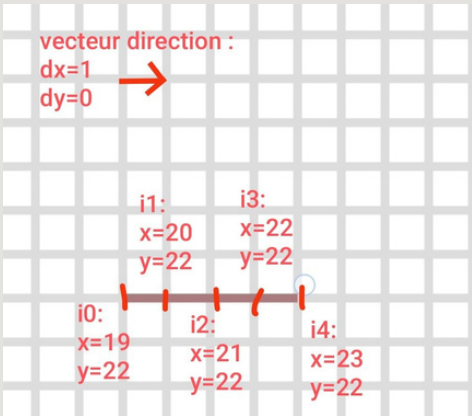


La longueur et l'orientation du trait est accepté si:

- dx et dy font tous les deux exactement la taille requise (4)
- une de ces deux longueurs vaut 0 et l'autre vaut la taille requise (4)

Pour le second filtre, on utilise les données de la partie. Pour stocker les informations de la partie on utilise un dictionnaire de ce type:
- Clef: position d'une croix
- Valeur: liste des traits qui passe par cette croix

Si le trait à passer le premier filtre, on calcul son vecteur direction. en gardant le signe de x et y (-1, 0 ou 1).

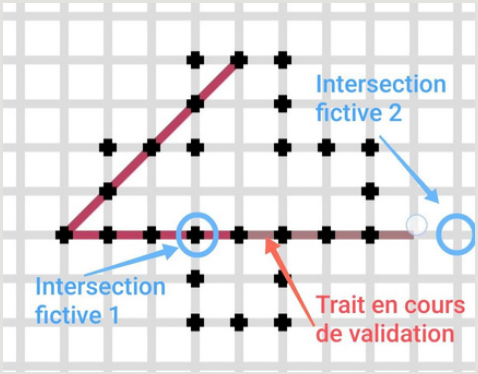


Ensuite, on parcourt en boucle chaque intersection du trait. Pour obtenir les points situés aux intersections, on part du point de départ, et on effectue une translation de ce point par le vecteur direction (précédemment calculé) pour chaque intersections que l'on veut obtenir.

Pour chaque intersections, on essaie de lire le dictionnaire. Si il nous renvoie quelque chose c'est que l'intersection en question est située sur une croix.

On compte donc le nombre de fois où le dictionnaire nous renvoie quelque chose pour connaître le nombre de croix sur lequel repose le trait. Si le trait ne repose pas sur le bon nombre de croix (4) alors il est directement éliminé.

En aucun cas, les traits ne doivent pouvoir se superposer. Il faut donc également vérifier que le trait que l'on trace n'est pas plus d'une intersection commune avec un autre trait. Pour cela, au fur et à mesure qu'on parcourt les intersections du trait en lisant notre dictionnaire, on conserve l'historique des traits croisés (donné par le dictionnaire). Si il y a un trait qu'on croise deux fois, alors c'est que notre trait est superposé sur un autre. Le trait que l'on est en train de tracer n'est donc pas validé. Lorsqu'on désactive l'option "prolongement des traits", il faut également veiller à ce que deux traits ne se prolongent pas. Exemple: sur l'image le trait translucide essaye de prolonger un trait existant. Pour éviter cela, lorsqu'on parcourt les intersections du trait, on parcourt aussi deux intersections fictives: avant l'intersection de départ et après l'intersection d'arrivée. En effet, si nous prolongeons un trait, ce trait aura forcément deux intersections communes avec la version agrandie du trait que nous essayons de tracer. On utilise alors le même algorithme que précédemment en ajoutant ces deux intersections fictives pour vérifier qu'il n'y a pas de prolongement.



Attention : Lorsqu'on compte le nombre de croix sur lequel repose un trait, il ne faut pas compter les croix des intersections fictives.

Pour calculer les positions des intersections fictives on utilise simplement le vecteur direction.

Si le trait est validé, on parcourt une seconde fois ses intersections et on ajoute dans le dictionnaire une référence vers le nouveau trait pour chacune des intersections.

Conclusion

Alexis WAMSTER

Développer l’application c’est toujours quelque chose que j'apprécie. Réfléchir aux algorithmes, corriger les bugs et voir le résultat final se concrétiser est pour moi une source de dopamine. En plus de ça, je trouve que notre équipe est plutôt complémentaire et c’est très appréciable de travailler dans ces conditions. Mais même si nous n’avions pas beaucoup de temps avec nos 3 projets à finir en 4 semaines, j’aurais aimé faire un jeu un peu plus ambitieux qu’on à envie de montrer aux gens. (Le morpions solitaire ça reste quand même un jeu moyennement impressionnant et très vite lassant).

Djabrail KHAPIZOV

Premier projet de développement mobile, une bonne découverte et interface différente, c’est motivant pour la suite et puis avec les autres projets en parallèle cela me donne des idées de projet perso avec l’utilisation d’API pour une app mobile. Délai assez court néanmoins on a su se départager les tâches et mener à bout ce projet.

Mikhail ANANI

Lancer notre premier projet Android en groupe, un jeu de morpion solitaire, ça a été franchement une aventure de fou. On a tous plongé là-dedans sans vraiment savoir où on mettait les pieds, mais on voulait créer un truc cool ensemble. Franchement, ça a été un mix d'apprentissage intense sur le dev Android et de galères à s'accorder sur tout. Mais au final, c'est ouf de voir ce qu'on a réussi à faire ensemble. On sort de cette expérience non seulement avec un jeu dont on est super fiers, mais aussi avec des potes sur qui compter et plein de skills en plus. C'était pas toujours facile, mais on l'a fait, et c'est ça qui compte.