API - Dobble

Alexis Rollin - Antoine Saget

Organisation

Nous avons commencé par prendre 1h pour réfléchir à nos différentes structure et à notre organisation. Nous avons utilisé un repository git pour pouvoir mettre en commun notre travail. Et nous avons décidé de commencer par les 3 grosses parties que nous avions identifié : la lecture des cartes dans un fichier, la détection du clic sur un icone, le positionnement aléatoire des icones sur une carte.

Les structures

Le jeu est représenté par un `plateau` qui contient des `cartes` dans une pioche ainsi qu'une carte `haut` et une carte `bas` ainsi qu'un `icone` commun aux deux images.

Une `carte` est représentée par des `icones`.

Un `icone` est représenté par une `image`, une `position`, une `rotation`, un `scale` mais aussi une `vitesse` et une `acceleration` nous expliquerons leur utilité dan la partie suivante). Afin de simplifier la détection d'un clic sur un icône et le placement des icônes, les icônes sont approximés à des cercles dans nos calculs.

Une `image` est simplement l'index de l'icone.

`vitesse`, `acceleration` et `position` sont des vecteurs (`vect2`).

Le placement aléatoire des icones

Pour le placement aléatoire des icones nous avons choisis la solution de la simulation physique où les icones sont attirés par le centre et ce repoussent entre eux :

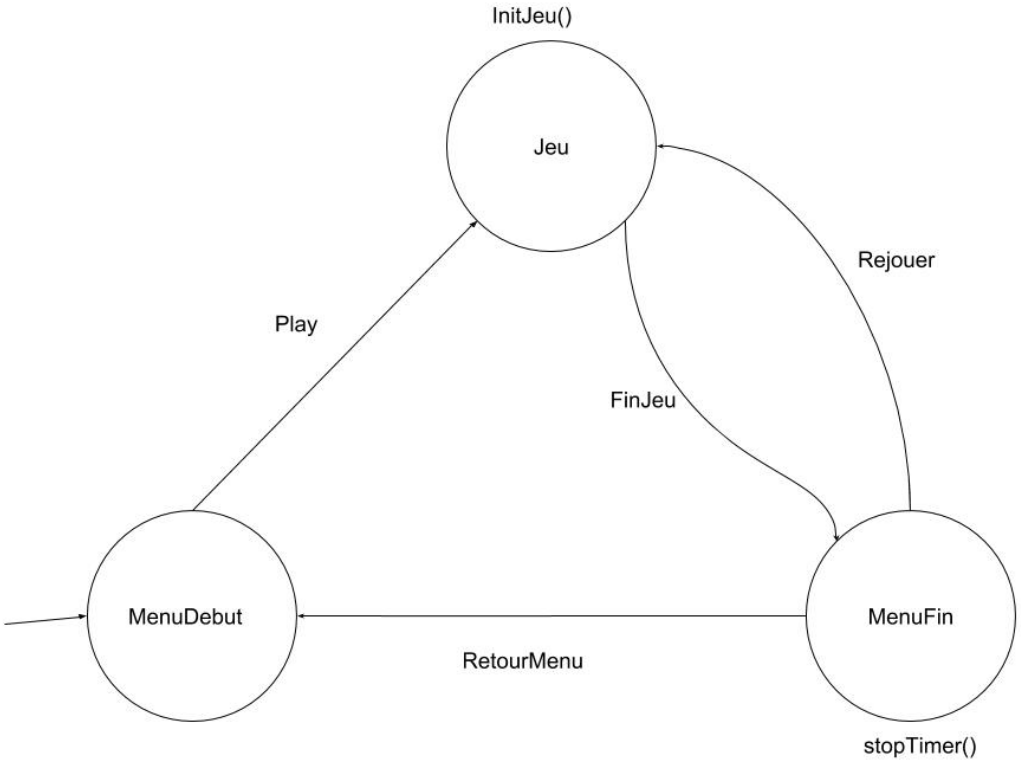
- On commence par placer chaque icone sur la carte à des angles prédéfinis mais des distances, rotation et taille aléatoire
- On fait évoluer les icones jusqu'à ce que la carte soit valide (qu'il n'y ai pas de chevauchement et pas de dépassement hors de la carte)
- Si trop d'étapes d'évolution on passées sans que la carte soit valide alors on recommence du début (cela peut arriver quand il y a beaucoup d'icones et qu'ils sont trop gros) -> Nous garantissons le fonctionnement de cet algorithme jusqu'à 10 icones par carte

Les forces appliqués à chaque étape d'évolution sont :

- Une force d'attraction vers le centre
- Une force de répulsion vers l'extérieur si on est trop prêt du centre
- Une force de répulsion si l'on est trop proche d'un autre icone
- Une force de frottement opposée à la vitesse de l'icone

La gestion des différentes phases de jeu

Pour gérer les différentes phase de jeu `MenuDebut`, `Jeu` et `MenuFin` nous avons décider d'utiliser un automate. Ce n'est pas forcément nécessaire car ici les comportements de menus sont très simple, mais cela à l'avantage d'être facilement améliorable si l'on veut ajouter des menus (et c'était l'occasion d'appliquer ce qui a été vu en cours d'automates).



On peut ainsi facilement passer d'un état à un autre avec la fonction `etatSuivant`.

Grâce à cet automate, on peut définir un comportement globale des fonctions : `onMouseMove`, `onMouseClick`, `onTimerTick` et `renderScene`. Et définir un comportement spécifique des chacune des fonctions propre à chaque état. Cela permet par exemple d'avoir facilement les 3 affichages différent.

Le clic sur un icone

Pour détecter si l'utilisateur a cliqué sur un icône donné, une fonction `estDansIcone` a été implémentée. Son principe: regarder si la distance entre le centre de l'icone et la position du clic est inférieure au rayon du cercle approximant l'icone. Si cette distance est bien inférieure, cela signifie que le clic est bien dans le cercle, et donc, dans l'icone. On applique cette fonction à tous les icones de la carte haute pour déterminer sur quel icone le joueur a cliqué. Une fois cet icone connu, il suffit de le comparer à l'icone commun aux deux cartes pour affirmer si le joueur a cliqué sur l'icone gagnant ou non.

Le choix aléatoire sans remise des cartes du plateau

Comme dit précédemment, le plateau doit avoir une carte haute et une carte basse, toutes eux différentes entre elles, et différentes des cartes précedemment utilisées. Pour cela, pour chaque carte haute et basse, on choisit aléatoirement une carte dans la pioche, et si elle a déjà été prise, on procède à nouveau à un tirage. On réitère le tirage aléatoire jusqu'à avoir une carte qui n'a pas été prise. Pour savoir quelles cartes ont déjà été utilisées, on se sert d'un tableau de marques: quand la carte d'indice i dans la pioche a été choisie, on met à la valeur à l'indice i du tableau de marques à 1. Quand toutes les cartes ont été jouées, on remet ce tableau de marques à 0. Un historique des 3 dernières cartes tirées a aussi été implémenté: il est utile après une remise à 0 du tableau de marques pour éviter d'avoir deux cartes identiques à la suite, la première tirée avant la remise à 0 et le deuxieme après.

Le temps

Tâche	Antoine	Alexis
Préparation	1h	1h
Lecture du fichier de cartes	1h40	
Trouver l'icône sur lequel on a cliqué		2h30
Debug et problème de compilation	1h	0h30
Recherche de la position d'un icone sur la matrice	0h20	
Placement aléatoire des icones	6h	
Commentaire, relecture	1h	1h
Création du programme principal	2h	
Optimisation placement aléatoire des icones	1h	
Choix aléatoire d'une carte sans remise et tests		2h
Initialisation, Score et timer		2h
Compte-rendu	2h	
Fichier d'icones et de cartes en paramètres du programme		1h
Recherche d'icones WATI-B		128h

Les améliorations possibles

Voici quelques améliorations possibles auquels nous avons pensé :

- Le clic sur la carte du haut ou sur la carte du bas
- Penser dès le départ aux automates et contruire notre code là-dessus car nous ne pensons pas en avoir utilisé tout le potentiel des automates
- Séparer dans une structure différente la vitesse et l'accélération des icones car ces deux vecteurs n'ont un intérêt qu'à l'initialisation du programme
- Améliorer la génération aléatoire des cartes pour que la taille soit fonction du nombre d'icone sur la carte
- Améliorer la gestion des ressources graphiques car actuellement on ne libère les ressources qu'au moment de quitter le programme...
- Tweaker les forces de la simulation physique pour que la génération de carte aléatoire soit encore plus rapide (nous l'avons déjà beaucoup améliorer)