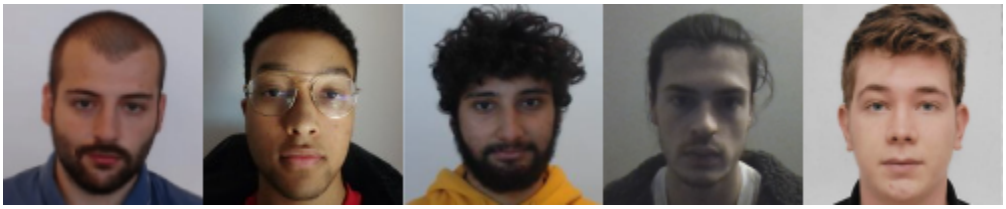
Rapport final de Projet
Table Interactive *PlayBoard*

Théo BERILLON, Jean-Baptiste CHEVRIER, Yohan ISMAEL,
Adrien LENOIR, Alexander LHUILIER

9 janvier 2022

Contents



Notre équipe se compose de 5 étudiants de 2e année, Théo BERILLON, Jean-Baptiste CHEVRIER, Yohan ISMAEL, Adrien LENOIR, Alexander LHUILIER, motivés à faire prendre vie au projet ***PlayBoard*** *et doués d'un éventail de compétences relativement large.*

1 Introduction

Le but de ce rapport est de permettre à d'autres élèves de pouvoir continuer à développer ce projet ou de s'inspirer pour développer un projet similaire.
L'idée initiale de ce projet était la création d'une table possédant un écran tactile, faisant de la reconnaissance d'objet, et le développement de jeux vidéo sur cette table qui utiliserait cette reconnaissance d'objet.
La création de ce produit complet en 80 heures était difficile à réaliser avec nos moyens, nous avions décidé de nous focaliser sur une seule partie de ce projet : la création de tactile avec de l'infrarouge, la création d'algorithme utilisant de l'intelligence artificielle pour faire de la reconnaissance d'objet ou le développement de jeux vidéo pour un produit de ce type.
Nous avions contacté une entreprise appelée Interactive Scape qui a bien voulu nous prêter son matériel afin que nous puissions développer des jeux vidéo ; et nous nous sommes donc attaqués à cet aspect du projet.

2 TUIO

TUIO est un protocole permettant la communication entre une **interface physique** : un écran déjà tactile utilisant une technologie capacitive ou encore une caméra détectant la position des doigts; et une **application logicielle**. Ce protocole vise à généraliser les communications entre ces deux types d'interfaces. Plusieurs profile existent au sein du protocole TUIO :

- Les **curseurs** correspondent à un point sur une interface physique (un doigt sur un écran par exemple).
- Les **objets** correspondent à un objet physique reconnaissable par l'interface physique : cela peut être un objet communiquant lorsqu'il est chargé par induction sur un écran ou un objet possédant un signe distinctif reconnaissable par une caméra.
- Les **blobs** correspondent à un ensemble de plusieurs curseurs pris dans une disposition particulière, reconnue par le logiciel. Par exemple, plusieurs doigts posés très proches les uns des autres sur un écran tactile peuvent être considérés comme un blob.

Le protocole TUIO est encodé avec le format OSC (Open Sound Control), puis envoyé en UDP au port 3333 par défaut.

Il existe plusieurs librairies suivant le protocole TUIO, et ce dans différents langages de programmation. On appelle TUIO Client une application écoutant et décodant des messages TUIO; et TUIO Tracker une application envoyant des messages TUIO.

3 Pratique

Dans notre projet le TUIO Tracker était intégré dans le hardware et le firmware de la table, qui possède un bouton ON/OFF pour la fonctionnalité TUIO (lorsqu'on est en OFF, la table fonctionne seulement comme un écran, le tactile et la reconnaissance d'objets ne fonctionnant pas). Il y avait un premier client afin de faire fonctionner le tactile sur Windows, mais nous avons utilisé des clients TUIO dans nos jeux afin de pouvoir accéder aux données (position, angle etc). Dans les jeux créés en Java nous avons utilisé un client défini dans la librairie TUIO_11 et pour les jeux créés en C#/Unity un client défini dans la librairie TUIOsharp.
Dans chacun de nos jeux, nous avons repris un modèle de classe dans laquelle on a défini une liste statique accessible depuis les autres classes. Lorsqu'un objet est posé/enlevé sur la table, un événement se déclenche et une méthode abonnée à cet évènement ajoute/supprime l'argument de l'évènement dans la liste. Cet argument est un objet de la classe TuioObject qui contient des informations comme la position et l'angle de cet objet, que nous utilisons ensuite dans nos jeux.

De notre côté, le protocole TUIO a un fonctionnement assez proche de la bibliothèque "awt" et ses divers listeners, assez classique du langage JAVA. Voici comment nous le manipulons dans la pratique. On instancie d'abord un *TuioClient*, auquel on ajoute via une méthode dédiée une instance de classe implémentant l'interface *TuioListener*, avant de lancer la communication entre les deux.
L'interface *TuioListener* fournit des méthodes telles que addTuioObject(TuioObject tobj) ou removeTuioObject(TuioObject tobj), appelées automatiquement lorsque l'on ajoute ou retire un tag TUIO sur/de la surface de la table. On se sert de ces fonctions de la façon décrite plus haut.

Les *TuioObject* en question représentent donc des tags TUIO, et sont munis de méthodes simple d'utilisation telles que getX(), getY() ou getAngle(), renvoyant les informations sur la position et l'orientation desdits tags.

Le protocole TUIO est issu d'un projet collaboratif et est totalement libre de droit.

4 Unity / C#

4.1 Gestionnaire de scènes et des informations récupérées via le protocole TUIO

Une scène en Unity correspond à une fenêtre qui nous sert à placer les différents éléments de notre décor. Dans le projet réalisé, il y a par exemple une scène pour le menu, une scène pour faire correspondre un objet physique à un joueur puis une scène pour un jeu.
La classe la plus importante est GameManager.cs . Il s'agit d'un singleton dont l'objectif est de gérer les différentes scènes, et de contenir les informations liées aux objets/curseurs Tuio, car le principe d'un singleton est de pouvoir être accessible depuis toutes les autres classes.
Afin de factoriser du code et de ne pas répéter une classe GameManager.cs au sein de chaque scène, il a été choisi de placer chacune des scènes à l'intérieur d'un GameObject et de placer toutes ces scènes contenues par un GameObject à l'intérieur d'une scène Unity unique nommé GameManager.

Il existe d'autres alternatives mais celle-ci a été choisi. Il y a une démarche à suivre lorsque l'on souhaite ajouter une nouvelle scène:

- La copier dans un GameObject
- Ajouter ce GameObject dans la liste "Scenes" de la classe GameManager.cs
- Ajouter un Tag pourtant le nom de cette scène à ce GameObject
- Ajouter ce nom dans l'énumeration "GameState" de la classe GameManager.cs
- Ajouter à la fonction *UpdateGameState* de la classe GameManager.cs un cas supplémentaire dans l'instruction de sélection (case switch)

Dans la méthode *ConnectTuio* de la classe GameManager.cs on écoute le port 3333 à l'aide de la ligne "tuioServer.connect();" . Ensuite on abonne nos méthodes *OnObjectAdded*, *OnObjectUpdated* et *OnObjectRemoved* aux évènements correspondants. Ainsi, lorsqu'un utilisateur pose un objet sur la table la méthode *OnObjectAdded* est appelée, lorsqu'un objet se trouve sur la table la méthode *OnObjectUpdated* est appelée et lorsqu'un objet est retiré de la table la méthode *OnObjectRemoved* est appelée. Nous avons défini une liste publique et statique "listObj" dans la classe GameManager.cs et on ajoute/retire un TuioObject lorsqu'un objet est posé/retiré sur la table. Cette liste est ainsi accessible depuis les autres classes correspondant par exemple au code d'un jeu.

4.2 Jeu du morpion

Le jeu possède deux scènes qui lui sont liées.
Dans la première scène *AddPlayerScene* on effectue un affichage de texte demandant aux joueurs de placer l'un après l'autre un objet sur la table afin d'assigner un objet au premier joueur puis un objet au deuxième joueur. La seconde scène "Morpion" sert au déroulement du jeu.
Pour la partie programmation au sein de la classe Morpion.cs, on teste les positions des objets (fonction *whichCase*), si la case sur laquelle l'objet se trouve est vide on dessine alors le pion (par l'instantiation d'un prefab Unity). On vérifie ensuite si trois pions du même joueur sont alignés ou encore si la partie s'est finie en match nul (fonction *grillTest*).

4.3 Jeu du SpaceSpeed

Le jeu possède également deux scènes: "SSAddPlayerScene" où chaque joueur doit placer un objet sur une planète bleu, et "SpaceSpeed" sert au déroulement du jeu. Le principe de ce jeu est de devoir enlever le plus rapidement un objet (appelé totem) de la table lorsque l'on a une carte en commun avec un autre joueur.
La partie programmation de ce jeu se décompose en deux classes : la classe Player.cs contenant les informations relatives aux joueurs : nombre de points, s'il est autorisé à jouer, si son totem est bien dans la zone ; et la classe SpaceSpeed.cs pour le fonctionnement du jeu.
Dans cette classe on vérifie qu'un joueur est prêt avec la fonction *CheckIfTotemInZone* pour le totem et *CheckIfCardInZone* pour la carte. Si tous les joueurs sont prêts, on fait un compte à rebours avec la fonction *CountDown* et on affiche les cartes de tous les joueurs dans la fonction *Update* en même temps que l'on vérifie quels joueurs peuvent potentiellement gagner des points avec *WhoIsAuthorized*. Si aucun joueur ne peut gagner, le jeu demande aux joueurs de placer une nouvelle carte au bout d'un certain temps, sinon le jeu reste dans un état d'attente de savoir quel joueur gagne le point. La détection de la levée du totem se fait dans la méthode *OnObjectRemoved* de la classe GameManager.cs.

On vérifie qui a gagné et attribut les points avec la méthode *WhoWon*. Le calcul des points gagnés se fait de la manière suivante:

- pointsGagné = nbre de joueurs ayant une carte en commun -1 + pointBonus
- pointBbonus = nombre de tour où le joueur n'a pas pu jouer

4.4 Jeu du TrigHockey sur Unity

Le jeu ne possède qu'une seule scène "TrigHockey". Ainsi l'association entre un joueur et un objet se fait directement dans le jeu, le premier joueur à placer un objet est le joueur 1, le deuxième le joueur 2... Le but du jeu est de marquer des points aux autres adversaires en tapant dans une balle virtuelle avec les objets réels. 3 classes sont utilisés afin de faire fonctionner ce jeu, ainsi que la physique du moteur de jeu Unity. La classe Goal.cs sert à compter le nombre de but. Lorsqu'un évènement de type 'choc' a lieu entre un but et la balle on ajoute un à ce but (fonction *OnTriggerEnter2D*). La classe THPlayer.cs sert à bouger le GameObject en forme de cercle dans le jeu en même temps que l'objet réel associé au joueur. La classe BouncyBall.cs calcule les trajectoires de la balle (lors de rebondissement par exemple): les calculs sont intégrés dans Unity.

5 Les petits chevaux

5.1 Généralités

Nous sommes partis de l'idée que le jeu des petits chevaux, un grand classique des jeux de société, méritais une adaptation numérique, d'autant plus que l'interaction humaine entre le plateau, les pions et le dé est possible grâce à la table que nous possédons.
Pour ce faire, nous avons codé en Java, en utilisant la librairie JavaFx. Le jeu a été mis à jour avec GitHub, ce qui nous permet de garder un historique des versions et des changements apportés au fur et à mesure de l'avancement du projet.
L'objectif est de réussir à, d'une part, coder le jeu des petits chevaux tel qu'il existe aujourd'hui, et d'autre part d'y ajouter toute la partie interaction avec la table connectée, en utilisant le protocole TUIO, et les Tags fournis par l'entreprise Interactive Scape.

5.2 La classe Board

Le principal atout de Java est la possibilité de faire interagir des objets, ce que nous avons fait en distinguant les différentes composantes du jeu. La première et principale classe du jeu est celle du plateau, le Board. C'est dans cette classe que l'on créé les équipes, que l'on initialise le plateau de jeu (l'image), les chevaux de chaque équipe, les positions de départ, et c'est aussi avec cette classe que l'on actualise le plateau après qu'un joueur ait joué.
On y retrouve donc le constructeur Board, qui est appelé en début de partie, qui permet donc d'initialiser tous les objets. Une des principales difficultés rencontrées concerne les positions des images, et leurs dimensions. En effet, le jeu doit être adapté à tous les écrans proposés par Interactive Scape, soit donc en résolution du Full HD ou de la 4K, et en dimension de 32 à 65 pouces. Pour cela, il est indispensable de récupérer les dimensions de l'écran sur lequel le jeu est joué. On utilise ensuite des variables qui seront utilisées pour placer les différentes images, notamment celles des chevaux et des résultats du dé.
La classe Board contient également les trois fonctions principales d'actualisation du jeu : *changePosition*, *associateAction* et *update*. Dans l'ordre d'importance, *update* vérifie si la partie est finie, c'est-à-dire si un des joueurs placé ses 4 chevaux sur les cases finales, et dans le cas contraire, récupère le résultat du dé si celui-ci a été lancé et cherche à y associer l'action correspondante. C'est alors qu'on fait appel à *associateAction*. Cette deuxième fonction va vérifier si avec le résultat du dé, un mouvement est possible (pour le joueur actuel). Si c'est le cas, elle cherche à récupérer la position du Tag du joueur concerné, et si celui si est sur un cheval déplaçable, elle fera appel à la fonction *changePosition*. Si le Tag n'est pas au bon endroit, on attendra de retourner dans cette fonction et de vérifier à nouveau sa position. Si aucun cheval du joueur ne peut être bougé, on passera automatiquement au tour suivant. Enfin, la fonction *changePosition* fera les interactions graphiques nécessaires, et actualisera au passage la variable position du cheval déplacé. On y vérifiera dedans si la case sur laquelle le cheval va est occupée, au quel cas il faudra en réinitialiser la position.

5.3 La classe Horse

Deuxième classe primordiale, la classe Horse. Elle contient toutes les informations relatives à chaque cheval, notamment sa position, son numéro, et son état. Le constructeur est simple, et n'initialise que la position, l'état et l'image associée au cheval. On retrouve dans la classe la fonction *isHorseMovable*, qui est appelée par exemple dans la fonction update, et qui vérifie si un cheval en particulier peut être déplacé.

On en déduira une deuxième fonction *isOneHorseMovable* qui renvoie True si un des 4 chevaux du joueur est jouable.
Deux fonctions ont été pénibles à coder mais sont absolument nécessaires au jeu. Ce sont les fonctions *getXPos* et *getYPos* qui, comme leur nom l'indiquent, renvoient des coordonnées correspondant à une position de cheval. C'est là ou le fait d'avoir récupéré les dimensions de l'écran est crucial, puisque ce sont les variables associées que nous utilisons dans cette fonction, à savoir la taille d'une case, la taille d'une ligne, et surtout le milieu horizontal et vertical de l'écran.
Cette classe Horse est associée à la classe Team, mais étant secondaire, on n'y accordera pas tout une partie. Elle regroupe simplement les 4 chevaux d'une même équipe, ainsi que leur couleur.

5.4 La classe Dice

Élément très important de notre intérêt pour les petits chevaux puisqu'il ramène l'interaction humaine avec la table connectée, le dé occupe une part très importante dans le jeu. C'est dans cette classe Dice que nous avons commencé à utiliser des Timers. Ils permettent de ne pas pratiquer la stratégie de polling, c'est-à-dire se placer dans une boucle while et vérifier en permanence si une action à eu lieu. Le timer nous permet de réguler cette vérification, en la faisant que lorsque le timer émet un signal, une action, en Java c'est une ActionEvent.
Lorsque c'est le tour d'un nouveau joueur, on démarre donc le timer (pour ne pas le laisser tourner quand le jeu n'en a pas besoin), et on fait une série de vérifications. Nous entrons alors dans les détails de l'utilisation de la librairie TUIO, que nous détaillerons ultérieurement. La fonction appelée par le timer est *actionPerformed*. On récupèrera l'angle du Tag associé au dé, et on transformera cette valeur qui est entre 0 et -2π en résultat de lancer de dé entre 1 et 6.
Après avoir actualisé les variables associées au lancer de dé, on traitera ce résultat lors de l'appel de la fonction *update* et on passera au tour suivant.

5.5 La classe TUIO

Cette classe ne porte pas très bien son nom puisqu'elle ne provient pas de la librairie en elle-même mais nous permet d'initialiser le nombre de joueurs, d'expliquer les règles et de récupérer les données des Tags qui seront utilisés pour le dé et chacun des joueurs. On retrouve également dans cette classe Tuio un timer, qui nous évite de rester coincé dans une boucle while. 3 fonctions principales sont présentes. La première est *getNumberOfPlayers*, qui nous permet comme son nom l'indique de récupérer via le tactile de l'écran le nombre de joueurs. On y utilise des JFrame, des JLabels et des JButtons afin d'afficher les informations pertinentes comme du texte ou des images pour le nombre de joueurs. Le résultat sera détecté par le timer, qui appellera à son tour la fonction *showRules* pour afficher les règles à l'écran. Enfin, la fonction *getTags* récupèrera par le biais du timer les Tags associés aux joueurs.
Nous détaillerons l'utilisation de la librairie TUIO et en particulier le contenu de la classe TestTuio2 qui nous permet d'actualiser et de récupérer toutes les données des différents Tags.

5.6 La classe Main

Cette dernière classe Main est appelée au lancement du jeu. Nous sommes obligés d'y inclure la fonction *start* de JavaFx, qui s'applique à un Stage, qui nous permet de lancer le jeu et de disposer des nombreuses fonctions fournies par cette librairie. Encore une fois, on utilise un timer dans cette classe. Ceci nous permet de vérifier si toutes les fonctions appelées dans Tuio sont terminées, c'est-à-dire si l'on connaît le nombre de joueurs, les Tags, et si les joueurs ont lu les règles.
On attendra donc que toutes les variables soient connues, avant d'appeler le constructeur Board, et de passer le jeu en plein écran pour démarrer la partie.

Enfin, une dernière classe nommée Executable fait simplement appel à la classe Main et nous permet de générer un artéfact, soit dans notre cas un .jar.

5.7 Interaction avec la librairie TUIO

Nous allons détailler dans cette partie comment nous avons relié le code du jeu à l'utilisation des Tags d'Interactive Scape, en utilisant le protocole TUIO. Plusieurs types d'objets sont définissables. Nous n'utilisons que les TuioObject, c'est ainsi qu'ils sont appelés en Java. Chaque objet possède de nombreuses variables associées. Nous n'avons utilisé que les suivantes :

- Le symbol ID, qui est un identifiant propre à chaque Tag, entre 1 et 24 ;
- Le session ID, qui est un identifiant secondaire, qui dépend de l'ordre dans lequel les Tags ont été placés ;
- Le hashcode, qui est un dernier identifiant, plutôt un long entier, associé à chaque Tag, et qui change dès que le Tag n'est plus en contact avec la table puis l'est à nouveau.

Pour récupérer donc un Tag par joueur en plus du dé, on modifie la fonction *addTuioObject* afin de vérifier que le Tag n'a pas déjà été placé et que nous en avons le bon nombre, et on stocke les objets dans une liste, dont nous auront besoin très régulièrement.
Pour la partie concernant le dé, le code a été fait de manière à détecter le mouvement puis l'arrêt de ce dernier. On vérifie pour cela que le dé a été ôté de la table, c'est même obligatoire avec le lanceur de dé que nous avons développé dans le cadre du projet, grâce au hashcode (il doit être différent du hashcode précédemment gardé en mémoire). Si c'est le cas, on vérifie que le dé est en mouvement (la table étant sensible, un Tag est automatiquement en mouvement lorsqu'il est posé sur la table), avec la fonction fournie par la librairie TUIO *getMotionAccel*. Par la suite, on attend que cette accélération soit nulle et on récupère alors l'angle, grâce à la fonction *getAngle*. On transforme enfin cet angle en entier entre 1 et 6 que l'on retourne.
On utilise également la position des Tags avec les fonctions *getX* et *getY*, pour notamment déplacer les petits chevaux. Pour cela, on utilise encore une fois les dimensions de l'écran car les positions des objets sont données entre 0 et 1. Par un calcul de distance minimum, on parvient dans la fonction *coordToPos* (dans la classe Horse) à renvoyer la position correspondant à l'emplacement du Tag.

6 Hardware

L'intérêt particulier du PlayBoard réside dans l'important sentiment d'interaction des joueurs avec le support et entre eux, et c'est dans cette optique que nous avons développé les interfaces H.-M. des différents jeux implémentés.
Dans le même sens, souhaitant conférer un maximum d'immersion aux utilisateurs, nous avons réalisé des mobiles dans lesquels *clipser* temporairement les tags TUIO, en fonction du jeu lancé. Des formats "carte à jouer", des totems type "jungle speed" et autres pions on pu être designé.
Tout ces objets ont été modélisés sur différents logiciels de *CAO*, dont SolidEdge (dimensions précises de forme géométriques classiques), Blender, et Sculptris (Parties moins précises, pièces plus "sculptées"). Ces modèles exportés en ".stl" ont ensuite été imprimés en 3D en plastique PLA*. Ce plastique se prête bien à l'impression, bien qu'il ne soit pas idéal. D'une part, parce qu'à l'impression, les couches ont des chances de se séparer, et d'autres part, parce que c'est un plastique tout simplement. Il est certe biodégradable, mais à notre époque, réaliser en série des grosses pièces comme celle-ci en plastique est assez inconcevable, et en dehors du cadre du prototypage, nous utiliserions plutôt du bois usiné. (Et si ce n'était pas viable, nous nous tournerions plutôt vers des mousses de plastiques biodégradables à faible densité, en moules à injection.)

*: Acide Polylactique

7 Conclusion

Le projet fini remplit les attentes énoncées par l'entreprise qui nous a prêté la table interactive, et les membres ont tous pu apprendre quelque chose de nouveau tout au long de ce projet, nous considérons donc celui-ci comme une réussite.
Nous tenons à remercier l'entreprise Interactive Scape de nous avoir fait confiance et de nous avoir permis de prendre part à ce projet, ainsi que l'équipe pédagogique qui s'est montrée réactive lorsque nous avions besoin d'émettre une convention de prêt avec cette entreprise.