GAMETRACKER : des traces d'utilisation pour la conception de jeux vidéo

Rémi Casado, Jonathan Cohen

Janvier 2015

Table des matières

Résumé

La collecte et le traitement de traces permet d'acquérir de la connaissance sur les expériences d'utilisateurs. Nous nous plaçons ici dans le contexte de développement de jeux vidéo. L'application GAMETRACKER

permet d'avoir un retour utilisateur pendant la période de développement. Les sessions de jeu sont tracées puis transformées pour extraire de la connaissance sur le comportement en jeu des testeurs.

1 Introduction

1.1 Contexte

Tout au long du cycle de développement d'un jeu vidéo, les phases de tests jouent un rôle primordial pour garantir non seulement l'absence de bugs, mais également une difficulté adéquate, une jouabilité adaptée et, d'une manière plus générale, un plaisir de jeu constant. Le *feedback* des testeurs aux developpeurs est donc une étape importante. On propose ici une application qui permet d'automatiser, au moins en partie, le travail de collecte et d'analyse des tests de jeux vidéo.

On se place donc dans le contexte de la collecte, de la transformation et de l'exploitation de traces d'utilisation. Ces traces d'utilisation vont permettre d'observer des types de comportement de joueurs sur le jeu en phase de développement. L'idée est de faire tester le jeu dans ses phases *alpha* et *bêta* afin d'observer le comportement des joueurs, pour déterminer les points forts du jeu, et les points faibles, *i.e.* les éléments à peaufiner. Chaque action élémentaire du joueur est collectée, et des comportements de haut niveau en sont déduits. Cela permet de répondre à des questions du type : *un joueur passe-t-il trop de temps dans une zone peu importante ou à la difficulté modérée ?*, ou encore *Faut-il revoir certains éléments de l'interface ?*, etc.

1.2 Problématique

La problématique que l'on se pose est de retrouver des comportements généraux de joueurs, à partir d'observations de bas niveau. Pour résoudre ce problème, nous avons procédé à la réalisation d'un projet, GAMETRACKER, prenant la forme d'une application Web liée à un enregistreur de touches clavier (*keylogger*) exécuté sur la machine de l'utilisateur (*i.e.* le testeur du jeu). Notre solution se base uniquement sur l'analyse et le traitement des touches du clavier de l'utilisateur.

Le reste de ce rapport est divisé comme suit : la section 2 présente la modélisation choisie pour collecter les traces d'utilisation. La section 3 décrit les différentes transformations de traces. La section 4 présente l'architecture utilisée et les choix d'implémentation fait pour la réalisation du projet. Enfin, on discute de tout ça dans la section 5, et on conclut dans la section 6.

2 Modélisation

2.1 Modèle de trace

Les traces d'utilisation lors d'un test d'un jeu sont définies par un ensemble d'éléments d'observation, ou *obsels*, qui décrivent les différentes actions réalisées par le testeur. Une trace d'utilisation est modélisée via des obsels de types différents ayant des attributs différents. Tous les obsels sont composés d'un temps de début et de fin, relatifs au temps de départ de l'expérience [1]. Ils possèdent également un identifiant unique. Les attributs de chaque obsel dépendent de son type, ou niveau. Nous distinguons trois niveaux d'obsels, qui correspondent à trois niveau d'abstraction des actions du testeur.

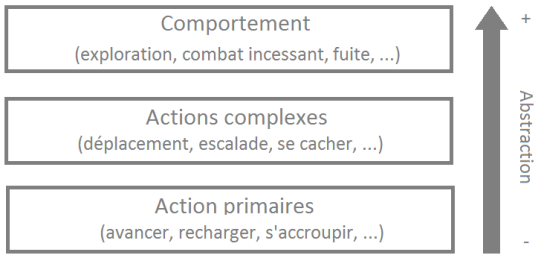

FIGURE 1 – Les trois couches d'abstraction correspondent aux trois niveaux d'obsels.

2.2 Niveaux d'obsels

2.2.1 Actions primitives

La collecte première d'obsels concerne les actions primitives, ou élémentaires, effectuées par l'utilisateur – son action de plus bas niveau lorsqu'il joue à un jeu. Par exemple, s'il s'agit du test d'un jeu de type *tir à la première personne*, les obsels de niveau élémentaire peuvent correspondre à `avancer`, `reculer`, `sauter`, `tirer`, etc. Ce type d'obsels possède deux attributs principaux : un attribut d'action, (comme énoncé précédemment) et un attribut de la touche utilisée par le testeur pour accomplir l'action voulue. Un exemple d'obsel de type élémentaire peut être :

```
{
    id    : "o-5o",
    type  : "PrimitiveAction",
```

1. Il est inutile de conserver le temps en absolu, car les traces ne concernent qu'une seule session d'alpha/bêta test

```
    key   : "Z",
    value : "move_forward",
    begin : 24,
    end   : 25
}
```

La champ `value` correspond à l'action du testeur. On note que cette action peut éventuellement être inconnue. Ainsi, si l'utilisateur se trompe de touche, on aura un type d'action inconnu, mais la touche actionnée sera sauvegardée dans le champ `key`. L'intérêt est de voir si l'utilisateur n'appuie pas sur une touche par réflexe. Si une touche non attribuée est actionnée de manière répétée, l'équipe de développeurs pourra demander au testeur pourquoi il appuyait sur cette touche, et en tenir compte dans le développement.

2.2.2 Actions complexes

Un deuxième type d'obsels est calculé à partir d'une première transformation de trace. On revient sur les transformations de traces dans la section 3. Ce type d'obsel correspond à des actions de plus haut niveau, des motifs – sorte d'agrégation locale – des actions primitives du testeur ; par exemple : `combat`, `déplacement`, `ravitaillement`, etc. Un exemple d'obsel de type complexe peut être :

```
{
    id        : "o-36",
    type      : "ComplexAction",
    label     : "Moving",
    value     : "move",
    begin     : 610,
    end       : 613,
    ambiguous : false
}
```

Les obsels d'actions complexes ne possèdent pas l'attribut correspondant à la touche clavier, mais possèdent un attribut booléen supplémentaire, `ambiguous`, qui va permettre de déterminer si cette action est sûre ou s'il faut la faire confirmer par un intervenant humain. Une action sûre est un motif d'actions primitives dont on a aucun doute sur le label attribué. À l'inverse, un motif dont on ne sait pas s'il à été correctement identifié nécessitera une confirmation d'un intervenant humain. Par exemple, pour le jeu utilisé lors des expérimentations [2], la touche *E* correspond à plusieurs actions possibles, dont `monter dans un véhicule`, `utiliser un objet`, etc. On ne peut pas déterminer de manière certaine si l'utilisateur se déplace à pied ou à véhicule, il faudra donc le faire confirmer *a posteriori*.

2. Farcry 4. `http://far-cry.ubi.com/fr-fr/home/index.aspx`

2.2.3 Comportement

Le troisième type d'obsel correspond à l'objectif du projet. Il s'agit d'obsels correspondants à un comportement local de l'utilisateur. De la même façon que les actions complexes sont des motifs d'actions primitives, le comportement local correspond à un motif d'actions complexes. Nous définirons cet obsel par l'exemple. Si un utilisateur se déplace pendant une durée assez longue, puis prend le temps d'observer son environnement, on pourra considérer qu'il *explore* le jeu. Si, dans la suite de la trace on constate qu'il se dirigeait en fait vers un objectif du jeu, on pourra se dire qu'il s'était perdu dans le jeu. Évidemment, ce type d'obsel de très haut niveau nécessitera une confirmation d'un humain.

3 Transformations

3.1 Types de transformations

Les transformations de traces correspondent aux modifications des traces d'utilisations primaires issues de l'activité de l'utilisateur. L'objectif des transformations est de détecter des motifs de bas niveau (répétitions de touches, typiquement) afin de décrire des comportements plus facilement interprétables par des humains.

Les deux principales transformations sont des réécritures de motifs. Il s'agit de chercher un motif de plus haut niveau dans les actions élémentaires collectées. Par exemple, imaginons que l'on observe une suite continue d'obsels correspondant à un déplacement, *i.e.* `avancer` ou `tourner`. On va donc créer un obsel commençant au début de ce motif, finissant à la fin de ce motif, et ayant pour type d'action `Déplacement`.

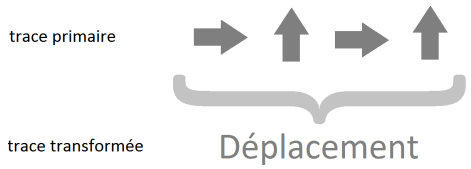

FIGURE 2 – Une réécriture de motifs

3.2 Méthode de transformation

Pendant une session de jeu, les traces sont dynamiquement enregistrées. Après cette session de jeu, une transformation de la trace récoltée est proposée à l'utilisateur. La méthode de transformation est assez simple : il s'agit de parcourir toute la trace et de créer un nouvel obsel de plus haut niveau chaque fois qu'un motif intéressant est détecté. Ces motifs intéressants peuvent être définis de deux manières, avec pour trait commun de posséder une collection d'obsels. Une première approche consiste à séparer les motifs en suite d'obsels

consécutifs, comme expliqué précédemment. Une autre approche consiste à séparer les motifs en fonction du temps passé à accomplir les actions relatives. Un exemple de motif pour passer d'une trace de niveau primaire à une trace d'actions complexes peut être :

```
{
    value       : "move",
    label       : "Move",
    pattern     : {
        actions : [
            "move_forward",
            "move_backward",
            "strafe_left",
            "strafe_right"
        ],
        times   : 3
    }
}
```

Dans cet exemple, on détectera une phase de déplacement dès qu'au moins une des `actions` aura été répétée au moins 3 fois de suite. Un exemple de motifs pour passer d'une trace d'actions complexes à une trace de comportements peut être :

```
{
    value       : "explore",
    label       : "Exploration",
    pattern     : {
        actions : [
            "move",
            "climb",
            "take_a_break"
        ],
        times   : 3
    }
}
```

4 Implémentation

4.1 Architecture de l'application

Le projet est une application Web liée à un *keylogger* exécuté sur la machine du testeur.

Le *keylogger*, écrit en Python, récupère les touches entrées par l'utilisateur et les envoie à un serveur NodeJS [3]. Celui-ci sert de passerelle entre l'interface

3. `http://nodejs.org/`

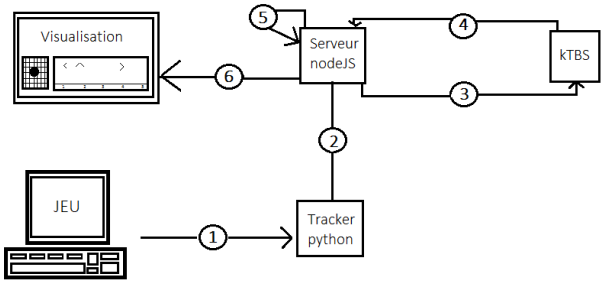

FIGURE 3 – Architecture logicielle GAMETRACKER
1 : récupération des touches clavier
2 : envoi des touches au serveur NodeJS
3 : envoi des obsels et des traces au kTBS
4 : récupérations des obels et des traces déjà enregistrés
5 : transformations locales des traces
6 : affichage de la trace

client, le *keylogger* et le kTBS [4], la plate-forme qui permet de stocker les traces de l'utilisateur. Par soucis de simplification et d'optimisation d'envoi des requêtes HTTP, le client effectue lui-même certaines requêtes vers le kTBS.

4.2 Méthode de collecte

Les *obsels* de bas niveau sont collectés via le *keylogger*. Chaque touche du clavier correspond, ou non, à une ou plusieurs actions du jeu. Cette correspondance entre une touche et une action est définie au préalable par le testeur, via l'interface Web à sa disposition. Le *keylogger* collecte les touches et les envoie par paquet de 20 frappes clavier. La collecte d'une touche commence quand l'utilisateur appuie sur la touche, et se termine quand l'utilisateur relâche la touche. La correspondance entre une touche et son identification en tant qu'action de jeu est faite sur le serveur NodeJS, qui crée et envoie ensuite un paquet de 20 *obsels* correspondants au kTBS.

4.3 Visualisation

Nous avons choisi de représenter les traces *via* deux visualisations différentes.

La première correspond à une visualisation classique : les obsels de la trace sont affichés sur une frise chronologique. Cette représentation est obtenue avec la librairie *Samotraces.js* [5]. Les motifs importants, ou les *obsels* ambigus, sont mis en valeur pour faire gagner du temps à la personne qui visualise ces traces.

4. kernel for Trace-Based Systems, `http://liris.cnrs.fr/sbt-dev/tbs/doku.php?id=tools:ktbs`
5. `http://dsi-liris-silex.univ-lyon1.fr/bmathern/samotraces/doc/`

L'autre visualisation représentation est moins traditionnelle. Nous avons décidé de simuler un personnage qui effectuerait les actions correspondantes aux obsels de niveau élémentaire. Cette visualisation est réalisée grâce à la libraire *Pixi.js*[6]. Avec cette méthode, lorsque l'utilisateur veut visualiser la trace, le personnage mime ce que le testeur aura fait durant sa session de jeu. L'idée ici est de dé-bruiter l'information amené par le trop grand nombre d'obsels de niveau élémentaire présents sur la frise chronologique. En effet, sur la première visualisation, les obsels sont peu lisibles de par leur nombre et leur rapprochement. Avec la simulation en direct, la personne qui visualise la trace peut se faire une idée rapide des actions effectuées par le testeur.

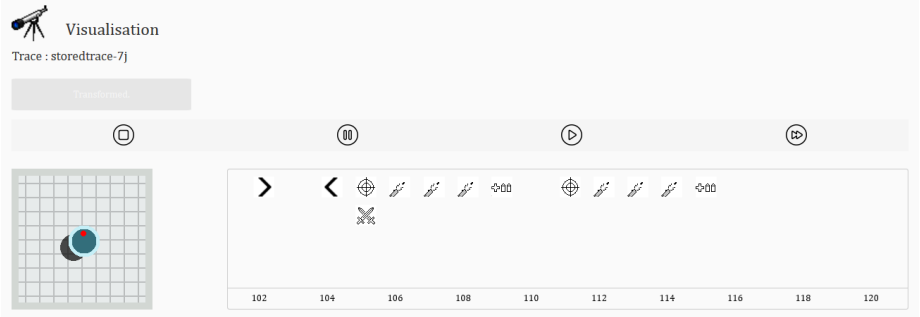

FIGURE 4 – Visualisation d'une trace. À droite, la visualisation avec *Samotraces.js*. À gauche, la visualisation avec *Pixi.js*

5 Discussion

On discute dans cette partie de certains problèmes rencontrés en cours de projet, ainsi que de certains choix de programmation et de modélisation. On présente également quelques pistes pour d'éventuelles futures améliorations.

5.1 Problèmes rencontrés

Notre méthode permet de visualiser des traces de haut niveau à partir du traitement d'actions élémentaires, les frappes clavier. Cependant, le fait de se servir d'actions très bas niveau restreint dès le départ l'information que l'on peut obtenir du jeu. Essayer d'abstraire les actions du joueur par réécriture successives de motifs fait que l'on perd le "grain fin" de certaines actions de jeu qui sont peut être importantes pour expliquer un comportement.

De plus, nous avons du faire face à une autre problématique : plusieurs actions de jeu peuvent s'activer par une même touche clavier. Cette problématique peut se résoudre de deux manières. On peut essayer d'inférer ce que veut dire la touche en question. Par exemple, si le joueur peut monter dans un véhicule, et

6. `http://www.pixijs.com/`

ouvrir une porte avec la touche E, il suffit d'analyser quelles touches il utilise ensuite (plutôt celles du déplacement en véhicule ou plutôt celles du déplacement à pied). Cette méthode est toutefois trop sensible à l'attribution des touches en jeu. L'autre solution serait que l'application soit reliée directement au jeu afin on d'étudier, non plus des frappes clavier, mais bien des évènements de jeu. Il faut cependant noter que de nombreux jeux se déroulent dans des environnements continus, et il faudrait donc discrétiser les événements élémentaires.

Chaque transformation de trace (passage d'un plus bas niveau vers un plus haut niveau d'abstraction) entraîne sur le kTBS la création d'une nouvelle trace. Ainsi, pour chaque session de jeu, trois traces sont utilisées en parallèle : la trace de base, la trace d'actions complexes et la trace de comportement. Ce choix de programmation relève uniquement de considérations techniques, permettant de faciliter la manipulation et la visualisation des traces.

On notera pour finir qu'envoyer toutes les touches clavier au kTBS n'est probablement pas la meilleur solution. Pour certaines sessions de jeu particulièrement longues, il faudrait pouvoir pré-traiter certaines données côté client. Les algorithmes de transformations de traces souffrent aussi de chutes de performances lorsque le nombre d'obsels dans la trace augmente.

5.2 Améliorations possibles

Quelques fonctionnalités présentées dans ce rapport et devant initialement faire partie du projet n'ont finalement pas pu être implémentées.

Initialement, pour pallier à la perte d'information due à l'analyse de traces bas niveau, il devait être possible pour la personne qui visualise les trace de pouvoir modifier les obsels considérés comme ambigus (comme la touche E). C'était l'utilité principale du champ `ambiguous` dans le modèle des traces. Par exemple, si, lors de la phase de transformation, le système n'a pas su distinguer si le testeur montait dans un véhicule ou ouvrait un coffre, il note cet obsel comme ambigu. L'utilisateur est ensuite libre de le modifier afin de garder le contrôle sur son activité. En effet, les obsels de haut niveau, bien que calculés, ne peuvent pas être aussi précis qu'une observation d'un être humain. En associant le *feedback* de l'utilisateur sur les actions ambiguës avec un apprentissage local du comportement du joueur, il peut ainsi être théoriquement possible d'inférer, avec une forte vraisemblance, l'action réelle effectuée par le joueur.

Sur un autre aspect, nous nous sommes rendus compte que la modélisation de traces n'était pas indispensable dans ce cadre d'application. En effet, les obsels étant limités à des frappes clavier ou une liste d'actions de plus haut niveau, nous n'avions pas nécessairement besoin de modélisation très poussée. Là où la modélisation aurait pu apporter de réels arguments, c'est si nous avions pu connecter le jeu à notre système. De cette manière nous aurions pu avoir des informations un peu plus complexes, et la modélisation aurait servie pour faire du raisonnement à partir de ces traces. De plus, l'utilisation du kTBS s'est principalement restreinte à une utilisation en tant que base de données de traces, ce qui n'est pas un mal en soi. Il aurait toutefois pu être intéressant d'utiliser les capacités du kTBS pour la transformations des traces. Peut-être était-il possible

de faire de la réécriture de motifs par le biais de requêtes SPARQL. Cela aurait permis d'alléger les traitements côté client et côté serveur NodeJS.

Également, quelques difficultés techniques liées au kTBS ont posé quelques contraintes de programmation, comme un soucis régulier de *cross-domain autorisation*, d'envoi de paquets, etc. Pour ce genre d'applications, où le nombre d'informations est très important dans un laps de temps assez court, il pourrait être intéressant que la base de traces se trouve sur le même support que le collecteur, pour éviter les temps de transfert.

6 Conclusion

L'utilisation des traces pour déterminer le comportement de joueurs lors du test d'un jeu vidéo en développement afin découvrir les aspects à améliorer s'avère être une méthode efficace. Même à notre échelle de développement de traces, nous n'avons pas été noyé sous un flot d'informations, comme cela aurait pu se produire. Cependant, une collecte de traces digne de ce nom ne peut pas se faire sans un lien direct entre le jeu et le système de traces. En effet, tout les événements en jeu qui ne passent pas par le clavier n'ont pas pu être pris en compte, et cela représente une limite non négligeable à GAMETRACKER.

A Captures d'écran

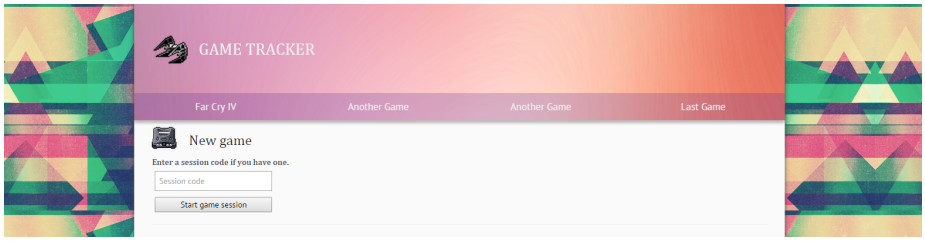

FIGURE 5 – Capture d'écran de l'interface de démarrage de l'application.

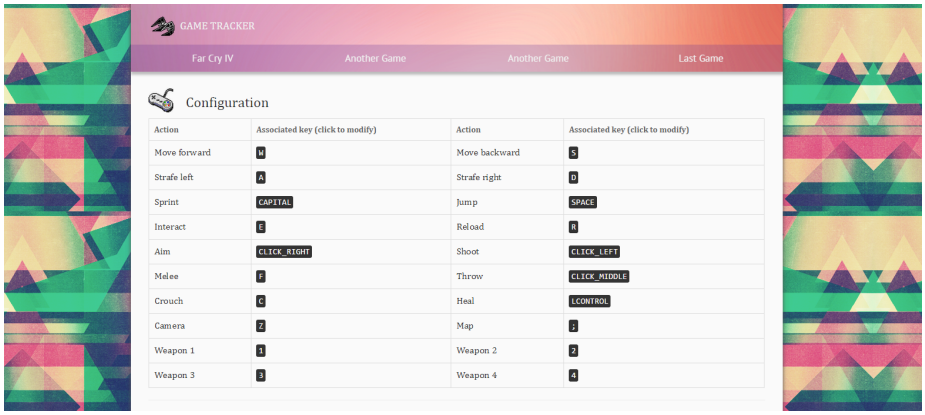

FIGURE 6 – Capture d'écran de l'interface de configuration des touches.

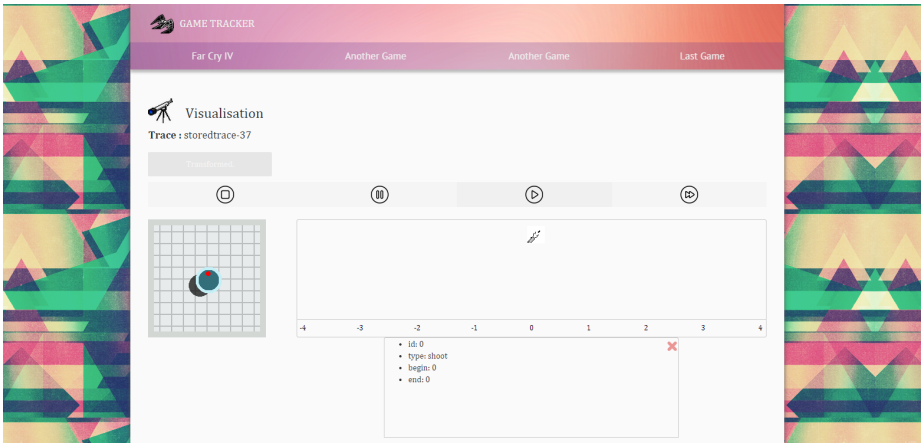

FIGURE 7 – Capture d'écran de l'interface de visualisation des traces.