Plateforme de simulation pour étudier la temporalité des interactions

Philippe Schmid

UNIVERSITÉ DE LILLE

MÉMOIRE DE STAGE

Plateforme de simulation pour étudier la temporalité des interactions

Auteur :
Philippe SCHMID

Tuteur :
Mathieu NANCEL

Stage de Master 2 en Informatique

chez

Inria, équipe-projet Loki
CRIStAL

Université de Lille

30 août 2019

Inria

Remerciements

Je remercie mon tuteur de recherche, Mathieu Nancel, ainsi que Sylvain Malacria pour leur disponibilité et leur aide tout au long de mon stage. Je souhaite également remercier Stéphane Huot et toute l'équipe Loki pour leur accueil, leur énergie et la bonne ambiance qui a rythmé ces 6 mois. Merci également à mon tuteur universitaire Jean Martinet pour son suivi.

Table des matières

Table des figures

Chapitre 1

Introduction

J'ai fait mon stage de fin de Master en Informatique au sein de l'équipe Loki chez Inria. Cette équipe-projet étudie l'IHM, l'interaction Homme-Machine.

Le bon échange d'informations entre l'utilisateur et la machine est crucial à l'accomplissement de la tâche entreprise par celui-ci. Les principaux moyens d'envoyer de l'information à l'ordinateur sont en utilisant des périphériques tel que la souris, le clavier ou la surface tactile. L'ordinateur nous renvoi de l'information à travers l'écran ou les hautparleurs, retranscrivant l'état actuel de celui-ci. Le cycle 'action de l'utilisateur' - 'réaction de l'ordinateur' - 'mise à jour des périphériques de sortie' - 'prise en compte des mises à jour par l'utilisateur' est ce qui compose l'interaction. La recherche en IHM se concentre sur l'amélioration de cette interaction.

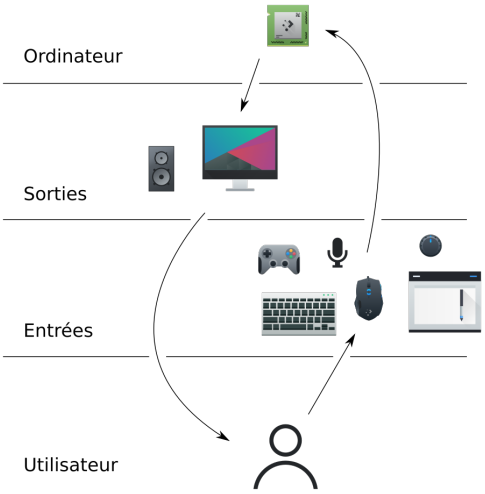

FIGURE 1.1 – Cycle d'interaction entre utilisateur et ordinateur

L'ordinateur réagit aux évènement de l'utilisateur, et l'utilisateur réagit aux mises à jour de l'ordinateur. Chaque étape introduit un moment de latence pour comprendre et interpréter ce qu'elle a reçu, et pour modifier ses plans en fonction de ces changements. Cependant, l'ordinateur ignore le délai introduit par l'utilisateur. Ainsi, une entrée jugée valide par l'état actuel de l'ordinateur sera interprétée dans le contexte de cet état, même si en réalité cet état n'est actif que depuis moins de temps que le délai dont a besoin l'utilisateur pour réagir. Et lorsque c'est l'ordinateur qui génère un évènement (basé sur l'heure par exemple) et qui fait une mise à jour sans action provenant de l'utilisateur, ce qui change l'état de l'ordinateur, l'utilisateur ne peut soupçonner qu'il va y avoir un changement, et risque de ne pas être compris correctement par l'ordinateur.

1.1 Mises à jour gênantes de l'interface

1.1.1 Divergences entre intention et résultat

Dans certains cas, ce changement d'état engendré par l'ordinateur vient interférer avec une action que l'utilisateur est en train de faire. C'est ce que nous avons appelé une interférence d'interaction.

Quelques exemples de mises à jour d'information ne provenant pas d'un évènement utilisateur sont la liste des réseaux wifi qui reflète la détection d'un nouveau réseau ou la perte d'un autre, la boîte mail qui, d'une manière similaire, ajoute les mails entrant, organise ceux qui sont dans la boîte, et supprime ceux qui en sortent. Mais également les apparitions et disparitions de notifications, selon ce qui se passe et leur temps d'affichage.

Si l'utilisateur décide d'agir au moment où l'ordinateur est en train d'effectuer une mise à jour, l'utilisateur pensera avoir fait quelque-chose, mais l'ordinateur comprendra autre chose. Tout cela parce que l'ordinateur ne prend pas en compte le délai introduit par l'utilisateur, et le suppose nulle.

Voici un exemple : L'utilisateur a décidé de se connecter à un réseau wifi. Il ouvre la liste de réseaux disponibles et les lit les différents noms disponibles. Il trouve celui qui l'intéresse. Il déplace le curseur jusqu'au réseau de son choix et clique. Une seconde après, l'ordinateur est en train de se connecter avec un réseau différent de celui qu'il avait choisit. Il n'y a cependant eu aucun problème technique ni d'erreur de la part de l'utilisateur.

Un instant avant que l'utilisateur ne clique, l'ordinateur a détecté un nouveau réseau (changement d'état) et l'a ajouté à la liste (mise à jour graphique), déplaçant la cible visée par l'utilisateur vers le bas.

Le résultat de l'action de l'utilisateur est différente de son intention, et l'utilisateur se retrouve à sélectionner un autre réseau wifi. Et si l'on pense à d'autres cas de figures, tel des fenêtres de dialogue qui apparaissent au dernier moment devant un bouton et qui prennent le clique de l'utilisateur comme une validation ou un refus, l'utilisateur n'aura non-seulement pas fait ce qu'il avait prévu, mais ne saura même pas ce qu'il a fait, ni comment annuler cette action.

1.2 Mise en place d'une plateforme d'étude des interactions graphiques

Ces problèmes de synchronisation, pouvant gêner l'utilisateur, sont à la raison de la création du projet de recherche auquel j'ai participé.

Ce projet a pour but d'explorer ce qui se passe au moment où une divergence se crée, d'étudier la potentielle réaction de l'utilisateur, pour pouvoir mettre en place des solutions améliorant l'expérience de l'utilisateur.

1.2.1 Un environnement d'interaction contrôlé

Pour pouvoir étudier ce phénomène d'interférence d'interaction, nous avons décidé de mettre en place une plateforme de création d'interfaces graphiques complètement contrôlables. Elle permet de créer une interface qui enregistrera tout évènement lié à celle-ci et à l'interaction de l'utilisateur avec elle. Et souhaitant pouvoir utiliser cet environnement dans différents contextes et à différents buts, nous avons opté en termes de technologie sous-jacente pour une implémentation en JavaScript, tournant dans le navigateur. Cela permet une bonne portabilité et une mise en place

simple. Cependant, une latence minimale étant cruciale pour obtenir des résultats reflétant ce qui s'est passé de manière juste, nous avons décidé d'utiliser un contexte WebGL pour plus de rapidité et de contrôle Et pour contrôler WebGL, nous avons décidé d'utiliser une librairie légère et la plus performante possible, permettant de nous concentrer sur la plateforme plus que sur l'implémentation d'une librairie d'interface graphique. Cela à condition qu'il y ait une librairie satisfaisant nos critères.

Le choix de la librairie à travers la mise en place d'un petit framework de benchmarking a fait l'objet de la première partie de mon projet.

1.2.2 Test de réaction lors du pointage

Une fois une librairie choisie, une première expérience a été mise en place, lors de la première itération de la plateforme. Cette expérience se base sur le principe de la tâche de Stop signal, paradigme utilisé dans les expériences en neuro-sciences pour étudier le temps de réaction d'une personne pour arrêter son action en cours(VERBRUGGEN, 2019). Le principe de l'expérience est de demander à l'utilisateur d'appuyer par exemple sur une touche dès qu'un signal apparait à l'écran. Si un second signal apparait (signal d'arrêt, différent du premier), il doit se retenir d'appuyer.

Nous nous sommes intéressé à l'existant dans ce domaine ainsi qu'aux consensus sur la création d'un tel teste. Nous nous sommes ensuite intéressé à l'ajout d'une tâche de pointage au test, car c'est un des contextes dans lequel survient le type de problème que nous étudions. Après avoir lu ce qui a été fait avec du pointage, nous nous sommes penché sur le protocole et nos attentes par rapport aux données résultantes de l'expérience. J'ai ensuite procédé à la mise en place des parties de la librairie nécessaires à l'expérience et implémenté l'expérience. Les résultats de cette étude seront soumis à la conférence internationale ACM CHI 2020.

Chapitre 2

Étude de librairies WebGL pour la plateforme et d'autres projets

Il existe plusieurs technologies pour implémenter la plateforme. Dans le but d'identifier la technologie la plus réactive et qui nous permet d'avoir le plus de contrôle sur la temporalité des évènements d'affichage et sur la mise à jour des données, nous avons procédé à des tests de performance. Les choix ont été guidés par les critères de rapidité et de flexibilité.

2.1 Décisions par rapport à nos critères

Pour valider notre choix de technologie web comme base de notre plateforme d'expériences, nous avons d'abord procédé à un test de performance, comparant une petite application développée avec Metal (l'API de rendu 3D sous Mac OS X, OpenGL n'est plus supporté depuis Mac OS X Mojave) et la même, développée avec WebGL.

2.1.1 Portabilité et délai minimum

Contexte

Une application développée en WebGL est portable, ce qui permet de faire des tests dans des conditions les plus réalistes possibles. Cependant, cette portabilité vient à un prix, celui d'une performance moindre par rapport à un programme natif.

Il a donc fallu faire un compromis. Pour cela, nous avons donc d'abord comparé une application native avec une application web identique. Le protocole de test est une variante du premier test (Click) utilisé pour comparer les différentes librairies WebGL, que nous verrons dans la prochaine partie.

Pour étudier la latence de pointage entre le mouvement de la souris et la mise à jour de l'écran, l'approche classique est de positionner une caméra haute fréquence devant l'ordinateur, filmant l'écran et la souris. La caméra enregistre les actions faites avec la souris et l'état de l'écran en même temps. Une fois l'enregistrement fini, on visionne la vidéo frame par frame et on identifie le début d'un évènement de la souris, et le moment où l'écran est mis à jour. On obtient ainsi le temps de latence.

Casiez et al. ont proposé une autre méthode, plus pratique et cyclique, où ce qui est affiché à l'écran contrôle directement les mouvements de la souris, et retransmet ces mouvements. Pour cela, il suffit de poser la souris sur l'écran, et d'afficher à l'écran un motif facile à identifier pour la souris. En suite, on procède en commençant cette fois-ci par la mise à jour de l'écran. C'est la mise à jour de l'écran qui, détectée par la caméra de la souris, va faire croire à celle-ci qu'elle a bougé. La

souris va donc envoyer un évènement de déplacement de souris, reçu par l'ordinateur. L'ordinateur a déclenché le début du cycle et reçoit l'évènement de la part de la souris à la fin de celui-ci. Il peut donc lui-même calculer immédiatement (pas de traitement de données comme le visionage de vidéo) le temps de latence entre évènement d'un clique souris et affichage à l'écran. De plus, cette méthode peut être répétée pour obtenir plus de valeurs de latence, permettant de voir sa constance par exemple. Aussi, cela permet d'éliminer tout paramètre extérieur pouvant jouer dans le calcul de latence (CASIEZ et al., 2015).

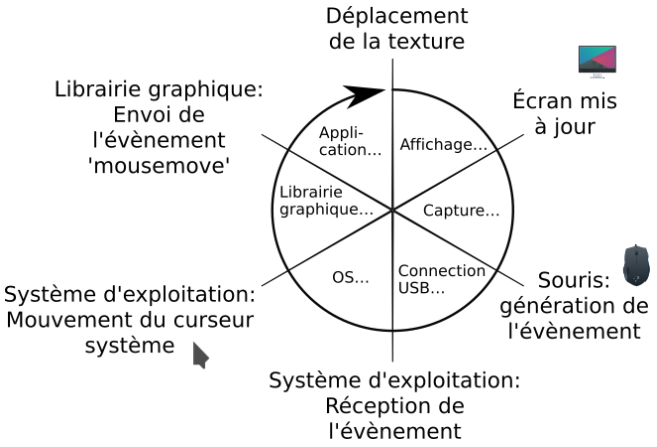

FIGURE 2.1 – Cycle des évènements, du déplacement de la texture à la réception du signal correspondant

Dans notre cas, nous nous intéressons exclusivement au temps de réaction de l'application à un évènement souris.

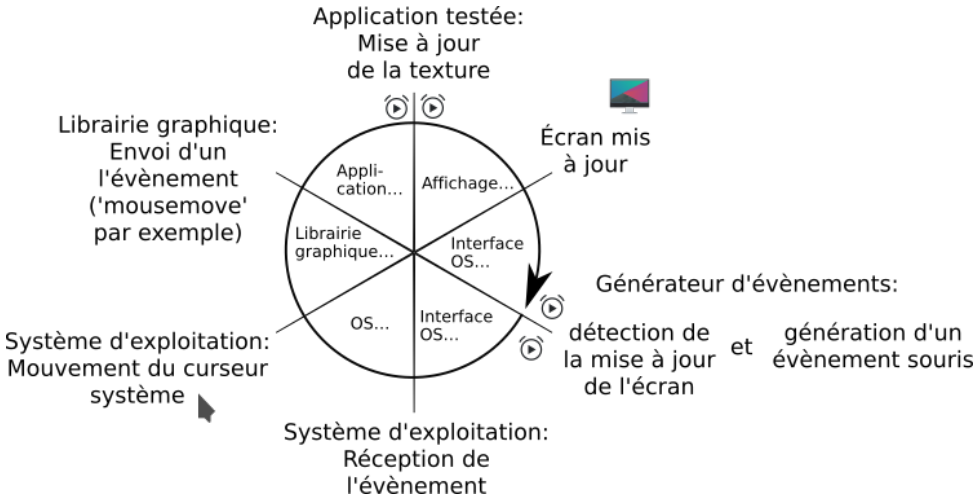

FIGURE 2.2 – Cycle des évènements, du déplacement de la texture à la réception du signal correspondant

C'est pourquoi nous avons remplacé la partie impliquant une souris par une application qui simule le comportement d'une souris et peut lire l'écran2.2. On aura ainsi la possibilité de contrôler précisément quand et quel évènement sera envoyé, et quel mise à jour est attendue à l'écran. On désignera cette application comme le générateur d'évènements.

Un paramètre limitant la vitesse de réaction d'une application est la fréquence de balayage de l'écran, ralentissant les échanges entre applications. Nous nous sommes

basé sur une fréquence fixée à 60, étant celle des moniteurs que nous utilisons pour ces tests, et la fréquence la plus répandue en ce moment.

L'expérience que nous avons utilisé pour comparer la réactivité d'une application native et une application WebGL suit le schéma de cycle, où l'on a deux parties distinctes échangeant des informations à travers des canaux d'entrée et de sortie. Ces informations sont interceptées par le générateur d'évènements, ce qui supprime le besoin de contrôler ces entrées et sorties depuis l'extérieur de l'ordinateur.

Le principe est d'alterner entre deux états, permettant de faire autant de cycles que nécessaire. Le premier état est déclenché par l'appui sur le bouton gauche de la souris. À ce moment, lorsque l'application reçoit l'évènement de pression du bouton, elle doit colorer une surface en vert. Le générateur d'évènement, ayant généré l'évènement de pression et détectant le moment où la surface devient vert, peut calculer le temps mit par l'application pour réagir. Le second état correspond au relâchement du bouton, où l'application doit peindre la surface en bleu.

Nous avons décidé d'enregistrer également les moments où l'application testée reçoit et a terminé sa procédure de mise à jour. Pour pouvoir comparer les temps obtenus par le générateur d'évènements et par l'application testée, il faut avoir une unité de temps commune. Nous avons donc choisi d'utiliser le nombre de millisecondes depuis l'époque UN*X.

Avec ce simple test de réaction, on peut obtenir une idée de la différence de latence introduite par la technologie sous-jacente.

Comparaison

En étudiant les résultats, il est clair que la vitesse de balayage de l'écran est inférieure aux temps de réaction des applications. On obtient ainsi en fin de compte des résultats très proches. Si l'on étudie les valeurs en excluant le passage par l'affichage, on remarque que la version native est plus rapide. Cependant, à ces vitesses, une grande quantité de facteurs commencent à avoir une importance non-négligeable (tel par exemple les choix de l'ordonnanceur du noyau). Ces facteurs n'étant pas contrôlables, il n'est pas très intéressant, ni utile, d'essayer d'obtenir des valeurs plus précises, celles-ci étant par conséquent fortement bruitées.

Ces résultats nous ont confortés sur ce que l'on peut atteindre comme performance en WebGL. Il reste maintenant à tester la performance de WebGL dans des cas plus complexes et plus réalistes.

Nous souhaitions utiliser une librairie graphique basée sur WebGL, pour pouvoir bénéficier des différents outils mis à disposition, ainsi que leur structure que l'on viendrait augmenter de différents outils de contrôle de temps et de création d'expériences.

C'est ce que l'on a donc fait ensuite. Nous avons donc fait un choix de librairies graphiques correspondant à nos attentes, et nous les avons testées dans des cas plus complexes, pour étudier leur performances dans différents cas de figure, et les comparer au cas simple précédemment étudié.

2.1.2 Choix d'une librairie WebGL

Il existe une très grande quantité de librairies graphiques utilisant WebGL. Nous avons ainsi par exemple une liste provenant du site de Khronos [1], de Wikipedia [2]

1. https://www.khronos.org/webgl/wiki/User_Contributions
2. https://en.wikipedia.org/wiki/List_of_WebGL_frameworks

et deux autres listes [3] [4]. Le nombre de librairies existant est considérable. Certaines librairies avaient des buts très différents des nôtres se spécialisaient dans des fonctionnalités qui ne nous seront pas utiles (particules, calculs mathématiques), ce qui a permis un premier élagage. De plus, certaines librairies se basaient sur d'autres librairies. Nous n'avons gardé que les librairies de plus bas niveau. Après quelques tests de démonstrations et de recherches, nous avons choisi trois librairies avec le plus de potentiel parmi toutes celles que l'on a pu tester ou qui ont une notoriété.

Ces librairies sont PixiJS, ThreeJS et TwoJS.

— PixiJS : Pour l'accent mis sur la 2D et ses démonstrations performantes
— ThreeJS : Pour ses démonstrations performantes
— TwoJS : Pour l'accent mis sur la 2D

Ces trois librairies offrent différents outils pour créer des graphismes, et permettent également d'utiliser d'autres contextes comme un rendu SVG ou l'utilisation directe du canvas.

En termes d'outils, nous avons retenu la classe Mesh pour ThreeJS, la classe Rectangle pour TwoJS, et les classes Graphics et Sprite pour PixiJS. On précisera par la suite la classe utilisée lorsqu'il s'agit de PixiJS.

Au sujet des contextes utilisés, le principe de SVG est de décrire (à l'aide de code en XML) les formes à dessiner, qui seront dessinées par le moteur de rendu web (le navigateur) et mises à jour si l'on zoom la page par exemple. Le Canvas est en quelque sorte une grande zone dans laquelle les formes sont tracées, et le résultat est une texture générée par le processeur affichée à l'écran. WebGL est un moyen d'utiliser la carte graphique pour générer cette texture. Ainsi, l'élément HTML affichant le résultat de WebGL est une balise Canvas, mais tout le rendu de l'image est fait par WebGL.

Nous avons choisi quelques unes des manières de créer des graphismes, selon l'implémentation de celles-ci, et avons testé tous les contextes différents. Le test de ceux-ci (WebGL, Canvas et SVG si disponible) permet de confirmer, ou d'infirmer, le choix d'utiliser WebGL comme technologie pour notre plateforme.

Enfin, une version codée en WebGL, sans librairie, sera également testée, faisant office de témoin.

2.2 Tests de performance

On a implémenté deux tests avec chacune des librairies (ainsi que la version témoin). Cela nous permet de tester la réponse des librairies en fonction de différents évènements déclencheurs et différentes tâches à accomplir.

Le premier test est basé sur le même principe que le test précédent, où l'évènement déclencheur est la pression du bouton gauche de la souris, et où l'application change la couleur de la scène. Cependant, on ajoute à la scène (jusqu'ici vide) une grille de carrés changeant de couleur de la même manière que l'arrière-plan. Cela permet de faire travailler la librairie et d'avoir un moyen de faire varier la complexité des tâches. Ce test est désigné par Click par la suite.

Le second a pour évènement déclencheur le déplacement de la souris, et dès qu'un tel évènement est détecté, la librairie devra déplacer la grille de manière à suivre le curseur. Ce test est nommé Move.

3. https://gist.github.com/dmnsgn/76878ba6903cf15789b712464875cfdc
4. https://github.com/sjfricke/awesome-webgl{DIESE}libraries

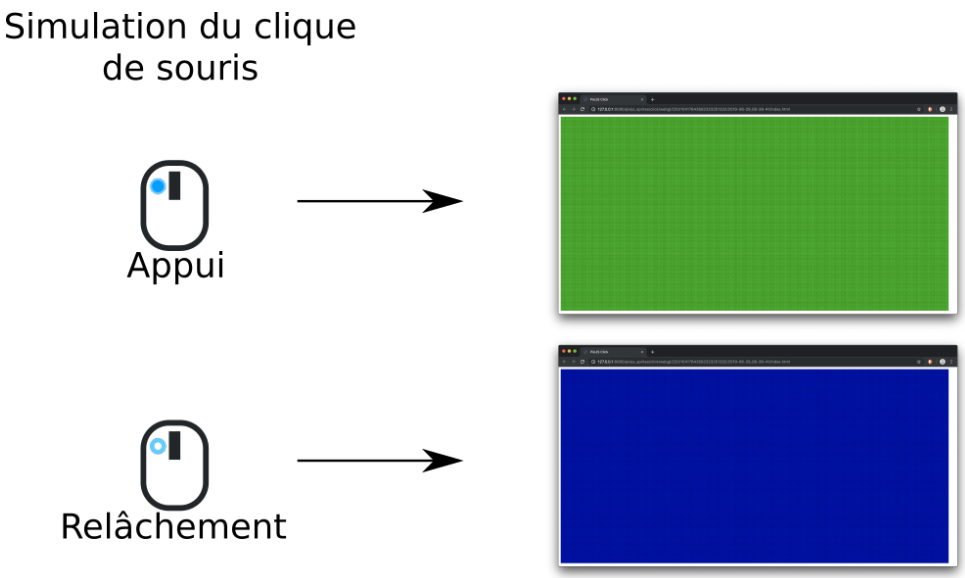

FIGURE 2.3 – Les deux phases de l'expérience Click

Paramètres de l'expérience

Nous avons donc testé ces trois librairies dans conditions et avec différents paramètres, dans le but de trouver la librairie ayant une latence la plus faible, quelque soit l'activité du système alentour, la tâche qu'elle doit accomplir et les types d'évènements qu'elle reçoit.

Pour cela, nous avons fait varier :

— Librairie utilisée : PixiJS avec Graphics, PixiJS avec Sprites, ThreeJS, TwoJS, témoin
— Contexte utilisé : WebGL, Canvas, SVG
— Type de test : Click, Move
— Complexité de la tâche à accomplir : Densité de la grille 20*10 jusqu'à 260*130
— Nombre de processus en arrière-plan : 1, 2, 3

En ce qui concerne la gestion des processus en arrière-plan, lors du lancement des tests, un minimum d'applications autres est en train de tourner, et nous allons ajouter au fur et à mesure des tests de plus en plus de processus "yes", ce processus ayant une consommation régulière de processeur. Cela nous permet de simuler le cas où l'application tourne dans des conditions difficiles, avec un processeur peu disponible.

Chaque cas de figure sera fait 200 fois de suite. On en extraira la moyenne et un intervalle de confiance à 95%.

Nous avons établi un protocole de test et implémenté les outils nécessaires pour mener ces tests.

2.2.1 Protocole d'évaluation

L'évaluation se base sur le temps mis par une librairie pour mettre à jour l'affichage, en fonction d'un évènement.

Tous les tests de librairie ont été menés sur un MacBook Pro 13-pouces de 2017 :

— Processeur : 2.3GHz Intel Core i5
— Memoire : 8GB 2133MHz LPDDR3
— Graphismes : Intel Iris Plus Graphics 640 1536MB

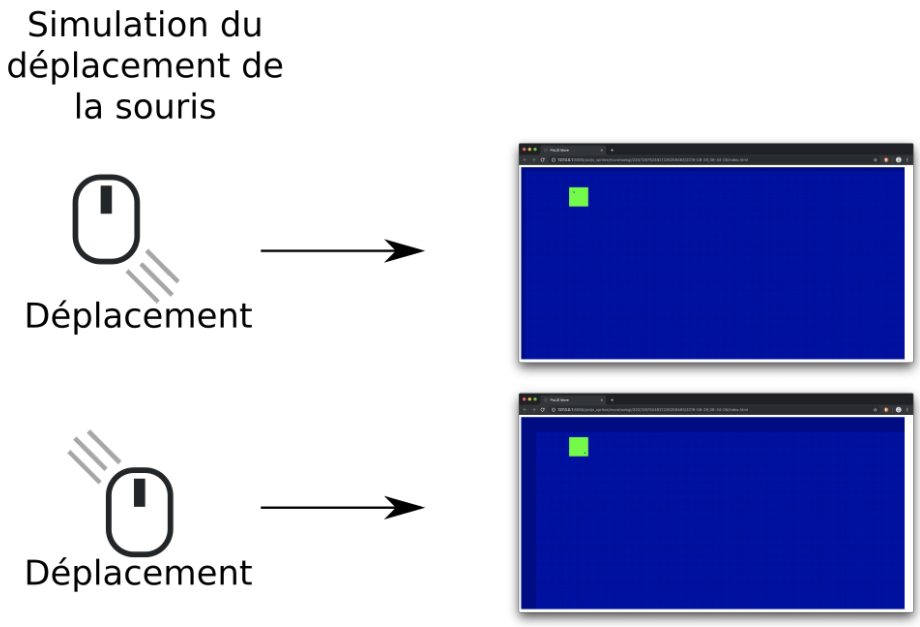

FIGURE 2.4 – Les deux phases de l'expérience Move

— Système d'exploitation : MacOS Mojave Version 10.14.5

Données enregistrées

Pour chacun des itérations, on enregistre des timestamps aux mêmes positions que lors des premiers tests, c'est-à-dire au moment de l'envoi de l'évènement déclencheur, au moment de la réception de celui-ci, lors de la fin de la procédure de mise à jour, et au moment où l'écran est à jour.

Comme mentionné précédemment, on utilisera le nombre de millisecondes depuis l'époque UN*X pour avoir une valeur dont l'origine est indépendante du lancement de l'application, et commune à l'application testée (applications web) et le générateur d'évènements.

2.2.2 Implémentation des outils de test

Pour implémenter ce protocole, nous avons automatisé tout le processus de test.

Pour un cas de figure

Chaque test d'un cas de figure nécessite un programme de simulation d'entrées et de contrôle de sorties, et le lancement de l'application à tester dans un navigateur. De plus, un serveur est utile pour enregistrer les données recueillies par l'application. Il permet également de charger et d'adapter le code de celle-ci aux paramètres sélectionnés. Chaque application sera lancée avec le navigateur Google Chrome, étant le navigateur le plus répandu et gérant le WebGL 2.

Pour toute l'évaluation

Un logiciel principal se charge de créer les combinaisons de paramètres, et la gestion des différents logiciels à lancer et arrêter. Pour chacun des cas de figures :

— Il lancera le serveur, les processus d'arrière-plan, ainsi qu'un moniteur de performance CPU
— Il lancera une instance de navigateur web et lui donnera l'URL du cas de figure, qui le chargera
— Si la page répond, le simulateur lance chaque tour l'un après l'autre.
— Une fois les 200 tours fini, le simulateur envoi un signal de fin à l'application, qui envoie les données récoltées au serveur, qui les enregistrent.
— Le simulateur enregistre ses données et s'arrête.
— Le programme principal reprend la main dès que les données sont enregistrées.
— Il arrête le navigateur.

Au tout début de la batterie de tests, le programme principal lance un petit moniteur système, enregistrant les différentes charges du processus tout au long de l'expérience et les enregistre à la fin.

2.3 Résultats

Pour ne pas surcharger les résultats, certains tests ont été désactivés, lorsque le temps de mise à jour dépassait les 1 seconde ou que le temps de chargement de la page était trop long.

2.3.1 Données

Nous avons comparé les valeurs de latence entre les différents cas de figures. Pour chacun des cas, on prend la moyenne et l'intervalle à 95% du temps nécessaire pour un cycle complet.

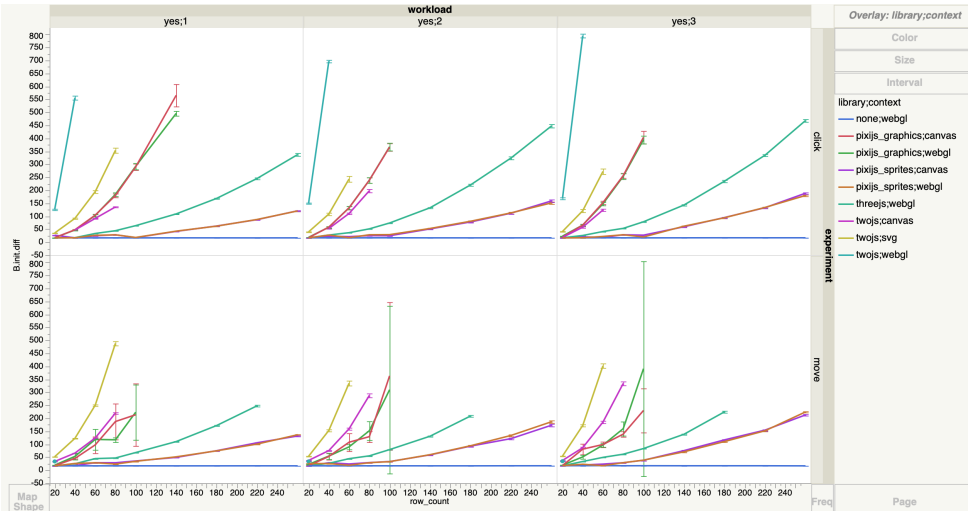

FIGURE 2.5 – Résultats des tests de performance

Faible densité

À densité faible (nombre de ligne de carrés allant de 20 à 80), quasiment tous les cas de figures ont eu des réponses inférieures à 1 seconde.

Nous remarquons 3 phénomènes. Tout d'abord, des trois librairies testées, seule TwoJS a des résultats extrêmement différents selon le contexte utilisé. Canvas est

le plus efficace, et WebGL est le moins efficace. Ensuite, en nous concentrant sur les deux autres librairies (ThreeJS & PixiJS(Graphics & Sprites)), on distingue deux groupes : le groupe de ThreeJS et PixiJS_Sprites, et le groupe de PixiJS_Graphics. TwoJS fait plus ou moins partie de ce second groupe, qui est de moins en moins constant, plus le nombre de carrés augmente. À 80*40 carrés dans la grille, le premier groupe, le plus performant, est toujours en dessous de 55 millisecondes de réaction pour un cycle complet, quelque soit le teste ou la charge CPU. Le second groupe est déjà au dessus des 120 millisecondes.

Finalement, le témoin semble insensible aux variations de paramètres. Il reste constamment à environ 16 millisecondes.

Ainsi, PixiJS_Sprites et ThreeJS sont jusqu'à 80*40 de carrés très stables et bas. TwoJS, de son côté a quasiment déjà dépassé les 1 secondes dans toutes les situations. La version WebGL en Move n'a passé que le premier teste.

Forte densité

Intéressons nous aux cas de figures avec un nombre de carrés entre 80*40 et 260*130 pour ThreeJS et PixiJS. PixiJS_Graphics sera abandonné après le test à 100 lignes, devenant trop inconstant.

On remarque que PixiJS_Graphics continue à ralentir et perdre en stabilité, contrairement aux autres tests, gardant leur constance par nombre de carrés. Cependant, l'écart se creuse entre PixiJS_Sprite et ThreeJS. Quelque soit le test à 180*90, ThreeJS a un délai d'environ 170 millisecondes avec une charge CPU de 1 (1 processus "yes" tournant en arrière-plan), 220 ms pour une charge de 2 et 230 ms pour une charge de 3. PixiJS_Sprite, à partir de 100*50, devient linéaire, mais reste très performant (55, 80, 100 millisecondes respectivement à la charge CPU (1, 2, 3)).

À 260*130, PixiJS_Sprites reste en dessous de 220 ms. ThreeJS en Click atteint les 475 ms, et ne les atteint pas en Move (dépassement des 1 secondes ou chargement trop long).

2.3.2 Interprétation

Divergences entre librairies et techniques

Tout d'abord, la raison de l'apparition des deux groupes de courbes (PixiJS_-Graphics & TwoJS; PixiJS_Sprites&ThreeJS) est que le groupe de PixiJS_Graphics utilise la technique de peinture de texture. Les carrés ne sont pas des maillages, mais sont dessinés sur une texture, apposée en suite sur une grande surface faisant la taille du contexte WebGL. Cette étape de dessin est bien plus lente que l'affichage de maillage, ce qui est fait par PixiJS_Sprites et ThreeJS, d'où la grande différence de performance.

Choix final de la librairie

PixiJS, en utilisant des Sprites, est la librairie la plus performante, introduisant néanmoins un peu de latence par rapport à une version WebGL sans librairie. Cette différence par rapport à ThreeJS peut s'expliquer de part le fait que PixiJS est spécialisée dans la 2D, optimisant donc les outils à cet effet. Elle met en place par exemple un système de batching permettant de mettre à jour toutes les sprites en parallèle

À partir de ces informations, nous avons décidé de choisir PixiJS, permettant d'avoir une très bonne performance et plusieurs outils à disposition, tel que Sprites, mais également si besoin est, Graphics ainsi que d'autres classes comme Mesh qui

est en train d'être optimisé et qui permet de créer des géométries directement, en spécifiant les vertices et leur coordonnées uv.

De plus, leur développeurs mettent de plus en plus l'accent sur le contexte WebGL, mettant de côté le Canvas, ce qui leur permet d'optimiser plus la version WebGL.

Chapitre 3

Première itération sur la librairie pour une première expérience

Comme nous avons pu le remarquer clairement dans un sondage que nous avons effectué (qui sera publié dans le document pour ACM CHI 2020), lorsque l'utilisateur n'a pas le temps de s'arrêter, des divergences entre intention et action (interférences d'interaction) apparaissent, et peuvent avoir des conséquences plus ou moins graves. On aura ainsi quelques cas de figures différents correspondant à un changement d'interface au dernier moment, tel que l'apparition, la disparition ou le déplacement d'un élément graphique inattendu. On voudra par exemple sélectionner un élément, qui disparaîtra ou se déplacera au dernier moment, ou un autre élément apparaîtra au dessus du premier, le masquant.

En effet, la réaction d'un être humain à un stimulus est complexe et possède plusieurs étapes avant d'exécuter une réponse à ce stimulus. Une première phase de perception précède l'interprétation du stimulus perçu, ce qui mène à une phase de décision de la réponse à avoir, pour finalement planifier cette réponse et l'exécuter. Si lors de l'exécution d'une réponse surgit un autre stimulus qui nécessite d'annuler ou de re-planifier son action mais qu'il n'y a plus assez de temps pour intégrer cette information, la réaction arrivera trop tard.

Un temps de réaction de ce genre se situe dans la centaine de millisecondes.

3.1 Stop Signal

3.1.1 Principe

L'étude de réaction à un signal peut être fait de plusieurs manières différentes, dont une qui est assez connue dans le monde des neurosciences : le paradigme du Stop Signal (VERBRUGGEN FREDERICK, 2008, VERBRUGGEN, 2019).

On demande à un utilisateur de réagir à un signal le plus rapidement possible. On aura par exemple un écran blanc sur lequel apparaît subitement une forme, signifiant qu'il faut appuyer sur une touche. Ces essais sont appelés des essais "Go".

De temps en temps, un second signal est envoyé, plus ou moins longtemps après le premier, signifiant à l'utilisateur d'arrêter d'exécuter sa réponse (appuyer sur une touche). Ces essais sont des essais "Stop".

Un essai Go est réussi lorsque l'utilisateur appuie sur la touche, et un essai Stop lorsqu'il n'appuie pas. Pour éviter un phénomène d'attente du signal stop, ces signaux sont placés aléatoirement dans les tests, et on leur demande de répondre aux signaux le plus vite possible.

Ce qui est recherché lorsque l'on fait ce genre de test en général est le seuil à partir duquel on n'a plus le temps d'annuler son action en cours. (La valeur correspondant

à la différence lancement du signal Stop - action théorique de l'utilisateur est appelée SSRT (stop signal response time))

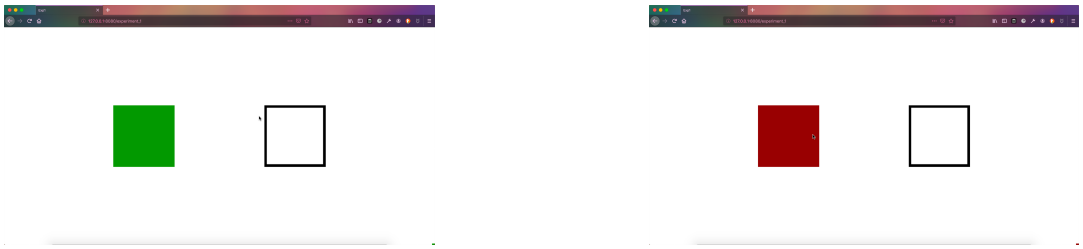

FIGURE 3.1 – Cas de figure Go, et cas de figure Stop

La figure 3.1 montre ce que l'on connait à priori, et ce que l'on ne peut connaitre qu'à la fin de l'essai éventuellement. Il est logique qu'au début d'un essai, on ne sache pas le moment exacte où l'utilisateur appuiera sur la touche. Cependant, c'est par rapport à cette valeur que l'on voudrait déterminer le moment où l'on lance le signal Stop.

N'ayant pas cette valeur, on va essayer de l'estimer en la rattachant à une valeur connue : le début de l'expérience (c'est-à-dire l'affichage du signal Go). Pour faire le lien entre signal Go et signal Stop, nous allons passer par une valeur estimée du moment où l'utilisateur clique, qui est la moyenne prise des essais Go réussis et Stop ratés (dans les deux cas de figures, l'utilisateur clique la cible). On peut ainsi estimer un délai entre signal Go et signal Stop qui correspond à un moment précis avant clique potentiel. Si l'utilisateur arrive à s'arrêter, on réduit le délai potentiel de réaction en augmentant celui entre signal Go et Stop, et on voit si l'utilisateur réussit encore. On peut ainsi adopter une stratégie de dichotomie pour trouver ce seuil.

Ces valeur étant sensibles aux bruits de l'environnement et difficiles à obtenir, l'expérience se doit d'être très cadrée. On doit donc choisir entre une expérience plus contrôlée avec des données plus claires, ou une expérience plus réaliste, avec un comportement de l'utilisateur plus naturelle.

3.1.2 Apport du pointage

On utilise des interfaces graphiques au quotidien, pour accomplir des tâches très diverses. Le fait d'être concentré sur ce que l'on veut faire, plus que sur l'action que l'on fait, change fortement notre niveau d'alerte face à un imprévu dans les actions effectuées.

Pour simuler la concentration sur une autre tâche, les tests en signaux Stop introduisent généralement une tâche à effectuer. Ainsi, au lieu de n'avoir qu'à appuyer sur une touche dès que le signal Go s'affiche, on demandera à l'utilisateur d'appuyer sur la touche directive correspondant à la forme du signal Go s'affichant (flèche pointant vers la gauche ou la droite). D'autres types de tâches ont également été testé (LEUNISSEN et al., 2017). L'utilisation de la tâche de pointage reste rare, et ce qui est évalué est le fait de commencer ou non à pointer (AL., 2008). Cependant, il serait intéressant d'étudier le comportement d'un utilisateur lors d'une tâche de pointage interrompue. Le pointage demande de la concentration, n'est pas un effort ponctuel et est complexe, car il possède plusieurs étapes . C'est dans ce contexte que l'on rencontre bien souvent une interruption.

La tâche de pointage étant plus complexe que l'appui d'une touche, elle est moins facile à prévoir. Nous abordons donc le sujet avec une démarche exploratoire, en

étudiant les phénomènes observables dans des cas de figures réalistes. Des études plus dirigées pourront éventuellement compléter ces études.

3.2 Plateforme d'expérience

3.2.1 Buts de la plateforme

Le but de la plateforme de tests et de pouvoir étudier ces phénomènes dans un environnement introduisant le moins de délais possibles entre les entrées et la mise à jour de l'écran, pour pouvoir obtenir des mesures de temps de réaction de l'utilisateur plus précises.

Pour cela, toute action faite y est enregistrée et formatée pour un usage ultérieur. Ces informations permettent de savoir ce qui s'est passé, de mener des études statistiques ou encore de rejouer l'expérience à la trace.

Basée sur la librairie performante PixiJS, une sur-couche d'éléments d'interface graphique est mise à disposition du créateur de l'expérience, pour lui permettre de reconstituer une interface graphique classique. Ces éléments sont contrôlables par code et par l'utilisateur. Tout changement subit ou effectué par l'élément est enregistré.

Finalement, pour gérer les expériences, un système de scènes permet au créateur d'expériences d'organiser les différents essais, blocs, pauses et écrans informatifs.

3.2.2 Structure

Les trois parties (logs, graphismes et scènes) sont indépendantes, permettant de n'utiliser que ce dont nous avons besoin. Pour les relier, un système d'évènements est mis en place. On pourra par exemple abonner un traceur à une mise à jour d'un élément de la scène.

L'objet principal de la partie traçage est le Logger, objet permettant d'enregistrer tout évènement qu'il reçoit. Il permet également une interaction avec cet historique, tel qu'obtenir un évènement précis, d'un type précis ou simplement le dernier enregistré. Le Logger interagit avec le serveur pour enregistrer les données récoltées lors de l'expérience.

L'objet principal de la partie graphismes est le Component, objet représentant un élément de la scène. Il peut émettre des évènements et s'abonner à ceux de son environnement. Chaque élément de la scène hérite de cet objet, et implémente ces particularités.

Enfin, l'objet principal de la partie organisation de scènes est bien-sûr l'objet Scene, contenant et gérant tous les autres objets graphiques et de traçage qui y sont utilisés. La scène gère sa visibilité et le démarrage et arrêt des autres objets la peuplant. C'est elle qui possède toute la chronologie des actions et évènements à venir lors de l'expérience.

3.3 Expérience intermédiaire

3.3.1 Buts de l'expérience

Lors de l'implémentation de la plateforme, nous avons développé une expérience permettant d'avoir une première série de résultats sur le sujet des interférences d'interaction. Cette expérience permet également de guider les choix pour les

versions suivantes de la plateforme. Le but final de la plateforme est de pouvoir simuler un bureau réaliste pour y étudier les comportements. Cette expérience n'est pas réaliste comme les expériences réalisées à l'aide de cette plateforme le seront, car la plateforme est encore en cours de développement. Elle se base sur les expériences en Stop signal et y introduit le pointage, élément d'interaction très courant.

3.3.2 Protocole de l'expérience

Nous allons donc procéder à une expérience contrôlée suivant un protocole du paradigme de Stop signal modifié.

Déroulement

L'étape principale est l'expérience à proprement parler, qui met en place le paradigme de Stop signal dans un contexte de pointage.

Pour cela, deux carrés (les cibles) sont disposés de part et d'autre du centre de la scène blanche. Les deux carrés sont identiques en termes de couleur et de taille.

À tout moment, un seul carré sera rempli en vert, et l'autre carré ne sera visible que par son contour (noir). L'utilisateur devra cliquer sur le carré visible vert. Si il devient rouge, l'utilisateur devra se retenir de cliquer, peu importe la position du curseur. Après un court instant, la cible repassera au vert et l'utilisateur appuiera dessus pour passer à l'essai suivant.

Une fois un essai terminé, on masque le carré courant et affiche l'autre.

Plusieurs paramètres varient par bloc : la taille des carrés, la distance entre les deux carrés, et le délai entre affichage du signal stop et clique du sujet (stop signal click delay, SSCD). Un bloc est donc la combinaison d'une valeur de taille et une valeur de distance, le SSCD variant au sein du bloc.

Si un signal Stop n'a pas le temps d'être affiché (l'utilisateur est plus rapide et clique avant, sur le carré encore vert), on prend ce test comme un essai Go réussi, on laisse passer un essai Go dans l'autre sens, et on remet le test Stop passé le tour suivant. Ce Stop sera donc rejoué lors du prochain essai dans le même sens de pointage.

Pour pouvoir estimer la vitesse de pointage du sujet, on lui demandera juste avant de faire l'expérience de faire un passage (avec moins de tours) sans qu'il n'y ait de signal Stop. On pourra ainsi obtenir une vitesse moyenne selon les paramètres du bloc. La corrélation de la difficulté de pointage, engendré par la distance entre les deux cibles et la taille de celles-ci, avec le temps mis à effectuer ce pointage est ce que l'on appelle une expérience de Fitts.

3.3.3 Mise en Œuvre de l'expérience

Cinq scènes sont créées, dont une principale, la scène des blocs. La première scène est la scène d'information, expliquant à l'utilisateur ce en quoi consiste l'expérience. Une scène de titres guide l'utilisateur à travers les différentes étapes du test. La seconde scène est la scène des blocs, dans laquelle sera passé les essais. Une fois un bloc passé, on affichera des statistiques à propos des essais à l'aide d'une scène de statistiques. Cela permet de motiver l'utilisateur à ne pas attendre un potentiel signal Stop mais d'agir au plus vite, car attendre ne fera qu'empirer ses résultats. Finalement, à la fin de l'expérience, la scène de conclusion clôturera la session.

Toutes ces scènes sont mises dans une Sequence (objet conteneur, possédant les informations communes aux scènes), et s'appellent lorsqu'elles ont fini par un système d'évènement.

À chaque fois que la scène des blocs est appelée, elle met en place un environnement de test. Avant chaque appel, la séquence met en place un jeu de paramètres pour ce bloc (distance entre carrés, et largeur d'un carré), selon un carré latin. Tout d'abord, elle initialise les éléments graphiques, les traceurs et les paramètres.

La fréquence des signaux Stop est ensuite fixée. Trois règles régissent la répartition ce ceux-ci :

— Il faut répartir les différentes valeurs de délai stop - clique de manière égale entre carré gauche et carré droit
— Un signal stop ne peut succéder à un autre
— 12% des tests sont des tests Stop

Avec ces règles, nous avons mis en place un système de contre-balancement des conditions.

Logs et réaction

Nous lançons ensuite deux loggers, l'un enregistrant tous les évènements de l'expérience (clique sur/à côté du bouton, affichage du signal stop, fin de l'essai, ...) et l'autre enregistrant tout les évènements de l'utilisateur (mousemove, mouseclick, keypress, ...). Le premier servira à récolter les résultats de l'expérience, et le second à pouvoir rejouer ce qui s'est passé au besoin.

Un clique sur le carré vert entrainera le passage au test suivant, de même que le clique sur un carré rouge, ou le dépassement du temps maximal donné pour un essai (1,5 seconde). Dans le cas où l'utilisateur a cliqué sur le carré, et sans s'être arrêté lors du signal Stop si il y en a eu un, la différence entre le début de l'essai et le clique est le temps total de pointage dont l'utilisateur a besoin (movement time, MT). On peut ainsi mettre à jour la moyenne des temps de pointage de l'utilisateur pour ce jeu de paramètres.

Fin d'un bloc et fin de l'expérience

À chaque fin de bloc, on demande aux traceurs d'enregistrer les données récoltées. Ceux-ci les envoient sous forme de json au serveur, qui les enregistrent. Le nombre d'essais Go manqués, de Stops ratés ainsi que la vitesse moyenne de pointage sont calculés puis affichés par la scène de statistiques, avant de lancer le bloc suivant.

Lorsque l'expérience est terminée, un programme de conversion lit tous les fichiers de données et les converti en deux fichiers CSV, l'un contenant essai par essai les données de l'expérience, et l'autre décrivant grâce aux évènements enregistrés le déroulement de la session.

3.3.4 État actuel de l'expérience

L'expérience est en train d'être finalisée et les dernières modifications de paramètres sont faites.

La prochaine étape sera de faire passer l'expérience à des sujets, pour ensuite étudier les données résultantes, ce qui nous permettra de mieux caractériser les interférences d'interaction.

Chapitre 4

Retour sur le stage et perspective à venir

Ce stage à été très formateur pour moi, et m'a permis de valider définitivement la direction que je prends pour la suite, qui est de faire un doctorat dans le domaine de l'IHM. Au long de ce stage, j'ai pu apprendre de nouvelles technologies et méthodologies, et mettre en pratique mes connaissances.

4.1 Compléments aux tâches menées

Nous avons testé la performance de PixiJS à manipuler du texte. En effet, le texte est omniprésent dans une interface graphique, mais n'est pas un élément facile à gérer. Pour du texte statique (ce qui sera souvent le cas), PixiJS est tout aussi rapide que n'importe quelle image. Cependant, si l'on souhaite changer le texte de manière imprévue, la librairie met plus de temps à rendre le texte, devant d'abord le transformer en texture (même procédé que celui de l'objet Graphics). Nous pouvons cependant, selon le cas de figure, optimiser cette vitesse en gérant nous-mêmes les caractères à partir de textures de texte générées par l'objet Text de PixiJS si vraiment besoin est, comme par exemple dans des champs de saisie de texte.

4.2 Apports, contretemps et solutions

L'implémentation d'un système de Benchmarking pour tester les librairies a renforcé mes connaissances dans le domaine du rendu graphique en les différentes APIs de communication avec la carte graphique (OpenGL, Metal), et dans le domaine du web en front-end comme en back-end. J'ai mis à profit les différents outils de développement de projets (Makefile, scripts, Git, …) pour créer un projet complètement automatisé simple d'utilisation. Aussi, j'ai utilisé mes connaissance en systèmes d'exploitation pour pouvoir simuler les entrées utilisateur et lire l'écran.

Ces différentes connaissances m'ont permis de m'adapter aux différentes situations que j'ai rencontré et d'en tirer profit.

Ainsi, lorsque la version de Mac OS X a changé et est passé à Mojave, qui ne supporte plus OpenGL, j'ai converti ma petite application en OpenGL pour remplacer ce dernier par Metal, l'API spécifique à la plateforme Mac.

Aussi, j'ai réussi à trouver un moyen d'accéder aux données de charge du CPU et un moyen simple mais efficace pour le surcharger de manière contrôlée. Cependant, les données de charge de la carte graphique intégrée sont restées inaccessibles, malgré mes différentes tentatives. J'ai dû m'en passer pour ces tests.

Le choix d'utiliser Google Chrome plutôt que Mozilla Firefox vient également d'un détail technique. Firefox laisse une frame de plus aux applications pour préparer leur contenu, ce qui, pour les cliques (non batchés), introduit une latence d'une frame supplémentaire. Lorsque l'on mesure les performances à l'échelle d'une frame, cette différence est non-négligeable.

Au cours de l'implémentation de l'expérience, les parties Traçage et Graphismes de notre plateforme se sont avérées adaptées. Cependant la partie Scène souffrait d'un manque de moyens de structurer le contenu de la scène en composant avec celle-ci. En effet, la Scene n'a pour l'instant qu'un moyen de coder son comportement, qui est la fonction d'initialisation. Ceci à engendré une grosse section de code au même endroit, rendant la gestion de celui-ci plus difficile. Une réorganisation de cette partie est en cours.

Finalement, un test avec et sans anti-aliasing a été mené et a montré une petite perte de performance de 4 millisecondes lorsque l'anti-aliasing est actif, ce qui est acceptable, si il est nécessaire d'en avoir. Cependant, pour notre expérience, nous l'avons désactivé car son utilité est limitée dans notre cas de figure.

4.3 Travaux futurs

En ce qui concerne l'expérience, elle est prête pour commencer les sessions. C'est la prochaine partie du projet qui aura lieu dans les semaines qui suivent.

Terminer de rédiger le document reprenant ces recherches qui va être soumis à CHI 2020.

Pour la plateforme, après une restructuration du système de Scènes, je complèterai la bibliothèque de composants graphiques mis à disposition du créateur d'expériences lors du mois à venir.

En ce qui concerne la plateforme de Benchmarking, elle pourra être utilisée pour d'autres projets, acceptant tout types d'application à tester. Pour la partie serveur, j'ai développé un petit langage permettant d'adapter le code d'une application à une situation. Je souhaiterai le remplacer par un standard de préprocessing plus répandu. La prochaine étape sera d'adapter le code pour d'autres plateformes, la gestion des processus, la création et l'interception des évènements et la récupération d'information sur la charge de CPU étant dépendant de la plateforme.

Bibliographie

AL., MIRABELLA Giovanni et (2008). « Context inuences on the preparation and execution of reaching movements ». In : *COGNITIVE NEUROPSYCHOLOGY*. DOI : 10.1080/02643290802003216.

CASIEZ, Géry et al. (2015). « Looking Through the Eye of the Mouse : A Simple Method for Measuring End-to-end Latency Using an Optical Mouse ». In : *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. UIST '15. Charlotte, NC, USA : ACM, p. 629-636. ISBN : 978-1-4503-3779-3. DOI : 10.1145/2807442.2807454. URL : http://doi.acm.org/10.1145/2807442.2807454.

LEUNISSEN, Inge et al. (avr. 2017). « Reliable estimation of inhibitory efficiency : To anticipate, choose, or simply react ? » In : *European Journal of Neuroscience* 45. DOI : 10.1111/ejn.13590.

VERBRUGGEN, Frederick et al. (2019). « A consensus guide to capturing the ability to inhibit actions and impulsive behaviors in the stop-signal task ». In : *eLife*. DOI : https://doi.org/10.7554/eLife.46323.002. URL : https://doi.org/10.7554/eLife.46323.002.

VERBRUGGEN FREDERICK, D. LOGAN Gordon (nov. 2008). « Response inhibition in the stop-signal paradigm ». In : *Trends in cognitive sciences*. DOI : 10.1016/j.tics.2008.07.005.