Université de Bordeaux
UF INFORMATIQUE MASTER
M2 Informatique pour l'image et le son

Aide de guidage en réalité virtuelle pour les spectateurs extérieurs

Projet de fin d'études

Mihai ANCA, Eddie GERBAIS-NIEF, Hugo LABASTIE

Cliente : Edwige CHAUVERGNE

Centre Inria de l'Université de Bordeaux (Equipe Bivwac)

Table des matières

Introduction

De nos jours, les technologies de réalité virtuelle deviennent de plus en plus répandues au vu de leurs utilisations dans divers domaines tels que le divertissement, les formations, la thérapie, la sensibilisation, la visualisation ou encore la collaboration à distance. Grâce aux avancements dans ce domaine, les casques VR sont devenus de moins en moins chers avec des prix autour des 300€ disponibles au grand public. Ceci a rendu un marché de niche plus accessible et a ouvert la voie vers des innovations.

Quant à la recherche dans les interfaces humain-machine, nous allons nous concentrer sur deux types d'expériences qui s'effectuent : les démonstrations et les expériences utilisateurs. Dans les deux cas, le développeur et l'expérimentateurs peuvent soit utiliser l'ordinateur, soit porter le casque VR. Chacun a sa place dans le domaine, le premier montrant le fonctionnement effectif d'une application et le second recueillant des résultats suite à l'interaction humaine avec l'application.

Un grand problème des expériences est que le porteur de casque est immergé dans un espace virtuel séparé de l'espace physique qui l'entoure [1]. Une barrière s'impose entre les spectateurs extérieurs et le porteur de casque, un aspect qui rend la collaboration difficile à mettre en place sans un second casque. Les spectateurs ne peuvent pas guider le testeur avec plus que la deixis - des indications orales qui montrent la proximité ou la distance entre interlocuteurs - qui est nettement plus laborieuse. Ceci est dû au manque d'un contexte commun partagé par les deux interlocuteurs. Nous avons également l'instinct de bouger nos mains pour rendre la communication plus expressive, mais il n'y a pas de moyen direct que le porteur de casque voie ces gestes.

Il nous est demandé de développer une solution qui crée la liaison entre les deux espaces afin de faciliter la communication des indications dans l'espace. Il s'agit, plus précisément, de capter les gestes des mains d'un utilisateur à l'aide d'un capteur, de trouver une représentation intuitive vers laquelle transposer ces mouvements en VR et d'en créer un produit final simple à prendre en main. Ceci sera indépendant de la VR, mais utilisable dans des projets VR.

Ce document présente en détail le processus de conception et de développement de ce logiciel en deux parties :

- Dans un premier temps, nous nous chargeons de trouver une solution de retranscription des mouvements d'une main en espace VR.
- Ensuite, nous pourrons l'augmenter avec des fonctionnalités complémentaires et des améliorations de l'expérience de l'utilisateur.

Dans les chapitres à venir, nous allons présenter l'état de l'art du domaine, des *user stories* accompagnées par un cahier des charges leur correspondant, les choix des logiciels utilisés pendant le développement, la structure de notre logiciel, le calendrier prévisionnel, la mise en place effective, le calendrier modifié suite aux imprévus, les tests développés pour ce logiciel, ainsi qu'une conclusion reprenant les éléments précédents.

Le développement de cet outil s'effectuera sur l'espace git suivant :
https://github.com/hyperion2022/Guidage-en-VR

1 Etat de l'art

Ce chapitre décrit notre recherche sur les techniques qui ont déjà été développées dans les domaines de la captation du mouvement, l'approximation des directions de pointage des doigts et les représentations possibles de ces mouvements en espace VR.

Détection de la position d'une main

Nous avons cherché sur la détection, soit de la position 2D d'une main par rapport à l'écran, soit en 3D afin de la placer également dans l'espace 3D. Ceci n'est réalisable qu'en utilisant du matériel spécialisé ou en entraînant un modèle d'IA (spécialisé en *Computer Vision*) afin de reconnaître des mains et éventuellement approximer des distances par rapport à la caméra (estimations monoculaires de cartes de profondeur) [2]. Ceci s'applique surtout au cas d'utilisation d'une *webcam* classique, une caméra 3D spécialisée ayant souvent ce type de traitement déjà implémenté dans son SDK.

Il existe différentes architectures de modèle de détection des mains à partir d'images 2D, dont celle-ci [3] qui se rapproche en termes de qualité de la détection à partir d'une *depth map*.

Pour inclure un modèle pré-compilé d'IA dans un projet Unity, une nouvelle fonctionnalité a été développée par ce dernier - Unity Sentis, mais elle reste en état bêta jusqu'à la fin de l'année [4]. Le package actuel d'Unity qui s'en charge s'appelle Barracuda et il fera partie des fonctionnalités de Sentis.

Dans un premier temps, nous avons trouvé le projet HaGRID [5] qui détecte des mains en 2D, ainsi que des gestes pré-définis. Une version améliorée de la détection 2D a été créée [6] afin de détecter les articulations individuelles de la main.

Le projet HandPoseBarracuda [7] détecte la main de façon plus précise jusqu'au niveau des articulations en 3D et les transpose déjà sur un modèle représentatif de la main en temps réel. Malgré son efficacité, l'auteur mentionne deux limitations : la détection ne peut se faire que sur une seule main à la fois et la transposition des gestes se fait en espace image.

Enfin, en regardant les technologies disponibles sur le marché et qui restent encore abordables pour faire l'abstraction des étapes de détection des positions 3D de mains, nous citons la Microsoft Kinect [8] qui reste utilisée à large échelle dans le domaine de la VR grâce à sa capacité de capter le mouvement de tout le corps et la Leap Motion [9] qui reste spécialisée dans la détection des mains.

Direction de pointage d'un doigt

Une fois la position de la main détectée, il faut extrapoler la direction d'un seul doigt afin de trouver le pixel pointé. Une fois calculée, trouver l'objet correspondant à ce pixel s'effectuera en lançant un rayon en direction orthogonale à l'aide des fonctions de l'API d'Unity.

Nous avons trouvé un article [10] qui décrit deux méthodes d'approximation de la direction de pointage basées sur la détection du pointeur et des yeux. Chacune des méthodes est basée sur une orientation différente de la caméra par rapport aux utilisateurs.

Un autre [11] définit une méthode de détection des mains à partir d'images 2D capturées pour être utilisées sur des appareils électroménagers. Une fois la zone de la main détectée par une chaîne de traitements d'image, des vecteurs sont calculés pour chaque doigt afin de trouver une approximation de ses contours.

Les articles suivants utilisent des méthodes moins triviales : le premier par J. Chun et S. Lee [12] décrit une détection à partir de trois caméras et le deuxième par M. Ürkmez and H. I. Bozma [13] utilise deux images, l'une classique en RGB et une autre image de profondeur.

Nous avons trouvé un autre article [14] qui présente une méthode inédite de détection des mains à partir du front de la personne. Cette méthode se base sur le principe de modèle d'histogramme.

Il existe également des solutions déployées telles que le projet `ManoDraw` [15] qui effectue des dessins à l'aide de la détection des mains.

En présence d'un appareil spécialisé de *hand tracking*, il est possible d'utiliser le `package XR Hands` [16] proposé par `Unity` pour accéder aux données de détection.

Représentation des gestes en espace `VR`

Ce qu'il reste à faire du côté du porteur de casque est de trouver une représentation facile à comprendre vers laquelle transposer les mouvements captés. L'article cité dans le sujet [1] propose plusieurs mécanismes efficaces pour attirer l'attention du porteur de casque.

Un exemple spécifique à l'initiation à la `VR` est celui de comment montrer à une personne qui porte le casque comment utiliser les manettes qu'elle est incapable de voir. Pour cela, il est possible d'afficher un modèle 3D des manettes et de changer la couleur des boutons à appuyer.

Pour mieux faire ressortir chacun des doigts sur un modèle 3D des mains du spectateur, il est possible de colorier chaque doigt en fonction de s'il est plié ou droit. En plus, chaque geste visuel pourrait être accompagné d'un retour auditif ou haptique.

Au-delà des mains, afin de rendre la communication plus efficace, un avatar 3D entier peut être utilisé pour que le porteur de casque puisse voir l'orientation dans l'espace du spectateur.

Du côté spectateur, la vue du porteur de casque sera reproduite à un grand angle `FOV` de 180° à 360°. Ainsi, une distortion panoramique est requise sous la forme d'un filtre *fish eye* ou *bird eye*. Ce qui reste important est d'éviter la pollution visuelle et ainsi la surcharge cognitive.

2 *User stories*

Alice souhaite présenter son application en `VR` à Bob, qui va la tester en portant un casque `VR` dans une scène `Unity`. Alice est assise devant l'ordinateur qui tourne cette application et qui dispose d'une `Kinect`. Elle souhaite trouver une façon plus naturelle de donner des indications à Bob vu qu'elle a tendance à expliquer en utilisant beaucoup ses mains et que donner des indications uniquement verbales ralentit l'expérience et nuit à l'immersion de Bob. Ainsi, Alice vient de trouver une solution qui lui capte les mains à l'aide de la `Kinect`, afin de les transmettre à Bob.

Les *stories* suivantes présentent des cas particuliers d'utilisation que nous souhaitons prendre en compte dans le développement de notre produit final.

Story 1 - Prise en main

Alice prépare l'expérience en installant un `plug-in` (`prefab`) facile à prendre en main et qui lui offre une documentation détaillée sur sa mise en place dans le projet. Elle cherche un `plug-in` qui ait un impact minimal sur les performances, sinon Bob risque de ressentir des effets négatifs sur sa perception et ainsi nuire au confort et à l'immersion. Alice n'a pas trop de temps pour apprendre à utiliser ce `plug-in`, alors elle se contente de voir qu'un tutoriel peut lui être affiché à n'importe quel moment à l'aide d'un bouton "*Help*".

Story 2 - Du geste à la visualisation

Alice lance l'application et effectue un pas de calibration qui lui demande de pointer vers chacun des quatres coins de l'écran et d'appuyer sur un bouton de validation une fois qu'elle trouve la position optimale. Une fois que le pointage est calibré, Alice commence à pointer vers un objet de la scène dans la vue de Bob pour qu'il puisse le manipuler. Elle se répère facilement dans la scène à l'aide d'un rayon qui lui montre vers quel objet elle est en train de pointer. L'objet en question est marqué de telle façon que même s'il est partiellement recouvert par un autre, il sera visible pour Alice. Bob visualise les gestes de pointage de Alice de façon naturelle et assez précise à l'aide d'un modèle 3D qui les réplique.

Story 3 - Communication avancée

Bob est encore débutant en matière de `VR`, alors Alice utilise des gestes supplémentaires pour lui montrer sur quels boutons appuyer quand il est perdu dans l'expérience. Quand Bob est occupé dans un coin, Alice change de point de vue sans le perturber en bougeant sa souris vers l'un des bords de l'écran. Elle place plusieurs marqueurs hors du champs de vision de Bob pour qu'il puisse les voir à la suite. Parfois, elle lui transmet des gestes sans qu'il puisse la voir, mais cela ne lui pose pas souci, comme Bob visualise une flèche qui lui montre la direction vers Alice.

3 Cahier des charges

Dans cette partie, nous allons présenter les contraintes imposées par notre cliente, les besoins non-fonctionnels et fonctionnels qui en découlent, ainsi que des extensions possibles destinées à augmenter l'expérience utilisateur.

3.1 Contraintes

Le sujet proposé s'effectue en espace de réalité virtuelle par nature et vu les fonctionnalités reconnues du moteur de jeu `Unity` dans ce domaine, ce dernier nous a été imposé pour sa robustesse et sa modularité à l'aide des `plug-ins` existants. `Unity`, à son tour, nous impose la programmation en langage `C#`.

Nous avons le choix du matériel limité aux *webcams* classiques et aux capteurs 3D `Leap Motion` et `Kinekt`, eux aussi reconnus dans le domaine des interactions homme-machine. Afin d'éviter de complexifier inutilement le projet et pour le rendre accessible au grand public, nous avons choisi de l'implémenter, dans un premier temps, sur une *webcam* classique dont tout ordinateur portable dispose de nos jours ou qui se trouve facilement à une fraction du coût des deux autres appareils. Cependant, les *webcam* ont leurs propres limitations : il faut rester dans un environnement suffisamment illuminé, avec un arrière-plan pas trop chargé et atteindre une résolution minimale pour que la détection des mains soit réalisable.

Il nous est finalement demandé un délivrable sous la forme d'un `plug-in Unity` accompagné par une documentation afin de rendre le projet immédiat à prendre en main dans un autre projet pré-existant. Il sera idéalement compatible avec n'importe quel `SDK` de `VR`, mais nous allons privilégier `SteamVR` selon la préférence de notre cliente.

Nous sommes aussi contraints de tester cette application à l'aide d'un casque `Oculus Quest 2` si nous souhaitons le faire en espace `VR` au-delà d'une simple simulation sur l'écran 2D.

3.2 Besoins non-fonctionnels

3.2.1 Fluidité

Description : Si les indications retranscrites sont irrégulières et sacadées, cela peut nuire fortement à leur perception et comprehension. L'analyse de la posture de la main doit produire une mise-à-jour au moins 10 fois par seconde. Et dans un second temps, le mouvement peut-être lissé par interpolation entre la position à l'instant t et celle à $t+1$.

Tests :

- Nous allons afficher le nombre de mises-à-jour de la position de la main par seconde.
- Nous allons voir si une fonction de lissage est capable de générer une transition satisfaisante entre les mises-à-jour.

Thème : perceptuel

Niveau de priorité : 1

3.2.2 Réactivité

Description : Nous souhaitons une latence entre la captation et la transmission des mouvements de la main qui n'affecte pas l'expérience des utilisateurs, alors nous fixons un objectif de la garder en-dessous de 200 millisecondes.

Gestion d'erreurs :

- Le manque de détection des gestes sera également marqué.

Tests :

- Nous allons tester en mesurant le temps à partir d'une première détection à la première réproduction de geste en espace `VR`.
- Nous pouvons pointer fixement une même direction, puis nous bougeons soudainement la direction pointée. En se filmant et en visionnant cet enregistrement, nous devrions pouvoir connaître précisément ce délai de retransmission.

Thème : perceptuel

Niveau de priorité : 2

3.2.3 Ergonomie

Description : L'expérience sera facile à prendre en main. Les gestes seront choisis afin d'être intuitifs et seront transmis de façon naturelle au porteur de casque. Les effets et éléments visuels introduits dans la vision du porteur de casque ne seront pas invasifs, lui laissant la vue aussi libre que possible. Le spectateur disposera d'une interface complète mais simple dans sa forme, lui permettant d'adapter les fonctionnalités à la situation, comme changer de mode de pointage entre la souris et la détection de la main, activer ou désactiver les gestes de main, mais aussi lui donnant un retour sur lequel il peut s'appuyer, tel le pourcentage de confiance en la reconnaissance de la posture de la main.
Tests :

- Approche empirique : Des tentatives pourront être effectuées en demandant l'avis de plusieurs personnes extérieures d'utiliser le `plug-in` dans un projet exemple afin de leur demander l'avis sur leur expérience.

Thème : perceptuel
Niveau de priorité : 2

3.2.4 Sensibilité du capteur

Description : L'une des nécéssités premières de disposer d'une telle méthode de pointage est le besoin d'une précision qui n'est pas obtenable avec seulement la parole ("à ta gauche", "plus haut", "un peu à droite").
La mesure qui semble la plus pertinante pour quantifier la précision (ou sensibilité) est l'angle minimum discernable entre deux pointages, depuis le point de vue du spectateur. Pour que cette méthode de pointage ait du sens, sa granularité ne doit pas dépasser les 5°. C'est à dire, si deux points espacés de 5 degrés ne peuvent pas être pointés distinctement, alors la méthode n'est pas viable.
La profondeur constitue un troisième degré de liberté, dont la résolution est quantifiable par deux mesures qui sont, la distance maximale atteignable (sa portée), ainsi que la plus petite distance discriminable (sa précision), les deux en mètres. Ces deux mesures ne forment qu'une seule grandeur, car augmenter l'échelle des distances de l'axe `z` augmente la portée, mais réduit la précision, alors que réduire cette échelle réduit la portée mais augmente la précision. Nous obtenons la grandeur qui nous intéresse (la résolution) en divisant la portée par la précision.
Tests :

- Nous allons tester en positionnant à la verticale un modèle de plateau d'échecs, à une certaine distance du point de vue du spectateur. Le spectateur devra pouvoir pointer une case de la grille donnée. Le porteur du casque devra donner le nom de la case pointée (par exemple "E8").
- Pour tester la profondeur, cette fois-ci, nous orienterons le plateau d'échecs de sorte que sa surface soit à angle rasant du point de vue du spectateur.

Thème : matériel
Niveau de priorité : 1

3.2.5 Facilité de la mise en place

Description : L'installation du `plug-in` sera facile, en ne requérant qu'un nombre minimal d'étapes de mise en place. Dans l'idéal, nous souhaitons avoir un unique `script` qui déclenche la création d'autres objets (qui eux aussi contiennent d'autres `scripts`) ou un `prefab` qui contient la plupart des ressources requises. Nous nous permettons de laisser l'utilisateur référencer les objets publics ou `SerializedField` manquants aux `scripts`.
Tests :

- Approche empirique : Une expérience pourra être effectuée en demandant l'avis de plusieurs personnes extérieures (qui connaissent `Unity`) d'inclure le `plug-in` dans un projet exemple afin de leur demander l'avis et de mesurer le temps requis.

Thème : pratique
Niveau de priorité : 1

3.2.6 Réutilisabilité

Description : Le `plug-in` sera composé de fichiers compatibles avec n'importe quel projet `Unity` produit à l'aide d'une version au moins égale à celle utilisée lors de son développement.
Tests :

- Il suffira de rajouter le `plug-in` à quelques projets `Unity` afin de vérifier que le `plug-in` fonctionne comme prévu.
- Tenter de charger le `plug-in` avec différentes versions d'`Unity`, sur différentes plateformes (`Windows`, `Linux`, `MacOS`) sur le même projet.

Thème : pratique
Niveau de priorité : 2

3.2.7 Légèreté

Description : Le rajout du `plug-in` à une application pré-existante ne doit pas impacter ses performances, afin d'éviter les effets négatifs sur l'immersion du porteur de casque [17]. L'analyse d'image en temps réel pour reconnaître une posture de main, pouvant être gourmande en calcul, doit rester suffisamment légère pour ne pas causer une baisse des fps supérieure à 10%. Dans le cas d'une application qui tourne en moyenne à 70 fps, ceci correspond à une chute maximum de 7 fps.
Tests :

- Il n'y a pas de façon uniforme de tester cela, alors il suffira de rajouter le `plug-in` à quelques projets `Unity` afin de mesurer leurs taux de rafraîchissement avant et après sa mise en place.

Thème : pratique
Niveau de priorité : 2

3.2.8 Documentation

Description : Le projet sera accompagné d'une documentation qui détaille la procédure d'installation, les fonctionnalités disponibles, ainsi que leur utilisation. Elle peut se présenter sous une forme textuelle ou vidéo.
Tests :
- Lors des tests de mise en place, nous prêterons attention à l'efficacite de guidage de l'outil en demandant l'avis des testeurs.

Thème : pratique
Niveau de priorité : 3

3.3 Besoins fonctionnels

3.3.1 Transmission des mouvements

Description : Nous allons introduire des éléments visuels dans la scène de sorte à ce que le porteur du casque les voit et en déduise des indications. Il peut s'agir de balises telles que des points lumineux ou des flèches, ou encore d'un modèle 3D (une paire de mains pouvant restituer les mouvements du spectateur en temps réel).
Gestion d'erreurs :
- En cas d'absence d'un mouvement détécté, nous pourrions soit supprimer l'élément visuel, soit le mettre en position figée dans le cas d'une représentation des mains.

Tests :
- Voir le besoin 3.2.5.

Thème : pointage
Niveau de priorité : 1

3.3.2 Captation de la position et du mouvement d'une main

Description : La position, ainsi que l'orientation de la main dans le champs de vision de la caméra, est correctement reconnue et permet le pointage d'un pixel sur l'écran. Une étape de calibration est nécessaire pour créer le lien entre la réalité physique et la scène `VR`, en demandant de pointer successivement les quatre coins de l'écran.
Ceci s'effectuera en 4 étapes :

- la calibration des mains ;
- la détection de la position de la main et de la direction de pointage ;
- la détection du pixel pointé ;
- la détection de l'objet de la scène 3D lui correspondant.

Gestion d'erreurs :

- Si l'interprétation est trop éloignée de la réalité physique, la calibration pourra être relancée.

Tests :

- Voir le besoin 3.2.5.

Thème : pointage
Niveau de priorité : 2

3.3.3 Mise en évidence de l'objet sélectionné

Description : En plus de la visualisation des gestes réalisés par le spectateur, nous souhaitons pouvoir faire ressortir l'objet/la zone pointée s'il en existe une, en le faisant apparaître plus clair ou en lui rajoutant un contour de couleur. Il faudra ainsi ajouter un moyen de supprimer ces éléments visuels, pour laisser la liberté à l'utilisateur d'à quel point il veut surcharger la scène avec des modifications visuelles et en conséquence gérer la visibilité par lui-même. Les objets seront sélectionnables par défaut, mais une option réglable lors de l'installation de notre `plug-in` permettra d'exclure certains objets selon leur *tag* (ou un `script` qui leur sera attaché) ou encore leur taille.

Tests :

- Colorier un objet sélectionné à différentes distances.
- Vérifier que rien n'est colorié quand la direction de pointage est vers une région vide.
- Faire en sorte que les objets tels que les murs ne soient jamais sélectionnables à l'aide de *tags* les discriminant ou d'une taille limite.

Thème : visualisation
Niveau de priorité : 1

3.3.4 Feedback pour le spectateur

Description : Le spectateur pointant vers un endroit de la scène sur son écran disposera d'un retour de ce qui est interprété et affiché. Du point de vue du spectateur, l'élément visuel, ainsi que le point de vue de celui qui porte le casque devront être visibles. En ayant une meilleure représentation, le spectateur qui pointe sera capable de connaître dans quelle mesure la retranscription est fidèle, fluide et réactive.
De plus, même s'il existe un décalage spatial (dû à une mauvaise calibration) ou temporel (dû à une forte latence), le spectateur devra quand même être capable d'indiquer l'objet souhaité en se fiant au retour visuel. Pour cela, nous envisageons de tracer une droite à partir du pixel détecté en direction orthogonale vers l'objet que l'on visualise sur l'écran.
Tests :

- Tester sur plusieurs objets à différentes distances et de différentes tailles afin d'en tirer la fiabilité de cette représentation.

Thème : visualisation
Niveau de priorité : 2

3.3.5 Affichage du pourcentage de confiance

Description : Un affichage en temps réel du pourcentage de confiance en l'interprétation de la position de la main peut grandement aider le spectateur à comprendre quelles conditions améliorent ou déteriorent cette interprétation. Par exemple, s'il est trop éloigné, il se rendra compte qu'en approchant ses mains, le pourcentage de confiance augmenterait.
Tests :

- Tester sur plusieurs objets à différentes distances et de différentes tailles afin d'en tirer la fiabilité de cette représentation.

Thème : visualisation
Niveau de priorité : 3

3.4 Extensions

3.4.1 Rajout d'un second point de vue contrôlable par le spectateur

Description : Il est préférable que le spectateur dispose de son propre point de vue depuis lequel il pointe, car pointer à partir de la vue du porteur de casque `VR` est problématique lorsqu'il bouge la tête ou que le spectateur souhaite sortir de son champs de vision. C'est d'autant plus génant si il y a une latence entre le pointage physique et le pointage interprété. Pour régler cela, nous allons rajouter une seconde caméra visualisable en même temps que celle de base. Elle sera contrôlable à l'aide du clavier et de la souris dans un premier temps.
Il faut s'assurer que le retour est pertinent depuis ce second point de vue. C'est à dire, que la visibilité et l'interprétation des éléments visuels ne dépendent pas du point vue. D'autres éléments et effets visuels peuvent être appliqués à cette seconde caméra, comme une délimitation du champs de vision du porteur du casque `VR`.
Tests :

- Vérifier que passer d'une caméra à une autre ne double pas les mouvements des mains dans les positions correspondantes aux deux affichages.
- Vérifier ce qui se passe quand le spectateur se rapproche trop du porteur de casque ou quand il sort de la scène (borner l'espace à parcourir).
- Vérifier que le retour pour le second point de vue soit pertinent, en utilisant les différents gestes dans diverses situations, avec des points de vue qui varient.

Thème : caméra
Niveau de priorité : 2

3.4.2 Rotation de la caméra pour le spectateur

Description : Bien que la seconde caméra (correspondant au spectateur) soit contrôlable via le clavier et la souris ou une manette, nous pouvons améliorer l'ergonomie en orientant le point de vue du spectateur pour qu'il soit toujours orienté vers le porteur du casque. Cette fonctionalité pourra être activée ou désactivée depuis l'interface. Nous considérons certains gestes comme pertinents pour ce type de contrôle, tel qu'un pointage complètement à gauche ou à droite de l'écran.
Gestion d'erreurs :

- Afin d'éviter des maux de tête pour le spectateur, la vitesse de rotation de la caméra sera changeable dans l'`UI`.

Tests :

- Voir le besoin 3.2.5.

Thème : caméra
Niveau de priorité : 3

3.4.3 Rajout d'autres gestes

Description : Le pointage ne suffit pas toujours pour faire passer toutes les informations pertinentes. Le spectateur disposera, donc, d'autres gestes, tels que l'entourage (désigner une zone fermée en décrivant une boucle du bout du doigt). L'animation complète du modèle des mains permettra de reproduire la pose des deux mains captées, permettant une liberté d'expression totale.
Gestion d'erreurs :
- Afficher un message d'erreur quand un geste n'est pas reconnu ou ambigu (voir le besoin 3.3.5).
- Si des données aberrantes (des positions irréalistes des articulations) sont retournées par l'analyse d'image, elles peuvent être corrigées.

Tests :
- Voir le besoin 3.2.5.

Thème : indications
Niveau de priorité : 1

3.4.4 Placement de marqueurs dans la scène

Description : Le spectateur peut souhaiter que certaines de ses indications soient persistantes. Pour cela, il pourra placer des marqueurs dans la scène. Afin d'éviter la surcharge visuelle, une limite de 10 marqueurs maximum sera imposée par défaut et sera modifiable dans les paramètres de l'`inspecteur`. De plus, les marqueurs pourront être supprimés à l'aide de la souris ou d'un geste spécifique tel que le pincement implémenté dans le `Microsoft HoloLens` quand ils seront atteints ou en rétroactif (une fois que de nouveaux seront placés, les plus anciens disparaîtront).
Gestion d'erreurs :
- Si la détection du geste de suppression produit de faux positifs, il sera possible de changer ou de désactiver le geste.

Tests :
- Poser des balises pour atteindre la limite, dans des positions différentes, à des distances différentes.
- Effectuer des séquences d'ajout et suppression de balises pour s'assurer que les actions désirées sont bien effectuées.

Thème : indications
Niveau de priorité : 2

3.4.5 Gestion des indications hors du champs de vision du porteur de casque

Description : Si une nouvelle indication ne se trouve pas dans le champs de vision du porteur de casque, nous souhaitons qu'il s'en rende compte grâce à un élément visuel à la périphérie de son champs de vision dans la direction de l'indication (comme une flèche ou une bande semi-transparente afin de garder l'`UI` non-invasive). Des pulsations visuelles, auditoires ou haptiques pourront être émises également.

Gestion d'erreurs :

— Si trop de gestes sont détectés les uns après les autres, nous mettrons en place une limite d'une seconde afin de limiter des affichages.

Tests :

— Effectuer plusieurs expériences où le spectateur se trouvera à différentes positions et rotations face au porteur de casque.

Thème : indications

Niveau de priorité : 3

4 Architecture logicielle

Ce chapitre présente l'organisation de notre code entre plusieurs classes et `prefabs` afin de créer un `pipeline` capable de sortir un pixel pointé à partir de données de détection générées par la `Kinect`. Le schéma sur la page suivante relève les relations entre les différentes entités du projet.

Une interface appelée `BodyPointsProvider` est implémentée par tout objet produisant, en temps réel, les positions 3D des articulations d'un corps.

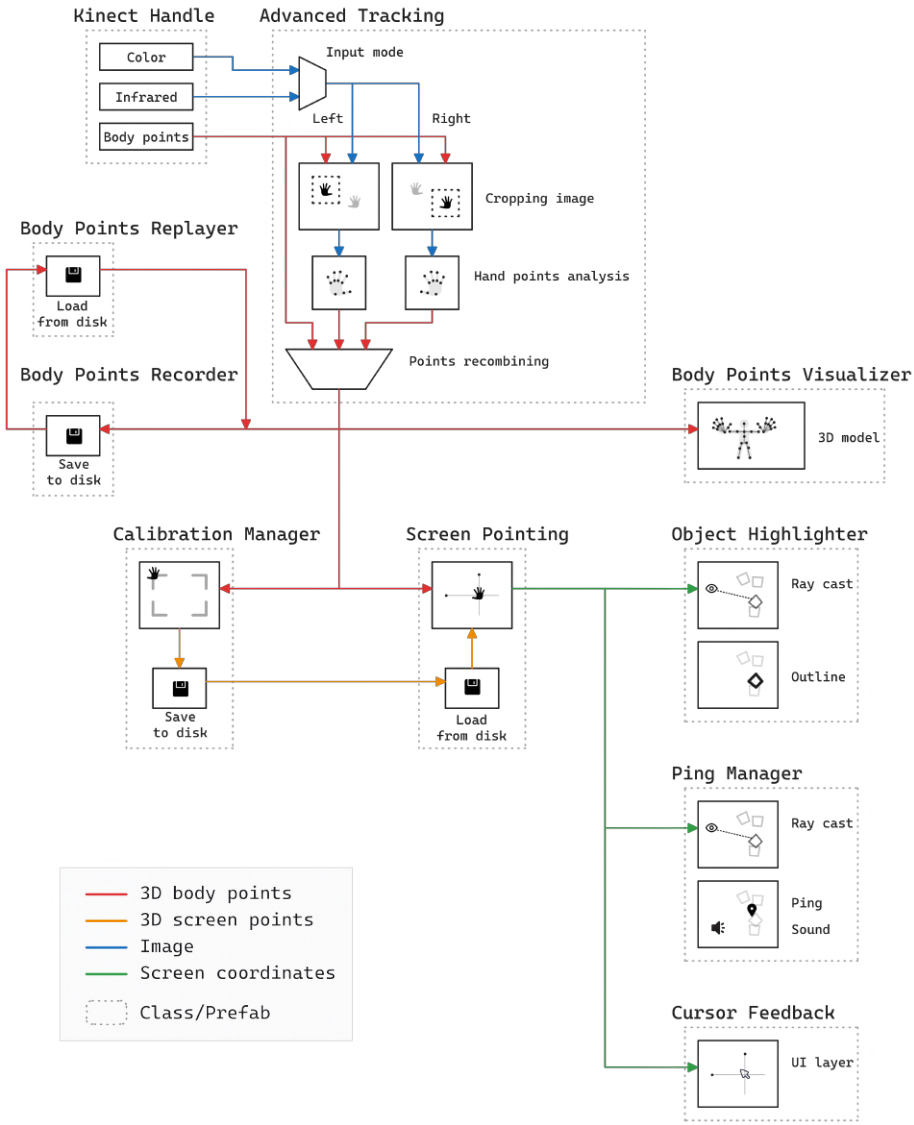
En partant de cette interface, il est possible de concevoir des producteurs et des consommateurs. Par exemple, nous pouvons implémenter cette interface pour `HandPoseBarracuda` ou encore pour la `Kinect`, puis les utiliser indifféremment dans une classe `BodyPointsVisualizer` qui en fait une représentation en animant un model 3D dans la scène.

Une classe (à partir de laquelle nous avons créé un `prefab` qui porte le même nom) appelée `AdvancedTracking` et implémentant `BodyPointsProvider` permet de combiner les points fournis par la `Kinect` avec ceux fournis par l'analyse `HandPoseBarracuda`, car la `Kinect` seule ne fournit pas de points de la main. Pour ce faire, nous récupérons les flux vidéo couleur et infrarouge, desquels nous découpons une image carrée englobant la main. Puis nous lançons l'analyse `HandPoseBarracuda`, pour enfin combiner les points de la main aux points obtenus par la `Kinect`. Cette opération est répétée pour les deux mains. Et optionnellement, plusieurs fois pour chaque main, si plusieurs modes d'entrée sont mis en concurrence.

Il est possible d'enregister une séquence de positions 3D vers un fichier et de la reproduire ultérieurement. Un `prefab` appelé `BodyPointsRecorder` accepte un `BodyPointsProvider` et enregistrera à une fréquence régulière les positions. Lorsque la scène est arrêtée, elles sont sérialisées au format `json`. Un second `prefab` appelé `BodyPointsReplayer`, désérialise ces données au démarrage de la scène, puis émet à un intervalle régulier les positions.

Enfin, nous avons des `prefabs` dédiés à l'étape de calibration requise à chaque changement de positionnement entre la `Kinect` et l'écran de l'ordinateur, au calcul du pixel pointé par l'utilisateur, ainsi que pour les différents types de *feedback* que nous offrons. Le rajout du `Object Highlighter` offre la possibilité d'entourer des objets pointés, le `Ping Manager` gère les notification du placement de marqueurs et le `Cursor Feedback` crée une couche d'`UI` pour afficher un curseur afin de nous repérer le pointage par rapport à l'espace de l'écran.

Ce découpage des différentes fonctionnalités en `prefabs` rend la prise en main du projet, ainsi que son intégration dans d'autres projets, plus simple et intuitive. Le format de `prefabs` nous facilite la tâche, en englobant plusieurs concepts sous une même couche d'abstraction.


Kinect Handle
Color
Infrared
Body points
Advanced Tracking
Input mode
Left
Right
Cropping image
Hand points analysis
Points recombining
Body Points Replayer
Load from disk
Body Points Recorder
Save to disk
Body Points Visualizer
3D model
Calibration Manager
Save to disk
Screen Pointing
Load from disk
Object Highlighter
Ray cast
Outline
Ping Manager
Ray cast
Ping Sound
Cursor Feedback
UI layer
3D body points
3D screen points
Image
Screen coordinates
Class/Prefab

5 Gantt prévisionnel

Cette partie du rapport présente le calendrier initialement prévu pour l'organisation des tâches. Nous avons préparé un tableau de type Gantt sur la page suivante, dont les tâches sont détaillées en Annexe 1.

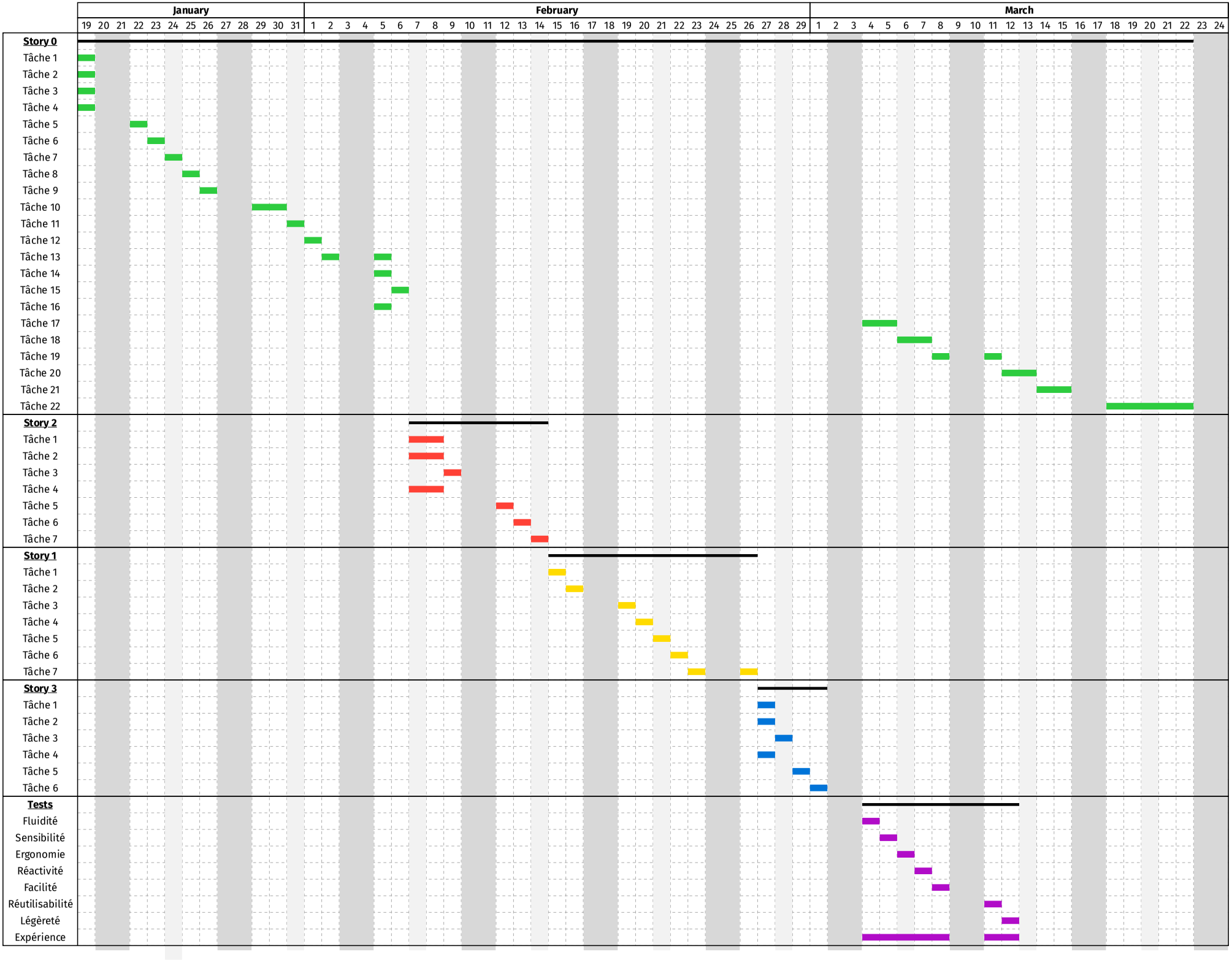
Nous avons prévu la période de travail entre le 18 janvier (le jour du premier rendez-vous avec la cliente) et le 25 mars (le jour convenu pour la présentation orale du projet) avec un suivi hebdomadaire non-marqué. Les bandes gris foncé représentent les *week-ends* que nous avons évités afin de respecter un horaire de travail "à temps plein". Les bandes gris clair correspondent aux journées de mercredi de chaque semaine et aident à mieux comprendre la répartition de chaque semaine individuellement.

Sur ce schéma, nous avons représenté les 3 *user stories* décrites dans la partie 2 (en jaune, rouge et bleu respectivement) dans un ordre qui inverse les *stories* 1 et 2 afin de rendre le développement plus logique. Les outils d'installation seront créés après les fonctionnalités de base. Les tâches ne devront pas demander plus de 6h selon notre planning, mais nous avons laissé la place pour certaines à effectuer sur deux journées dû aux imprévus. Elles seront composées de plusieurs sous-tâches qui seront réparties entre les membres en fonction des besoins spécifiques au moment de leur démarrage.

Nous y avons également ajouté une "*Story 0*", qui correspond aux différentes parties de la mise en place de l'environnement de travail et à la rédaction des chapitres du rapport. Elle été divisée en deux parties, l'une avant le développément effectif du `plug-in` concernant la planification et la recherche initiale et l'autre à la fin, pour reprendre le processus de développement afin de le décrire en détail. Une semaine à part sera entièrement dédiée à la création de la présentation du 25 mars.

En même temps que la rédaction de la seconde moitié du rapport, nous voulons effectuer une période de test correspondant aux besoins non-fonctionnels afin de rendre l'application plus conviviale. Ainsi, nous souhaitons consacrer un jour pour chacun des éléments pour les vérifier et faire les ajustements nécessaires sans devoir refaire des parties entières du projet. Une expérience développée en Tâche 7 de la *Story* 1 sera déployée afin de récolter des avis de différents utilisateurs et de les traiter.

Gantt prévisionnel



6 Implémentation

Ce chapitre présentera les algorithmes utilisés pendant le développement de notre projet, ainsi que les projets externes que nous avons intégrés et les difficultés que nous avons rencontrées.

Quant aux fonctionnalités de base, autrement dit la détection des mains et leurs utilisations, nous avons conçu la chaîne de traitement en figure 1.

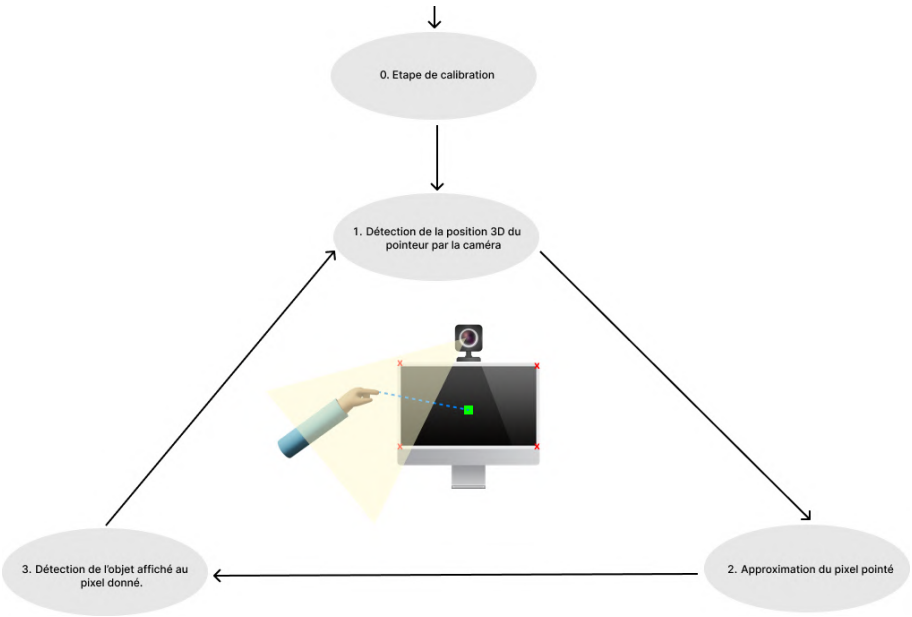


FIGURE 1 – Chaîne de traitement de la détection de la main. (source image : Freepik)

Elle sera ensuite enrichie avec différentes extensions possibles qui ont été décrites dans le cahier des charges.

6.1 Détection de la main

Le premier aspect nécessaire au démarrage du projet a été une façon de détecter la main et d'en tirer des positions 3D à l'aide d'une caméra classique. Nous avons intégré le projet `HandPoseBarracuda` [7] mentionné dans l'état de l'art afin d'avoir l'option la plus puissante pour démarrer notre projet.

Une fois installé, plusieurs scènes nous ont été fournies, dont une qui affichait le flux vidéo de la *webcam* à côté d'une représentation 3D d'une seule main. La détection est assez fiable en fonction de la qualité du matériel et des conditions d'éclairage, avec une détection de chaque articulation (figure 2).

Par contre, ce projet n'est capable de représenter les positions 3D que par rapport au poignet, qui a toujours une profondeur nulle. Nous avons tenté de

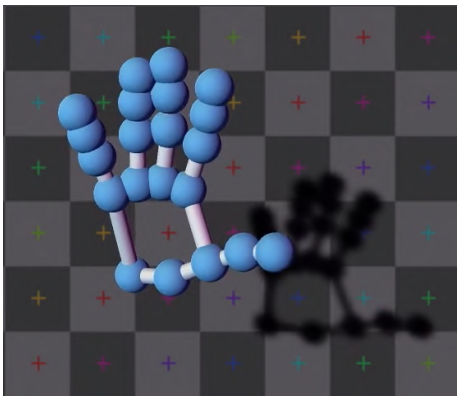


FIGURE 2 – Représentation d'une main détectée à l'aide de `HandPoseBarracuda`.

combiner cette solution avec un autre modèle d'`IA` d'estimation monoculaire de la profondeur à partir d'une image 2D en utilisant toujours le `package Unity Barracuda`, mais nous avons dû abandonner l'idée en manque de temps pour apprendre à faire fonctionner le code d'intégration du modèle qui ne nous fournissait qu'une image noire.

Ainsi, nous avons demandé à mi-parcours des caméras 3D et des `Kinect v2` nous ont été fournies. Elles disposent chacune de 3 composantes [18] : une caméra RGB, un émetteur infra-rouge et une caméra infra-rouge. Ceci assure une captation 2D accompagnée par une carte de profondeur pour passer à la 3D.

La prise en main nous a demandé un temps considérable, mais encore raisonnable. Il a fallu installer le `Kinect for Windows SDK 2.0`, ainsi qu'inclure un projet `Unity` créé par `Microsoft` en utilisant les bibliothèques spécifiques à la `Kinect` afin d'avoir un exemple d'utilisation.

La `Kinect` nous offre un squelette 3D qui reste assez limité en ce qui concerne la détection des mains - nous ne disposons que des positions du poignet, de la paume, du pouce et du bout de la main (figure 3).

Ainsi, nous pouvons détecter la direction de pointage à partir de deux vecteurs : celui qui commence au poignet et qui s'arrête au pointeur et celui qui commence à la tête et qui s'arrête au pointeur. Les deux points de vue de la personne captée sont visibles dans la figure 4, mais nous avons trouvé qu'utiliser la tête au lieu du poignet, malgré que ce soit moins intuitif, est nettement plus précis.

6.2 Détection avancée

La `Kinect` n'analyse pas la posture des mains et les quelques points fournies (poigné et extrémité) sont assez imprécis. Il est possible de lancer l'analyse `HandPoseBarracuda` sur l'image fournie par la `Kinect`, en utilisant la position d'une main pour découper un carré la contenant afin de combiner les points avec

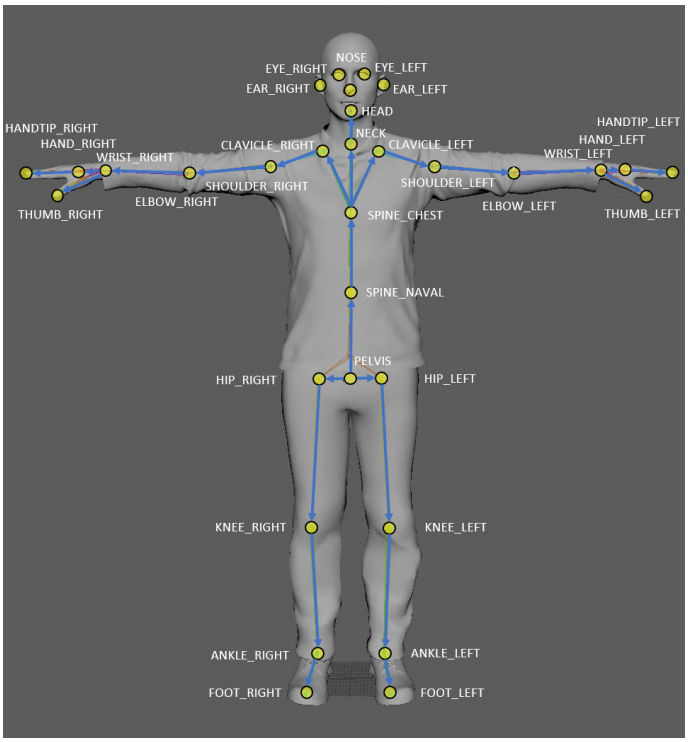


FIGURE 3 – Le squelette 3D généré par la détection de la `Kinect`. (source image : https://learn.microsoft.com/en-us/azure/kinect-dk/body-joints)

le squelette fourni par la `Kinect`. Cette procédure peut être appliquée aux deux mains.

Pour que la détection des mains fonctionne même avec un mauvais éclairage, il devient nécessaire d'utiliser l'image infrarouge de la `Kinect`. L'idéal serait de pouvoir combiner l'image couleur et infrarouge pour améliorer la détection des mains en général. De plus, il est nécessaire d'appliquer un contraste dynamique sur l'infrarouge, ce qui est possible en divisant l'intensité de chaque pixel par l'intensité maximale dans l'image. Ceci nécessite un `shader` particulier pour extraire l'intensité maximale.

L'utilisation combinée des images infrarouge et couleur pose problème, car chacune des caméras a une position différente et même une orientation légèrement divergente (figure 5). La capture d'image n'est pas non plus synchronisée, ce qui signifie qu'il est difficile, lorsque la main est en mouvement, d'obtenir les deux images capturant le même instant. Cependant, ces deux problèmes peuvent être solutionnés, le premier en calibrant la position des deux caméras dans l'espace de la `Kinect` et le second en s'assurant que les marqueurs de temps associés à chaque image soient égaux, en stockant une image en l'attente de sa "jumelle".

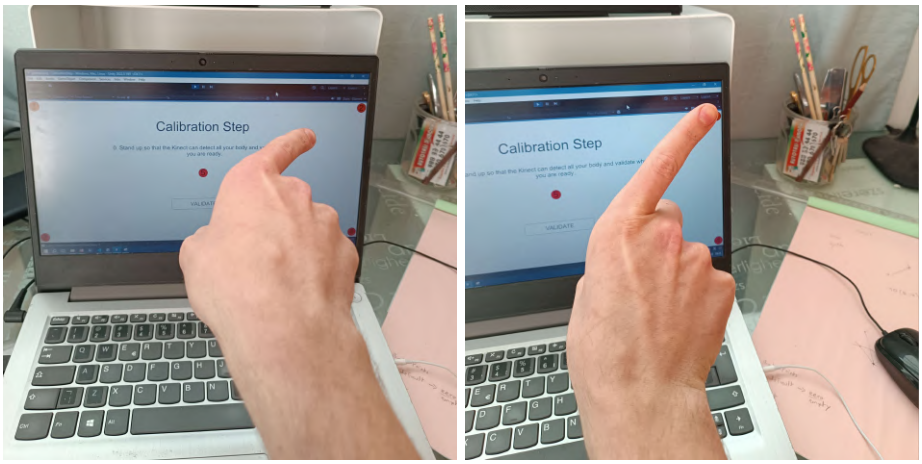


FIGURE 4 – Les deux types de détection appliqués au pointage vers le coin haut-droit de l'écran : à gauche, à partir du poignet (l'élément pointé reste visible) et à droite, à partir de la tête (le doigt cache l'élément pointé).

FIGURE 5 – La détection combinée entre la `Kinect` et `HandPoseBarracuda`, les images des mains décalées visibles à gauche et à droite.

D'ailleurs, il y a un troisième défi lié à comment combiner les deux informations - les canaux RGB et le canal infrarouge. Notre première idée était de multiplier les canaux de l'image couleur par l'intensité infrarouge pour faire ressortir des contrastes normalement absents sur l'image couleur. Ceci est notamment dû au fait que sur l'infrarouge, il y a souvent un trés bon contraste entre la main et le fond, puisque l'éclairage est actif depuis la `Kinect` et fait ressortir la main en plus clair du fond plus sombre. Cependant, cette méthode, tant que les deux images ne peuvent pas être parfaitement superposées, introduit de doubles contours qui semblent plutôt pertuber la détection des mains.

La deuxième approche pour éviter ces doubles contours était de ne pas prendre en compte l'intensité lumineuse de l'image couleur, mais uniquement sa teinte et saturation qui se combineraient à l'intensité lumineuse de l'image infrarouge. Il a fallu implémenter dans un `shader` deux fonctions `rgb_to_hsv` et `hsv_to_rgb`, mais cela ne mène non plus vers le bon résultat. Il semble qu'un alignement parfait des deux images est véritablement une nécéssité, mais il faut choisir la profondeur sur laquelle nous alignons l'image (phénomène de parallaxe). Ainsi, au cas où la main est orientée de sorte que certaines parties sont plus éloignées et d'autres plus proches, la superposition des deux images ne peut pas être parfaite sur l'intégralité de la main. Il est possible, grâce aux données de la `Kinect`, de supposer une inclinaison globale de la main et d'appliquer un alignement progressif dans la direction où l'éloignement varie.

La dernière piste est simplement de faire deux analyses à chaque fois, l'une avec la couleur et l'autre avec l'infrarouge, pour ne garder que le résultat de celle qui obtient le meilleur score. Nous avons de la résilience, mais une utilisation de ressources doublées avec une possible perte de fluidité sur des machines peu puissantes. Cependant, cette méthode est utilisée par défaut. Le type d'entrée peut être sélectionné dans l'interface de paramétrage du `prefab`. Une nouvelle analyse n'est pas démarrée tant que la précédente est encore en cours.

D'autres pistes sont envisageables pour améliorer la captation, notamment en lissant les positions captées par la `Kinect` et en se basant sur les positions précédentes et leur inerties dans l'espace 3D. Il se trouve que `HandPoseBarracuda` utilise déjà une telle méthode.

6.3 Etape de calibration

Une fois la détection fonctionnelle, il faut définir la position du cadre de l'écran par rapport à la `Kinect`. Nous l'avons utilisée sur des ordinateurs portables et nous avons trouvé que les meilleurs positions pour la placer sont à gauche/droite (mais utiliser les deux mains crée le risque que l'une cache l'autre) ou en haut derrière l'écran. Nous avons également observé qu'éloigner la `Kinect` résulte en une meilleure détection. En utilisant un ordinateur classique, nous envisageons une mise en place en-dessous de l'écran.

Nous avons ainsi envisagé deux méthodes de calibration possibles. Nous pourrions toucher les quatre coins de l'écran et mémoriser la position de l'index. Cette solution reste assez efficace et facile à implémenter, mais elle requiert que la `Kinect` soit posée de sorte que les mains restent visibles lorsqu'elles touchent l'écran.

Ainsi, nous retiendrons plutôt la seconde méthode suivante, qui est insensible au placement de la `Kinect` :

— Nous détectons le pointage vers les quatre coins de l'écran (sans les toucher physiquement), nous gardons une paire de points (tête et index) à chaque fois.

— Nous détectons le pointage vers le centre de l'écran à partir de deux positions différentes (depuis sa gauche et sa droite).

Nous obtenons l'éloignement de l'écran en trouvant où les droites pointant le centre, depuis deux positions différentes, se croisent (figure 6).

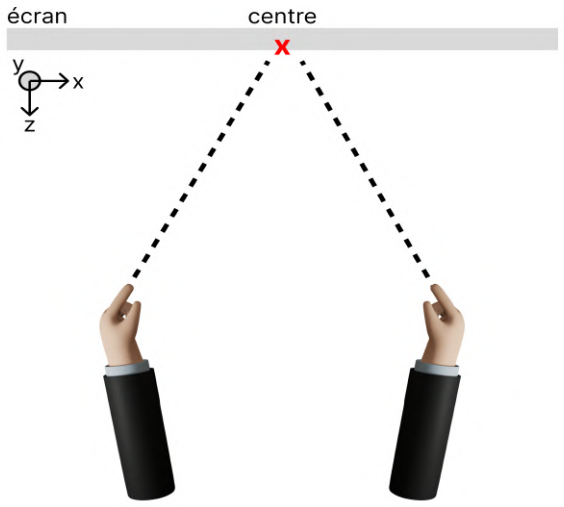


FIGURE 6 – Schéma du calcul du centre de l'écran. (Vue du dessus)

Nous supposons que le plan de l'écran est perpendiculaire à la tête. Alors, nous cherchons les quatre points d'intersection des quatre droites pointant les coins de l'écran avec le plan passant par le centre et orthogonales à la tête (figure 7).

Même si seulement trois coins d'écran sont requis pour définir sa position, nous demandons quand même à l'utilisateur de poiter les quatre coins pour que le processus de calibration soit symétrique et ainsi intuitif pour lui.

Cependant, deux problèmes nous ont forcé à revisiter cette méthode de calibration. Premièrement, supposer que l'écran fait face à l'utilisateur est incorrect, car la plupart du temps, même si l'écran est horizontalement orienté vers l'utilisateur, il l'est rarement verticalement. Deuxièmement, l'imprécision de la captation par la `Kinect` nécéssite une grande redondance dans la capture des points et d'une méthode intelligente de moyennage pour réduire l'effet des données aberrantes.

Au final, nous demandons à l'utilisateur de pointer vers chaque coin de l'ecran depuis trois positions différentes. De plus, lorsqu'un pointage est capturé, l'utilisateur doit rester immobile durant environ deux seconds, car une vingtaine de captures sont effectuées. La position de pointage est estimée en calculant la médianne géométrique plutôt que le centre de masse, car elle est plus résiliante au données aberrantes. De même, en ayant trois positions différentes, il est possible de mieux moyenner là où les trois droites se croisent, de nouveau via la médianne géométrique. Puis, la position de l'écran est calculée à partir de la position estimée des quatre coins sans ignorer le quatrième. Ce calcul profite

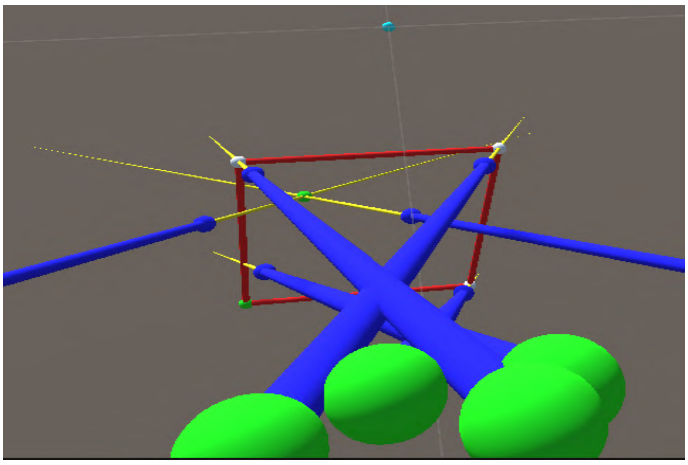


FIGURE 7 – Visualisation des droits qui figurent dans le calcul de la calibration : la `Kinect` en bleu clair, les coins de l'écran en blanc reliés par des droites rouges, les 6 détections de pointage depuis les sphères vertes vers les bleues.

de la redondance d'information et fournit la position de l'écran sous forme d'un repère encodé avec un point correspondant au coin haut gauche, ainsi que deux vecteurs, l'un pour l'horizontale et l'autre pour la verticale.

L'interface de calibration donne les instructions à l'utilisateur au centre, lui demande de valider en cliquant sur un boutton «VALIDATE» et lui indique où pointer en affichant une cible sous forme d'un disque bleu. Pour attirer le regard de l'utilisateur et lui indiquer qu'une action est requise de sa part, ce disque est animé en voyant sa taille légèrement augmenter et diminuer périodiquement (figure 8).

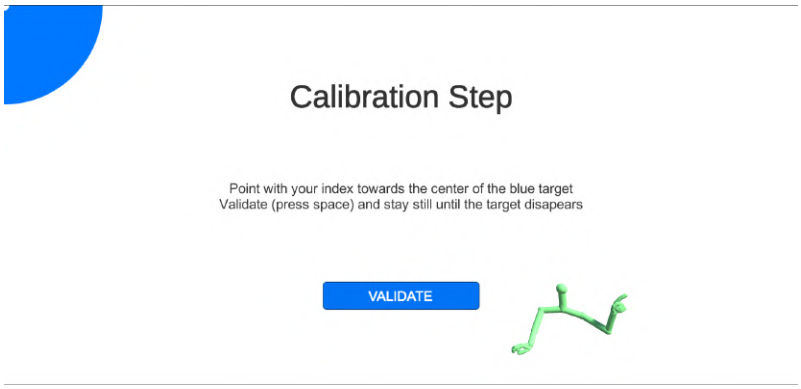


FIGURE 8 – L'interface de l'étape de calibration.

Lorsque que la capture est en cours, pour indiquer qu'elle progresse, la cible

réduit pendant deux secondes jusqu'à disparaitre, indiquant que la capture est réussie.

6.4 Intersection entre deux droites

Il est important de noter que la probabilité que deux droites en espace 3D se croisent est quasiment nule. Lorsque nous parlons de "croisement" de deux droites, il s'agit de la paire de points les plus prôches sur ces droites.

Soient a et b deux droites non-parallèles et ne s'intersectant pas. Soient p et q des points appartenant à a et b respectivement. Soit c le segment reliant p et q. Si le segment c est perpendiculaire à a et b, alors il est de longueur minimale.

L'intuition pour cette propriété est que si le segment c n'était pas perpendiculaire à a (ou b), il serait possible de déplacer p (ou q) du coté où l'angle d'intersection est inférieur à 90° pour obtenir un segment c plus court. En répétant ce procédé, les deux angles d'intersection tendraient vers 90° et nous aurions atteint la longueur minimum de c. Pour être sûr que ce minimum n'est pas local, il faudrait prouver que la longueur de c est une fonction concave.

Pour trouver cette paire de points p et q, nous les écrivons sous la forme $o+tv$ (avec o un point sur la droite et v un vecteur parallèle), puis nous cherchons t de telle sorte que le segment c soit perpendiculaire à a et b ; autrement dit, le produit scalaire serait nul. Cela donne un système à deux équations et deux inconnues, résoluble en inversant une matrice de taille 2x2 :

$$\begin{cases} (o_b + t_b v_b - o_a - t_a v_a) \cdot v_a = 0 \\ (o_b + t_b v_b - o_a - t_a v_a) \cdot v_b = 0 \end{cases}$$

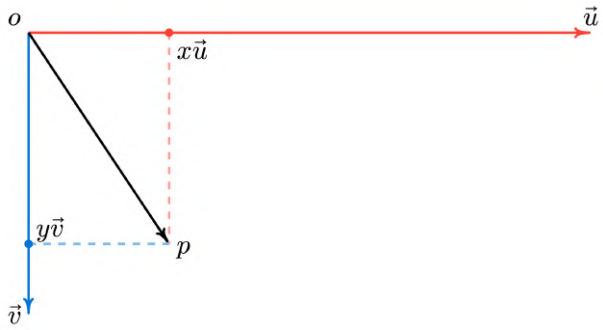
$$\begin{bmatrix} v_a \cdot v_a & v_b \cdot v_a \\ v_a \cdot v_b & v_b \cdot v_b \end{bmatrix} \times \begin{bmatrix} -t_a \\ t_b \end{bmatrix} = \begin{bmatrix} (o_a - o_b) \cdot v_a \\ (o_a - o_b) \cdot v_b \end{bmatrix}$$

$$\begin{bmatrix} -t_a \\ t_b \end{bmatrix} = \begin{bmatrix} v_a \cdot v_a & v_b \cdot v_a \\ v_a \cdot v_b & v_b \cdot v_b \end{bmatrix}^{-1} \times \begin{bmatrix} (o_a - o_b) \cdot v_a \\ (o_a - o_b) \cdot v_b \end{bmatrix}$$

6.5 Détection du pixel pointé

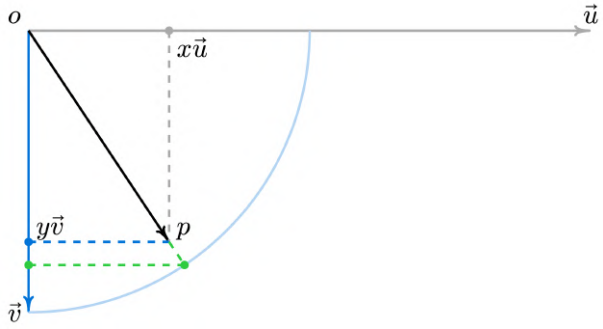
Une fois la position de l'écran connue en espace `Kinect`, il est possible de connaître où l'utilisateur pointe sur l'écran. Pour cela nous capturons la position de sa tête et de son index. Nous trouvons le point d'intersection p entre la droite, passant par la tête et l'index, et le plan de l'écran. Nous nous intéressons au vecteur partant du coin haut-gauche o allant à ce point p, car à partir de ce vecteur, il faut pouvoir déterminer quelle portion x de l'axe horizontal $\vec{u}$ et quelle portion y de l'axe vertical $\vec{v}$ permettent d'obtenir ce vecteur :

$$p - o = x\vec{u} + y\vec{v}$$



Pour trouver les deux facteurs x et y, nous utilisons le produit scalaire, car il permet d'obtenir le cosinus de l'angle entre deux vecteurs. Cependant, le cosinus de ces angles ne représente pas tout à fait les facteurs x et y que nous cherchons.

Regardons l'exemple à partir de l'axe vertical :



Pour obtenir le facteur y, nous appliquons le théorème de Thalès, en divisant le cosinus par la longueur de $\vec{v}$ et en multipliant par la longueur de $(p - o)$. Alors, en employant la formule de produit scalaire $\vec{i} \cdot \vec{j} = \|\vec{i}\|\|\vec{j}\|cos(\theta)$ qui fait apparaître le cosinus, nous obtenons :

$$(p - o) \cdot \vec{v} = \|p - o\|\|\vec{v}\|cos(\theta) \qquad | \ : \|\vec{v}\|$$

$$\frac{(p - o) \cdot \vec{v}}{\|\vec{v}\|} = \|p - o\|cos(\theta) \qquad | \ : \|\vec{v}\|$$

$$\frac{(p - o) \cdot \vec{v}}{\|\vec{v}\|^2} = \frac{\|p - o\|cos(\theta)}{\|\vec{v}\|} \iff$$

$$\frac{(p - o) \cdot \vec{v}}{\|\vec{v}\|^2} = y$$

Alors, la position normalisée sur l'écran est obtenue à l'aide de la formule suivante :

$$\begin{cases} x = \frac{(p-o)\cdot\vec{u}}{\|\vec{u}\|^2} \\ y = \frac{(p-o)\cdot\vec{v}}{\|\vec{v}\|^2} \end{cases}$$

Les positions `(0, 0)`, `(0, 1)`, `(1, 0)` et `(1, 1)` correspondent aux coins haut-gauche, bas-gauche, haut-droit et bas-droit respectivement. Pour obtenir les coordonnées effectives du pixel, il reste seulement à multiplier les x et y résultants par la largeur, respectivement la hauteur, de l'écran en pixels et de les tronquer pour avoir des valeurs entières.

6.6 Détection de l'objet sélectionné

Une fois le pixel calculé précisément, `Unity` nous offre la fonctionnalité de `Raycast`. Ceci nous offre une façon simple de lancer des rayons à partir d'un point donné vers une destination arbitraire en espace 3D. Nous pouvons soit partir du pixel détecté, soit utiliser `Input.mousePosition`, pour lancer un rayon en direction orthogonale à partir de l'écran dans la scène.

Dans le schéma de fonctionnement (figure 9), nous pouvons observer que la direction de pointage physique ne peut pas être extrapolée vers l'espace `Unity` à cause de la projection perspective de la caméra. Ainsi, ce qui est visible sur l'écran correspond à une réalité distordue. Cet aspect nous a fait séparer la détection en deux parties : celle depuis le doigt vers l'écran et celle depuis l'écran vers le premier objet rencontré par le rayon.

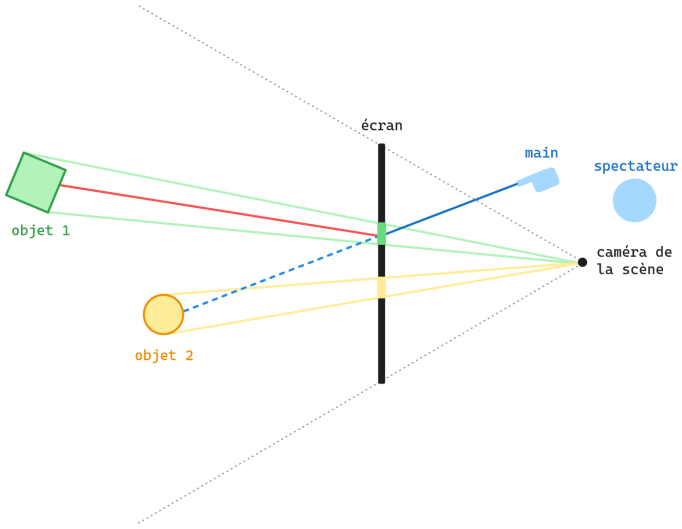


FIGURE 9 – Schéma de pointage qui se sépare entre la réalité physique et la vue de la scène.

Nous pouvons ainsi changer la position d'une image 2D (comme une croix, par exemple) sur l'écran pour simuler un curseur et ensuite avoir un retour visuel pour l'utilisateur. Un essai antérieur avec la souri ne donnait pas de résultats satisfaisants : nous essayions de lancer un rayon depuis la position de la souri sur l'écran, mais tout le rayon était caché par la souri.

Dans la scène, nous utilisons l'`asset` gratuit «`Quick Outline`» [19] du magasin d'`Unity`. En quelques lignes de code, tout objet peut être entouré par une ligne de couleur et d'épaisseur arbitraires. Nous avons conçu un simple `script` de quelques lignes qui effectue ce traitement à chaque fois que la souri est mise sur un objet et qui désactive cet effet une fois que la souri sort de cet objet.

Ce qui est particulièrement utile est le fait que tout le bord de l'objet est visible, même si un autre le cache partiellement (figure 10).

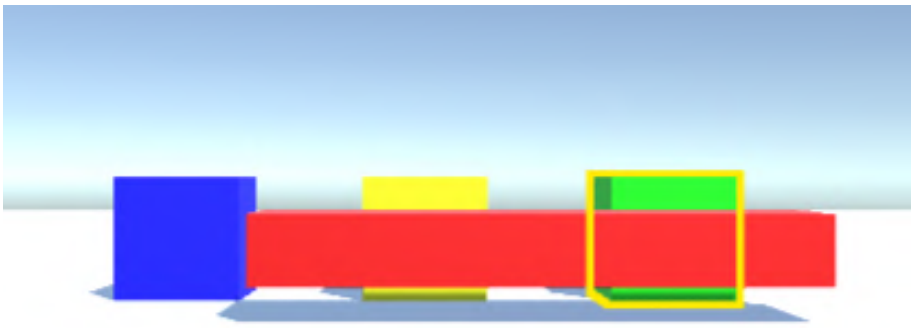


FIGURE 10 – Utilisation du `package` «`Quick Outline`» dans une scène de test.

Pour éviter de pouvoir sélectionner des objets tel que le sol ou des murs, nous avons d'abord pensé à utiliser des `tags`, soit pour *blacklister*, ou pour *whitelister* les objets de la scène. Cependant, cela semble avoir un coût trop élevé pour l'utilisateur final qui devrait manuellement *taguer* adéquatement chaque objet. La solution retenue est d'utiliser la taille de l'objet pointé comme propriété discriminatoire. Une taille maximale est paramétrable depuis l'éditeur `Unity`. Tout objet dépassant cette taille sera ignoré par l'outil de sélection. Par défaut, cette taille limite est de 10 mètres.

6.7 Représentation 3D des mains

Une fois que nous disposons en temps réel des points de corps, nous pouvons animer un model 3D. Pour ce faire, nous utilisons deux types de primitives, des cylindres et des sphères. Tout point est représenté par une sphère et connecté à un autre par un cylindre.

Afficher une sphère à une position souhaitée est assez trivial, mais ce n'est pas le cas pour un cylindre qui en relie deux. Pour y arriver, il faut premièrement positioner le cylindre sur l'un des points, puis l'orienter vers l'autre pour enfin l'étirer et le décaler de la moitié de la distance séparant les deux positions.

Ceci nous offre une représentation abstraite, mais encore reconnaissable, d'un

humain. La combinaison entre le tronc généré par la `Kinect` et les articulations des mains détectées par `HandPoseBarracuda` crée un modèle suffisamment expressif pour recréer des gestes tels que des nombres comptés sur les doigts, des salutations, des signes "OK" et même le "signe des cornes" associé à la musique rock (figure 11).

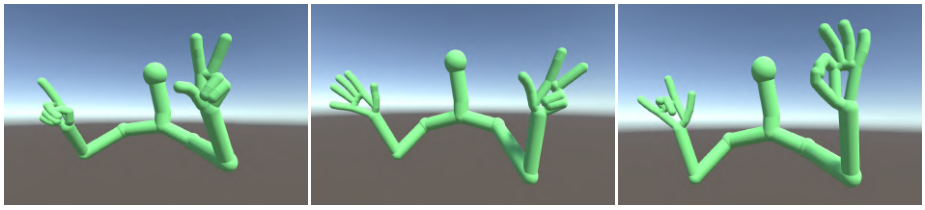


FIGURE 11 – Le modèle 3D de tronc fourni par la `Kinect` combiné avec celui de mains fourni par `HandPoseBarracuda`.

6.8 Marqueurs et outils pour la communication

Nous avons pensé à deux moyens de communication entre les deux acteurs possibles pour notre projet : un système de marqueurs pouvant être placés précisément dans une scène pour attirer l'attention sur une zone plus ou moins précise et un autre qui permettrait de marquer un objet spécifique de la scène grâce à un contour (avec le `package` «`Quick Outline`» décrit plus haut).

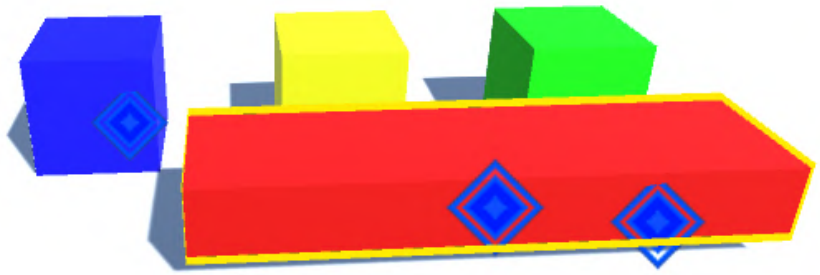


FIGURE 12 – Utilisation de marqueurs dans une scène de test.

Quant aux marqueurs, nous avons fait le choix d'utiliser des `sprites` (images en 2D) pouvant être placés dans la scène. Pour cela, nous nous sommes inspirés des fonctionnalités de *ping* de certains jeux vidéo, utilisées pour faciliter la communication entre différents joueurs. Nous lançons un rayon depuis la position du curseur (contrôlé soit à la souris, soit grâce à notre système de détection de pointage à l'aide de la `Kinect`) et nous plaçons le marqueur avec un léger

décalage par rapport au premier objet rencontré afin de l'en faire ressortir (figure 12).

L'utilisation des `sprites` 2D a cependant apporté des problèmes liés à la lisibilité. Par exemple, un élément 2D dans une scène 3D peut s'avérer problématique si nous nous plaçons à un angle rasant qui le rend invisible. Pour pallier à ce problème, nous orientons, à tout moment, le `sprite` vers la caméra principale.

Un autre problème rencontré par rapport aux marqueurs est que malgré le décalage appliqué à leur placement, nous avons remarqué que ceci n'était pas toujours suffisant. Trouver la valeur de décalage parfaite à appliquer n'était pas une solution viable non plus, puisque cela dépendait de la position de la caméra. Ainsi, nous avons conçu une solution consistant à utiliser une seconde caméra directement dupliquée à partir de la caméra principale, mais restrainte à la vue des marqueurs uniquement. Ces dernier se distinguent des autres objets grâce à l'utilisation de la fonctionnalité de `tags` dans `Unity`. Cette seconde caméra nous permet de rendre les marqueurs visibles à travers tous les autres objets en créant un masque séparant les deux.

Cependant, au final ce sera une autre méthode qui sera utilisée, car la précédente était trop contraignante, notamment dans sa mise en place, puis présentait une difficulté de dimensionnement des marqueurs. Nous utilisons finalement une couche d'`UI` pour afficher les marqueurs. Une liste des marqueurs est maintenue et leurs positions sur l'`UI` sont mises à jour à chaque rendu.

6.9 `Plug-in` et documentation

Nous avons mis ensemble les fichiers utilisables dans un `plug-in` sous forme d'un `package` à importer depuis la fenêtre de `Package Manager` d'`Unity`. Il contient un fichier `.json` à importer et ne prend que 8.5Mo d'espace. Son contenu est montré en Annexe 2.

Nous avons enfin créé une documentation concise que nous laissons sous forme de fichier `README` en format `Markdown` à la racine du projet.

Au début, nous décrivons le but du projet et nous listons les prérequis nécessaires à son démarrage. Ensuite, chaque scène, interface, `prefab` et composante qui puissent être glissés dans la scène sont décrits. Leurs propriétés éventuelles sont également listées.

Elle a été rédigée en anglais pour rendre notre projet disponible à un plus grand public. Nous mettons son contenu en Annexe 3.

Un tutoriel vidéo est également disponible au lien suivant :
`https://youtu.be/Vo7gqL1Ylao?si=I29aPzQCEq8DCsWa`.

7 Gantt effectif

Cette partie est complémentaire au chapitre 5 et reprend le même tableau mis à jour à la fin du projet. Ce dernier sera comparé à sa version prévisionnelle afin d'en tirer le résultat qui reflète dans quelle mesure la planification a été respectée.

Le tableau sur la page suivante utilise les mêmes tâches qu'avant (décrites en Annexe 1), mais les colorie en vert si elles ont été accomplies ou en rouge si elles n'ont pas été tentées ou qu'elles n'ont pas eu de finalité.

Les premières semaines ont bien commencé en respectant le planning sur la mise en place de l'environnement de développement (*Story 0*, en vert jusqu'au 6 février). Nous nous sommes pleinement concentrés sur les parties du rapport que nous pouvions rédiger dès le départ : l'introduction, l'état de l'art, le cahier des besoins, les *user stories* et le Gantt prévisionnel et nous avons créé les espaces collaboratifs nécessaires au dévelopement.

Ensuite, nous avons développé le cœur de l'application, ce qui nous a également pris le plus de temps à cause de différents soucis liés à comment intégrer un second modèle d'`IA` pour en tirer la profondeur et avoir une détection en 3D. Cette impasse nous a mené à demander des `Kinect`, ce qui nous a remis à zéro du point de vue du développement. Une fois l'`API` prise en main, nous avons pu concevoir un modèle de calibration et un calcul d'approximation du pixel pointé par l'utilisateur-guide. Pour pouvoir encore profiter de notre travail sur les *webcams* classiques en utilisant `HandPoseBarracuda`, nous l'avons combiné avec la détection de la `Kinect` pour avoir la détection de toute la main.

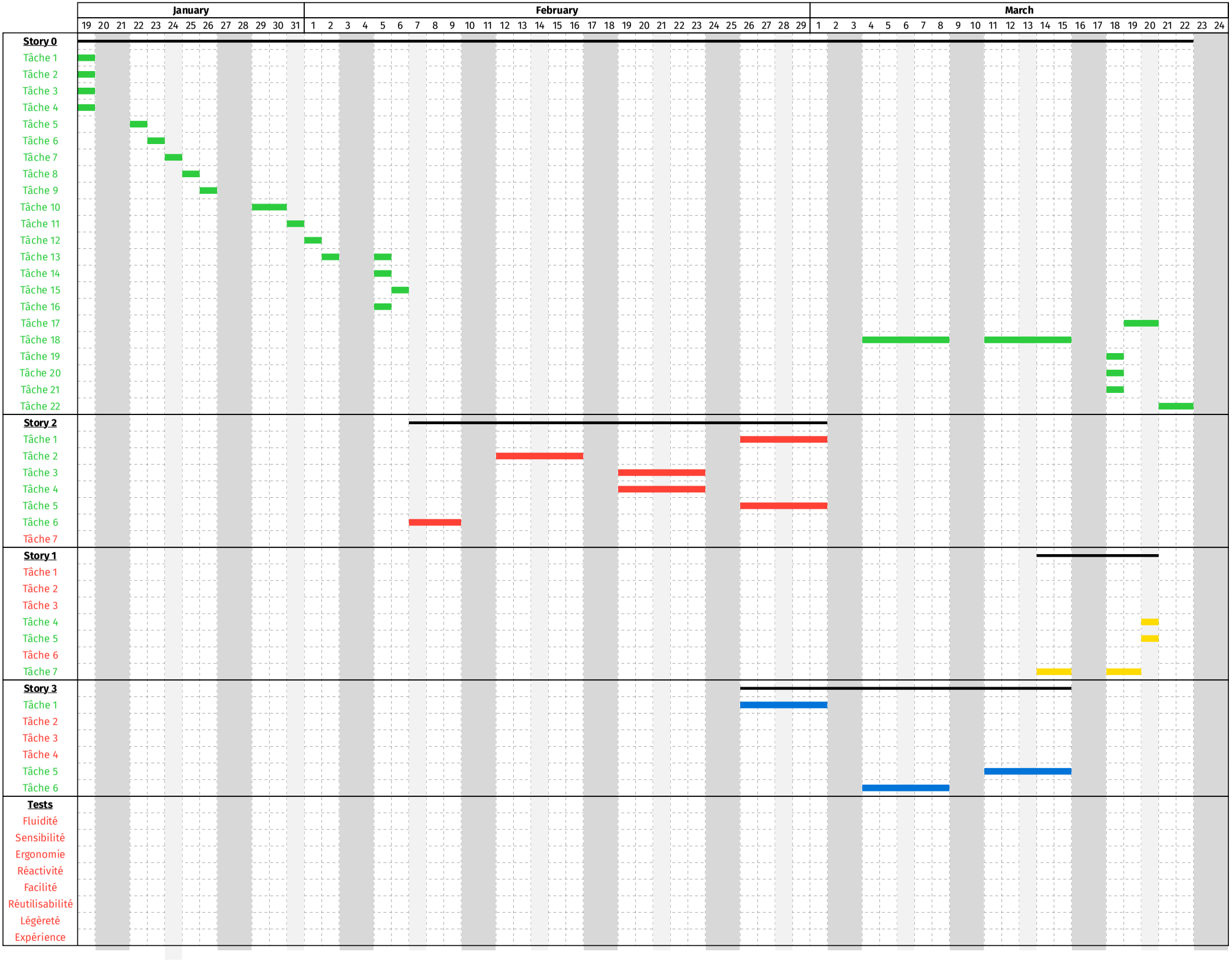
Nous avons continué en rajoutant des extensions possibles (pour la *Story 3* en bleu), notamment la mise en place des marqueurs visuels et la notification de leur placement en dehors du champs de vision. Ce qui n'est pas marqué est la captation des deux mains à la fois, qui est fonctionnelle, est présente dans une scène à part, mais qui n'a pas pu être intégrée dans la scène sans pouvoir tester leur placement en 3D à l'aide d'un casque `VR`. Une implémentation de seconde caméra qui puisse au choix suivre le porteur de casque ou s'en séparer a été tentée, mais pas finalisée. En même temps, nous avons continué sur la rédaction du rapport en le complétant avec l'architecture logicielle, l'implémentation, le Gantt effectif, la partie des tests et la conclusion.

Une fois ceci terminé, nous avons pris la décision de réallouer notre temps restant en créant une démo interactive pour montrer les possibilités futures du projet. Ceci a entraîné l'exclusion d'autres types de tests plus "exacts", mais vu la nature de notre projet, nous avons considéré qu'une expérience serait plus représentative de notre travail qui n'est pas facilement visualisable depuis des images 2D. La démo a été implémentée (comme vu en *Story 1*, Tâche 7), mais elle n'a pas été déployée en recueillant des avis de plusieurs utilisateurs (comme décrit dans «Tests»). Le pourcentage de confiance de la détection n'est non plus affiché, mais il est utilisé pour choisir entre la détection RGB et infra-rouge, et indiqué sur le model 3D animé en changeant la couleur de vert pour capté, bleu pour inféré et rouge pour perdu.

Enfin, nous avons créé un `plug-in` englobant tous les fichiers et les scènes

pertinentes et nous avons rédigé une documentation concise afin qu'une autre personne puisse reprendre le projet et l'utiliser à la suite. Les deux dernières journées ont été dédiées à la rédaction de nos diapositives de présentation.

Gantt effectif



8 Démo

Dans cette partie, nous allons présenter une scène de démo qui englobe les fonctionnalités implémentées dans notre projet. Nous avons décrit des tests possibles dans la partie 3, mais comme évoqué précédemment, la nature complexe du projet et les interactions entre le monde physique et la scène 3D ne nous laissent pas le choix pour effectuer des tests unitaires. Des tests donnant des valeurs de mesure peuvent s'effectuer, mais nous nous sommes concentrés sur la création d'une expérience qui puisse rassembler les fonctionnalités développées : la détection des pointages et leur visualiation à l'aide d'un curseurs, la mise en évidence des objets et le placement de marqueurs.

Nous avons initialement souhaité intégrer le fonctionnement du casque `VR` décrit parmi les contraintes du projet, mais nous avons vite rencontré des erreurs manquant de documentation qui faisaient que l'application `SteamVR` ne détecte pas le casque qui était visible depuis l'application `Oculus Rift`. Vu les contraintes de temps, nous avons décidé de simuler le fonctionnement du casque à l'aide d'une caméra contrôllable depuis la première personne.

Nous nous sommes inspirés des jeux où deux joueurs sont les deux en boîte noire ; autrement dit, chacun a son propre matériel et l'un ne voit pas l'autre ou son matériel, mais ils peuvent communiquer de vive voix. Le but de ce type de jeu est d'avoir un utilisateur qui possède un manuel et qui doit guider l'autre pour atteindre le but en respectant les règles imposées sans jamais voir les opérations qu'il effectue. Ce style de jeu s'appuie fortement sur l'efficacité des instructions communiqués oralement.

Le but de notre expérience est de créer un jeu à deux joueurs, partiellement en boîte noire, dont la fin est atteinte quand une série d'objets numérotés (parfois même de la même couleur) est choisie en l'ordre, tout en évitant un nombre de boîtes "bombe". Le jeu est perdu quand l'ordre n'est pas respecté ou quand un objet "bombe" est choisi.

En début de chaque partie, après la calibration, un nombre arbitraire de cubes sont marqués en tant que "bombe" et le reste sont permutés afin d'en trouver un ordre aléatoire. Le guide a une vue fixe "du dessus" de la scène en temps réel avec une `UI` propre à lui pour voir l'ordre des cubes. Il est aussi capable de voir l'utilisateur qui bouge sur la table de jeu (figure 13).

Le joueur "principal" (qui doit effectuer la tâche décrite par le guide) est représenté par une capsule avec un cube partiellement visible qui décrit son orientation, afin que le guide sache où le joueur est en train de regarder. Il sera contrôllable à l'aide des touches WASD pour la position et des flèches gauche-droite pour la rotation autour de l'axe y (*tank controls* en anglais, souvent utilisées en manque de souri ou de *sticks* analogiques sur les manettes).

Un problème que nous avons considéré est celui de comment contrôler le jeu par deux joueurs en même temps sans le sur-compliquer et sans utiliser d'appareils additionnels. Nous avons procédé ainsi :

— Le joueur principal utilisera le clavier comme décrit plus haut, en utilisant les touches WASD et les flèches gauche-droite pour le mouvement, ainsi que la souri pour cliquer sur les objets afin de les sélectionner. Nous avons

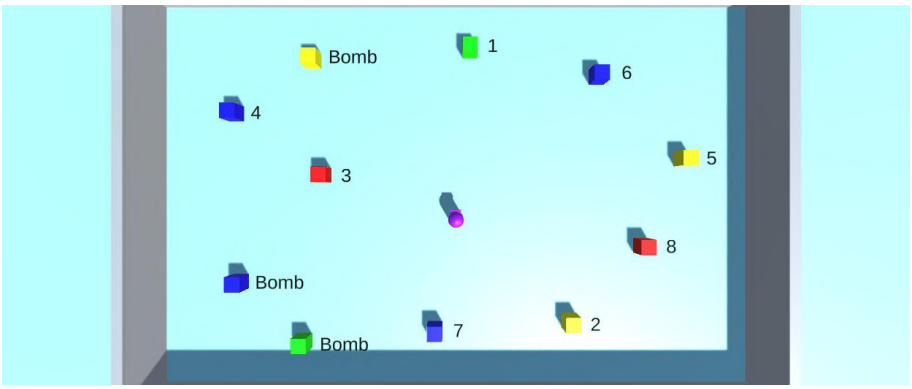


FIGURE 13 – La vue du dessus de la scène qui comporte l'`UI` avec la solution désirée (le joueur principal se trouve au milieu).

fait ce choix pour garder une scène simple, mais des modes alternatifs de contrôle peuvent être envisagés, comme tourner à l'aide de la souri et utiliser ses boutons pour marcher vers ou depuis un point de vue visé.

— Le guide n'a besoin que d'une seule touche (séparable du clavier du joueur principal en utilisant un second clavier, qui ne risque pas de croiser les `inputs` de l'autre, comme nous avons choisi la touche `P` (comme Ping). Elle sera nécessaire pour placer des marqueurs visuels accompagnés d'un *ping* sonore sur les cubes "bombe". Ensuite, la `Kinect` effectuera le travail d'interface "guide-machine". Les cubes sur lesquels il pointe seront entourés pour être visibles par le joueur principal.

Enfin, nous avons créé trois vidéos de ce jeu : l'une pour l'étape de calibration et deux autres pour chacun des points de vue dans la démo effective. Elles sont disponibles aux liens suivants :

— https://youtu.be/3mrPiO7Izbg?si=TqAnH0tzmxjRZ8YI
— https://youtu.be/3LOeghUUq1E?si=M4hZ5L9VA3i7DYoC
— https://youtu.be/51O50HOYqTQ?si=LVh63zBdTX8y6qf0

Avec le temps nécessaire, nous aurions pu faire tester cette démo par plusieurs personnes et recueillir des données à l'aide d'un formulaire avec des questions telles que :

— Trouvez-vous l'application utile/ludique ?
— Est-ce qu'elle est stable/Avez-vous rencontré des *bugs* ?
— Vous la trouvez intuitive/facile à prendre en main ?
— Vous trouvez ce moyen de communication plus efficace qu'uniquement utiliser des commandes vocales ?
— Quels aspects (ne) vous ont (pas) plus ?
— Qu'est-ce qui pourrait être amélioré ?

Conclusion

Nous venons de voir le processus d'exploration, de planification, d'implémentation et de déploiement d'un projet `Unity` qui profite de la détection d'un squelette humain assez basique au niveau des mains afin de permettre à un utilisateur de pointer sur l'écran en utilisant son doigt. Cette fonctionnalité de base a été complétée en étant combinée avec le `package HandPoseBarracuda` pour avoir chacune des articulations des doigts.

Notre projet est déployé sous forme d'un `plug-in` facile à prendre en main pour de futurs projets nécessitant la détection du pointage vers l'écran. Il est également accompagné d'un fichier `README` qui liste les prérequis, ainsi que chaque fichier qui peut être tiré dans la scène et utilisé directement et ses paramètres.

Des augmentations destinées aux expériences visant la communication non-verbale entre un guide et un porteur de casque `VR` ont également été créées : la mise en évidence des objets sélectionnés à l'aide d'un contour, l'utilisation d'un curseur pour évaluer la qualité du pointage et se repérer sur l'écran, ainsi que le placement de marqueurs visuels avec un système de *ping* sonore pour notifier le porteur de casque des événements hors de son champs de vision sans lui polluer la vue avec d'autres éléments d'`UI`.

Le plus grand avantage de notre implémentation de la détection de pointage est qu'elle ne dépend que de l'existance de la `Kinect`, un appareil largement commercialisé à un coût assez faible. La partie `VR` ne constitue qu'un cas d'utilisation possible parmi plusieurs, qui ne demanderait pas un temps considérable pour être mis en place.

Ce qui influe le plus sur la qualité de notre détection est la qualité de détection de la `Kinect`, qui n'est pas toujours satisfaisante. Il y a plusieurs facteurs qui peuvent rendre la détection de très bonne jusqu'à quasiment inutilisable : les conditions d'éclairage, le positionnement de la `Kinect` et la distance face à l'utilisateur, l'écart entre les deux images générées qui nous donnent la détection des points, ainsi que le nombre de personnes détectées.

Avec plus de temps, nous aurions pu soit recréer la détection à l'aide d'une `Leap Motion` ou augmenter le `package HandPoseBarracuda` avec un modèle d'estimation monoculaire de carte de profondeur (à partir d'une *webcam* classique) et utiliser plus de points sur la main pour avoir une meilleure estimation des gestes. Avec un budget alloué, nous aurions également pu échanger les `Kinect v2` pour des `Kinect Azure` pour avoir une meilleure détection dans un format nettement plus petit et simple à mettre en place. Ce type d'application est également susceptible d'être améliorée en utilisant des algorithmes de détection plus puissants que ceux déployés sur la `Kinect` et dans `HandPoseBarracuda` et en utilisant des machines plus puissantes que les nôtres.

Nous aurions également pu implémenter un système de visualisation des mains sur d'autres modèles 3D dans la scène, afficher plus de métriques dans une interface spécifique, rajouter une seconde caméra qui peut suivre la première ou bouger indépendamment, utiliser des gestes plus précis et les associer à des actions. Le projet devrait ensuite être testé de façon plus exacte en dehors de la démo décrite dans le chapitre 8.

L'utilisation de notre projet est prévue dans divers domaines, que ce soit des simulations en `VR` ou non. Il est possible de créer des expériences destinées au divertissement, ainsi que des outils destinés à l'usage industriel. L'une des utilisations les plus évidentes est celle du travail collaboratif, que ce soit en salle de réunion (pour que le présentateur ne soit pas entouré par les spectateurs) ou à distance (dans des situations telle que la pandémie de Covid). Tout un tas de possibilités sont rendues possibles grâce à cette méthode inédite qui sert d'interface humain-machine non-triviale.

Annexes

Annexe 1 - Liste des tâches

***Story 0* : Mise en place de l'environnement de développement**

- Tâche 1 - Création du Trello (15min)
- Tâche 2 - Création du dépôt git (30min)
- Tâche 3 - Création de l'Overleaf pour le rapport (30 min)
- Tâche 4 - Création du groupe sur Discord (5min)
- Tâche 5 - Rédaction des besoins non-fonctionnels (2h)
- Tâche 6 - Rédaction des besoins fontionnels (2h)
- Tâche 7 - Rédaction des extensions (2h)
- Tâche 8 - Rédaction des contraintes (30 min)
- Tâche 9 - Rédaction des *user stories* (1h)
- Tâche 10 - Recherche sur l'existant (1h)
- Tâche 11 - Rédaction des tâches par besoin (1h)
- Tâche 12 - Création du Gantt (1h)
- Tâche 13 - Lecture de l'article cité dans le sujet (4h)
- Tâche 14 - Rédaction de l'état de l'art (2h)
- Tâche 15 - Rédaction de Gantt prévisionnel (30 min)
- Tâche 16 - Rédaction de l'introduction (30 min)
- Tâche 17 - Rédaction d'architecture logicielle (1h)
- Tâche 18 - Rédaction d'implémentation (2h)
- Tâche 19 - Rédaction de Gantt effectif (1h)
- Tâche 20 - Rédaction des tests (1h)
- Tâche 21 - Rédaction de la conclusion (1h)
- Tâche 22 - Préparation de la présentation orale (6h)

***Story 1* : Prise en main**

- Tâche 1 - Extraction du pourcentage de confiance du modèle de détection (1h)
- Tâche 2 - Calcul de métriques (2h)
- Tâche 3 - Création de l'`UI` (3h)
- Tâche 4 - Création du `plug-in` (30 min)
- Tâche 5 - Rédaction de la documentation (2h)
- Tâche 6 - Création du tutoriel (1h)
- Tâche 7 - Création d'une expérience (4h)

***Story 2* : Du geste à la visualisation**

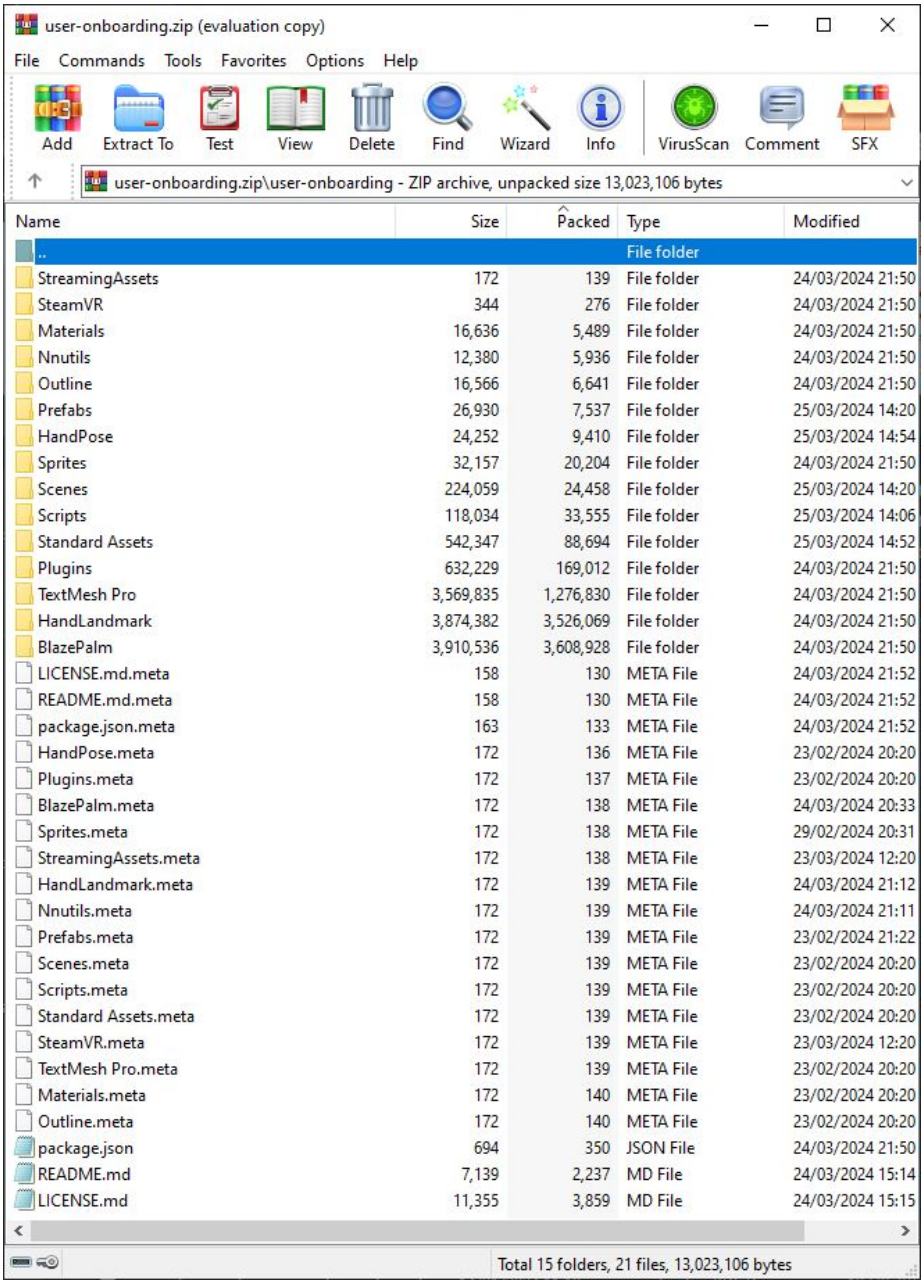
- Tâche 1 - Détection de l'objet sélectionné (1h)
- Tâche 2 - Détection de la main (6h)
- Tâche 3 - Création de l'étape de calibration (2h)
- Tâche 4 - Détection du pixel pointé (4h)
- Tâche 5 - Affichage du curseur (30 min)

— Tâche 6 - Marquage visuel de l'objet sélectionné (1h)
— Tâche 7 - Intégration d'un modèle 3D qui réplique les mouvements (3h)

Story 3 **: Communication avancée**

— Tâche 1 - Création/manipulation de marqueurs (2h)
— Tâche 2 - Implémentation de gestes pour entourer une zone (4h)
— Tâche 3 - Implémentation de gestes pour un tutoriel d'utilisation de manette (4h)
— Tâche 4 - Changement de point de vue (2e caméra + contrôle) (2h)
— Tâche 5 - Notification de gestes hors champs (3h)
— Tâche 6 - Captation des deux mains à la fois (2h)

Annexe 2 - Contenu du `plug-in`



Annexe 3 - Documentation

Projet de fin d'études: Aide de guidage en réalité virtuelle pour les spectateurs extérieurs

Developers: Mihai ANCA, Eddie GERBAIS-NIEF, Hugo LABASTIE

Client: Edwige CHAUVERGNE

User Guide

This project implements a screen pointing direction detection method using your pointer. It is based on the hand detection of a Kinect device coupled with the HandPoseBarracuda project.

It detects when you are pointing at your screen with your index finger and allows you to highlight objects and place pings/tags in the scene.

You can introduce a 3D model of your tracked body in the scene in real time and break the isolation from other users by using hand gestures.

An installation tutorial (in French) is available here: https://youtu.be/Vo7gqL1Ylao?si=NF2aNvuAIeH9yE28

3 demo videos are also available here:

- https://youtu.be/3mrPiO7Izbg?si=TqAnHotzmxjRZ8YI
- https://youtu.be/3LOeghUUq1E?si=M4hZ5L9VA3i7DYoC
- https://youtu.be/51O50HOYqTQ?si=LVh63zBdTX8y6qfo

Prerequisites

- Unity version 2022.3.19f1 or later
- Kinect v2 (SDK to install: https://www.microsoft.com/en-us/download/details.aspx?id=44561)

Installation Instructions

1. Download the user-onboarding.zip file from the root of the project.
2. Unzip it.
3. In your Unity project, open the Package Manager and press on +, then Import from disk.
4. Look in the unzipped folder for package.json and choose it.
5. You will now find the files inside the Packages folder.

Below, you will find descriptions for all the different prefabs you can drag and drop into your desired scene.
For more info, please refer to the tutorial video.

Prefab: KinectHandle

Manages the Kinect.

Properties	Type	
Enable Logs	`bool`	Kinect state will be printed to the console

Scene: Calibration Step

This scene can be launched and will perfom a calibration step. The calibration is saved in a .json file. The calibration is there to determine where the screen is positioned in space. As long as the Kinect and the screen are not moved relative to each other, the saved calibration will remain valid.

The instructions are given as text. The Kinect has to be connected. The validation button can be activated with the space bar or with the right mouse button, even when the mouse is not over it.

Interface: BodyPointsProvider

The KinectHandle prefab inherits from an abstract class called BodyPointsProvider. The KinectHandle is not the only one inheriting from it. Any object that produces real-time body points tracking may implement it (or inherit from it).

Members		
Method	`GetBodyPoint`	takes `BodyPoint`, returns `(PointState, Vector3)`
Enumeration	`BodyPoint`	List of all body points: `Head`, `LeftWrist`, `RightIndex`, etc...
Enumeration	`PointState`	`Tracked`, `Inferred`, `NotTracked`, `NotProvided`
Event	`BodyPointsChanged`	This event is raised every time the points positions or states change

---

Prefab: BodyPointsVisualizer

Reads body points from a BodyPointsProvider and animates a 3D model.

Properties	**Type**	
BodyPointsProvider	`BodyPointsProvider`	An object that produces body points

Prefab: ScreenPointing

Loads the saved calibration from the .json file. Then, from a BodyPointsProvider, determines where the user is pointing on the screen with its index finger. It also considers the mouse as a means of pointing.

Properties	**Type**	
TargetCamera	`Camera`	Which camera is the point of view of the pointing user
CalibrationFilePath	`String`	File path to the saved calibration
SmoothFactor	`float`	A factor between `0.` and `1.` of how much to smoothen the pointing position over time

Members		**type**	
Getter	`pointing.mode`	`PointingMode`	
Getter	`pointing.atPixel`	`Vector2`	Screen position in pixels
Getter	`pointing.atNorm`	`Vector2`	Normalized screen position
Enumeration	`PointingState`		`None`, `Body`, `Mouse`

This prefab comes with several components that make use of the pointing:

CursorFeedback

Gives a pointing feedback by showing a cursor where the pointing position is inferred on the screen.

ObjectHighlighter

Highlights the pointed object in the scene, just by hovering over it or by clicking on it.

Properties	Type	
KeyCodes	`List<KeyCode>`	Which key press triggers object selection
Hovering	`bool`	While an object is only pointed at, it is highlighted
SizeLimit	`float`	The highlight will ignore any object with a size exceeding the limit

PingManager

Places tags (or pings) in the scene where the user is pointing. Note that in order for the pings to be seen from another camera, it is necessary to also add the PingLayer prefab (see below).

Properties	Type	
KeyCodes	`List<KeyCode>`	Which key press triggers ping placement/removal

---

Prefab: PingLayer

Shows the pings placed by a Ping Manager to a camera.

Properties	Type	
PingManager	`PingManager`	The PingManager from which we want the pings to appear
TargetCamera	`Camera`	The camera from which the pings have to appear

---

Prefab: BodyPointsRecorder

Given a BodyPointsProvider, it records the body points while the scene is playing and saves them in a .json file when it stops.

Properties	Type	
BodyPointsProvider	`BodyPointsProvider`	An object that produces body points
CapturesPerSecond	`float`	The frequency of capture
OutputFilePath	`String`	File path where to write the recorded points

---

Prefab: BodyPointsReplayer

Implements BodyPointsProvider. Produces body points by reading previously saved ones from the given .json file.

Properties	Type	
InputFilePath	`String`	File path where to read the recorded points

Bibliographie

[1] E. Chauvergne, M. Hachet, and A. Prouzeau, "User Onboarding in Virtual Reality : An Investigation of Current Practices," in *CHI 2023 - Conference on Human Factors in Computing Systems* (ACM, ed.), (Hamburg, Germany), ACM, Apr. 2023.

[2] F. Al Farid, N. Hashim, J. Abdullah, M. R. Bhuiyan, W. N. Shahida Mohd Isa, J. Uddin, M. A. Haque, and M. N. Husen, "A Structured and Methodological Review on Vision-Based Hand Gesture Recognition System," *Journal of Imaging*, vol. 8, no. 6, 2022.

[3] C. Zimmermann and T. Brox, "Learning to Estimate 3D Hand Pose from Single RGB Images," in *IEEE International Conference on Computer Vision (ICCV)*, 2017. https ://arxiv.org/abs/1705.01389.

[4] "Sentis overview | Sentis | 1.3.0-pre.3 — 1.3." https://docs.unity3d.com/Packages/com.unity.sentis@1.3/manual/index.html. [Accédé le 28/02/2024].

[5] "GitHub - cj-mills/icevision-openvino-unity-tutorial : This tutorial covers training an object detector with the IceVision library and implementing it in a Unity game engine project using the OpenVINO Toolkit. — github.com." https://github.com/cj-mills/icevision-openvino-unity-tutorial. [Accédé le 28/02/2024].

[6] "GitHub - kinivi/hand-gesture-recognition-mediapipe : This is a sample program that recognizes hand signs and finger gestures with a simple MLP using the detected key points. Handpose is estimated using MediaPipe. — github.com." https://github.com/kinivi/hand-gesture-recognition-mediapipe. [Accédé le 28/02/2024].

[7] "GitHub - keijiro/HandPoseBarracuda : Hand and finger tracking solution (MediaPipe Hands) for Unity Barracuda — github.com." https://github.com/keijiro/HandPoseBarracuda. [Accédé le 28/02/2024].

[8] "Azure Kinect DK – Develop AI Models | Microsoft Azure — azure.microsoft.com." https://azure.microsoft.com/en-us/products/kinect-dk/. [Accédé le 29/02/2024].

[9] "Digital worlds that feel human | Ultraleap — ultraleap.com." https://www.ultraleap.com/. [Accédé le 29/02/2024].

[10] M. Lee, D. Weinshall, E. Cohen-Solal, A. Colmenarez, and D. Lyons, "A computer vision system for on-screen item selection by finger pointing," in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, vol. 1, pp. I–I, 2001.

[11] D. Lee and S. Lee, "Vision-Based Finger Action Recognition by Angle Detection and Contour Analysis," *ETRI Journal*, vol. 33, no. 3, pp. 415–422, 2011.

[12] J. Chun and S. Lee, "A vision-based 3D hand interaction for marker-based AR," *International Journal of Multimedia and Ubiquitous Engineering*, vol. 7, pp. 51–58, 01 2012.

[13] M. Ürkmez and H. I. Bozma, "Detecting 3D Hand Pointing Direction from RGB-D Data in Wide-Ranging HRI Scenarios," in *2022 17th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pp. 441–450, 2022.

[14] G. Tofighi, N. A. Afarin, K. Raahemifar, and A. N. Venetsanopoulos, "Hand Pointing Detection Using Live Histogram Template of Forehead Skin," 2014.

[15] "ManoDraw - ManoMotion Community Project — manomotion.com." https://www.manomotion.com/manodraw-a-manomotion-community-project/. [Accédé le 04/03/2024].

[16] "XR Hands | XR Hands | 1.1.0 — 1.1." https://docs.unity3d.com/Packages/com.unity.xr.hands@1.1/manual/index.html. [Accédé le 04/03/2024].

[17] "The Importance of Frame Rates." https://help.irisvr.com/hc/en-us/articles/215884547-The-Importance-of-Frame-Rates. Accédé le 25/01/2024.

[18] R. Brancati, C. Cosenza, V. Niola, and S. Savino, "Experimental Measurement of Underactuated Robotic Finger Configurations via RGB-D Sensor," 06 2018.

[19] "Quick Outline | Particles/Effects | Unity Asset Store — assetstore.unity.com." https://assetstore.unity.com/packages/tools/particles-effects/quick-outline-115488. [Accédé le 10/03/2024].