Université de Bretagne Occidentale

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

RAPPORT DE STAGE

EN ENTREPRISE

«Interagir avec un ruche virtuelle au moyen d'une vraie ruche jouant le rôle d'interface matérielle tangible»

HENARD Aymeric

11 mai - 31 juillet
2020

Tuteur de stage : Mr.DUVAL Thierry
Enseignant référant : Mr.RIVIÈRE Jérémy

Établissement / Formation : Université Bretagne Occidentale - M1 Informatique SIIA
Entreprise d'accueil : IMT Atlantique - CS 83818-Technopôle Brest-Iroise 29238 BREST

Table des matières

Introduction

L'objectif de ce stage est de réaliser un simulateur 3D permettant d'agir sur une ruche virtuelle au travers d'interactions tangibles grâce à l'utilisation d'une ruche réelle. Ce simulateur servira de passerelle pour la communication entre l'utilisateur et le modèle multi-agent simulant l'activité des abeilles de la ruche. Les mouvements de l'utilisateur seront captés par un système de tracking 3D et des interactions tangibles seront mises en place afin d'interagir avec le modèle multi-agent.

La ruche :

La ruche utilisée est similaire à l'illustration 1. Elle peut contenir plusieurs cadres, rangés de la même manière. Les cadres peuvent être retirés et placés ailleurs.
Il y a un toit (absent de l'illustration 1) permettant de couvrir la ruche.

Illustration 1: Photo d'une ruche réelle

Unity[1] :

Unity est un moteur de jeu très répandu dans l'industrie du jeu vidéo permettant de déployer des jeux sur tous les supports[2]. Afin d'offrir à l'utilisateur un outil très simple et efficace, la grande majorité des traitements sont cachés ce qui permet à l'utilisateur de se concentrer sur l'essentiel de son projet.
Le développement d'un projet Unity est basé sur l'utilisation des « GameObjects » et des « scripts ».
Les GameObjects sont des objets activés régulièrement ayant un comportement défini, généralement dans un script.
Les scripts sont des fichiers écrits en C# décrivant le comportement d'un GameObject ou réalisant un traitement autre, utile à l'application (exemple : la gestion de niveau dans un jeu).
Dans tous les cas, pour être utilisé, les scripts doivent être rattaché à un ou plusieurs GameObject.

Le travail :

Le début de mon stage a commencé en télétravail. Le simulateur 3D est développé en langage C# sur Unity, offrant de nombreux outils facilitant le développement d'applications 3D. L'objectif du simulateur est d'afficher différentes informations pour l'utilisateur, détecter des actions particulières et proposer un monitoring non intrusif.
Quant au suivi du stage à distance, mon maître de stage et moi avons convenu de faire un point hebdomadaire. Pour ce faire, nous avons utilisé l'application Discord, offrant des salons de communications textuels, vocaux et vidéos.

En annexe de ce document se trouve des exemples, ainsi que le code ayant permis de réaliser ce travail.

1 Pour plus d'informations, le site web officiel Unity est disponible avec le lien suivant : https://unity.com/fr
2 Un support est la combinaison d'une machine et d'un OS. Par exemple un PC avec windows, linux ou macOs, smartphone ou tablette avec Android ou IOS etc...

Le concept de manipulation 3D au travers d’un outil d’interaction logique :

Quant on utilise un équipement de réalité virtuelle de type casque (HTC vive, OcculusRift), la navigation dans l’environnement virtuel se fait de manière spécifique. Le point de vue de l’utilisateur change en fonction de l’orientation de son casque, et il possède généralement deux manettes lui permettant d’interagir avec son environnement. De plus, couplé à un système Opti Track 3D, l’utilisateur à la possibilité de se déplacer dans le monde virtuel simplement en se déplaçant dans le monde réel. L’Opti Track 3D permet aussi de colocaliser des objet dans le monde réel, et il est ainsi possible de les superposer avec leur représentation dans le monde virtuel. Cela permet d’interagir avec le monde virtuel en manipulant de vrais objets, et ainsi offrir un semblant de réalisme à l’utilisateur.

Néanmoins, sans l’accès à ce matériel, il est tout de même important de pouvoir interagir avec le monde virtuel. Il est nécessaire de mettre en place des interactions similaires par l’utilisation d’autres dispositifs physiques comme le clavier et la souris.

Ainsi, pour permettre à l’utilisateur de se déplacer, un script de déplacement est mis en place. Il détecte l’activation des touches directionnelles verticales (vers le haut et vers le bas) et réalise une translation du positionnement de l’utilisateur dans le monde virtuel, en fonction du point de vue de l’utilisateur. Le déplacement se fera donc en marche avant ou en marche arrière, dans la direction du point de vue de l’utilisateur.

Pour modifier ce point de vue, le script détecte l’activation des touches directionnelles horizontales (droite et gauche), et réalise une rotation autour de l’axe vertical en fonction de la direction de la touche enclenchée. L’utilisateur a alors la possibilité de déplacer son point de vue sur la droite et sur la gauche.

Pour la manipulation d'objet, afin de proposer un système réutilisable, mon maître de stage m'a demandé de mettre en place un curseur permettant de saisir les objets. Dans le cas d'une utilisation avec un clavier et une souris, le curseur est positionné devant la caméra, et déplacé sur l’écran avec les mouvements de la souris. Sa distance par rapport à la caméra est modifiable en utilisant la molette de la souris.

Unity permet de faire de la détection d’objet 3D et d’obtenir les caractéristiques de l’objet touché. Une des caractéristiques obtenues est l’étiquette. On peut donner une étiquette (une catégorie) à un ou plusieurs GameObjects. Ainsi, lors de la détection de collision, on peut vérifier l’étiquette de l’objet touché, et dans notre cas, si il possède l’étiquette "Catchable" (attrapable). Si l'utilisateur décidait d'attraper l'objet, il pourrait cliquer sur le bouton gauche de la souris pour attraper l'objet. On change alors l’objet d’affiliation et on place l’objet détecté en tant que fils du curseur. Unity calcule alors automatiquement le changement de repère de l’objet afin qu’il reste à la même place dans le monde lors du changement d’affiliation et il se déplace maintenant avec le curseur.
Si le changement de repère n’était pas fait automatiquement, l’objet se trouverait aussi loin du curseur qu’il se trouvait loin de l’origine de son précédent repère.

Pour éviter le problème des collisions, l'objet devient fantomatique. Il passe à travers les murs tant qu'il est porté, et ne subit plus les lois de la physique. Cela permet de faciliter la manipulation d'objet.

Le curseur sera ensuite piloté par la position des manettes de réalité virtuelle, lors d'une utilisation du simulateur avec un casque de réalité virtuelle à la place de son utilisation avec un clavier et une souris.

Illustration 2: Le positionnement de la caméra et du curseur sur Unity

Sur la photo ci-dessus, le curseur est entouré en rouge, et la position de l'utilisateur est représentée par la caméra.

Actions à détecter

L’utilisateur du simulateur doit pouvoir interagir avec le modèle de la colonie d’abeilles. Par le biais de la ruche tangible, l’utilisateur va transmettre ses ordres par des actions, détectables par la simulation. En conséquence, il faut prévoir à l’avance les actions qui doivent être détectées, et mettre en place dans le programme Unity des moyens de les reconnaître.
Les actions qu’un apiculteur réalise lorsqu’il manipule la ruche et qui nous sont utiles sont les suivantes :

-retirer le toit de la ruche, et le repositionner à sa place ;
-retirer un cadre de la ruche, le repositionner dans la ruche à sa place, ou à une autre place ;
-frapper d’un coup sec le cadre, pour faire tomber les abeilles.

L’objectif est de détecter des actions et de permettre de perturber le modèle de la colonie d’abeilles et d’ainsi étudier la conséquence des actions virtuelles d’un utilisateur sur la colonie d’abeilles à court ou long terme.

Détecter le positionnement du toit de la ruche :

La position du toit de la ruche est simple à détecter. Sa position dans l'espace correspond à son centre. En ajoutant un "GameObject" positionné à l’endroit où doit être positionné le toit sur la ruche pour être correctement installé, cela permet de calculer facilement la distance entre les deux positions et ainsi savoir si le toit est à sa place. Il faut laisser une marge d'erreur bien sûr, mais une distance théorique de 0 serait la condition idéale.

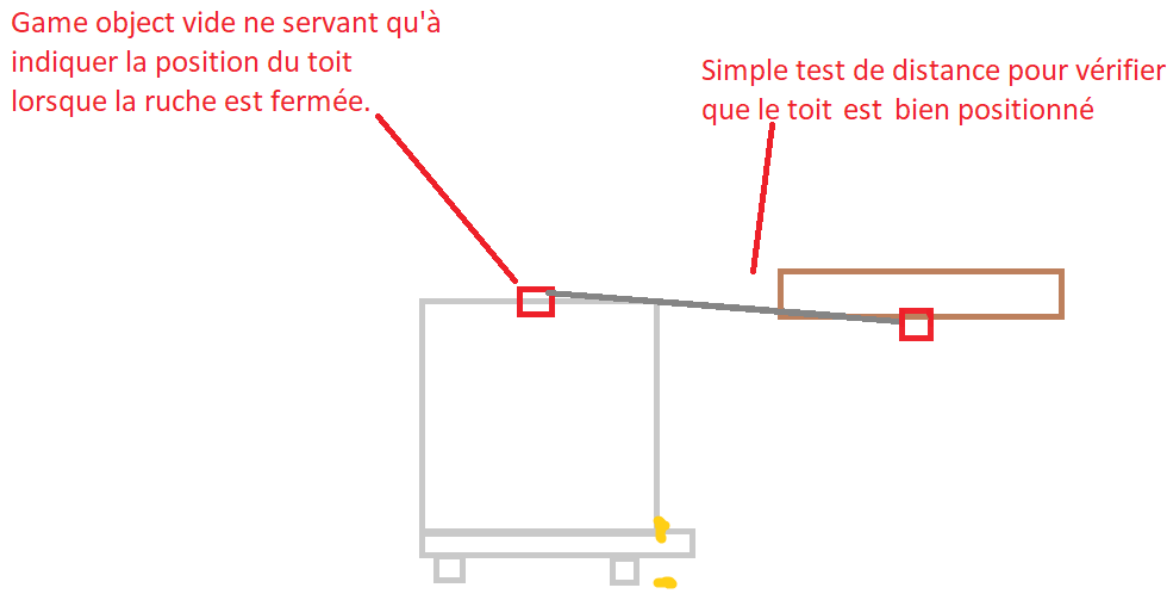


Illustration 3: Schéma montrant le fonctionnement de la détection du positionnement du toit

On pourrait penser que cette technique ne permet pas de vérifier la rotation autour de l'axe vertical, et donc avoir le toit positionné en travers de la ruche. Mais ce n'est pas le cas. Le toit est fait pour s'emboîter sur la ruche, et grâce à la physique et aux collisions gérées par Unity, la distance entre les deux positions sera correcte seulement si le toit est bien emboîté.

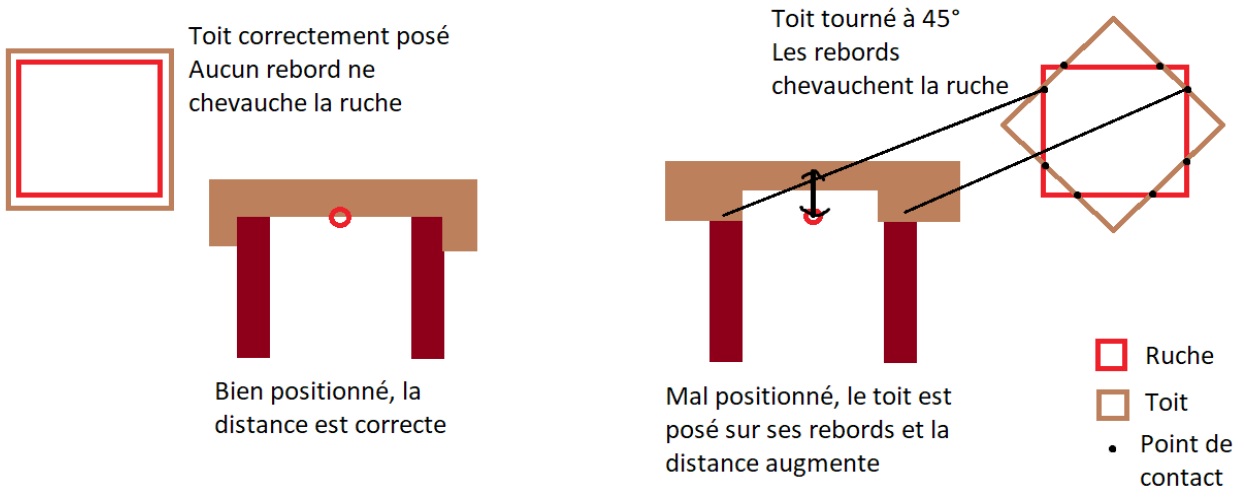


Illustration 4: Schéma montrant qu'un toit mal emboîté ne peut pas être validé par le modèle 3D.

Détecter le positionnement des cadres :

La position des cadres dans la ruche doit être détectée. Un cadre peut se trouver hors de la ruche mais aussi sur n'importe quel emplacement disponible de la ruche. Il faut donc trouver un moyen de détecter un cadre positionné dans la ruche, mais aussi de l'identifier et d'identifier son emplacement.

Je suis donc parti d'un volume de détection positionnée au fond de la ruche, que la partie basse du cadre atteint lorsqu'il est rangé. Ce volume n'est pas tangible. Il peut donc être traversé par tous les objets de l'environnement. Par contre, il prend en compte les événements quand un objet le traverse. Les cadres peuvent donc détecter les volumes de détections lorsqu'ils les traversent. Ainsi, grâce aux méthodes offertes par le script des volumes de détection et connaissant sa condition, un panneau peut être identifié grâce son numéro de panneau ainsi que sa position dans la ruche si il est rangé. Bien sur, il sait aussi si il n'est pas placé dans la ruche.

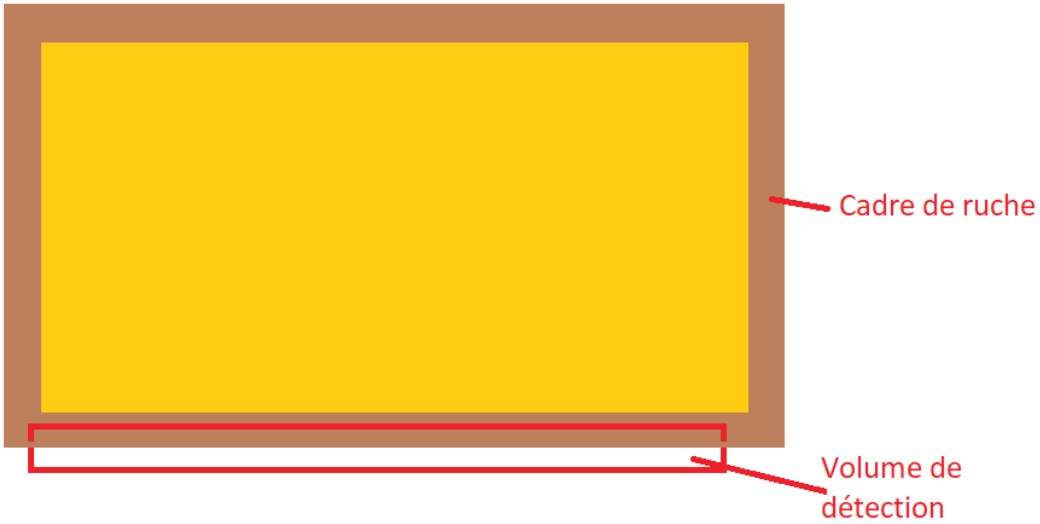


Illustration 5: Schéma du principe de détection d'un cadre par l'utilisation d'un volume de détection

Néanmoins, l'utilisation d'un seul volume de détection n'est pas suffisant. Il y a plusieurs configurations provoquant un faux positionnement d'un panneau dans la ruche.

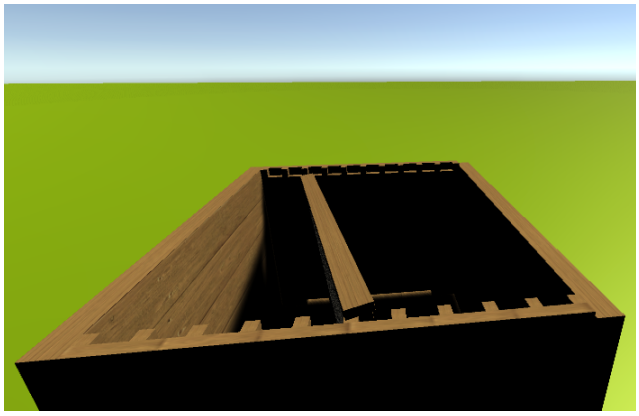


Illustration 7: Exemple de mauvais placement

Illustration 6: Exemple de mauvais placement

Pour palier ce problème, il faut s'assurer que le panneau soit bien droit dans la ruche et que ses extrémités soient bien en appui sur les rebords de rangement de la ruche. J'ai donc ajouté deux autres aires de détection par rangement de la ruche. De fait, le placement ne sera valide que si le panneau est bien posé dans un compartiment.

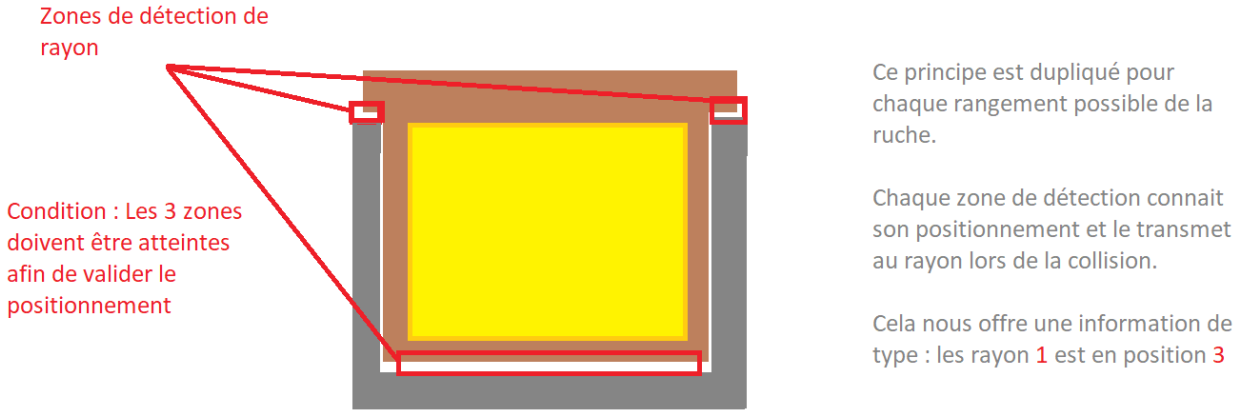


Illustration 8: Schéma décrivant le principe de détection d'un cadre

Ces 3 aires ont pour identifiant commun le numéro de rangement dans la ruche. Le cadre doit donc toucher 3 aires ayant le même identifiant. Autrement, cela signifie que le cadre est mal placé, et chevauche un autre emplacement.

Détecter si un cadre subit un choc :

Lorsque les apiculteurs retirent un cadre et veulent faire évacuer les abeilles de celui-ci, ils donnent un coup sec sur la partie haute du cadre. Ce mouvement, provoquant une perturbation de la colonie, est utile pour la simulation et doit donc être détecté.

Lors de l'analyse du mouvement, on peut constater que le cadre commence par descendre puis remonter avec la résistance du bras qui tient le cadre. On peut généraliser ce mouvement par le déplacement du cadre dans un sens, puis dans le sens opposé, sans que ce soit forcément vers le bas puis vers le haut. Les abeilles n'attendent pas ce mouvement particulier pour s'enfuir, elles subissent un choc, elles réagissent.

Mais ce n'est pas tout, il faut aussi prendre en compte la force du mouvement. Le cadre subit une forte accélération dans un sens, puis dans l'autre.

Cela peut se détecter par la détection d'une accélération positive puis négative. La sensibilité est modifiable, pour accepter une accélération plus ou moins élevée, c'est-à-dire un choc plus ou moins violent.

Affichage

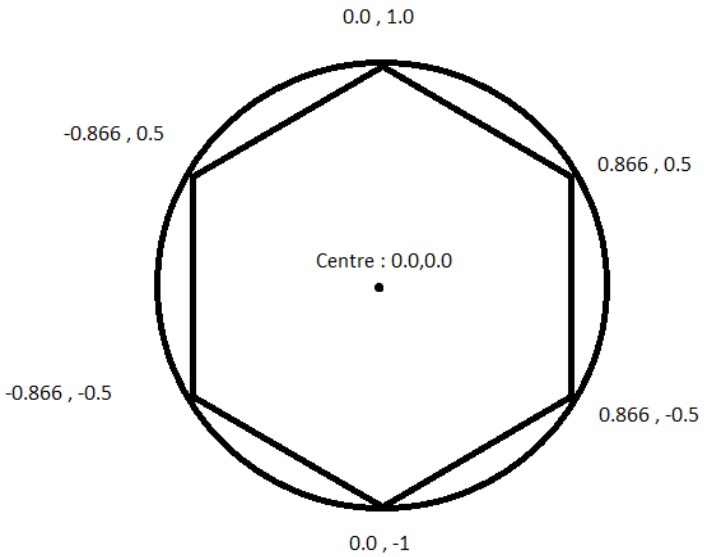
Le modèle de la colonie d'abeilles transmet de nombreuses informations à l'utilisateur. L'objectif de la simulation est d'afficher efficacement ces informations afin que l'utilisateur comprenne l'ensemble d'un simple coup d'œil. Le contenu des alvéoles est l'un des différentes information que le modèle peut envoyer.

Les alvéoles :

Lors de l'affichage des alvéoles des cadres des ruches, l'information essentielle à représenter est le contenu. Chaque alvéole doit montrer une information, et cela peut être réalisé à l'aide de couleurs, car très simple à comprendre.

La technique mise en place dans un ancien projet était de générer la texture représentant les alvéoles en coloriant chaque pixel un à un. Cette technique n'est pas réutilisable en terme de performance car il faut 400 ms pour colorier la texture. Cela correspond à 2.5 images par seconde alors qu'il faudrait au minimum 60 images par seconde pour éviter que l'utilisateur ne souffre de cinétose (mal de mer ou mal des transports). Cette cinétose est provoqué par un décalage entre les actions physiques de l'utilisateur et ce qu'il perçoit visuellement à cause de la lenteur d'actualisation de l'image.

La représentation va donc se faire à partir d'un shader, qui va représenter chaque alvéole par un hexagone coloré. La position et la couleur de chaque alvéole seront déterminées par un GameObject, qui va utiliser un nuage de points pour stocker la position des alvéoles et la valeur correspondant à leur contenu, qui sera par la suite représenté par une couleur. Pour gagner en performance, les différentes positions des sommets d'une alvéole (un hexagone) ont été préalablement calculées pour n'avoir qu'a l'exécution que des calculs simples



Utilisation des formules trigonométriques pour calculer les coordonées des points.
Pour modifier la taille de l'hexagone, il faut multiplier les coordonnées par la taille voulue divisée par 2

Illustration 9: Schéma des valeurs des sommets d'un hexagone de taille 1

Ensuite, pour placer chaque hexagone à sa position sur le cadre, on modifie le centre de l'hexagone (centre + position) et on modifie donc chaque sommet de la même façon (sommet + position).
Ces positions sont calculées préalablement dans le script gérant les données du shader, et correspondant aux différentes cases du tableau envoyé par la simulation de la colonie d'abeilles.

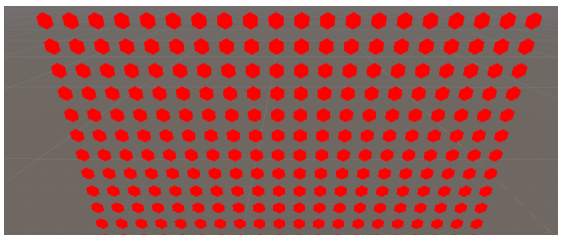


Illustration 10: Résultat de l'affichage des alvéoles sans décalage

D'ailleurs, il faut savoir que les alvéoles sont décalées d'une ligne à l'autre, donc il a fallu préparer ce décalage avant d'envoyer les positions au shader.

Pour mettre en place ce décalage, il faut déplacer chaque alvéole sur la droite d'une distance correspondant à la moitié du diamètre de l'alvéole. Mais ce déplacement n'a lieu que pour les lignes ayant un identifiant pair (ou impair). Cela nous donne la formule suivante : (Diametre/2)*(ligne modulo 2)+position.

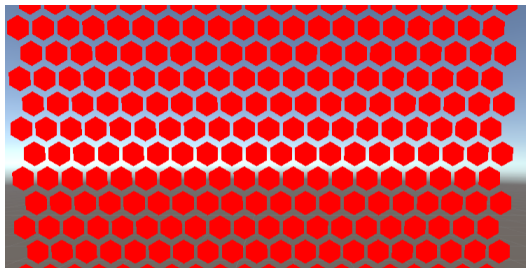


Illustration 11: Résultat de l'affichage des alvéoles avec décalage

Diametre/2 : la moitié du diamètre à ajouter pour le décalage
(ligne modulo 2) : retourne 0 ou 1, correspondant à pair ou impair.

Monitoring non intrusif

Plan de clipping (plan de coupe) :

Un plan de clipping permet de n'afficher que les formes présentes d'un coté du plan positionné dans l'espace.

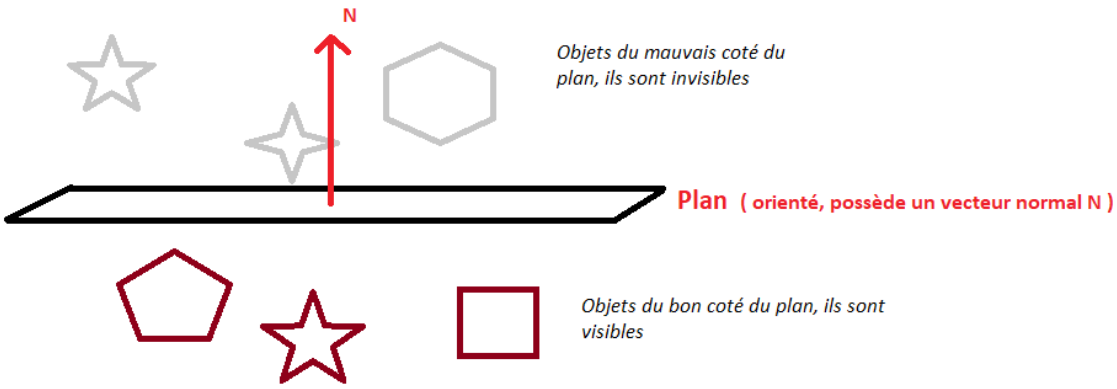


Illustration 12: Schéma du principe de fonctionnement d'un plan de clipping

Cela permet d'obtenir des visualisations qu'il serait impossible d'obtenir dans le monde réel. Par exemple, cela permet d'avoir une coupe de la ruche simulée, sans la perturber, ni réellement découper la ruche.

Il y a plusieurs façons de réaliser un shader de clipping sur Unity : soit en utilisant l'extension ShaderGraph, soit en écrivant le shader soi-même. Dans les deux cas, les fonctions/méthodes utilisées sont les mêmes, mais appelées de façon différentes. L'utilisation de l'un ou de l'autre dépend simplement des préférences du développeur : soit il préfère écrire le code, soit il préfère utiliser une interface graphique lui permettant de manipuler les méthodes sous forme de boites.

Le clipping pour les alvéoles :

Nous avons maintenant un shader pour réaliser un clipping et un shader pour créer les alvéoles. Il serait maintenant intéressant de fusionner les deux. Je n'ai pas trouvé le moyen (si il existe) de mettre deux shaders ensemble, un shader de surface et un shader géométrie. Dans le doute, j'ai réalisé un shader mélangeant les deux. Il utilise l'équation cartésienne d'un plan pour déterminer si une alvéole est du bon coté du plan ou non. Si elle ne l'est pas, l'alvéole n'est simplement pas dessinée.

Pour savoir si un point est du bon coté du plan, il faut d'abord obtenir l'équation cartésienne du plan : $ax + by + cz + d = 0$.

Pour trouver les valeurs des variables a, b, et c, rien de plus simple si l'on possède le vecteur normal du plan, c'est-à-dire le vecteur perpendiculaire au plan, orienté dans un sens (deux sens sont

possibles). Les variables a, b et c correspondent alors respectivement aux valeurs x, y et z du vecteur normal.

Ensuite, pour trouver la valeur de d, il faut connaître la position du plan dans l'espace, et calculer la formule : d=-(a*x + b*y + c*z) avec (x,y,z) correspondant à la position du plan dans l'espace.

Maintenant que nous avons l'équation, il suffit de remplacer x, y et z de l'équation par le point que nous voulons évaluer, dans notre cas, la position d'une alvéole.

En fonction du résultat, positif ou négatif, on peut choisir de dessiner l'alvéole ou non.

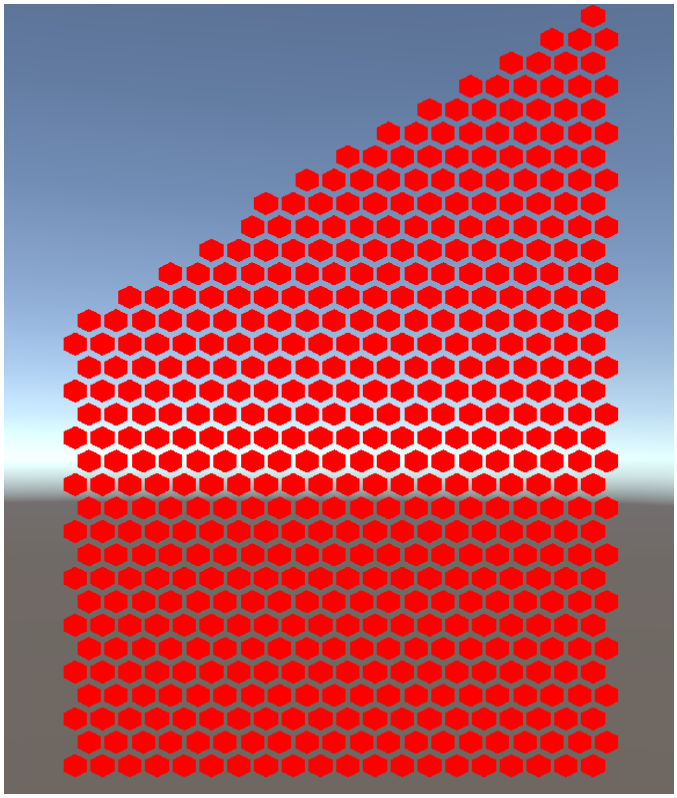


Illustration 13: Résultat de la mise en place du clipping sur le shader d'alvéoles

Conclusion

Plusieurs briques essentielles du simulateur sont maintenant en place.
Des interactions sont possibles par la mise en place d'un curseur permettant d'interagir avec le monde virtuel. De plus, il est possible de naviguer dans la simulation 3D à l'aide du clavier.
L'utilisateur de la simulation 3D peut réaliser des actions apicoles qui sont désormais détectables. Il est possible de détecter si l'utilisateur retire ou place le toit de la ruche, si il déplace un cadre ou le retire de la ruche, et si il donne un coup sur le cadre.
On peut aussi visualiser le contenu des alvéoles de chaque cadre par l'intermédiaire d'un shader de géométrie.
De plus, la mise en place d'un plan de clipping nous offre des perspectives de visualisation qui peuvent être très intéressantes et propres à la simulation 3D.

Néanmoins, il reste encore du travail.
Il faut encore rajouter des interactions, comme frapper contre le toit pour faire tomber les abeilles, ainsi que modifier la détection des cadres. Après une discussion avec un apiculteur, la manipulation des cadres qu'il nous a présenté diffère de la détection qui est actuellement en place.
De plus, en fonction des besoins du modèle SMA, d'autres interactions peuvent être demandées.
Dans tous les cas, il faudra quelques jours pour mettre en place les interactions connues.

Autrement, il faudra en juillet adapter la simulation pour pouvoir être utilisée avec un équipement de réalité virtuelle de type casque, ainsi que mettre en place la colocalisation et la superposition de la ruche réelle avec la ruche virtuelle grâce au tracking 3D.

D'un point de vue personnel, ce stage correspond parfaitement à mes attentes initiales. Je voulais valider mon choix de poursuivre vers le développement d'application en environnement 3D et découvrir Unity. J'ai pu réaliser ces deux objectifs, et je vais pouvoir continuer à progresser sur Unity durant la suite de mon stage.

Annexe

Définition :

Système multi-agent (SMA) : « est un système composé d'un ensemble d'agents (un processus, un robot, un être humain, etc ..) situés dans un certain environnement et interagissant selon certaines relations. Un agent est une entité caractérisée par le fait qu'elle est, au moins partiellement, autonome. »

https://fr.wikipedia.org/wiki/Système_multi-agents

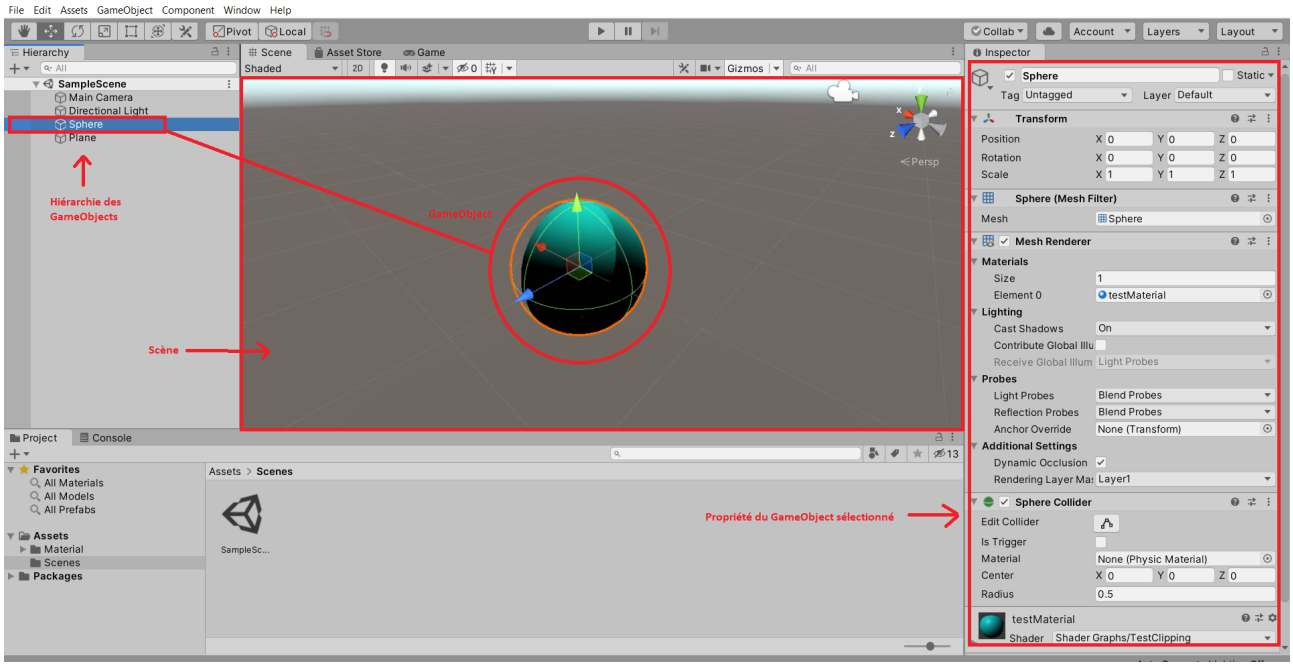
Tracking 3D : Système permettant de suivre une cible et d'y extraire l'information de mouvement sur 3 dimensions, à l'aide de capteurs, caméras ou autres dispositifs.

Shader : « C'est un programme informatique, utilisé en image de synthèse, pour paramétrer une partie du processus de rendu réalisé par une carte graphique ou un moteur de rendu logiciel. Il sert à décrire l'absorption et la diffusion de la lumière, la texture à utiliser, les réflexions et réfractions, l'ombrage, le déplacement des primitives et des effets post-traitement. »

https://fr.wikipedia.org/wiki/Shader

Interaction tangible : L'utilisateur interagit avec l'information numérique par le moyen de l'environnement physique.

Interface de l’éditeur Unity et GameObject



Exemple de script :

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

0 références
public class MouseLook : MonoBehaviour
{
    [SerializeField]
    private float mouseSensitivity = 100f;
    public Transform cursorbody;
    float xRotation = 0f;
    // Start is called before the first frame update
    0 références
    void Start()
    {
        Cursor.lockState = CursorLockMode.Locked;
    }

    // Update is called once per frame
    0 références
    void Update()
    {
        float mouseX = Input.GetAxis("Mouse X") * Time.deltaTime * mouseSensitivity;
        float mouseY = Input.GetAxis("Mouse Y") * Time.deltaTime * mouseSensitivity;

        float zDistance = 1.5f;
        cursorbody.localPosition =
            new Vector3(cursorbody.localPosition.x+mouseX, cursorbody.localPosition.y + mouseY, zDistance);
    }
}
```

La méthode Start() est appelée à l’initialisation du GameObject contenant le script.

La méthode Update() est appelée à chaque actualisation de la scène contenant les GameObjects.

Contenu du script permettant la détection d'un coup sur le cadre :

```
private Vector3 previousPosition;
private float previousSpeed;

[SerializeField]
private float sensibility=250f; //Correspond to the minimum acceleration to detect action

[SerializeField]
private float timer = 0.5f; //Correspond to high limit time of the action ( analysis time in seconds )
```

```
private void detectChock()
{
    float dist = Vector3.Distance(previousPosition, transform.position);
    float speed = dist / Time.deltaTime;
    float accele = (speed - previousSpeed) / Time.deltaTime;

    //Started action
    if (accele >= sensibility)
    {
        inAction = true;
    }

    //Second phase of the action
    if (inAction && (accele <= (-sensibility)))
    {
        Debug.Log("There was a hit");
        endedAction();
    }

    //Timer update
    if (inAction)
    {
        timer -= Time.deltaTime;
    }

    //Ended timer
    if (timer <= 0)
    {
        endedAction();
    }

    previousPosition = transform.position;
    previousSpeed = speed;
}
```

Contenu du script permettant la vérification du positionnement d'un cadre :

```
[SerializeField]
private int id;


private GameObject[] honeycombDetectionArea = { null, null, null }; //Contains last area of each type detected
private bool[] wellPositionedAt = { false, false, false };
private int[] storageValue = { -1, -1, -1 };    //Contains id of last area of each type detected
private bool tidy = false; //true if honeycomb his well stored in the hive
```

Détection de la collision avec une zone de détection :

```
private void OnTriggerEnter(Collider other)
{
    switch(other.tag)
    {
        case "HoneycombDetectionArea":
            HoneycombDetectionArea area = other.gameObject.GetComponent<HoneycombDetectionArea>();
            int position = area.getPositionInHive();

            //Stock needed informations to check later if honeycomb is well positioned
            int type = area.getDetectionAreaType();
            storageValue[type] = position;
            wellPositionedAt[type] = true;
            honeycombDetectionArea[type] = other.gameObject;

            break;

        default:
            break;
    }
}
```

Contenu du script « HoneycombDetectionArea » utilisé dans le script de la capture d'écran ci-dessus :

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

2 références
public class HoneycombDetectionArea : MonoBehaviour
{
    [SerializeField]
    private int positionInHive;


    1 référence
    enum DetectionAreaType : int
    {
        Down = 0,
        UpRight = 1,
        UpLeft = 2
    }

    [SerializeField]
    DetectionAreaType type;

    1 référence
    public int getPositionInHive()
    {
        return positionInHive;
    }

    1 référence
    public int getDetectionAreaType()
    {
        return (int)type;
    }
}
```

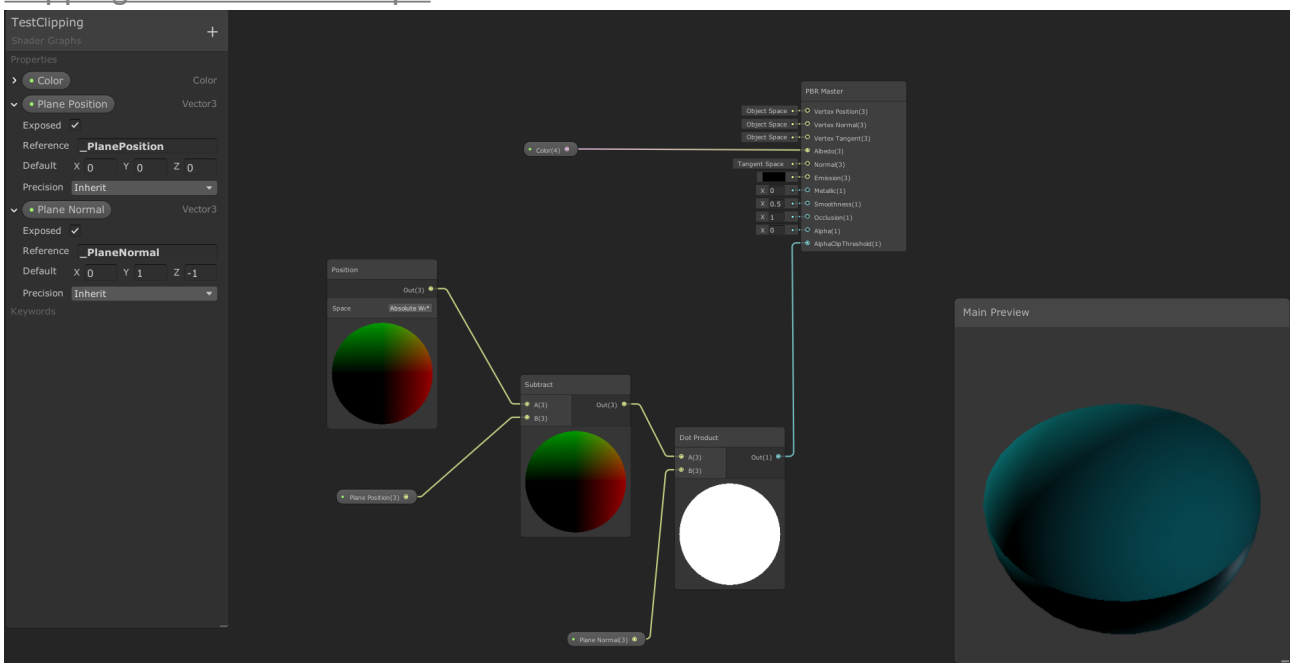
Enfin, la méthode permettant la vérification du bon positionnement du cadre dans 3 zones de détections ayant le même identifiant :

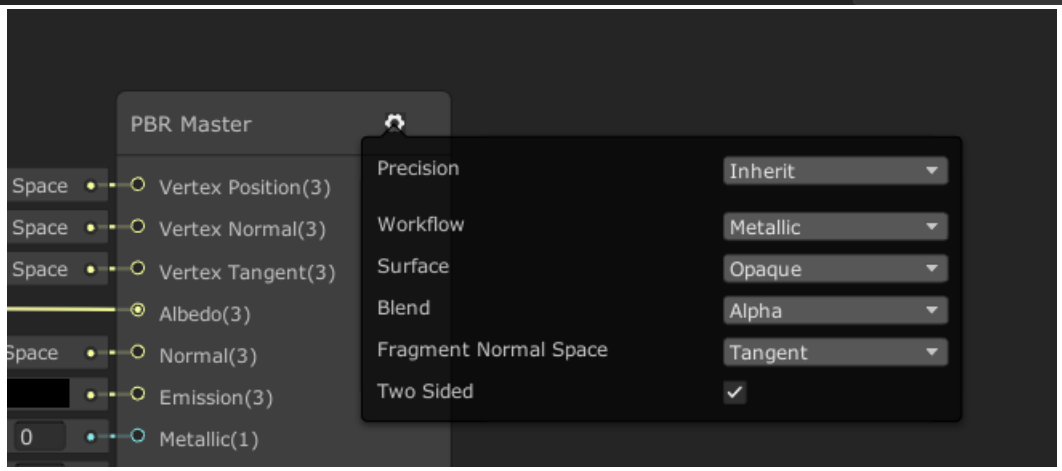
```
private void isStillTidy()
{
    bool res = true;
    for(int i=0;i<3;i++)
    {

        if (honeycombDetectionArea[i] != null)
        {
            Vector3 nearestPointOfHoneycomb = honeycombDetectionArea[i].GetComponent<Collider>().ClosestPointOnBounds(gameObject.transform.position);
            Vector3 nearestPointOfArea = gameObject.GetComponent<Collider>().ClosestPointOnBounds(nearestPointOfHoneycomb);
            float dist = Vector3.Distance(nearestPointOfHoneycomb, nearestPointOfArea);
            if (dist > 0.05f)
            {
                wellPositionedAt[i] = false;
                res = false;
            } else
            {
                wellPositionedAt[i] = true;
            }
        } else
        {
            res = false;
        }
    }
    for(int i=0;i<2;i++)
    {
        if (storageValue[i] != storageValue[i + 1]) res = false;
    }

    if(tidy && !res)
    {
        tidy = false;
        Debug.Log("I am " + id +"and I moved");
    } else if (!tidy && res)
    {
        Debug.Log("I am " + id + " and I found an area for me in position : " + storageValue[0]);
        tidy = true;
    }
}
```

Contenu du script permettant la détection du positionnement du toit par rapport à la ruche :

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

0 références
public class HiveRoof : MonoBehaviour
{
    [SerializeField]
    private Transform roofOriginalPosition;

    private bool wellPositioned = false;
    // Start is called before the first frame update
    0 références
    void Start()
    {
    }

    // Update is called once per frame
    0 références
    void Update()
    {
        float dist = Vector3.Distance(roofOriginalPosition.position, gameObject.transform.position);
        if (!wellPositioned && dist <= 0.1f)
        {
            Debug.Log("Roof is in place");
            wellPositioned = true;
        }

        if (wellPositioned && dist > 0.1f)
        {
            Debug.Log("Roof is not in place");
            wellPositioned = false;
        }
    }
}
```

Clipping avec ShaderGraph






```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class TestClipping : MonoBehaviour
{
    public Material mat;

    // Update is called once per frame

    void Update()
    {
        mat.SetVector("_PlanePosition", transform.position);
        mat.SetVector("_PlaneNormal", transform.forward);
    }
}
```