CLERMONT AUVERGNE INP Isima

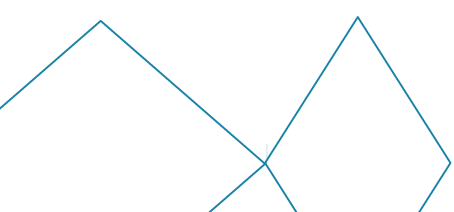
# Rapport de Projet en Réalité Virtuelle

## Titre : "Burger Lord" (GITHUB)


***Élèves :***
Jacques DUKUZEMUNGU

***Enseignants :***
Jules ROYER


22 janvier 2025



# Table des matières

# 1 Introduction

Ce projet porte sur la création d'un jeu pseudo-immersif à la 1ère personne dans lequel le joueur doit empiler des aliments suivant une recette donnée.

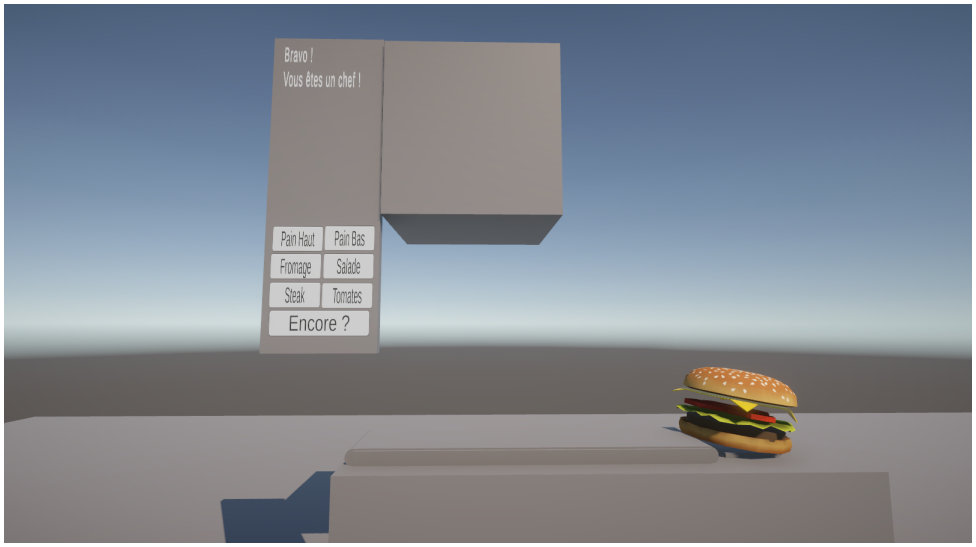



FIGURE 1 – Écran de jeu

L'intérêt de ce projet réside dans l'application concrète de notions liées au développement sur le moteur graphique Unity telles que :

- Collider
- Raycast
- Rigidbody
- Script
- Prefab
- Asset 3D
- UI

Les autres notions bonus telles que les particules et les animations n'ont pas pu être incorporées à temps.

# 2 Explication du gameplay

Premièrement, passons en revue les différentes fonctionnalités de gameplay à implémenter pour obtenir un prototype de jeu jouable.

## 2.1 Boucle principale du jeu

Dans ce jeu, le joueur incarne l'employé d'une chaîne de fast-food renommée nommée "Burger Lord" (nom choisi pour nommer le jeu). Pour espérer rester dans cette entreprise quelques jours de plus, il doit correctement préparer les burgers phares de la chaîne de restauration à l'aide d'une machine, puis les envoyer sur un convoyeur afin que ses collègues puissent les servir aux clients.

Ainsi, la boucle de gameplay principale nécessite plusieurs étapes :

- Proposer une recette au joueur.
- Permettre au joueur de sélectionner les ingrédients nécessaires à la préparation.
- Permettre au joueur de valider sa proposition de burger en l'envoyant sur le convoyeur.
- Valider ou non le burger sur l'assiette.
- En cas de validation, reproposer une recette différente ou bien, en cas d'échec, proposer une nouvelle tentative.

## 2.2 Contrôles du joueur

Burger Lord se joue à la première personne. Pour permettre une interactivité intuitive avec le jeu, deux fonctionnalités ont été implémentées : un contrôle des déplacements au clavier via les touches 'ZQSD' et un input control basé sur la sélection et la validation de bouton par la vue du joueur.

Afin de perdre le moins de temps possible sur le script de déplacement, j'ai utilisé le script template disponible à cette adresse écrit par AlucardJay.

La méthode de sélection et validation par vue est rendue possible par le script UIInteraction.cs.

**Code 1 : UIInteraction.cs C#**

```
private void Update()
{
    gazeRay = new Ray(mainCamera.transform.position,
                      mainCamera.transform.forward);
    Debug.DrawRay(gazeRay.origin, gazeRay.direction *
                      gazeDistance, Color.red);

    //-------Tâche du Graphic Raycaster---------
    PointerEventData pointerData = new PointerEventData(eventSystem)
    {
        position = Input.mousePosition
    };

    List<RaycastResult> results = new List<RaycastResult>();
    raycaster.Raycast(pointerData, results);

    foreach (var result in results)
    {
        if (result.gameObject.CompareTag("Button"))
        {
            result.gameObject.SendMessage("OnSelected");

            if (Input.GetMouseButtonDown(0))
            {
                result.gameObject.SendMessage("OnClicked");
            }
        }
    }
    //------------------------------------
}
```

En récupérant la position de la souris, le script demande au GraphicRaycaster de lancer un rayon depuis la position de la souris vers l'avant. Tous les boutons touchés par ce rayon reçoivent alors le message "OnSelected", qui active une fonction contenue dans leur script permettant de les griser. Si en plus la souris est cliquée lors de la sélection, alors le message "OnClicked" leur est aussi envoyé.

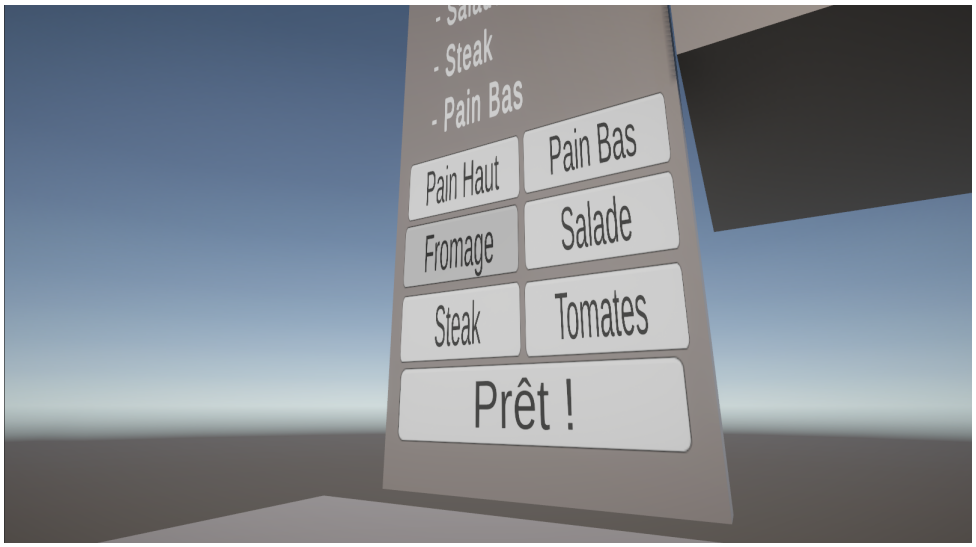



FIGURE 2 – Sélection par vue dans le jeu

# 3 Implémentation sur Unity

## 3.1 Architecture du jeu

L'architecture du projet est basée autour de deux objets nécessaires pour les communications entre objets et la logique du jeu : l'event manager et la burger machine.

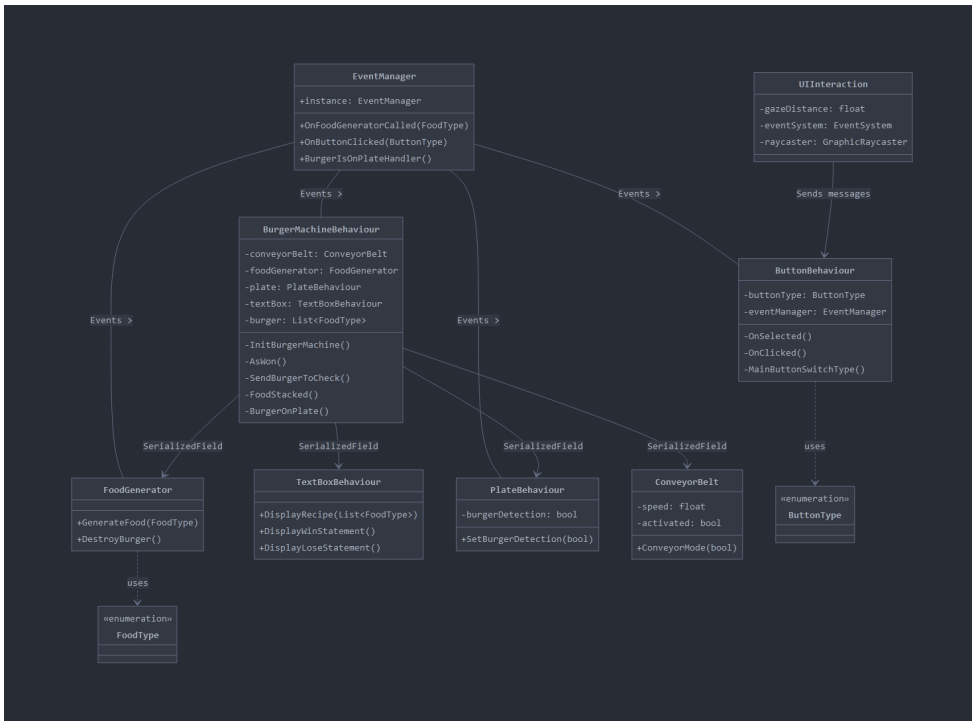



FIGURE 3 – Diagramme simplifié

Dans un premier temps, je comptais me baser entièrement sur une architecture centrée autour de l'Event Manager, avec des interactions entre classes et game objects uniquement possibles via l'abonnement à des événements, afin de rendre le code le plus propre et modulable possible. Cependant, je me suis heurté à des difficultés d'implémentation qui m'ont souvent fait choisir le passage par SerializedField comme solution de facilité pour la communication entre scripts et game objects.

## 3.2 Gestionnaire d'événements : Event Manager

L'event manager permet donc la communication entre les différentes classes et objets tout en respectant le principe d'encapsulation. Son design pattern en singleton assure son rôle de gestionnaire unique. Les différents délégués créés ainsi que les événements associés permettent de gérer correctement :

- La génération de nourriture en fonction des retours de l'UI.
- La marche à suivre lorsque le burger est prêt.
- La marche à suivre lorsque le burger est sur l'assiette.
- La marche à suivre lorsque le joueur demande à rejouer.

**Code 2 : Différents événements du script EventManager.cs C#**

```
public delegate void                FoodGeneratorCall(FoodType food);
public event FoodGeneratorCall      OnFoodGeneratorCalled;

public delegate void                BurgerReadyAction();
public event BurgerReadyAction      BurgerReadyEvent;

public delegate void                BurgerOnPlateAction();
public event BurgerOnPlateAction    BurgerOnPlateEvent;

public delegate void                PlayAgainAction();
public event PlayAgainAction        PlayAgainEvent;
```

## 3.3 Logique de jeu : Burger Machine Behaviour

Toute la logique du jeu est implémentée dans le script BurgerMachineBehaviour.cs en plus de la gestion des objets enfants : Conveyor, Plate, FoodGenerator et Text Box.

**Code 3 : Différents objet liés au script BurgerMachineBehaviour.cs C#**

```
[SerializeField] private ConveyorBelt conveyorBelt;
[SerializeField] private FoodGenerator foodGenerator;
[SerializeField] private PlateBehaviour plate;
[SerializeField] private TextBoxBehaviour textBox;
```

- Rôle de BurgerMachineBehaviour.cs : Générer les recettes à proposer et gérer l'état du jeu.
- Rôle de ConveyorBelt.cs : Translater le burger.
- Rôle de PlateBehaviour.cs : Vérifier que le burger est bien sur l'assiette.
- Rôle de FoodGeneratorBehaviour.cs : Générer des instances de prefab de nourriture pour confectionner le burger et renvoyer la liste de nourriture générée.
- Rôle de TextBoxBehaviour.cs : Afficher la recette et un message en cas de succès ou d'échec.

# 4 Conclusion

Le jeu fonctionne tel qu'il est censé fonctionner, et toutes les notions d'Unity requises sont présentes. Cependant, il ne propose que 3 recettes et gagnerait donc à pouvoir générer des recettes aléatoirement. De plus, il manque une direction artistique pour être plus divertissant ainsi qu'un système de scoring.

Merci à M. Jules Royer et M. Emmanuel Mesnard pour leurs précieux enseignements qui m'ont aidé à la réalisation de ce travail.

# 5 Annexe

**Code 4 : Script BurgerMachineBehaviour.cs C#**

```
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using static FoodType;

public class BurgerMachineBehaviour : MonoBehaviour
{
    [SerializeField] private ConveyorBelt conveyorBelt;
    [SerializeField] private FoodGenerator foodGenerator;
    [SerializeField] private PlateBehaviour plate;
    [SerializeField] private TextBoxBehaviour textBox;

    private List<FoodType> burger;
    private int currentRecipe = 0;


    private List<FoodType> recipe0 = new List<FoodType>()
    {
        DOWNBREAD,
        STEAK,
        SALAD,
        TOMATO,
        FROMAGE,
        UPBREAD,
    };

    private List<FoodType> recipe1 = new List<FoodType>()
    {
        DOWNBREAD,
        FROMAGE,
        STEAK,
        TOMATO,
        SALAD,
        UPBREAD,
    };

  ...
```

**Code 5 : Script EventManager.cs C#**

```
using System;
using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.UI;
using static ButtonType;
using static FoodType;

public class EventManager : MonoBehaviour
{
    public static EventManager                  instance { get; private set; }

    public delegate void                        FoodGeneratorCall(FoodType food);
    public event FoodGeneratorCall              OnFoodGeneratorCalled;

    public delegate void                        BurgerReadyAction();
    public event BurgerReadyAction              BurgerReadyEvent;

    public delegate void                        BurgerOnPlateAction();
    public event BurgerOnPlateAction            BurgerOnPlateEvent;

    public delegate void                        PlayAgainAction();
    public event PlayAgainAction                PlayAgainEvent;


    void Awake()
    {
        if (instance == null)
        {
            instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }

...
```

**Code 6 : Script BasicFPCC.cs par Alucard Jay Kay Github C#**

```
// ----------------------------------------
// BasicFPCC.cs
// a basic first person character controller
// with jump, crouch, run, slide
// 2020-10-04 Alucard Jay Kay
// ----------------------------------------

// source :
// https://discussions.unity.com/t/855344
// Brackeys FPS controller base :
// https://www.youtube.com/watch?v=_QajrabyTJc
// smooth mouse look :
// https://discussions.unity.com/t/710168/2
// ground check : (added isGrounded)
// https://gist.github.com/jawinn/f466b237c0cdc5f92d96
// run, crouch, slide : (added check for headroom before un-crouching)
// https://answers.unity.com/questions/374157/character-controller-slide-action-scr
// interact with rigidbodies :
// https://docs.unity3d.com/2018.4/Documentation/ScriptReference/CharacterControlle

// ** SETUP **
// Assign the BasicFPCC object to its own Layer
// Assign the Layer Mask to ignore the BasicFPCC object Layer
// CharacterController (component) : Center => X 0, Y 1, Z 0
// Main Camera (as child) : Transform : Position => X 0, Y 1.7, Z 0
// (optional GFX) Capsule primitive without collider (as child) : Transform : Posit
// alternatively :
// at the end of this script is a Menu Item function to create and auto-configure a
// GameObject -> 3D Object -> BasicFPCC

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

#if UNITY_EDITOR // only required if using the Menu Item function at the end of thi
using UnityEditor;
#endif

...
```

Code 7 : Script ConveyorBelt.cs par Jules Royer C#

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

public class ConveyorBelt : MonoBehaviour
{
    [SerializeField] private float speed = 1;
    [SerializeField] private bool activated;
    [SerializeField] private float stopDelay = 1;

    private Rigidbody rigidBody;
    private float chrono;
    private float zSize;
    private Transform baseTransform;
    private Transform boutTransform;



    void Start()
    {
        rigidBody = GetComponent<Rigidbody>();
        chrono = 0;
        zSize = 0;

        baseTransform = transform.Find("Base");
        if (baseTransform == null)
        {
            Debug.Log("ConvoyorBelt : La base n'a pas été trouvée");
        }
        else
        {
            zSize += baseTransform.localScale.z;
        }

        boutTransform = transform.Find("Bout");
        if (baseTransform == null)
        {
            Debug.Log("ConvoyorBelt : Le bout n'a pas été trouvée");
        }
    ...
```

Code 8 : Script FoodGeneratorBehaviour.cs C#

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

public class FoodGenerator : MonoBehaviour
{
    [Header("Nouritures")]
    [SerializeField] private GameObject upBread;
    [SerializeField] private GameObject downBread;
    [SerializeField] private GameObject fromage;
    [SerializeField] private GameObject salad;
    [SerializeField] private GameObject steak;
    [SerializeField] private GameObject tomato;

    [Header("Other")]
    [SerializeField] private Transform _model;

    private List<GameObject> burger;

    public void GenerateFood(FoodType foodType)
    {
        GameObject food;

        switch (foodType)
        {
            case UPBREAD:
                food = GameObject.Instantiate(upBread, _model.position,
                                              _model.rotation);
                burger.Add(food);
                break;

            case DOWNBREAD:
                food = GameObject.Instantiate(downBread, _model.position,
                                               _model.rotation);
                burger.Add(food);
                break;

            case FROMAGE:
                food = GameObject.Instantiate(fromage, _model.position,
                                              _model.rotation);
                burger.Add(food);
                break;
```

Code 9 : Script PlateBehaviour.cs C#

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

public class PlateBehaviour : MonoBehaviour
{
    private bool burgerDetection = false;

    private void OnTriggerEnter(Collider other)
    {
        if (burgerDetection)
        {
            //Debug.Log("PlateBehaviour : Un burger est sur moi");
            EventManager.instance.BurgerIsOnPlateHandler();
        }
    }

    public void SetBurgerDetection(bool _burgerDetection)
    {
        burgerDetection = _burgerDetection;
    }
}
```

Code 10 : Script TextBoxBehaviour.cs C#

```
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using static FoodType;

public class TextBoxBehaviour : MonoBehaviour
{
    private TMP_Text textBox;

    private void Start()
    {
        textBox = GetComponent<TMP_Text>();

        if ( textBox == null )
        {
            Debug.Log("TextBoxBehaviour : Je ne trouve pas le texte");
        }
    }

    public void DisplayRecipe(List<FoodType> burgerRecipe)
    {
        Clear();

        textBox.text += "Recette :\n";

        string foodWord = "";

        for (int i = burgerRecipe.Count - 1; i >= 0; i--)
        {
            switch (burgerRecipe[i])
            {
                case UPBREAD:
                    foodWord = "Pain Haut";
                    break;
                case DOWNBREAD:
                    foodWord = "Pain Bas";
                    break;
                case FROMAGE:
                    foodWord = "Fromage";
                    break;
...
```

**Code 11 : Script UIInteraction.cs C#**

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

public class UIInteraction : MonoBehaviour
{
    [SerializeField]
    private float gazeDistance = 5f;

    public EventSystem eventSystem;
    public GraphicRaycaster raycaster;
    private Camera mainCamera;
    private Ray gazeRay;

    private void Start()
    {
        mainCamera = Camera.main;
    }

    private void Update()
    {
        gazeRay = new Ray(mainCamera.transform.position,
                          mainCamera.transform.forward);
        Debug.DrawRay(gazeRay.origin,
                      gazeRay.direction * gazeDistance, Color.red);

        //-------Tâche du Graphic Raycaster---------
        PointerEventData pointerData = new PointerEventData(eventSystem)
        {
            position = Input.mousePosition
        };

        List<RaycastResult> results = new List<RaycastResult>();
        raycaster.Raycast(pointerData, results);

        foreach (var result in results)
        {
            if (result.gameObject.CompareTag("Button"))
            {
                result.gameObject.SendMessage("OnSelected");
```