

Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Інститут комп'ютерних систем
Кафедра інформаційних систем

Кебіков Дмитрій Романович,
студент групи AI-215

ДИСЦИПЛІНА
Об'єктно-орієнтоване програмування

КУРСОВА РОБОТА
Створення онлайн гри про поварів

Спеціальність:
122 Комп'ютерні науки

Освітня програма:
Комп'ютерні науки

Керівник:
Годовиченко Микола Анатолійович,
кандидат технічних наук, доцент

Одеса – 2023

ЗМІСТ

Анотація.....	3
Вступ.....	4
1 ПРОЦЕС СТВОРЕННЯ ОДИНОЧНОЇ ГРИ.....	5
1.1 Перші кроки.....	5
1.1.1 Механіки чи онлайн?	5
1.1.2 Запуск.....	5
1.1.3 Шаблон проєкта.....	5
1.1.4 Неправильний шлях.....	6
1.1.5 Перші скрипти.....	6
1.2 Імпорт моделей.....	8
1.3 Анімація при повороті.....	9
1.4 Колізії і сковзання.....	9
1.4.1 Колізія.....	9
1.4.2 Теоретичний аспект.....	10
1.4.3 Варіант вирішення проблеми.....	11
1.4.4 Слайдінг.....	12
1.5 Взаємодія і що це переді мною.....	13
1.6 Підсвічування.....	16
1.7 Піднімання об'єктів і використання інтерфейсів.....	28
1.8 Нарізка, смаження, прогрес, смітник.	33
1.8.1 Нарізання.....	36
1.8.2 Смаження.....	37
1.9 Тарілka і UI для неї.	39
2 «Netcode»	43
2.1 ServerRPC і ClientRPC в Unity.....	43
2.2 Про мою реалізацію.....	45
Висновки.....	47
Перелік використаних джерел.....	48

АНОТАЦІЯ

Курсова робота присвячена розробці онлайн гри, яка базується на концепції гри "Overcooked". Гра спрямована на кооперативний геймплей і надає розважальний досвід користувачам. Основна мета гри полягає в тому, щоб гравці виконували замовлення та готувати різні види бургерів в обмежений час. Гра передбачає наявність головного меню, можливість створення лобі з вибором публічного або приватного доступу та можливістю підключення гравців. Після того, як всі гравці підтвердять готовність, гра переходить до основної сцени, де гравці в кухні намагаються виконати якнайбільше замовлень шляхом виконання певних кулінарних операцій.

В ході розробки гри буде використана програма Unity, а мова програмування C#. Для досягнення необхідного функціоналу використовуватимуться такі бібліотеки, як System, UI, 2DSprites, Unity Network, Unity Lobby та Relay. Також для створення всіх 3д моделей я використовував програму Blender.

ABSTRACT

The coursework is dedicated to the development of an online game based on the concept of the "Overcooked" game. The game focuses on cooperative gameplay and aims to provide an entertaining experience for the players. The main objective of the game is for players to fulfill orders and prepare various types of burgers within a limited time. The game includes a main menu, the ability to create lobbies with options for public or private access, and the ability to connect players. Once all players confirm their readiness, the game transitions to the main scene where players in the kitchen strive to fulfill as many orders as possible by performing specific culinary operations.

The development of the game will be carried out using the Unity engine, with the programming language C#. To achieve the desired functionality, libraries such as System, UI, 2DSprites, Unity Network, Unity Lobby, and Relay will be utilized. I also used Blender to create all 3D models.

ВСТУП

У сучасному світі, комп'ютерні ігри стали невід'ємною частиною нашого розважального дозвілля. Існує безліч різноманітних ігрових концепцій, серед яких "Overcooked" є однією з найяскравіших кооперативних ігор.

З урахуванням цього, було вирішено розробити комп'ютерну гру, яка буде мати схожість з "Overcooked". Головна мета гри полягає в тому, щоб надати гравцям забавний та співпрацюючий геймплей. Гравці повинні будуть виконувати замовлення та готувати різні види бургерів протягом обмеженого часу.

Метою даної курсової роботи є створення комп'ютерної гри. Вона повинна надати користувачам розважальний досвід і сприяти спільній грі та взаємодії гравців. Для досягнення цієї мети будуть проведені наступні завдання:

- Вивчення концепції схожих кооперативних ігор та аналіз основних особливостей.
- Розробка головного меню та інтерфейсу гри.
- Створення 3д моделей для гри.
- Реалізація механік гри, включаючи приготування бургерів, керування персонажами та створення різних User Interfaces(активні замовлення, пауза, початок гри тощо).
- Встановлення мережевого з'єднання для спільної гри кількох гравців.
- Тестування та виправлення помилок.

Виконання цих завдань дозволить створити захоплюючу гру з кулінарними викликами та можливістю спільної гри з друзями.

1 ПРОЦЕС СТВОРЕННЯ ОДИНОЧНОЇ ГРИ

1.1 Перші кроки

1.1.1 Механіки чи онлайн?

Перш за все, необхідно вирішити, на що будемо зосереджуватися в першу чергу - на реалізації механік гри чи на створенні онлайн режиму. Оскільки є деякі важливі аспекти, які потрібно зрозуміти та реалізувати до створення онлайн режиму, я обираю пріоритет у розробці механік гри. Причина полягає в тому, що спочатку я хочу створити всі анімації, предмети та забезпечити їх взаємодію з гравцями. Також, не вважаю необхідним наразі зосереджуватися на онлайн режимі, оскільки це можна реалізувати пізніше. Деякі розробники також застосовують такий підхід, який мені підходить.

1.1.2 Запуск

Для розробки гри використовуватимуться програми Unity та Visual Studio. Завантажена остання повна версія Unity та відповідна версія Visual Studio.

1.1.3 Шаблон проєкта

Першим кроком після запуску Unity є вибір шаблону проєкту. За планом гра повинна бути в тривимірному форматі, і для цього можна використовувати стандартний тривимірний шаблон. Однак, також доступна версія зі шляхом "The URP" (Universal Render Pipeline)(рис. 1.1), яка має певні переваги. Вона придатна для будь-якої платформи, включаючи ПК, телефони та консолі. Я обрав цей шаблон, оскільки мені цікаво, як він працює, і сподіваюся, що це не призведе до непередбачуваних проблем.



Рис 1.1 The URP

ху, з, тоді як Vector2 використовує лише ху. Для вирішення цієї проблеми, створюється Vector3, який приймає значення з Vector2, де $z = 0$.



```
private void Update()
{
    Vector2 inputVector = new Vector2(0, 0);

    if (Input.GetKey(KeyCode.W))
    {
        inputVector.y = +1;
    }

    if (Input.GetKey(KeyCode.S))
    {
        inputVector.y = -1;
    }

    if (Input.GetKey(KeyCode.D))
    {
        inputVector.x = +1;
    }

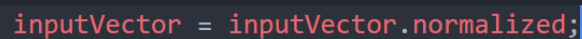
    if (Input.GetKey(KeyCode.A))
    {
        inputVector.x = -1;
    }

    transform.position += inputVector;
    Vector3 moveVector = new Vector3(inputVector.x, 0f, inputVector.y);
    transform.position += moveVector;
}
```

Рис.1.4 Простий скрипт руху граця. Правий скріпшот замінює підкреслену строчку.

Також зустрілися дві поширені помилки:

- Якщо гравець рухається по діагоналі, його швидкість збільшується. Цю проблему легко вирішити, нормалізуючи вектор руху(рис. 1.5).



```
inputVector = inputVector.normalized;
```

Рис.1.5 Нормалізація вектора.

- У Unity швидкість руху залежить від кількості кадрів в секунду (fps). Це означає, що якщо різні гравці грають з різною кількістю кадрів в секунду (наприклад, 60 fps та 120 fps), то швидкість руху буде різною. Цю проблему можна вирішити, використовуючи множник Time.deltaTime, який розраховує час, необхідний для завантаження одного кадру. Це забезпечує стабільну швидкість руху незалежно від кількості кадрів в секунду. До: <https://youtu.be/na0REJvb7cg> .Я тут додаю множник і роблю можливість настройки moveSpeed: <https://youtu.be/eJ4D3KDZrSA> .

Після внесення змін у код, гравець може рухатись у всіх напрямках (wasd)(рис.1.6).

```
[SerializeField] private float moveSpeed = 11f;
0 references
private void Update()
{
    Vector2 inputVector = new Vector2(0, 0);

    if (Input.GetKey(KeyCode.W))
    {
        inputVector.y = +1;
    }

    if (Input.GetKey(KeyCode.S))
    {
        inputVector.y = -1;
    }

    if (Input.GetKey(KeyCode.D))
    {
        inputVector.x = +1;
    }

    if (Input.GetKey(KeyCode.A))
    {
        inputVector.x = -1;
    }

    inputVector = inputVector.normalized;

    Vector3 moveVector = new Vector3(inputVector.x, 0f, inputVector.y);
    transform.position += moveVector * Time.deltaTime * moveSpeed;
}
```

Рис.1.6 Кінцевий код цього пункту

1.2 Імпорт моделей

Важливо імпортувати моделі графічних об'єктів у розробкове середовище Unity. Для цього я використовую формат .fbx, який є поширеним стандартом для обміну та імпортування 3D-моделей. В інтерфейсі Unity, я додаю ці моделі разом з відповідними текстурами, забезпечуючи візуальну якість ігрових об'єктів. І це ігрова модель гравця(рис 1.7):



Рис. 1.7 Гравець ♥

1.3 Анімація при повороті

У даному розділі розглянемо реалізацію повороту гравця в напрямку його руху з міні анімацією повороту. Замість того, щоб гравець різко змінював напрямок за один кадр, ми забезпечимо плавний перехід між початковим положенням і новим напрямком руху.

Використовуючи алгоритм Lerp(рис.1.8) (лінійна інтерполяція), здійснюється поступовий поворот гравця в бажаний напрямок. Для цього використовуються два вектори: початкове положення гравця (transform.forward - вектор a) і напрямок руху (moveDirection - вектор b).

```
transform.forward = Vector3.Lerp(transform.forward, moveVector, Time.deltaTime * 15);
```

Рис.1.8 Lerp - реалізація у коді

Процес розрахунку повороту відбувається на основі значення типу float (Time.deltaTime)(рис.1.9), яке вказує, скільки часу потрібно витратити для досягнення бажаного повороту.

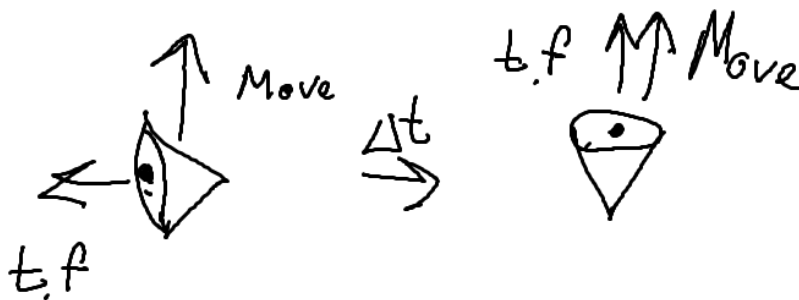


Рис.1.9 Lerp – гравець повертається з часом

1.4 Колізії і сковзання

Перед розглядом проблеми з колізією, я виконав попередню підготовку, що включала налаштування камери та внесення невеликих змін до кольорів (кольори=нейтральні, контраст++, ність++), що покращило вигляд гри.

1.4.1 Колізія

Проблема виникла після написання наступного коду:

```
RaycastHit hit;
if (Physics.Raycast(transform.position, moveVector, out hit, moveVector.magnitude
    * Time.deltaTime * moveSpeed))
{
    // Якщо є колізія, не рухатись
    return;
}
```

Однак, цей підхід був невдалим. Коли гравець зіткнувся з колізією (після додавання до куба компонента Box Collider), він застрягав. Ввід користувача ігнорувався, якщо гравець намагався рухатись у напрямку колізії. Крім того, гравець застрягав наполовину в кубі через те, що колайдер гравця був точкою в центрі об'єкта.

У підсумку, щоб вирішити цю проблему, потрібно переробити або знайти оптимальний спосіб.

1.4.2 Теоретичний аспект

Одним з потенційних рішень для цієї проблеми є використання функції Raycast. Raycast - це функція для виявлення зіткнень між об'єктами в тривимірному просторі. Вона дозволяє визначити, чи перетинається промінь з певної початкової точки у заданому напрямку з іншими об'єктами, які можуть мати колайдери.

Ця функція приймає три аргументи:

- origin - початкова точка, з якої починається промінь. Це може бути точка в просторі, де знаходиться об'єкт або його частина (у нашому випадку - положення гравця transform.position).
- direction - напрямок променя. Це може бути вектор, який показує, куди спрямований промінь (напрямок руху гравця moveVector).
- distance - відстань, яку промінь буде пройтись в заданому напрямку (ширина гравця і товщина стегна).

Якщо промінь зіткнеться з яким-небудь об'єктом, функція поверне true, в іншому випадку - false.

Після вивчення інформації з Reddit та документації Unity, стало зрозуміло, що одна з проблем полягає у використанні Raycast. На рисунку(рис.1.10) було показано, що гравець, здається, "прогризає" куб. Рейкаст фактично сприймає нижню частину об'єкта як точку, тому зображення(рис 1.11) нижче є нормальним,

оскільки червона крапка не зіткнулась з колізією і все ж залишилася на моделі. Хоча ми можемо визначити відстань, все одно це неідеальне рішення. Raycast не має поняття висоти, і ми можемо побачити візуалізацію на наступному малюнку(рис 1.12).



Рис. 1.10 «Гриз» куб

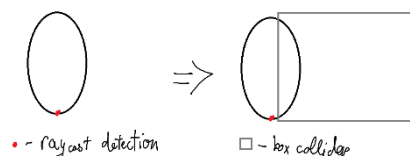


Рис. 1.11 Raycast крапка

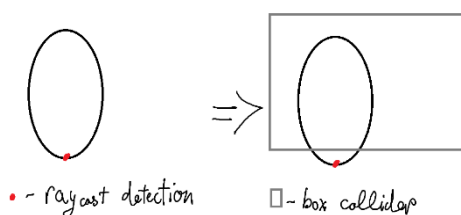


Рис. 1.12 Raycast не чіпає колізію коробки

1.4.3 Варіант вирішення проблем

У даному випадку, нам може допомогти використання методу CapsuleCast замість Box або Sphere cast, оскільки гравцем виступає капсула, і наш персонаж має схожість з циліндром.

Загальна ідея полягає в тому, що CapsuleCast повертає значення типу BOOLEAN. Якщо немає зіткнень, рух є можливим (movePossible = true). Якщо є зіткнення, рух неможливий (movePossible = false) (рис 1.13).

```
if(movePossible)
```

```
{//можна рухатись змінюючи transform.}
```

```
if (!movePossible || // але гравець натискає клавіші, які рухають його по діагоналі)
```

```
{//то слайдини по кубу}
```

```
bool movePossible = !Physics.CapsuleCast(transform.position, transform.position + Vector3.up * playerHeight,
playerWidth, moveVector, Time.deltaTime * moveSpeed);
```

Рис. 1.13 Raycast не чіпає колізію коробки

У даному випадку, параметр `transform.position` визначає позицію початку капсули, а `transform.position + Vector3.up * playerHeight` - кінцеву позицію.

`Vector3.up` - вектор, що показує напрям вгору від глобальної координатної системи.

`playerHeight` - це висота гравця.

`playerWidth` - це ширина капсули.

`moveVector` - вектор, який вказує напрямок руху гравця.

Якщо метод `CapsuleCast` повертає значення `true` – немає колізій з іншими об'єктами, тобто рух можливий (`movePossible = true`). Якщо `false`, то наявність колізій (`movePossible = false`).

Відео результат: <https://www.youtube.com/watch?v=-EV0qRQEXG8>

1.4.4 Слайдінг

Тут ситуація наступна: коли ми не можемо рухатись напрямку через перешкоду, ми перевіряємо можливість руху вбік (ліворуч-праворуч). Якщо зіткнень не виявлено, ми можемо рухатись вбік.

Або ми перевіряємо можливість руху вгору-вниз. Якщо зіткнень не виявлено, ми можемо рухатись.

Таким чином, ми використовуємо цей код(рис.1.14) для руху з урахуванням колізій.

```
if (!movePossible)
{
    //axis x after stuck
    Vector3 moveAxisX = new Vector3(inputVector.x, 0f, 0f);

    movePossible = !Physics.CapsuleCast(transform.position, transform.position + Vector3.up * playerHeight,
                                        playerWidth, moveAxisX, Time.deltaTime * moveSpeed);

    //if yes we will slide x axis
    if (movePossible)
    {
        moveVector = moveAxisX;
    }
    //axis z after stuck
    else
    {
        Vector3 moveAxisZ = new Vector3(0f, 0f, inputVector.y);

        movePossible = !Physics.CapsuleCast(transform.position, transform.position + Vector3.up * playerHeight,
                                            playerWidth, moveAxisZ, Time.deltaTime * moveSpeed);

        //if yes we will slide z axis
        if (movePossible)
        {
            moveVector = moveAxisZ;
        }
    }
}

//if we stuck here our moveVector changed in moveAxis(x/z)
if (movePossible)
{
    transform.position += moveVector * Time.deltaTime * moveSpeed;
}
```

Рис. 1.14 Код слайдінгу без нормалізацій

Пояснення, якщо ми зіткнулися з колізією (`movePossible = false`), ми створюємо новий вектор3, який відповідає за рух по відповідній осі (x або z). Потім ми замінюємо змінну `moveVector` змінною `moveAxisX` (або `moveAxisZ`) у нашій логіці. Потім ми знову перевіряємо можливість руху, і якщо вона зберігається, ми застосовуємо слайдінг по цій осі. Код для руху по осі X і Z є практично ідентичним, за винятком назв векторів та положення `inputVector.x` або `inputVector.z`.

Результати можна переглянути за посиланням: <https://www.youtube.com/watch?v=TS8axpqKRHc>.

Також, як можна побачити, гравець все ще рухається з різною швидкістю по діагоналі. Цю проблему можна вирішити, нормалізуючи вектори `moveAxisX` та `moveAxisZ`. Це можна зробити, додаючи наступний код:

```
moveAxisX = moveAxisX.normalized;
moveAxisZ = moveAxisZ.normalized;
```

Фінальний код для пункту 1.4 (рис.1.15):

```
private void Movement()
{
    Vector2 inputVector = new Vector2(0, 0);

    if (Input.GetKey(KeyCode.W)) --
    if (Input.GetKey(KeyCode.S)) --
    if (Input.GetKey(KeyCode.D)) --
    if (Input.GetKey(KeyCode.A)) --
    inputVector = inputVector.normalized;

    Vector3 moveVector = new Vector3(inputVector.x, 0f, inputVector.y);

    float playerHeight = .7f;
    float playerWidth = .3f;

    bool movePossible = !Physics.CapsuleCast(transform.position, transform.position + Vector3.up * playerHeight,
        playerWidth, moveVector, Time.deltaTime * moveSpeed);

    if (!movePossible)
    {
        //axis x after stuck
        Vector3 moveAxisX = new Vector3(inputVector.x, 0f, 0f);
        moveAxisX = moveAxisX.normalized;

        movePossible = !Physics.CapsuleCast(transform.position, transform.position + Vector3.up * playerHeight,
            playerWidth, moveAxisX, Time.deltaTime * moveSpeed);

        //if yes we will slide x axis
        if (movePossible)
        {
            moveVector = moveAxisX;
        }
    }

    //axis z after stuck
    else
    {
        Vector3 moveAxisZ = new Vector3(0f, 0f, inputVector.y);
        moveAxisZ = moveAxisZ.normalized;

        movePossible = !Physics.CapsuleCast(transform.position, transform.position + Vector3.up * playerHeight,
            playerWidth, moveAxisZ, Time.deltaTime * moveSpeed);

        //if yes we will slide z axis
        if (movePossible)
        {
            moveVector = moveAxisZ;
        }
    }

    //if we stuck here our moveVector changed in moveAxis(x/z)
    if (movePossible)
    {
        transform.position += moveVector * Time.deltaTime * moveSpeed;
    }

    transform.forward = Vector3.Lerp(transform.forward, moveVector, Time.deltaTime * 15); // число відповідає за призначення повороту
}
```

Рис.1.15 Вигляд коду

1.5 Взаємодія і що це переді мною

Перед гравцем розташована сирboard, з якою він може взаємодіяти. Якщо гравець зупиняється і перестає рухатись, то зберігається останній об'єкт, з яким можлива взаємодія. Для цього ми заносимо цей об'єкт в змінну.

Проходячи до реалізації в Unity, ми стикаємося з проблемою, що Unity відображає лише передню частину "Faces" моделі(рис.1.16). Тому перед тим, як

імпортувати модель з Blender, потрібно перевірити орієнтацію всіх "Faces" так, щоб вони правильно відображались, наприклад, забезпечуючи, щоб всі вони мали синю колірну маркування.

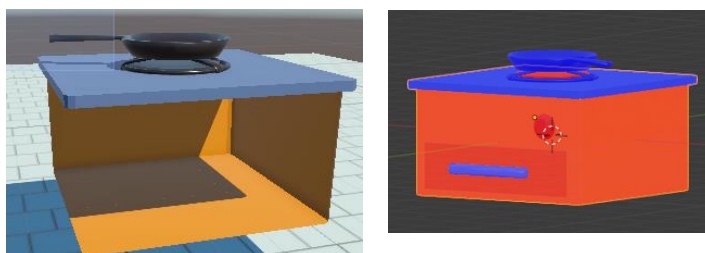


Рис.1.16 Як виглядає в юніті і як виглядають в блендері номалі.

Знову ми повертаємося до складного процесу Raycast. Використовуючи метод `Physics.Raycast`, ми перевіряємо наявність перешкод у напрямку, в якому гравець намагається взаємодіяти. Метод приймає наступні аргументи(рис.1.17):

Початкова позиція перевірки (`transform.position`).

Вектор напрямку перевірки (`moveVector`).

Довжина луча перевірки (`interactionDistance` - 1.5).

Параметр `raycastHit`, який зберігає результати перевірки.

```
float interactionDistance = 1.5f;
Physics.Raycast(transform.position, moveVector, out RaycastHit raycastHit, interactionDistance);
```

Рис.1.17 Physics.Raycast.

Змінна `raycastHit` містить наступну інформацію: точку зіткнення, вектор нормалі у точці зіткнення, відстань до перешкоди та колайдер, з яким стикається луч. Ми використовуємо цю інформацію для визначення типу перешкоди, наприклад, різних типів `surboard` або інших об'єктів. В майбутньому можуть бути додані перешкоди між персонажами, але поки що ця частина коду ще не розглядається.

Далі ми переходимо до розгортання практики(рис.1.18). Оскільки нам потрібні значення `moveVector` і `inputVector` з функції `Movement()` в функції `Interact()`, ми зробили їх глобальними змінними в цьому класі. Це може виглядати

не дуже елегантно, оскільки вимагає копіювання коду, але наразі це єдиний спосіб отримати доступ до цих значень. Значення `moveVector` і `inputVector` передаються з `Movement()` і тепер можуть бути використані у функції `Interact()`.

```
0 references
public class NewBehaviourScript : MonoBehaviour
{
    4 references
    [SerializeField] private float moveSpeed = 13f;
    11 references
    private Vector2 inputVector = new Vector2(0, 0);
    7 references
    private Vector3 moveVector = Vector3.zero;

    0 references
    private void Update()
    {
        Movement();
        Interact();
    }

    1 reference
    private void Interact()
    {
        float interactionDistance = 1.5f;
        if (Physics.Raycast(transform.position, moveVector, out RaycastHit raycastHit, interactionDistance))
        {
            Debug.Log(raycastHit.transform);
        }

        if (Input.GetKeyDown(KeyCode.E))
        {
            Debug.Log("Aramusa");
        }
    }

    private void Movement()
    {
        inputVector = new Vector2(0, 0);

        if (Input.GetKey(KeyCode.W))
        {
            inputVector.y = +1;
        }
        if (Input.GetKey(KeyCode.S))
        {
            inputVector.y = -1;
        }
        if (Input.GetKey(KeyCode.D))
        {
            inputVector.x = +1;
        }
        if (Input.GetKey(KeyCode.A))
        {
            inputVector.x = -1;
        }

        inputVector = inputVector.normalized;

        moveVector = new Vector3(inputVector.x, 0f, inputVector.y);
        для логіки діагональних слایدів
    }
}
```

Рис.1.18 Показ коду для цього пункту

Однак, якщо ми хочемо зробити ці вектори унікальними для функцій, можуть виникнути проблеми з багами.

Фікс багу включає глобальну змінну `lastMove` (рис.1.19), яка зберігає останній напрямок руху. Це дозволяє гравцю "бачити" об'єкт перед собою, коли гравець не рухається.

Далі слід створити скрипт для самої surfboard. Це простий скрипт, який викликається гравцем, тому його оголошено як публічний.

```

float interactionDistance = 1.5f;
if (moveVector != Vector3.zero)
{
    lastMove = moveVector;
}

if (Physics.Raycast(transform.position, lastMove, out RaycastHit raycastHit, interactionDistance))
{
    Debug.Log(raycastHit.transform);
}

```

Рис.1.19 LastMove code.

Нижче наведений код зі скрипта гравця(рис.1.20). В ньому ми перевіряємо, чи взаємодіємо з компонентом Cupboard, і якщо так, то викликаємо його функцію.

Таким чином, ми продовжуємо вдосконалювати логіку гри та взаємодію гравця з об'єктами у середовищі.

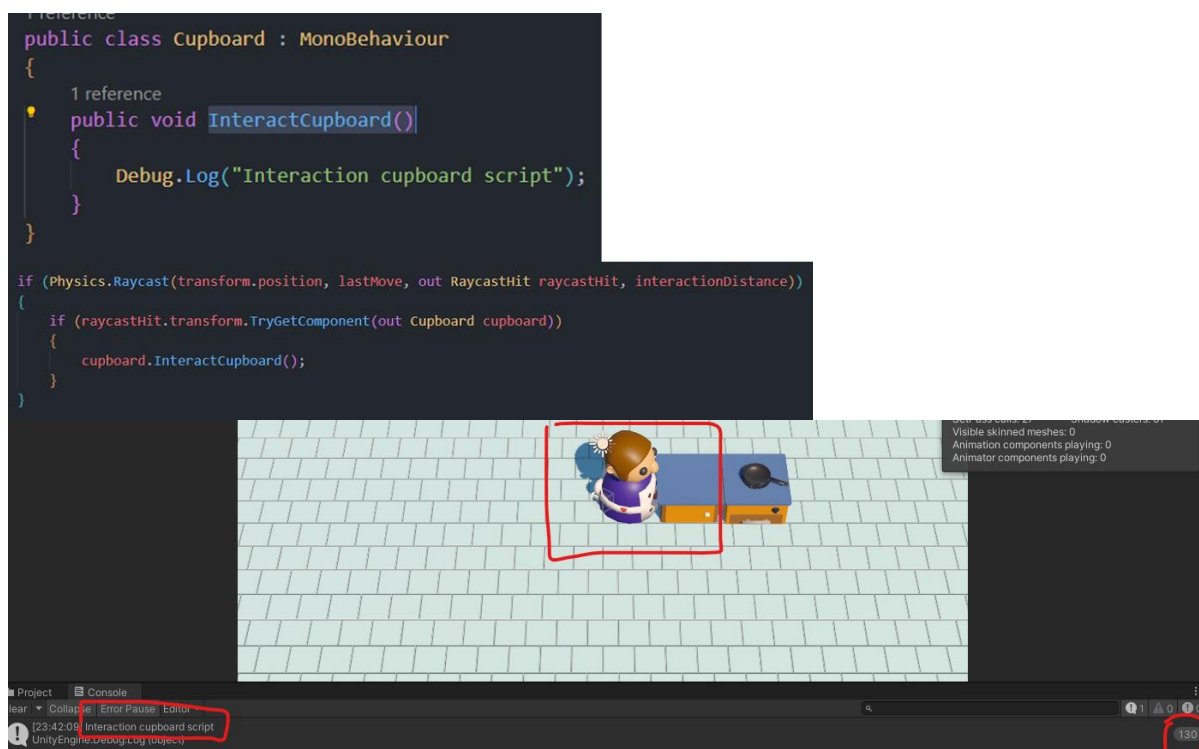


Рис.1.20 Демонстрація того, що гравець бачить перед собою cupboard

1.6 Підсвічування

Ми вже успішно реалізували функціонал, щоб наш плеєр міг бачити та розрізняти різні типи Cupboard. Тепер ми переходимо до додавання візуальної складової, щоб сам гравець також міг бачити, куди дивиться плеєр.

Для цього ми створюємо папку з префабами і додаємо «нову» модель(рис 1.21) для звичайного Cupboard. Ця модель трохи більша за звичайний Cupboard і

має матеріал з білою альфа-мапою. Це дає ефект виділення, що об'єкт обраний або виділяється. Таким чином, ми створюємо візуальний ефект, що показує, куди спрямований погляд плеєра.

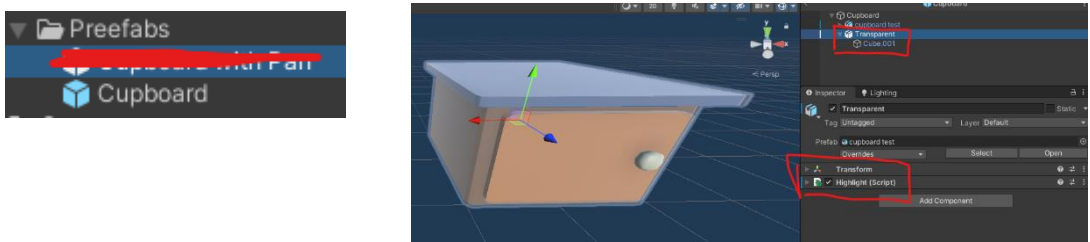


Рис.1.21 Префаб і «нова» модель

Після створення моделі з білою альфа-мапою, ми прикріплюємо до неї новий скрипт, в якому розписуємо функціонал.

У цій розділі ми провели зміни в самому коді і його переробці, що зайняло кілька днів. Враховуючи це, я вирішив, що в кінці цього розділу я повторно розпишу весь код та його значення, не вдаючись до опису процесу, як це було раніше. Результат розділу можна переглянути за посиланням: <https://youtu.be/IIciv9hFvKU>.

Розберемо весь код у виді таблиць:

Код	Мій коментар/пояснення
<pre>using System; using System.Collections; using System.Collections.Generic; using UnityEngine;</pre>	<p>Наш початок роботи полягає у простому копіюванні кольорової палітри з Visual Studio до нашої таблиці. Це не тільки зробить її більш привабливою, але й полегшить мені розуміння.</p> <p>Тут ми включаємо простір імен System, який містить основні класи C#, такі як EventHandler та EventArgs. Вони використовуються нами для реалізації підсвічування в даний</p>

	момент. Але потім будуть десятки Events.
<pre> public class Player : MonoBehaviour { //highlight stuff public static Player Instance { get; private set; } </pre>	<pre> public static Player Instance { get; private set; } </pre> <p>Це статична властивість, яка надає доступ до єдиного екземпляру класу Player. Оскільки я створюю онлайн-гру, це є тимчасовим рішенням, і його необхідно буде змінити в подальшому. Проте наразі воно працює для одного гравця. Важливо зазначити, що SETTER є приватним, тобто його можна встановити лише в цьому розділі. Але ми зможемо отримувати дані з іншого скрипта, який відповідає за підсвічування.</p>
<pre> private void Awake() { if (Instance != null) { Debug.Log("Instance != null. Error"); } Instance = this; } </pre>	<p>Якщо інший екземпляр вже існує, це означає, що сталася помилка. Записуємо цю помилку до консолі(але вона виникне тільки при імплементуванні мултіплеєра).</p> <p>Якщо ж інший екземпляр не існує, поточний екземпляр встановлюється як єдиний екземпляр, використовуючи статичне поле Instance. Це дозволяє будь-якому іншому об'єкту в програмі отримати</p>

	<p>доступ до цього єдиного екземпляру через це поле.</p> <p>Важливо розуміти, що без встановлення екземпляру програма викликатиме помилку "object reference not set to an instance of an object" (об'єкт посилення не вказує на екземпляр об'єкта).</p>
<pre>public EventHandler<OnHighlightCupboardEventArgs> OnHighlightCupboard;</pre>	<p>Використовуємо подію - це механізм, який дозволяє класу повідомляти про зміни інформації, що відбуваються в ньому, і забезпечує зв'язок між об'єктами. Ми також використовуємо подію в скрипті гайлайту, але про це поговоримо пізніше, після розгляду скрипта Player.</p>
<pre>public class OnHighlightCupboardEventArgs : EventArgs { public ClearCupboard selectedCupboard; } //end highlight stuff part 1</pre>	<p>Ця частина відповідає за вибірку cupboard для підсвічування та виконання необхідних дій з ними.</p> <p>Примітка: Ми використовуємо клас, оскільки без нього не можемо передати інформацію в інший скрипт. Клас OnHighlightCupboardEventArgs визначає тип аргументу, який буде переданий разом з подією. У даному випадку, цей клас містить властивість selectedCupboard, яка містить інформацію про виділений cupboard.</p>

	Цей клас дозволяє передавати цю інформацію разом з подією.
<pre>//layerMask that player could interact with only markeed objects as cupboards *note it is also must be done in unity ui [SerializeField] private LayerMask cupboardsLayerMask; [SerializeField] private float moveSpeed = 13f; private Vector2 inputVector = new Vector2(0, 0); private Vector3 moveVector = Vector3.zero; private Vector3 lastMove = Vector3.zero; private ClearCupboard selectedCupboard;</pre>	Тут це дані, які я думаю, зрозумілі і просто з назв
<pre>private void Update() { Movement(); Interact(); }</pre>	Ubdate в Юніті визивається кожен фрейм.
<pre>private void Movement() { inputVector = new Vector2(0, 0); if (Input.GetKey(KeyCode.W)) { inputVector.y = +1;</pre>	Movement ми розбирали раніше (у 1.1 перші кроки і по кроковому додавані). Дивитися коментарі в коді, там написано що за що відповідає.

```
}

if (Input.GetKey(KeyCode.S))
{
    inputVector.y = -1;
}

if (Input.GetKey(KeyCode.D))
{
    inputVector.x = +1;
}

if (Input.GetKey(KeyCode.A))
{
    inputVector.x = -1;
}

inputVector = inputVector.normalized;

moveVector = new Vector3(inputVector.x, 0f,
inputVector.y);

float playerHeight = .7f;

float playerWidth = .9f;

bool movePossible =
!Physics.CapsuleCast(transform.position,
transform.position + Vector3.up * playerHeight,
playerWidth
h, moveVector, Time.deltaTime * moveSpeed);
```

```
        if (!movePossible)

        {

            //axis x after stuck

            Vector3 moveAxisX = new Vector3(inputVector.x,
0f, 0f);

            moveAxisX = moveAxisX.normalized;

            movePossible =

!Physics.CapsuleCast(transform.position,
transform.position + Vector3.up * playerHeight,

                                playerWidth
h, moveAxisX, Time.deltaTime * moveSpeed);

            //if yes we will slide x axis

            if (movePossible)

            {

                moveVector = moveAxisX;

            }

            //axis z after stuck

            else

            {

                Vector3 moveAxisZ = new Vector3(0f, 0f,
inputVector.y);

                moveAxisZ = moveAxisZ.normalized;
```

```

        movePossible =
!Physics.CapsuleCast(transform.position,
transform.position + Vector3.up * playerHeight,
                                player
Width, moveAxisZ, Time.deltaTime * moveSpeed);

        //if yes we will slide z axis

        if (movePossible)

        {

            moveVector = moveAxisZ;

        }

    }

}

//if we stuck here our moveVector changed in
moveAxis(x/z)

    if (movePossible)

    {

        transform.position += moveVector *
Time.deltaTime * moveSpeed;

    }

    transform.forward =
Vector3.Lerp(transform.forward, moveVector, Time.deltaTime
* 15); // число-відповідає за пришвидшення повороту

}

```

```

private void Interact()
{
    float interactionDistance = 2f;
    //lastMove needed if our player stand still and we
    want that interaction would be possible

    if (moveVector != Vector3.zero)
    {
        lastMove = moveVector;
    }

    if (Physics.Raycast(transform.position,
lastMove, out RaycastHit raycastHit,
interactionDistance, cupboardsLayerMask))
    {
        if
(raycastHit.transform.TryGetComponent(out
ClearCupboard clearCupboard))
        {
            //interaction after key input

            if (Input.GetKeyDown(KeyCode.E))
            {
                clearCupboard.InteractCupboard()
;

            }

            //highlight

```

Interact. Ще раз наголошую, що в цій частині ми визначаємо, що знаходиться перед гравцем, яким об'єктом, і якщо це звичайний clearCupboard, то ми його помічаємо як selectedCupboard, щоб потім додати підсвічування (highlight) на нього.

if (Physics.Raycast

ми використовуємо метод Raycast з модуля Physics, щоб визначити, чи є об'єкт, на який спрямований останній рух гравця. Якщо промінь зіткнувся з об'єктом зі шару cupboardsLayerMask, то ми продовжуємо перевірку.

Далі, в рядку другому, ми використовуємо метод TryGetComponent, щоб перевірити, чи має об'єкт компонент ClearCupboard.

У рядку третьому ми перевіряємо, чи є об'єкт clearCupboard, тим сами чи ні. Якщо об'єкт clearCupboard не є обраним об'єктом,


```

        if (clearCupboard !=
selectedCupboard)

        {

            SetSelectedCupboard(clearCupboard
d);

        }

    }

    else

    {

        // Check if the player is leaving
the selected cupboard area

        if (selectedCupboard != null)

        {

            SetSelectedCupboard(null);

        }

    }

}

else

{

    // Check if the player is leaving the
selected cupboard area

    if (selectedCupboard != null)

    {

        SetSelectedCupboard(null);

    }

}

}

```

ТО МИ ВИКЛИКАЄМО МЕТОД
SetSelectedCupboard(clearCupboard).

Далі ми в інших випадках
обнуляєм selectedCupboard.

Далі буде метод
SetSelectedCupboard(clearCupboard).

<pre> }</pre>	
<pre> private void SetSelectedCupboard(ClearCupboard selectedCupboard) { this.selectedCupboard = selectedCupboard; OnHighlightCupboard?.Invoke(this, new OnHighlightCupboardEventArgs { selectedCupboard = selectedCupboard }); }</pre>	<p>Перший рядок методу встановлює вибраний cupboard на об'єкт, який передається як аргумент методу. У цьому випадку, метод отримує об'єкт "selectedCupboard".</p> <p>Другий рядок методу викликає подію "OnHighlightCupboard" (яка, як ми розглядали раніше, є типом "EventHandler<OnHighlightCupboardEventArgs>"), якщо вона не є рівною null (позначається знаком питання ?). Ця подія використовується для відображення вибраного cupboard в грі.</p>

Тепер розглянемо скрипт Highlight:

<pre> using System.Collections; using System.Collections.Generic; using UnityEngine; public class HighlightCupboard : MonoBehaviour { //for insert prefabs and visual [SerializeField] private ClearCupboard clearCupboard; //prefab of cupboard visual</pre>	<p>Цікавий факт: Нам потрібно, щоб екземпляр (Instance) зі скрипта Player активувався перед частиною Player.Instance... з цього скрипта. Тому відповідний метод у Player скрипті називається Awake(), а тут ми використовуємо метод Start().</p>
---	--

```

[SerializeField] private GameObject
highlightObject; //prefab of transparent visual

private void Start()
{
    Player.Instance.OnHighlightCupboard +=
Instance_OnHighlightCupboard;
}

private void
Instance_OnHighlightCupboard(object sender,
Player.OnHighlightCupboardEventArgs p)
{
    if (p.selectedCupboard == clearCupboard)
    {
        highlightObject.SetActive(true);
    }
    else
    {
        highlightObject.SetActive(false);
    }
}
}

```

Коли в Player станеться якась подія, він може повідомити про це. Player використовує обробник подій OnHighlightCupboard для інформування всіх підписаних на цю подію.

У випадку HighlightCupboard, метод Start() встановлює обробник подій Instance_OnHighlightCupboard для події OnHighlightCupboard в класі Player. Це дозволяє HighlightCupboard реагувати на зміни в Player.

Метод Instance_OnHighlightCupboard() є обробником події OnHighlightCupboard. Цей метод приймає об'єкт-відправник і екземпляр OnHighlightCupboardEventArgs, який містить вибраний cupboard гравцем. В методі перевіряється, чи обраний cupboard є поточним clearCupboard, і якщо так, то активується об'єкт highlightObject, що відповідає за

	виділення cupboard. Інакше highlightObject деактивується.
--	---

1.7 Піднімання об'єктів і використання інтерфейсів

У цій главі ми використовуємо інтерфейси, наслідування (Inheritance) і зміну батьківських об'єктів (parents).

У нас є скрипт InheritCupboard(рис. 1.21), до якого підключений інтерфейс I_PropParent. Цей скрипт використовується для різних типів шафок (cupboards), які успадковують його функціональність, але мають власну унікальну логіку. Наприклад, контейнер може створювати об'єкти, тоді як пуста шафка може брати або ставити об'єкти.

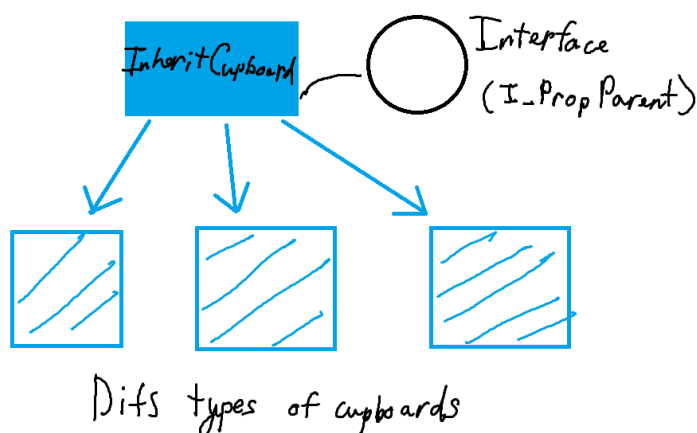


Рис.1.22 Схема. В нас є інтерфейс який прикріплений до Базового префабу InheritCupboard, а всі інші типи cupboards будуть розширювати його.

Відео, яке показує, як це все працює, можна переглянути за посиланням:
<https://youtu.be/UzB2X9urgYA>

Давайте коротко розберемо вже перероблений код.

```
public interface I_PropParent
{
    //here lays setters, getters, the one which make null
    and boolean to understand is there prop at all.

    public Transform GetPropTransform();

    public void SetPropParent(Props prop);

    public Props GetProp();

    public bool HasProp();

    public void NullProp();}
```

Хто імпліментує цей інтерфейс матиме такі 5 методів. В нашому випадку матимуть цей інтерфейс InheritCupboard і Player (Бо предмети ми беремо з cupboards у руки player, тобто змінюємо parents. *це не робиться в інтерфейсі, а робиться ЗА ДОПОМОГОЮ його, бо ми розділяєм дані між тими хто його імплементив)

Тепер InheritCupboard з якого наслідую всі інші cupboards

```
public class InheritCupboard : MonoBehaviour, I_PropParent
{
    [SerializeField] protected Transform topPoint; //point where
    things will spawn/carried etc

    //Props class which contain info of PropSort where we connected
    cheese/onion/tomato etc

    //we need this so player understand what object is not just simple
    PROP but a TOMATO.

    protected Props prop;

    public virtual void InteractCupboard(Player player)
    {

    }

    public Transform GetPropTransform()
```

Клас містить метод InteractCupboard, який приймає об'єкт гравця та буде override в класах-спадкоємцях, щоб забезпечити різні функціональність при взаємодії з cupboard.

```

{
    return topPoint;
}

public void SetPropParent(Props prop)
{
    this.prop = prop;
}

public Props GetProp()
{
    return prop;
}

public bool HasProp()
{
    return prop != null;
}

public void NullProp()
{
    prop = null;
}
}

```

Пояснення також можна дивитись в самому коді, бо я впевнений, що вони прості і зрозумілі.

Далі роздивимся два схожих кода і наслідників цього класу.

Container = spawner(спавнер об'єктів), ClearCupboard = put/take object(звідсіля ми беремо і кладемо об'єкти).

```

public class Container : InheritCupboard
{

```

```

public class ClearCupboard : InheritCupboard
{

```

```

[SerializeField] private PropsSort propSO;
//so = scriptable object

public override void InteractCupboard(Player
player)
{
    //check that player can pick up only one
object
    if (!player.HasProp())
    {
        //creation of prop prefab in topPoint
        //and link the prop to a
clearCupboard object
        Transform propTransform =
Instantiate(propSO.prefab, topPoint);
        propTransform.GetComponent<Props>().S
etPropParent(player);
    }
}

```

```

[SerializeField] private PropsSort propSO;
//so = scriptable object

public override void InteractCupboard(Player
player)
{
    //check is there any object on a cupboard
    if (!HasProp())
    {
        //no prop found so check if player
holding smth drop here
        //else do nothing
        if (player.HasProp())
        {
            player.GetProp().SetPropParent(this);
        }
    }
    //so there is smth in cupboard. If player
has smth on hands do nothing
    //if player empty handed PICK IT
UP!
    else if (!player.HasProp())
    {
        {get from cupboard to player hands
        GetProp().SetPropParent(player);
    }
}
}

```

І скрипти які я згадував, але детально не розбирав. Саме PropsSort і Props:

```

[CreateAssetMenuAttribute()]
public class PropsSort : ScriptableObject

```

PropsSort на початку містить атрибут [CreateAssetMenuAttribute()],

<pre> { public Transform prefab; public string propName; } </pre>	<p>що дозволяє створювати нові екземпляри цього класу</p>
<pre> public class Props : MonoBehaviour { [SerializeField] private PropsSort propsSort; private I_PropParent propParent; public PropsSort GetPropsSort() { return propsSort; } public void SetPropParent(I_PropParent propParent) { // If this Props object already has a parent object, // remove the reference to the Props object from that parent if (this.propParent != null) { this.propParent.NullProp(); } this.propParent = propParent; propParent.SetPropParent(this); </pre>	<p>GetPropsSort(), повертає змінну propsSort.</p> <p>SetPropParent(), приймає параметр propParent. У цьому методі перевіряється, чи існує вже батьківський об'єкт для екземпляру Props. Так - видаляє зв'язок між ним та попереднім батьківським об'єктом. Ні - новий батьківський об'єкт стає батьківським для Props.</p> <p>Далі викликається метод SetPropParent() на propParent з параметром this, що передає власний екземпляр Props як child об'єкт до propParent.</p>


```

        //when changing parents we get new
topPoint.

        //Also make local coordinates 0 0 0

        transform.parent =
propParent.GetPropTransform();

        transform.localPosition = Vector3.zero;

    }

    public I_PropParent GetProp()

    {

        return propParent;

    }

}

```

1.8 Нарізка, смаження, прогрес, смітник.

Тут буде йтись про додавання місця для нарізки трьох об'єктів. Смаження м'яса і пересмаження його. Progress bar, де можна бачити скільки часу смажиться м'ясо. І звичайно ж смітник.(рис 1.22)

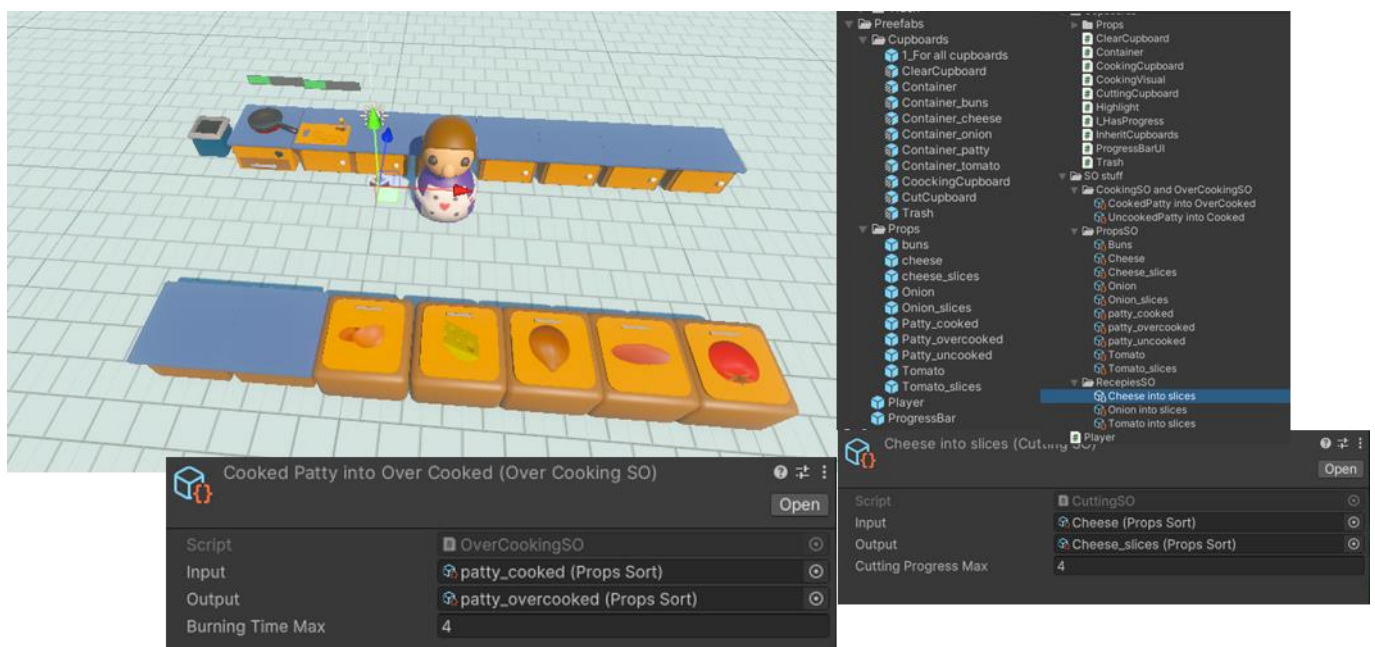


Рис.1.22 Вигляд. Кількість скриптів і SO.

Якнайкоротше про наступні коди з точки зору механік. Рухатись wasd. Об'єкти з 5 контейнерів можна брати і ставити на звичайні clearCupboards (Ability 1). З інгредієнта можна нарізати на cuttingCupboard і інгредієнти стануть нарізаними їхніми версіями(нарізання клікання Ability 2). М'ясо можна покласти на пательню, через деякий час воно перейде у стан cooked і якщо м'ясо не прибрати вчасно то воно згорить. І смітник, туди викидають непотрібні об'єкти.

Перейдемо до розгляду скриптів, в основному це будуть картинки з пояснювальними коментарями в них.

У Player додався момент з встановкою Ability2, в данному випадку нарізання виконується після натискання на LeftControl.(рис.1.23)

```
//interaction after key input
if (Input.GetKeyDown(KeyCode.E))
{
    basicCupboard.InteractCupboard(this);
}

//interaction after key input
if (Input.GetKeyDown(KeyCode.LeftControl))
{
    basicCupboard.InteractCutting(this);
}
```

Рис.1.23 Додано нова взаємодія.

Тепер InheritCupboard і Interface PropParent. Всі пояснення в коментарях. І останнє фото це простий clearCupboard(рис.1.24).

```

10 references
public class InheritCupboard : MonoBehaviour, I_PropParent
{
    1 reference
    [SerializeField] protected Transform topPoint; //point where things will spawn

    //Props class which contain info of PropSort where we connected cheese/onion/tomato etc
    //we need this so player understand what object is not just simple PROP but a TOMATO.
    4 references
    protected Props prop;
    1 reference
    public virtual void InteractCupboard(Player player)
    {
    }

    1 reference
    public virtual void InteractCutting(Player player)
    {
    }

    1 reference
    public Transform GetPropTransform()
    {
        return topPoint;
    }

    1 reference
    public void SetPropParent(Props prop)
    {
        this.prop = prop;
    }

    18 references
    public Props GetProp()
    {
        return prop;
    }

    13 references
    public bool HasProp()
    {
        return prop != null;
    }

    2 references
    public void NullProp()
    {
        prop = null;
    }
}

6 references
public interface I_PropParent
{
    //here lays setters, getters, the one which make null and boolean to understand is there prop at all.
    1 reference
    public Transform GetPropTransform();
    1 reference
    public void SetPropParent(Props prop);
    18 references
    public Props GetProp();
    13 references
    public bool HasProp();
    2 references
    public void NullProp();
}

public class ClearCupboard : InheritCupboard
{
    [SerializeField] private PropsSort propSO; //so = scriptable object

    public override void InteractCupboard(Player player)
    {
        //check is there any object on a cupboard
        if (!HasProp())
        {
            //no prop found so check if player holding smth drop here
            //else do nothing
            if (player.HasProp())
            {
                player.GetProp().SetPropParent(this);
            }
        }

        //so there is smth in cupboard. If player has smth on hands do nothing
        //if player empty handed PICK IT UP!
        else if (!player.HasProp())
        {
            //get from cupboard to player hands
            GetProp().SetPropParent(player);
        }
    }
}

```

Рис.1.24 InheritCupboard, Interface PropParent, clearCupboard

Смітник, який видаляє прийманий предмет(рис.1.25).

```

0 references
public class Trash : InheritCupboard
{
    1 reference
    public override void InteractCupboard(Player player)
    {
        //if player holding something delete that
        if (player.HasProp())
        {
            player.GetProp().Despawn();
        }
    }
}

```

Рис.1.25 Trash

Тут показані SO, тобто сюди ми вписуємо рецепти. Наприклад, рецепт нарізання помідора, ми вводимо помідор, а повинні отримати нарізану версію його. Одні префаби замінюються на інші, саме префаби, а не тільки 3д моделі.(рис.1.26)

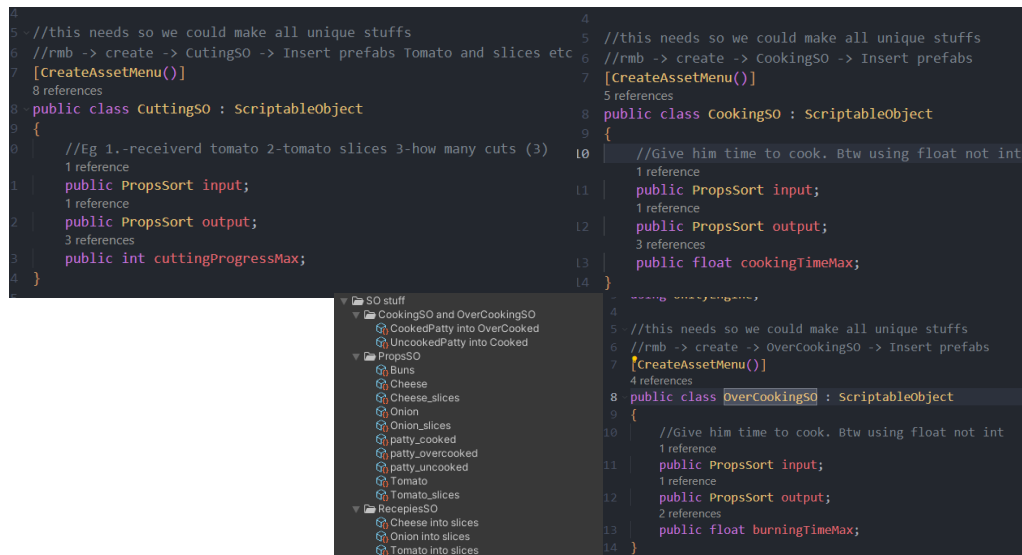


Рис.1.25 Нарізання SO і смаження з пересмаженням SO.

1.8.1 Нарізання.

Далі розглядається скрипт для нарізного cupboard.(рис. 1.26)

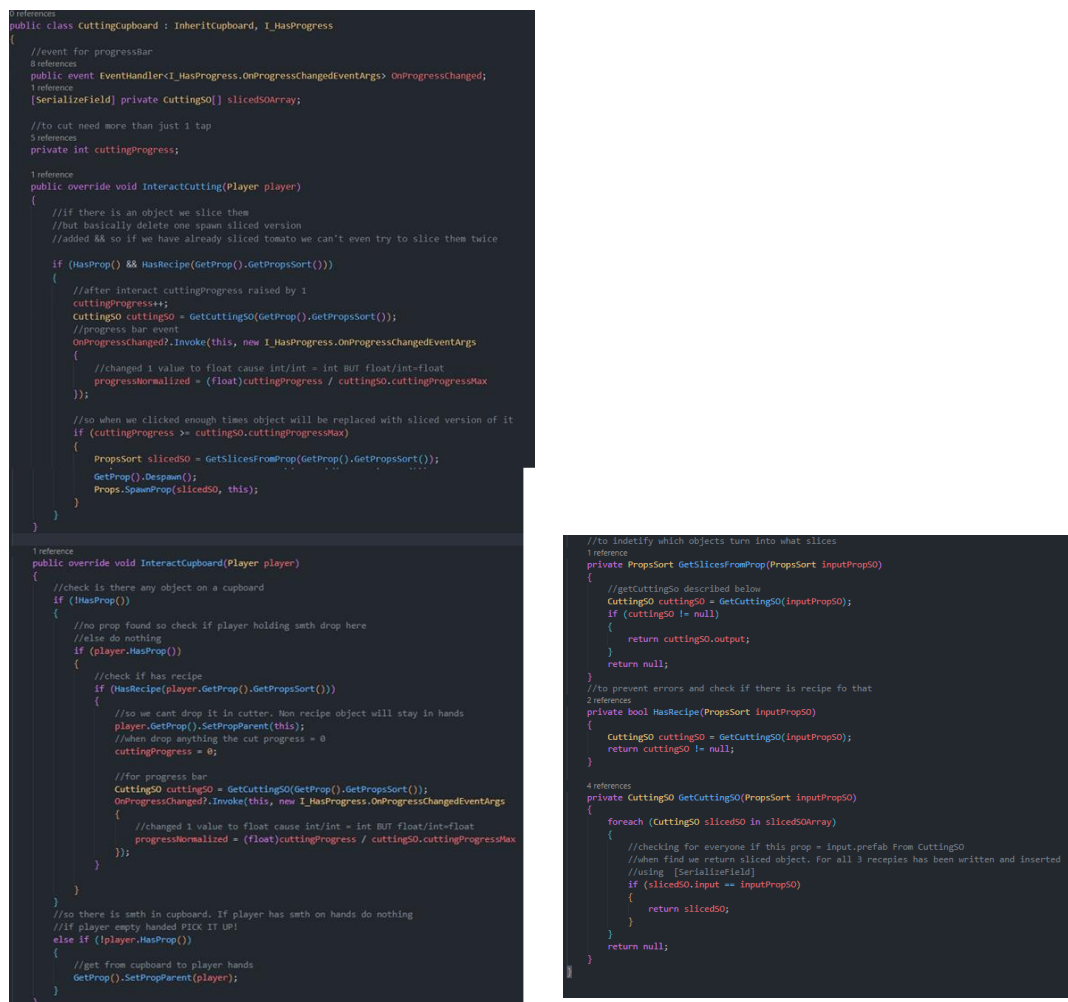


Рис.1.26 Нарізний cupboard скрипт.

Реалізація інтерфейсу HasProgress(рис.1.27)

```

14 references
public interface I_HasProgress
{
    //event for progressBar
    8 references
    public event EventHandler<OnProgressChangedEventArgs> OnProgressChanged;
    11 references
    public class OnProgressChangedEventArgs : EventArgs
    {
        //normalized means that value is from 0 to 1. And events usually called beginning in ON
        10 references
        public float progressNormalized;
    }
}

```

Рис.1.27 Інтерфейс HasProgress.

Тут створювався скрипт для показання прогресу процесу дії(рис. 1.28), використовувалось у нарізанні і смаженні. В юніті зроблено так, що ці UI дивляться у камеру.

```

1 reference
public class ProgressBarUI : MonoBehaviour
{
    2 references
    [SerializeField] private Image barImage;
    1 reference
    [SerializeField] private GameObject hasProgressGameObject;

    //this created cause unity dont show [SerializeField] interfaces
    2 references
    private I_HasProgress hasProgress;

    // Start is called before the first frame update
    0 references
    private void Start()
    {
        //progress bar
        hasProgress = hasProgressGameObject.GetComponent<I_HasProgress>();

        hasProgress.OnProgressChanged += HasProgress_OnProgressChanged;
        //so the bar will always start with 0 progress
        barImage.fillAmount = 0;
        //dont show the bar if noone interacting with it
        gameObject.SetActive(false);
    }

    // Define a method to handle changes to the Cupboard's progress
    1 reference
    private void HasProgress_OnProgressChanged(object sender, I_HasProgress.OnProgressChangedEventArgs p)
    {
        // Set the fill amount of the progress bar to the normalized progress value
        barImage.fillAmount = p.progressNormalized;
        //if bar is full or zero dont show it. Else show the bar
        if (p.progressNormalized == 0f || p.progressNormalized == 1f)
        {
            gameObject.SetActive(false);
        }
        else
        {
            gameObject.SetActive(true);
        }
    }
}

```

Рис.1.28 ProgressBar.

1.8.2 Смаження.

А тепер cookingCupboard(рис.1.29), де ми використовуватимемо State. Це створення станів, що відображають певний етап. Містить назви можливих станів. У нашому випадку використовується для відстеження процесу готування їжі.

- State.Idle: коли не використовується;
- State.Cooking: коли готується їжа;
- State.Cooked: коли їжа готова;

- State.Burned: коли згоріла їжа.

```

4 references
public class CookingCupboard : InheritCupboard, I_HasProgress
{
    //event for progress bar
    8 references
    public event EventHandler<I_HasProgress.OnProgressChangedEventArgs> OnProgressChanged;
    0 references
    [SerializeField] private CuttingSO[] slicedSOArray;

    //stuff to turn red when cooking
    5 references
    public event EventHandler<OnStateCheckEventArgs> OnStateCheck;
    6 references
    public class OnStateCheckEventArgs : EventArgs
    {
        6 references
        public State state;
    }

    13 references
    public enum State
    {
        3 references
        Idle,
        3 references
        Cooking,
        3 references
        Cooked,
        2 references
        Burned,
    }

    1 reference
    [SerializeField] private CookingSO[] cookingSOArray;
    1 reference
    [SerializeField] private OverCookingSO[] overCookingSOArray;

    //timer moment
    10 references
    private State currState;
    4 references
    private float burningTimer;
    5 references
    private float cookingTimer;
    5 references
    private CookingSO cookingSO;
    4 references
    private OverCookingSO overCookingSO;

    0 references
    private void Start()
    {
        //we are always starting in Idle state
        currState = State.Idle;
    }

    //timer will update every frame
    0 references
    private void Update()
    {
        if (HasProp())
        {
            switch (currState)
            {
                case State.Idle:
                    break;

                case State.Cooking:
                    cookingTimer += Time.deltaTime;

                    // event that shows how much need to be in the bar
                    OnProgressChanged?.Invoke(this, new I_HasProgress.OnProgressChangedEventArgs
                    {
                        progressNormalized = cookingTimer / cookingSO.cookingTimeMax
                    });

                    //if it is bigger than max than it cooked
                    if (cookingTimer > cookingSO.cookingTimeMax)
                    {
                        //cause it fried we Despawn this version of patty and create +cooked version
                        GetProp().Despawn();
                        Props.SpawnProp(cookingSO.output, this);

                        //cooking is finished so we put patty to cooked state
                        currState = State.Cooked;
                        burningTimer = 0f;
                        overCookingSO = GetOverCookingSO(GetProp().GetPropsSort());

                        //writing down event so know when to activate red part of cooking process
                        OnStateCheck?.Invoke(this, new OnStateCheckEventArgs
                        {
                            state = currState
                        });
                    }
                    break;

                case State.Cooked:
                    burningTimer += Time.deltaTime;

                    // event that shows how much need to be in the bar
                    OnProgressChanged?.Invoke(this, new I_HasProgress.OnProgressChangedEventArgs
                    {
                        progressNormalized = burningTimer / overCookingSO.burningTimeMax
                    });

                    //if it is bigger than max than it cooked
                    if (burningTimer > overCookingSO.burningTimeMax)
                    {
                        //cause it fried we Despawn this version of patty and create +cooked version
                        GetProp().Despawn();
                        Props.SpawnProp(overCookingSO.output, this);

                        //cooking is finished so we put patty to cooked state
                        currState = State.Burned;

                        //writing down event so know when to activate red part of cooking process
                        OnStateCheck?.Invoke(this, new OnStateCheckEventArgs
                        {
                            state = currState
                        });

                        // and now we make timer = 0
                        OnProgressChanged?.Invoke(this, new I_HasProgress.OnProgressChangedEventArgs
                        {
                            progressNormalized = 0f
                        });
                    }
                    break;

                case State.Burned:
                    break;
            }
        }
    }
}

```

```

public override void InteractCupboard(Player player)
{
    //check is there any object on a cupboard
    if (!HasProp())
    {
        //no prop found so check if player holding smth drop here
        //else do nothing
        if (player.HasProp())
        {
            //check if has recipe
            if (HasRecipe(player.GetProp().GetPropsSort()))
            {
                //so we cant drop it in cookingCupboard. Non recipe object will stay in hands
                player.GetProp().SetPropParent(this);

                //changing this variable so we would understand what Prop is there
                cookingSO = GetCookingSO(GetProp().GetPropsSort());

                //player dropped smth so we put in cooking state
                //and to prevent bugs we reset timer
                currState = State.Cooking;
                cookingTimer = 0f;

                //the state here also changed so we need to rewrite it in event
                OnStateCheck?.Invoke(this, new OnStateCheckEventArgs
                {
                    state = currState
                });

                // event that shows how much need to be in the bar
                OnProgressChanged?.Invoke(this, new I_HasProgress.OnProgressChangedEventArgs
                {
                    progressNormalized = cookingTimer / cookingSO.cookingTimeMax
                });
            }
        }

        //so there is smth in cupboard. If player has smth on hands do nothing
        //if player empty handed PICK IT UP!
        else if (!player.HasProp())
        {
            //get from cupboard to player hands
            GetProp().SetPropParent(player);

            //and when player pick ups the object state is idle
            currState = State.Idle;

            //the state here also changed so we need to rewrite it in event
            OnStateCheck?.Invoke(this, new OnStateCheckEventArgs
            {
                state = currState
            });

            // if player take smth patty, and now we make timer = 0
            OnProgressChanged?.Invoke(this, new I_HasProgress.OnProgressChangedEventArgs
            {
                progressNormalized = 0f
            });
        }
    }
}

```

```

//to prevent errors and check if there is recipe fo that
1 reference
private bool HasRecipe(PropsSort inputPropSO)
{
    CookingSO cookingSO = GetCookingSO(inputPropSO);
    return cookingSO != null;
}

2 references
private CookingSO GetCookingSO(PropsSort inputPropSO)
{
    foreach (CookingSO cookedSO in cookingSOArray)
    {
        //checking for everyone if this prop = input.prefab from CookingSO
        //when find we return cooked or overcooked version of it
        //using [SerializeField]
        if (cookedSO.input == inputPropSO)
        {
            return cookedSO;
        }
    }
    return null;
}

1 reference
private OverCookingSO GetOverCookingSO(PropsSort inputPropSO)
{
    foreach (OverCookingSO overCookedSO in overCookingSOArray)
    {
        if (overCookedSO.input == inputPropSO)
        {
            return overCookedSO;
        }
    }
    return null;
}

```

```

0 references
public class CookingVisual : MonoBehaviour
{
    1 reference
    [SerializeField] private CookingCupboard cookingCupboard;
    2 references
    [SerializeField] private GameObject redHighlight;

    // Start is called before the first frame update
    private void Start()
    {
        // Register the CookingCupboard_OnStateCheck method as an event handler for the OnStateCheck event
        cookingCupboard.OnStateCheck += CookingCupboard_OnStateCheck;
        // Deactivate the red highlight at the beginning of the game
        redHighlight.SetActive(false);
    }

    // Define a method to handle the OnStateCheck event
    1 reference
    private void CookingCupboard_OnStateCheck(object sender, CookingCupboard.OnStateCheckEventArgs p)
    {
        // Determine whether to activate the red highlight based on the state of the cooking cupboard
        bool activated = p.state == CookingCupboard.State.Cooking || p.state == CookingCupboard.State.Cooked;
        redHighlight.SetActive(activated);
    }
}

```

Рис.1.29 CookingCupboard скрипт(читати перший стовпчик потім другий).

І це кінцевий результат пункту 1.8: <https://youtu.be/zlKXuH4yras>

1.9 Тарілка і UI для неї.

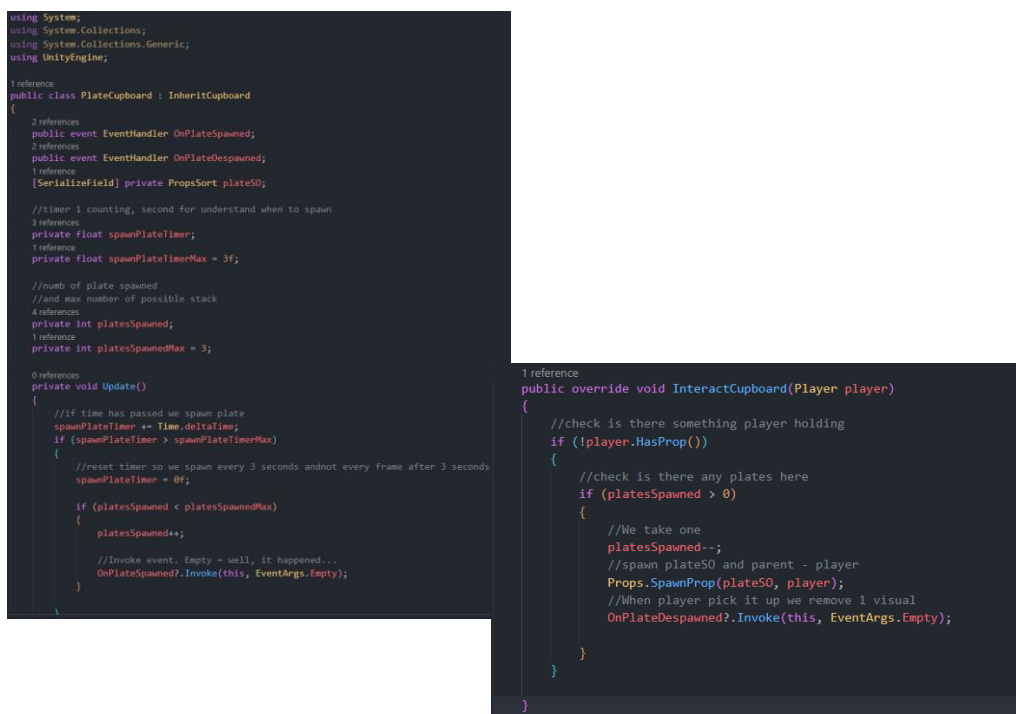
У нас є шафка `clearCupboard`, яка має функціонал спавнера тарілок. Максимальна кількість тарілок, які можуть спавнитись, обмежена до 4 штук, і вони спавняються по одній кожні X секунд.

Тарілка має 3D модель, яка складається з двох частин: візуальної 3D моделі і справжнього префабу зі скриптами. Візуальна частина використовується на спавнері для простого відображення тарілки. Коли взаємодіють зі спавнером, він видає справжній префаб зі скриптами та забирає одну візуальну тарілку зі спавнера.

Справжній префаб тарілки складається з трьох частин: перша частина - сама тарілка, друга - 3D частини бургера, які розміщуються на тарілці, і третя - 2D UI, де показується, які інгредієнти вже присутні на тарілці.

З тарілкою можна взаємодіяти, беручи інгредієнти або складаючи інгредієнти на тарілку. Наразі ми перейдемо до пояснення скриптів, і я розумію, що їх може бути важко зрозуміти, оскільки вони взаємодіють між собою.

Спавнер тарілок `Cupboard`(рис. 1.30):



```

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

1 reference
public class PlateCupboard : MonoBehaviour
{
    2 references
    public event EventHandler OnPlateSpawned;
    2 references
    public event EventHandler OnPlateDespawned;
    1 reference
    [SerializeField] private PropsSort plateSO;

    //timer 1 counting, second for understand when to spawn
    1 reference
    private float spawnPlateTimer;
    1 reference
    private float spawnPlateTimerMax = 3f;

    //numb of plate spawned
    //and max number of possible stack
    4 references
    private int platesSpawned;
    1 reference
    private int platesSpawnedMax = 3;

    0 references
    private void Update()
    {
        //If time has passed we spawn plate
        spawnPlateTimer += Time.deltaTime;
        if (spawnPlateTimer > spawnPlateTimerMax)
        {
            //reset timer so we spawn every 3 seconds and not every frame after 3 seconds
            spawnPlateTimer = 0f;

            if (platesSpawned < platesSpawnedMax)
            {
                platesSpawned++;

                //Invoke event. Empty = well, it happened...
                OnPlateSpawned?.Invoke(this, EventArgs.Empty);
            }
        }
    }

    1 reference
    public override void InteractCupboard(Player player)
    {
        //check is there something player holding
        if (!player.HasProp())
        {
            //check is there any plates here
            if (platesSpawned > 0)
            {
                //We take one
                platesSpawned--;
                //spawn plateSO and parent - player
                Props.SpawnProp(plateSO, player);
                //When player pick it up we remove 1 visual
                OnPlateDespawned?.Invoke(this, EventArgs.Empty);
            }
        }
    }
}

```

Рис.1.30 Спавнер тарілок.

Тарілка(рис. 1.31):

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

//Enhance Props
12 references
public class Plate : Props
{
    //event for burger visual
    //It will work when we add some of the burger's parts.(patty, onion, buns...
    3 references
    public event EventHandler<OnAddPartBurgerEventArgs> OnAddPartBurger;
    4 references
    public class OnAddPartBurgerEventArgs : EventArgs
    {
        2 references
        public PropsSort propsSort;
    }

    1 reference
    [SerializeField] private List<PropsSort> validPropSortList;
    4 references
    private List<PropsSort> propsSortList;
    0 references
    private void Awake()
    {
        propsSortList = new List<PropsSort>();
    }

    4 references
    public bool TryAddProp(PropsSort propsSort)
    {
        // If it is not a valid prop -> false
        if (!validPropSortList.Contains(propsSort))
        {
            return false;
        }
        // if there is already this object in a plate -> false
        if (propsSortList.Contains(propsSort))
        {
            return false;
        }
        //means we can put it on a plate
        else
        {
            propsSortList.Add(propsSort);
            //when we adding burger part we invoke the event
            OnAddPartBurger?.Invoke(this, new OnAddPartBurgerEventArgs
            {
                propsSort = propsSort
            });
            return true;
        }
    }

    //this to get List of object ON a plate right now
    //using this in a ui part where we see icons
    3 references
    public List<PropsSort> GetPropsSortList()
    {
        return propsSortList;
    }
}
```

Рис.1.31 Тарілка.

Бургер на тарілці(рис.1.32). Оскільки в скрипті тарілки прописано логіку додавання чогось туди тепер залишалась візуальна частина. SetActive певний об'єкт який додався і він стає видимим.

```
public class PlateBurgerVisual : MonoBehaviour
{
    //it appears without Ser-able it will not show in editor 0.0.
    //to identify what we are adding
    [Serializable]
    3 references
    public struct PropsSort_G0
    {
        1 reference
        public PropsSort propsSort;
        2 references
        public GameObject gameObject;
    }

    1 reference
    [SerializeField] private Plate plate;
    2 references
    [SerializeField] private List<PropsSort_G0> PropsSortG0List;

    0 references
    private void Start()
    {
        plate.OnAddPartBurger += Plate_OnAddPartBurger;
        //disable every part of burger
        foreach (PropsSort_G0 propsSort_G0 in PropsSortG0List)
        {
            propsSort_G0.gameObject.SetActive(false);
        }
    }

    1 reference
    private void Plate_OnAddPartBurger(object sender, Plate.OnAddPartBurgerEventArgs p)
    {
        //so logic is: we cycle through all(5 in this time) parts of burger and we
        //activate the 1 we put on plate
        foreach (PropsSort_G0 propsSort_G0 in PropsSortG0List)
        {
            if (propsSort_G0.propsSort == p.propsSort)
            {
                propsSort_G0.gameObject.SetActive(true);
            }
        }
    }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
//connected to IconTemplate btw, not burgerIcons

public class PlateBurgerIconsConnection : MonoBehaviour
{
    //here we gram image of the icon and set proper one from PropsSort sprites
    [SerializeField] private Image image;

    public void SetPropsSortConnection(PropsSort propsSort)
    {
        image.sprite = propsSort.sprite;
    }

    //also here we could add some animation things, that every time icons appeared
    // we could add patrticles or smth, but there is no time for that
}
```

Рис.1.32 Бургер візуальна частина на тарілці.

Тепер додаємо іконки(рис. 1.33), які б зверху показували що несе гравець.

Тому для цього і UI(це в основному) я додав Спрайт можливість у PropsSort частину. (очевидно що всі картинку мені прийшлося перероблювати в блендері і потім робити квадрати з них в фотошопі).



Рис.1.33 Приклад вигляду іконок.

Тепер розглянемо кодову частину:

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

0 references
public class PlateIcons : MonoBehaviour
{
    //2 references. 1 - plate. 2 - ui, icon + bg
    2 references
    [SerializeField] private Plate plate;
    3 references
    [SerializeField] private Transform iconTemplate;

    //disable visual so it will not make a fuss when the plate is empty
    0 references
    private void Awake()
    {
        iconTemplate.gameObject.SetActive(false);
    }

    //listening to event the same as plate
    //when some new part is added
    0 references
    private void Start()
    {
        plate.OnAddPartBurger += Plate_OnAddPartBurger;
    }

    //when event called we summon UbdateIcons()
    1 reference
    private void Plate_OnAddPartBurger(object sender, Plate.OnAddPartBurgerEventArgs p)
    {
        UpdateIcons();
    }

    //Here we Update Icons -_-
    //running through all ingredients tha added and adding specific icon(sprite) for this part of burger

    1 reference
    private void UpdateIcons()
    {
        //Deleting children UNLESS it is IconTemplate
        //we do this cause we DO NOT want spawning every icon each time 1 object added
        foreach (Transform child in transform)
        {
            if (child == iconTemplate) continue;
            Destroy(child.gameObject);
        }

        //running through all ingredients
        foreach (PropsSort propsSort in plate.GetPropsSortList())
        {
            //Instantiate to create new object
            //transform not null cause we DO NOT need spawn in 0 0 0 cordinats.
            //So in our case it will spawn as a child object of Plate(reference)
            Transform iconTransform = Instantiate(iconTemplate, transform);
            //and here we activate visual. We deactivate it on Awake.
            iconTransform.gameObject.SetActive(true);
            //setting correct sprite here. Tomato -> tomato, buns -> buns
            iconTransform.GetComponent<PlateBurgerIconsConnection>().SetPropsSortConnection(propsSort);
        }
    }
}
```

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

0 references
public class PlateUiLock : MonoBehaviour
{
    3 references
    private enum Mode
    {
        1 reference
        LookAt,
        1 reference
        LookAtInverted,
    }

    1 reference
    [SerializeField] private Mode mode;

    0 references
    private void LateUpdate()
    {
        switch (mode)
        {
            case Mode.LookAt:
                transform.LookAt(Camera.main.transform);
                break;

            case Mode.LookAtInverted:
                Vector3 dirFromCamera = transform.position - Camera.main.transform.position;
                transform.LookAt(transform.position + dirFromCamera);
                break;
        }
    }
}
```

Також було створено величезна кількість різних UI, але розбирати тут я їх не буду. Скрипти до них знаходяться у папці Assets->Scripts->UI.

Результат одно користувальницької гри: https://youtu.be/SPgm_ZpEmy4

2 «NETCODE»

Netcode в Unity відноситься до набору інструментів і функціональності, яка дозволяє створювати онлайн-ігри зі зниженою затримкою та високою продуктивністю. Це включає в себе різні технології та підходи для оптимізації мережевої взаємодії між гравцями, синхронізації даних та управління мережевими процесами.

У Unity існує кілька різних підходів до реалізації Netcode:

Unity Multiplayer: Це вбудована в Unity система для створення мережеских ігор. Вона надає можливості для створення серверних та клієнтських додатків, обробки введення гравців, синхронізації об'єктів, а також реалізації мережеских лобі і з'єднання з допомогою NetworkManager.

Крім цього, важливо вивчити основні концепції та терміни, пов'язані з Netcode, такі як:

Сервер і клієнт: Мережева гра зазвичай має один або кілька серверів, які керують грою та синхронізують дані між гравцями. Клієнти - це програми, що виконуються на комп'ютерах гравців і підключаються до сервера для участі в грі.

Relay-сервери: Це проміжні сервери, які допомагають установити з'єднання між клієнтами, коли пряма пірінгова комунікація неможлива. Вони допомагають забезпечити стабільність та безпеку мережевого з'єднання.

Лобі: Лобі - це місце, де гравці можуть знаходити та приєднуватись до доступних громадських ігор або створювати свої власні. Лобі може надавати можливості для фільтрації громадських ігор за різними критеріями, такими як рівень складності, регіон тощо.

2.1 ServerRPC і ClientRPC в Unity

ServerRPC (Server Remote Procedure Call) і ClientRPC (Client Remote Procedure Call) - це механізми, які дозволяють викликати функції на стороні сервера або клієнта в мережевій грі в Unity.

ServerRPC викликається на стороні сервера і відправляє повідомлення клієнтам, щоб вони виконали певну функцію. Це дає можливість серверу керувати грою та викликати функції на клієнтах з метою синхронізації даних, розсилки повідомлень або виконання певних дій.

ClientRPC викликається на стороні клієнта і відправляє повідомлення серверу, щоб він виконав певну функцію. Це дає можливість клієнтам сповіщати сервер про певні події, викликати функції на сервері або взаємодіяти з іншими клієнтами.

ServerRPC і ClientRPC дозволяють реалізувати взаємодію між клієнтами та сервером в мережевій грі. Вони часто використовуються для синхронізації стану об'єктів, передачі ігрових подій, обміну даними та виконання специфічних функцій.

Синтаксис ServerRPC:

Функція, яка повинна бути викликана на клієнтах, повинна бути анотована атрибутом [ServerRPC].

Виклик ServerRPC здійснюється за допомогою методу CallRpc на стороні сервера.

ServerRPC може приймати параметри, що будуть передані клієнтам.

Синтаксис ClientRPC:

Функція, яка повинна бути викликана на сервері, повинна бути анотована атрибутом [ClientRPC].

Виклик ClientRPC здійснюється за допомогою методу CallRpc на стороні клієнта.

ClientRPC може приймати параметри, які будуть передані серверу.

Нюанси:

ServerRPC і ClientRPC можуть бути викликані тільки з компонентів, що успадковують NetworkBehaviour.

Виклики ServerRPC та ClientRPC можуть бути здійснені тільки на об'єктах, що мають мережевий авторитет.

Аргументи `ServerRPC` і `ClientRPC` повинні бути серіалізовані, тобто вони повинні бути примітивними типами даних або об'єктами, що успадковують `MonoBehaviour` або `ScriptableObject`.

`ServerRPC` і `ClientRPC` є потужними інструментами для реалізації взаємодії між клієнтами та сервером у мережевій грі в Unity. Вони дозволяють передавати повідомлення та викликати функції з одного боку на інший, що допомагає синхронізувати стан гри та забезпечити взаємодію між гравцями.

2.2 Про мою реалізацію.

На жаль, процес створення реалізації в мене не був записаний. І тут були змінені скрипти до кінцевих, які можна переглянути на репозиторії на GitHub, але я можу дати загальну інформацію щодо процесу реалізації і пошуку помилок у мережових додатках в Unity.

При створенні реалізації мережової гри з використанням `Netcode` і `Network Manager` у Unity, процес включав наступні етапи:

- Підключення `Netcode` та `Network Manager`: Встановлення необхідних пакетів та імпортування потрібних модулів для роботи з `Netcode` і `Network Manager`.
- Створення сервера і клієнтів: Ініціалізація сервера та підключення клієнтів до нього. На цьому етапі можуть бути визначені правила і налаштування для мережової гри.
- Створення мережових об'єктів: Визначення об'єктів, які будуть взаємодіяти по мережі, і налаштування їх мережевого авторитету.
- Використання `ServerRPC` і `ClientRPC`: Визначення функцій, які будуть викликатися на сервері або клієнті, та анотування їх атрибутами `ServerRPC` або `ClientRPC` відповідно. Ці функції викликаються з використанням методів `CallRpc` на стороні сервера або клієнта.
- Синхронізація стану гри: Використання `ServerRPC` і `ClientRPC` для синхронізації стану об'єктів між сервером і клієнтами. Це може включати передачу даних, розсилку повідомлень або виконання певних дій для забезпечення відповідності гри на всіх пристроях.

- Пошук та усунення помилок: Коли виникають помилки під час використання ServerRPC і ClientRPC, процес пошуку причини може включати перегляд та аналіз коду, встановлення точок зупину, відладку та внесення необхідних змін для виправлення проблем.

ВИСНОВКИ

У даній курсовій роботі було розроблено комп'ютерну гру «Chef Run», яка має схожість з відомою грою "Overcooked". Гра спрямована на кооперативний геймплей і має на меті надати користувачам розважальний досвід та сприяти спільній грі та взаємодії гравців. Основною метою гри є виконання замовлень та приготування різних видів бургерів в обмежений час.

Для розробки гри було використано програму Unity та мову програмування C#. Для досягнення необхідного функціоналу використовувалися такі компоненти, як Netcode, Network Manager, Relay та Lobby, що забезпечили мережевий функціонал гри та можливість створення лобі для гравців.

Процес розробки гри включав створення головного меню з можливістю створення публічних та приватних лобі, де гравці могли підключатися та готуватися до гри. Головною сценою гри була кухня, де гравці могли взаємодіяти з об'єктами, брати і нарізати інгредієнти, готувати бургери та здавати замовлення.

Під час розробки було враховано різноманітні аспекти, такі як обмеження часу та можливість згорання м'яса, необхідність правильного складання бургерів на тарілку та їх подачі на спеціальне місце.

У процесі роботи використовувалися 3D-моделі, які були створені за допомогою програми Blender.

Окрім того, було проведено тестування гри та виправлення помилок для забезпечення якісного геймплею та оптимальної роботи мережевого з'єднання.

Результатом даної курсової роботи є створена комп'ютерна гра, яка надає користувачам можливість насолодитися захоплюючим геймплеєм, спільною грою з друзями та виконанням кулінарних викликів. Розроблена гра може бути використана як основа для подальшого розвитку та додавання нових функціональних можливостей.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Unity Technologies. "Unity Multiplayer and Networking". URL: <https://unity.com/solutions/multiplayer> (дата звернення 13.04.2023).
2. Unity Technologies. "Networking Overview". URL: <https://docs.unity3d.com/Manual/UNetOverview.html> (дата звернення 15.04.2023).
3. Unity Technologies. "UNET - Unity Networking". URL: <https://learn.unity.com/tutorial/unet-overview> (дата звернення 14.04.2023).
4. Unity Technologies. "UNET - High-level API". URL: <https://learn.unity.com/tutorial/unet-high-level-api> (дата звернення 15.04.2023).
5. Unity Technologies. "UNET - Low-level API". URL: <https://learn.unity.com/tutorial/unet-low-level-api> (дата звернення 15.04.2023).
6. Unity Technologies. "UNET - HLAPI vs LLAPI". URL: <https://learn.unity.com/tutorial/unet-hlapi-vs-llapi> (дата звернення 15.04.2023).
7. Unity Technologies. "UNET - Player Connection". URL: <https://learn.unity.com/tutorial/unet-player-connection> (дата звернення 15.04.2023).
8. Unity Technologies. "UNET - Matchmaking". URL: <https://learn.unity.com/tutorial/unet-matchmaking> (дата звернення 15.04.2023).
9. Unity Technologies. "UNET - Lobby". URL: <https://learn.unity.com/tutorial/unet-lobby> (дата звернення 15.05.2023).
10. Unity Technologies. "UNET - Relay". URL: <https://learn.unity.com/tutorial/unet-relay> (дата звернення 15.05.2023).
11. Unity Technologies. "UNET - Network Discovery". URL: <https://learn.unity.com/tutorial/unet-network-discovery> (дата звернення 15.04.2023).
12. Unity Technologies. "UNET - Customizing Network Manager". URL: <https://learn.unity.com/tutorial/unet-customizing-network-manager> (дата звернення 13.05.2023).

- 13.Unity Technologies. "UNET - Custom Network Messages". URL: <https://learn.unity.com/tutorial/unet-custom-network-messages> (дата звернення 17.05.2023).
- 14.Unity Technologies. "UNET - Syncing Player Positions". URL: <https://learn.unity.com/tutorial/unet-syncing-player-positions> (дата звернення 17.05.2023).
- 15.Unity Technologies. "UNET - Network Transform". URL: <https://learn.unity.com/tutorial/unet-network-transform> (дата звернення 11.06.2023).
- 16.Unity Technologies. "UNET - Network Animator". URL: <https://learn.unity.com/tutorial/unet-network-animator> (дата звернення 11.06.2023).
- 17.Unity Technologies. "UNET - Network Transform Child Objects". URL: <https://learn.unity.com/tutorial/unet-network-transform-child-objects> (дата звернення 13.06.2023).
- 18.Unity Technologies. "UNET - Network Identity". URL: <https://learn.unity.com/tutorial/unet-network-identity> (дата звернення 13.06.2023).
- 19.Unity Technologies. "UNET - Network Manager HUD". URL: <https://learn.unity.com/tutorial/unet-network-manager-hud> (дата звернення 13.06.2023).
- 20.Unity Technologies. "UNET - Network Lobby Manager". URL: <https://learn.unity.com/tutorial/unet-network-lobby-manager> (дата звернення 13.06.2023).