

ЧЕРКАСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ім. Богдана Хмельницького

Факультет Обчислювальної техніки, інтелектуальних та
управляючих систем

Кафедра Програмного забезпечення автоматизованих систем

КУРСОВА РОБОТА з дисципліни "Програмування та алгоритмічні мови" НА ТЕМУ: «Гра Теніс»

Студента _____ 1 _____ курсу, групи _____ КН-19
напряму підготовки "Інформаційні технології"
спеціальності "Комп'ютерні науки"
_____ Літвінова Р.Р.

Керівник _____ кандидат технічних наук,
старший викладач кафедри ІТ Царик Т.Ю.
Національна шкала: _____
Кількість балів: _____ Оцінка: ECTS _____

Члени комісії

(підпис)

(прізвище та ініціали)

(підпис)

(прізвище та ініціали)

(підпис)

(прізвище та ініціали)

Черкаси, 2020

Зміст:

Вступ.....	3
Розділ 1. Правила та умови гри. Джерела знань	4
Розділ 2. Проектування гри	6
2.1. Основні сцени	6
2.2. Головне меню	7
2.3. Ігрове поле.....	9
Розділ 3. Реалізація.....	14
3.1. Меню.....	14
3.2. Ігрове поле.....	16
Розділ 4. Висновки	20
Розділ 5. Список літератури	21

Вступ

Вибираючи тему для курсової роботи я відразу звернув увагу на ігри. На мою думку це чудова тема тому, що:

- Високий рівень інтерактивності
- Можливість практичного застосування навичок з геометрії та фізики
- Можливість розробки хоча б примітивного ШІ (штучного інтелекту)

Найвідоміший комп'ютерний варіант настільного тенісу (Понг) вперше був розроблений Аллоном Алкорном в 1972 р. як тренувальний проект в компанії Атарі. Екран був розділений навпіл вертикально, тож гравці розміщувались зліва та справа. Це було більш зручно, адже ця гра була створена для 2 гравців, які стояли поруч біля ігрового автомату. Свою гру я створював по мотивам Понгу, але вирішив, що краще буде розділити екран горизонтально, тому що передбачається гра лише проти ШІ.

Отже, мета роботи – зробити якісний пінг понг, побудований на основі фізичних взаємодій об'єктів (платформ гравця та супротивника, стін та самого м'яча). Хоча й фізика гри досить проста, але все ж для того, щоб зробити якісну і безвідмовну систему фізичних взаємодій може знадобитися багато часу. Тож я вирішив, що буду робити гру на ігровому движку. Є багато сучасних движків, але я вибрав Godot Engine через 3 причини: по-перше, цей движок повністю безкоштовний та з відкритим кодом; по-друге, я маю досвід роботи з ним, по-третє, він має можливість крос-компіляції для майже всіх сучасних ОС, в тому числі мобільних при цьому не жертвуючи швидкістю.

Розділ 1. Правила та умови гри. Джерела знань

Правила пінг понгу досить прості:

- 1) На ігровому полі розміщені 2 горизонтальні платформи гравців з верхнього та нижнього краю вікна та 2 вертикальні статичні стіни з лівого та правого краю вікна, через які м'яч та гравці не можуть пройти.
- 2) Якщо м'яч залітає у "ворота" суперника, гравцю на протилежній частині поля зараховують 1 очко і починається новий раунд.
- 3) Ціль гри набрати більше очок ніж суперник-штучний інтелект.
- 4) Гра не має обмежень ні по часу ні по очкам.

Я вирішив, що гра буде мати 4 рівня складності, від Easy (легкий) до Insane (безумно важкий). Рівні складності будуть впливати на швидкість пересування суперника.

Так як гра по суті дуже проста, шукати прототипи мені не знадобилось, я сконцентрувався на тому, як гра відчувається, щоб при потребі змінити потрібні параметри.

Godot Engine, як і більшість сучасних ігрових движків включає в себе фізичний движок, який здатен симулювати об'єкти у відповідності до їхніх параметрів. Також по стандарту він передбачає гравітацію, але пінг понг відбувається у невагомому середовищі, тож ця функція не знадобиться.

Godot Engine має архітектуру "сцен" в яких розміщуються "ноди", складові ігрової сцени, які можуть вміщувати в себе інші ноди та сцени. Завдяки цьому вибудовується чітка структура кожної сцени. За допомогою цього буде дуже легко перетворити правила гри на саму сцену цієї гри.

Як і будь-якої хорошої гри у мого пінг понгу буде головне меню. Воно буде відкриватися при запуску гри та дасть змогу провести первинне налаштування рівня складності та будь яких інших параметрів які буде потреба налаштувати.

Також, авжеж, повинна бути кнопка для запуску основної ігрової сцени та кнопка для виходу із гри.

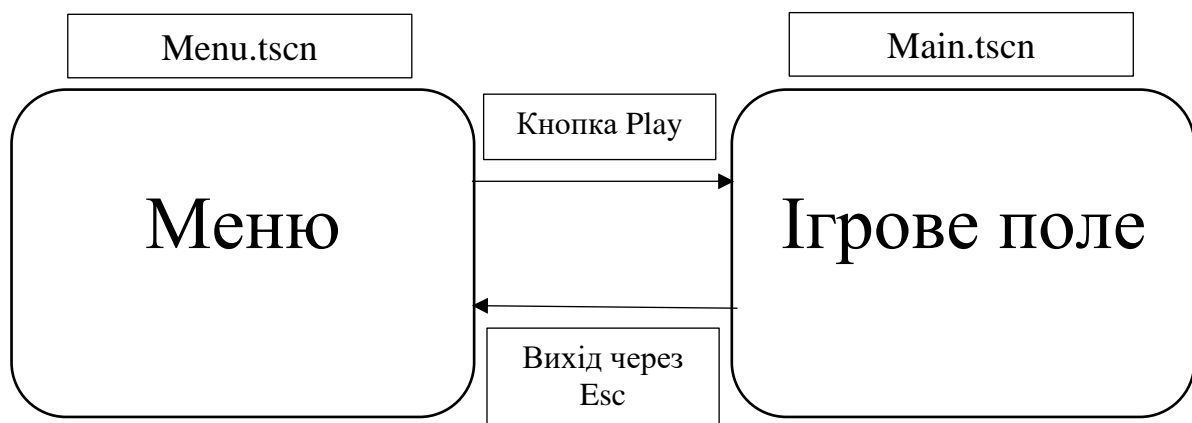
Усі знання по Godot Engine, зокрема по GDScript, мови програмування, яка використовується для написання скриптів (логіки) гри, я черпав з офіційної документації <https://docs.godotengine.org/en/stable/index.html> а також форуму <https://godotengine.org/community>.

Розділ 2. Проектування гри

2.1. Основні сцени

Перш за все потрібно визначити основні сцени гри. Виходячи з правил, ігрова сцена буде лише 1 тому, що не передбачається будь-яких змін у розміщенні елементів сцени (стін, гравців, тощо). Окрім основної ігрової сцени, повинна також бути сцена головного меню.

Ці 2 сцени будуть мати двосторонній зв'язок:



Через те, що гра повинна виглядати добре на всіх моніторах, проектування інтерфейсу дещо ускладняється. Поки відношення сторін вікна зберігається, можна все просто пропорційно розтягувати і все буде виглядати добре, але, як тільки відношення сторін зміниться, всі елементи будуть виглядати сплюсненими, або розтягнутими. Тож потрібно застосувати динамічні методи компоновки інтерфейсу. Для меню і ігрового поля вони будуть різними.

2.2. Головне меню

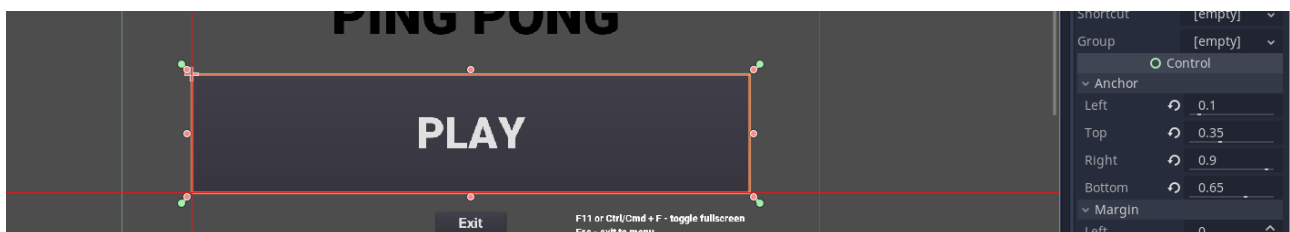
У розділі 1 вже визначено основні задачі, поставлені перед головним меню, підсумовуючи:

- 1) Кнопка для початку гри
- 2) Кнопка для виходу із гри
- 3) Кнопки для налаштування рівня складності

Але, все ж, для завершення композиції варто додати назву гри і підпис автора.

Також я додав перемикач режиму вертикальної синхронізації (VSync), який програмним чином синхронізує частоту кадрів (FPS) із частотою оновлення дисплею. Це значно знижує використання ресурсів ЦП (центрального процесора) і ГП (графічного процесора), що в свою чергу знижує енергоспоживання акумулятора на ноутбуках. Також VSync зменшує кількість розривів зображення, які виникають якраз завдяки відмінності FPS і частоти оновлення монітора.

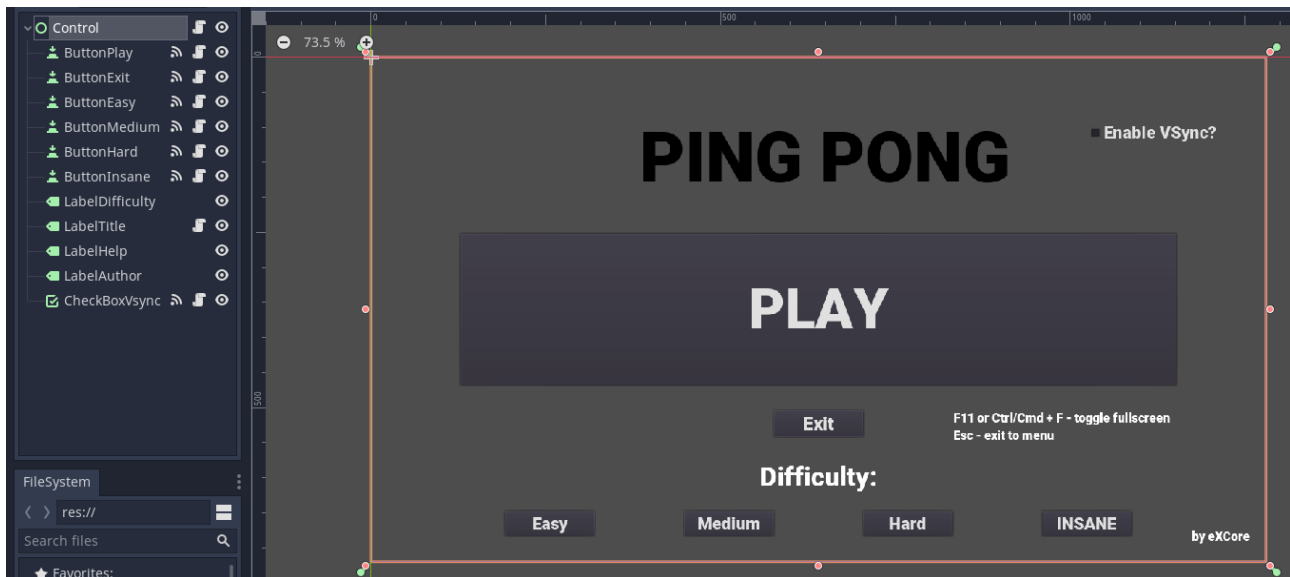
Так як я поставив задачу робити компоновку інтерфейсу динамічно, я буду використовувати можливість Godot Engine, яка називається Anchors (якорі). Вони дозволяють задавати положення і розмір елементів інтерфейсу не у абсолютних координатах, а у відносних, від 0 до 1.



Наприклад для кнопки "Play": лівий якір дорівнює 0,1, що означає що кнопка буде починатися зліва на $(0,1 * \text{розмір вікна по горизонталі})$ пікселі. Правий який дорівнює 0,9, що означає, що кнопка закінчується справа на $(0,9 * \text{розмір вікна по горизонталі})$ пікселі.

Таким чином можна компоувати елементи незалежно від розмірів вікна. Єдине обмеження, яке варто поставити – мінімальні висоту і ширину вікна, щоб у ньому могли розміститися усі елементи. На жаль, у поточній версії Godot Engine відсутня така стандартна можливість, тож її треба реалізувати вручну.

Отже дизайн головного меню в цілому буде виглядати так:



Зліва знаходиться панель дерева сцени. У кожної сцени повинен бути один "корінний" елемент, у даному випадку Control, який є базовим для всіх елементів інтерфейсу, в даному випадку ніяких функцій крім кореня не виконує. Розміщені кнопки про які написано на початку підрозділу а також допоміжні написи для ознайомлення з гарячими клавішами.

Для позначення рівня складності, який вибраний у даний момент я буду використовувати різні кольори шрифту на кнопках. Усі крім вибраного будуть мати білий колір, вибраний буде мати колір в залежності від вибраного рівня, для кращого розуміння. Наприклад Easy – зелений, Hard – помаранчевий.

Для перемикача VSync я вибрав елемент CheckBox, який по стандарту має 2 стани – увімкнено та вимкнено.

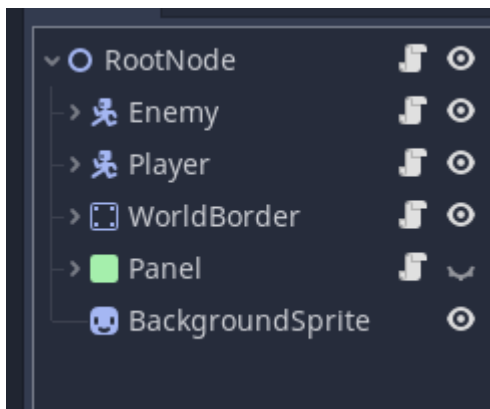
Для назви гри я планую зробити плавне переливання кольорів, для деякої динаміки в меню.

2.3. Ігрове поле

Завдяки архітектурі сцен, дуже легко перетворити правила гри у дизайн, і з нього у структуру сцени. Якщо представити собі те, що бачить перед собою гравець, то ці елементи і будуть складати основні ноди сцени. Отже гравець має бачити перед собою:

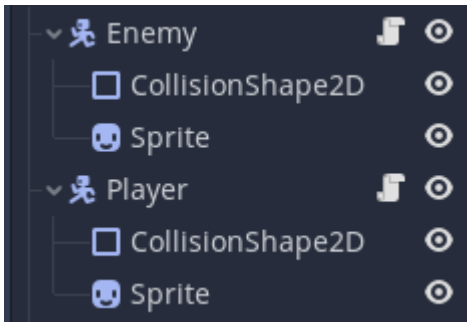
- 1) Платформу свого суперника
- 2) Свою платформу
- 3) М'яч
- 4) Стіни
- 5) Ворота
- 6) Задній фон


Для зручності я об'єднав 3-ій і 4-ий пункти у 1 – границю.



У дереві цієї сцени є все, крім м'яча, тому що він з'являється через деякий час після початку гри (для зручності). 1 додатковий елемент – це панель для відображення написів, зокрема рахунку та відліку початку нового раунду.

Платформи гравця та суперника майже однакові, за винятком кольору.

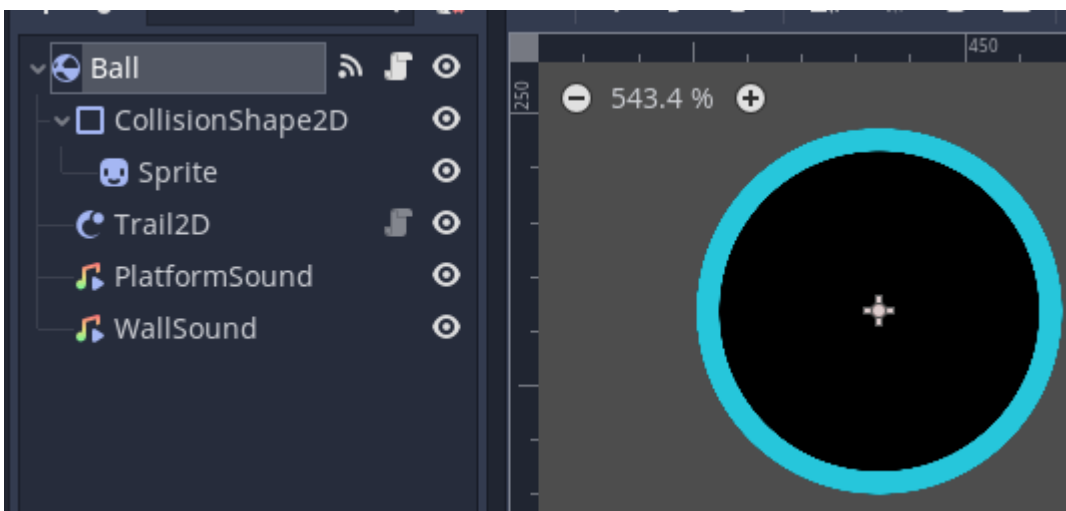



Тут з'являється перший фізичний об'єкт: **KinematicBody2D** . Він не має фізичних характеристик крім розміру, весь його рух повинен описуватися вручну.

CollisionShape2D тут і далі – фізична форма об'єкту, невидима при грі, у даному випадку має форму капсули (2 протилежні сторони – прямі, 2 інші – півкола).

Sprite тут і дали – видима форма об'єкта, у більшості випадків текстури. В моєму випадку у форматі ".png".

М'яч виконано у вигляді окремої додаткової сцени, тому що повинна бути можливість додавання його на поле з усіма компонентами (**Sprite**, **CollisionShape2D**, тощо).



М'яч має інший фізичний тип – **RigidBody2D** . Це фізичне тіло яке має більшість фізичних характеристик (масу, вагу, коефіцієнт тертя...) і повинно

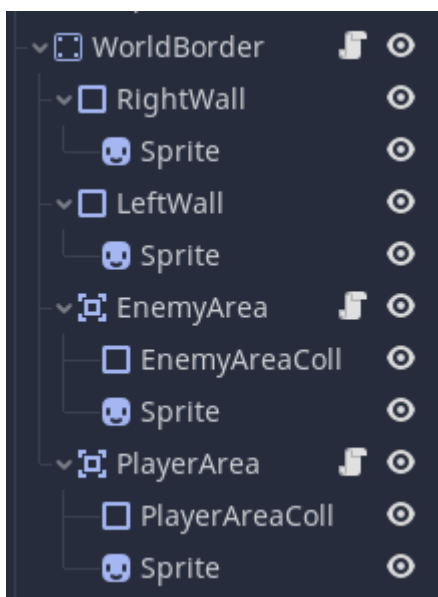
оброблятися фізичним движком (є можливість повністю самостійної обробки діючих фізичних сил, але в даному проєкті це не потрібно). На цей об'єкт можна діяти надаючи йому імпульси, але в даному випадку це не використовується тому, що движок самостійно їх обраховує при взаємодії з рухомим KinematicBody2D (гравцем, супротивником), або статичними стінами, про які буде мова далі.


CollisionShape2D тут має форму круга.

Trail2D – безкоштовне розширення (add-on), створене іншим розробником, за допомогою якого можна створювати анімовані "хвости".

Також м'яч має 2 ноди (AudioStreamPlayer2D) для відтворення звуків ударів об платформу (PlatformSound) і стіну (WallSound).

Що стосується границі (WorldBorder), то ось з чого вона складається:

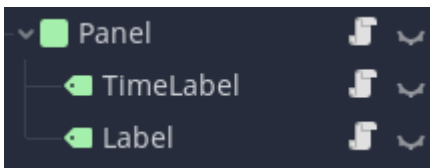


WorldBorder – StaticBody2D , фізичний нерухомий об'єкт який може взаємодіяти з RigidBody2D та KinematicBody2D.

RightWall, LeftWall – його CollisionShape2D, у вигляді прямокутників.

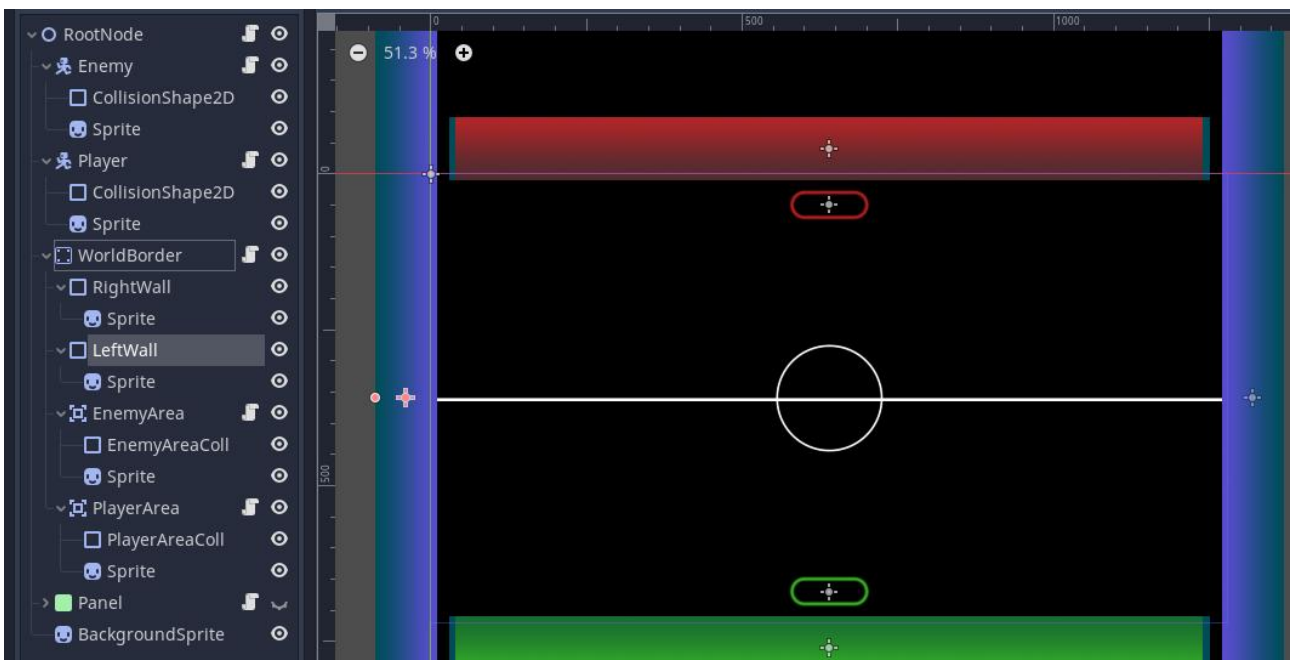
EnemyArea, PlayerArea – новий тип Area2D, який не взаємодіє фізично, але реєструє тіла що входять та виходять з його CollisionShape2D.

Остання група – Panel.



Panel – елемент що надає місце та задній фон об'єктам які можуть показуватись на ньому.

Отже в цілому дизайн поля з прихованою панеллю:



З показаною:



У цій сцені динамічну компоновку доведеться реалізовувати вручну, за допомогою формул.

Розділ 3. Реалізація

Godot Engine підтримує багато мов, але основними вважаються C++ та GDScript, та C# для Godot Mono. GDScript – строго типізована мова високого рівня за синтаксисом схожа на мову Python. Була створена спеціально для цього движка і оптимізована під архітектуру сцен. На цій мові я і буду реалізувати логіку своєї гри.

3.1. Меню

GDScript реалізує подійно-орієнтовану парадигму програмування. Отже для обробки таких дій як натискання кнопки, перемикання CheckBox, тощо, треба використовувати обробники подій.

Для всіх кнопок я обробляю подію `pressed()`, що означає, що кнопка натиснена.

Для кнопки Play обробник простий – змінити сцену на ігрове поле і обнулити рахунок.

Для кнопки Exit ще простіший – закрити програму.

Усі кнопки рівнів складності прив'язані до 1 скрипту. Функція `_ready()` виконується після ініціалізації об'єкту, в ній за допомогою конструкції `match` (аналог `switch`) виставляється колір шрифту в залежності від імені кнопки та того, чи вибрана складність яка відповідає цій кнопці.

Обробник `pressed()` кнопок рівнів складності складається з трьох `match`: в першому виставляється нова складність в залежності від імені кнопки, в другому оновлюється (викликається `_ready()`) кнопка попередньої складності (для зміни кольору), в третьому оновлюється кнопка яка була натиснена.

Остання цікава функція в меню – переливання кольору напису. Вона реалізована дуже просто:

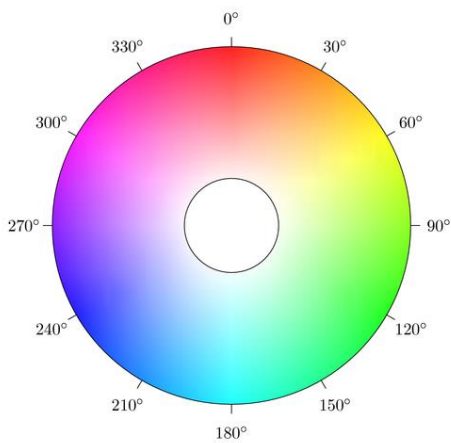
```

var i = 0.0

func _process(delta):
>I  var color = Color.from_hsv(i, 1, 1)
>I  i += delta * 0.33
>I  if i > 1:
>I    i = 0
>I  self.add_color_override("font_color", color)

```

`_process()` викликається при кожному новому кадрі. Параметр `delta` – час у секундах який пройшов з моменту попереднього кадру. Таким чином можна реалізувати анімацію стабільну в часі, не залежно від кількості FPS. Отже, анімуючи параметр Hue з простору HSV, я отримую переливання кольорів.



круг Hue, Saturation, де Hue – в градусах, Saturation – відстань від центру до конкретного кольору.

Також у функції `_ready()` кореня сцени меню прив'язується обробник події `"size_changed"` – зміна розмірів вікна.

3.2. Ігрове поле

Перш за все при запуску сцени ініціалізуються всі об'єкти. Після цього починають виконуватись функції `_ready()` всіх об'єктів за правилом: для того щоб виконалась функція `_ready()`, вона повинна виконатись для всіх дітей даного об'єкту.

Отже спочатку виконується `_ready()` зон кожного гравця. В них для таймера відліку прив'язується callback (функція що викликається при закінченні чогось), в якій реалізована зміна написів рахунку і відліку часу до початку нового раунду.

Наступною виконується функція `_ready()` `WorldBorder`, в якій і реалізовано динамічне розміщення ігрових об'єктів:

```
1 extends StaticBody2D
2
3 func _ready():
4     $RightWall.position[0] = GlobalVars.currentResolution[0] + 40
5     $RightWall.position[1] = GlobalVars.currentResolution[1] / 2
6     $PlayerArea.position[0] = GlobalVars.currentResolution[0] / 2
7     $PlayerArea.position[1] = GlobalVars.currentResolution[1] + 40
8     $PlayerArea/PlayerAreaColl.shape.extents = Vector2(GlobalVars.currentResolution[0] / 2 - 20, 50)
9     var spritescalex = GlobalVars.currentResolution[0] / 1280
10    $PlayerArea/Sprite.scale = Vector2(0.41 * spritescalex, 0.2)
11    $EnemyArea.position[0] = GlobalVars.currentResolution[0] / 2
12    $EnemyArea/EnemyAreaColl.shape.extents = Vector2(GlobalVars.currentResolution[0] / 2 - 20, 50)
13    $EnemyArea/Sprite.scale = Vector2(0.41 * spritescalex, 0.2)
14
```

Як видно, всі значення `position`, тобто координати і `scale`, тобто коефіцієнти розмірів об'єкта прив'язані до `GlobalVars.currentResolution`, що являє собою `Vector2`, тип який містить 2 значення з плаваючою крапкою, і містить розміри вікна гри.

Аналогічно, `_ready()` гравця:

```
func _ready():
> position[0] = GlobalVars.currentResolution[0] / 2
> position[1] = GlobalVars.currentResolution[1] - 50
```

Соперник може мати різні рівні складності, тож його `_ready()` присвоює йому швидкість пересування залежно від цього:


```

7 ▾ func _ready():
8   ▸ DIFFICULTY = GlobalVars.currentDifficulty
9   ▾ ▸ match DIFFICULTY:
10  ▾ ▸ ▸ 0:
11    ▸ ▸ ▸ moveSpeed = 3
12  ▾ ▸ ▸ 1:
13    ▸ ▸ ▸ moveSpeed = 4
14  ▾ ▸ ▸ 2:
15    ▸ ▸ ▸ moveSpeed = 5
16  ▾ ▸ ▸ 3:
17    ▸ ▸ ▸ moveSpeed = 6
18  # ▸ yCompensate = moveSpeed * 10
19  ▸ self.position[0] = GlobalVars.currentResolution[0] / 2

```

Після того як все це завершило роботу, виконується останній `_ready()` кореня сцени:

```

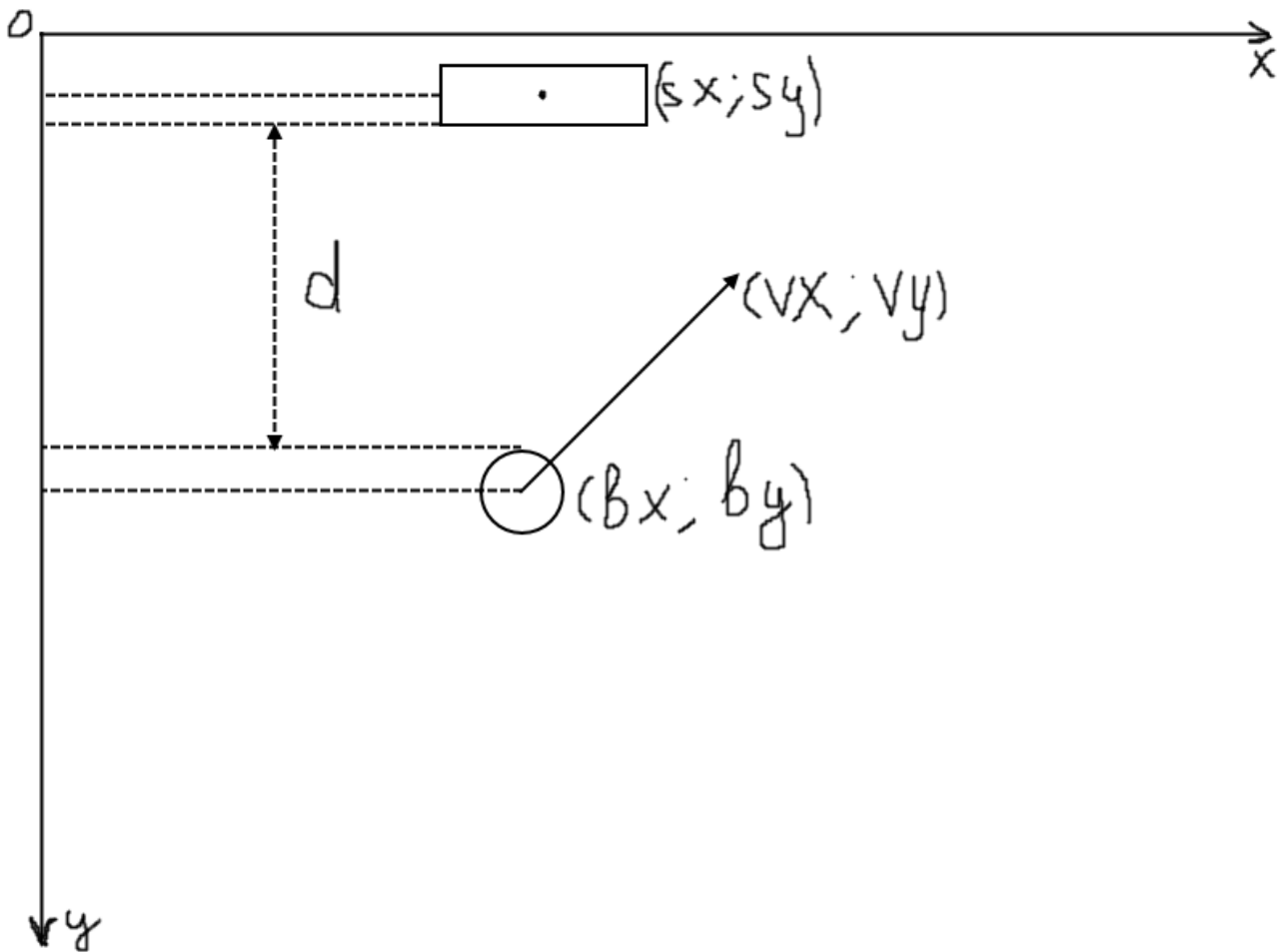
▾ func _ready():
▾ ▸ if firstDelay:
▸   ▸ firstDelay = false
▸   ▸ $WorldBorder/PlayerArea.count_down()
▸   ▸ var _con = get_viewport().connect("size_changed", self, "_on_resize")
▸   ▸ $BackgroundSprite.scale = Vector2(0.425*GlobalVars.currentResolution[0]/1280, 0.425*GlobalVars.currentResolution[0]/1280)
▸   ▸ $BackgroundSprite.position = GlobalVars.currentResolution / 2
▸   ▸ OS.vsync_enabled = GlobalVars.vSyncEnabled

```

`firstDelay` викликає таймер відліку при першому запуску сцени. Аналогічно сцені меню, приєднується обробник події `"size_changed"`. Обчислюються параметри заднього фону, присвоюється значення параметру `OS.vsync_enabled`, що перемикає режим VSync.

На цьому ініціалізація завершується і починає виконуватись основний ігровий цикл (Game Loop). На цьому етапі гравець намагається переграти противника.

На завершення хочу розповісти про "інтелект" суперника.



Задача інтелекту – перемістити свою платформу на те місце куди прилетить м'яч. Формально – знайти координату x перетину продовження нижнього краю платформи і продовження вектору швидкості м'яча прикладеного до верхнього краю м'яча.

Час t який знадобиться м'ячу для проходження відстані d дорівнює:

$$t = \frac{b_y - ball\ radius - s_y - capsule\ radius}{v_y}$$

За цей час м'яч переміститься по координаті x на:

$$d_x = v_x t$$

Тобто його кінцева координата:

$$x = b_x + v_x t$$

Це i є координата на яку треба перемістити платформу. Слід зазначити, що цей розв'язок не враховує відбиття від стін тому, що в такому випадку було б не можливо перемогти цього суперника.

```
var bPosx_predict = bPos[0] + ((bPos[1] - ballShape.radius - self.position[1]
>| >| >| >| >| >| >| >| - selfShape.radius) / (-bVel[1])) * bVel[0] # Predicting ball's intersection
bPosx_predict = clamp(bPosx_predict, 0, OS.window_size[0])
```

В коді все як і у формулі, додатково я обмежую цей прогноз краями екрану.

Розділ 4. Висновки

На даний момент гра є досить завершеною, режим гри із ШІ досить цікавий і підходить для гравців будь-якого рівня за рахунок вибору рівня складності.

Враховуючи реалізовані можливості у гру досить легко буде за потреби додати режим гри проти реальної людини, але мені бракує досвіду роботи з мережами, тож я цього не робив.

За рахунок реалізації на движку Godot Engine, ця гра є крос-платформною, за рахунок динамічної компоновки може працювати на моніторах с різними відношеннями сторін, а за рахунок досить простої та оптимізованої графіки та фізики може працювати на майже будь-якому сучасному комп'ютері. Також, на мою думку важливою є функція VSync, яка може бути використана на ноутбуках для зниження споживання енергії акумулятора.

Розділ 5. Список літератури

<https://docs.godotengine.org/en/stable/index.html>

<https://godotengine.org/community>