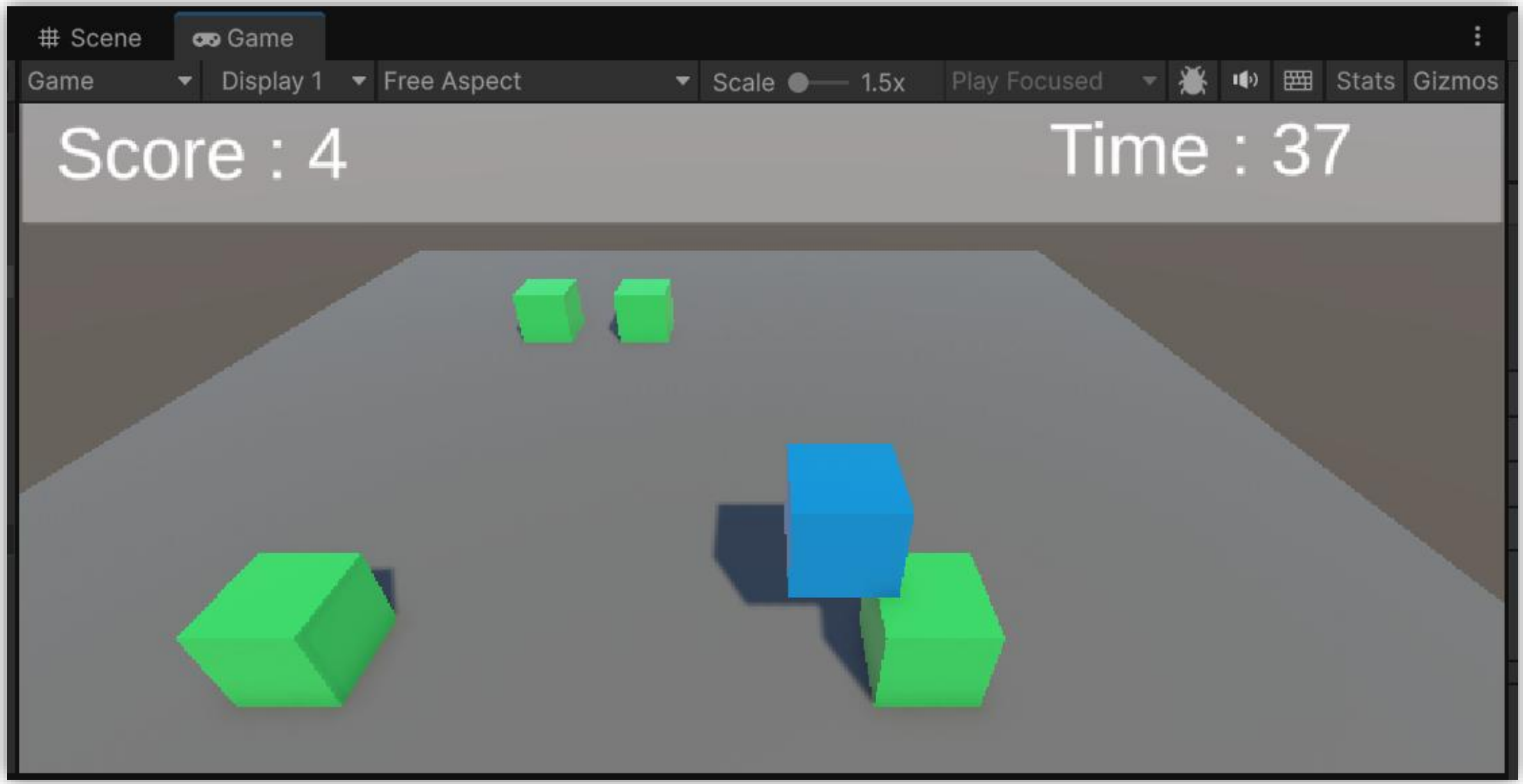
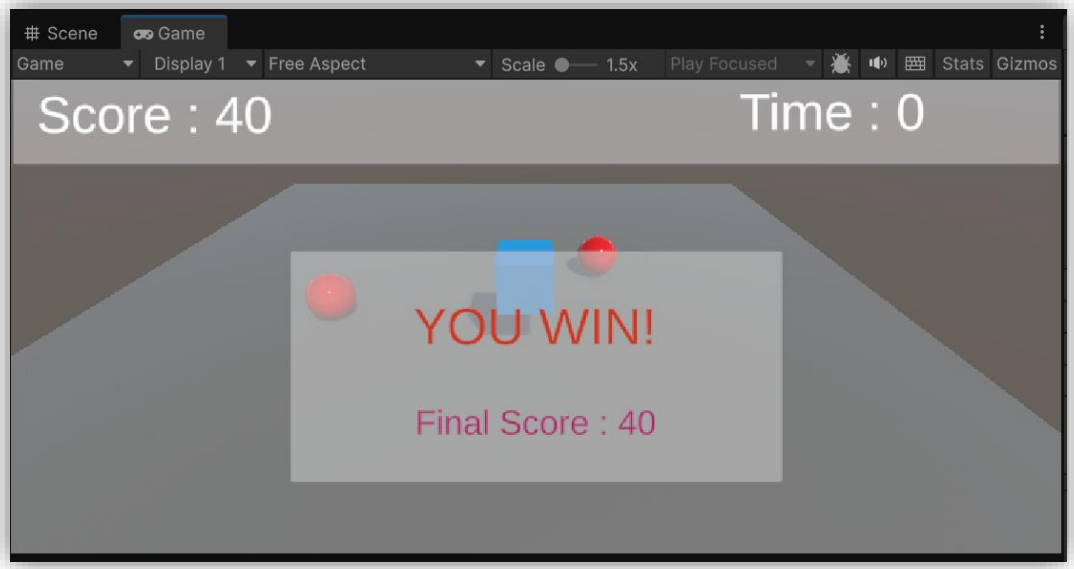
part
1

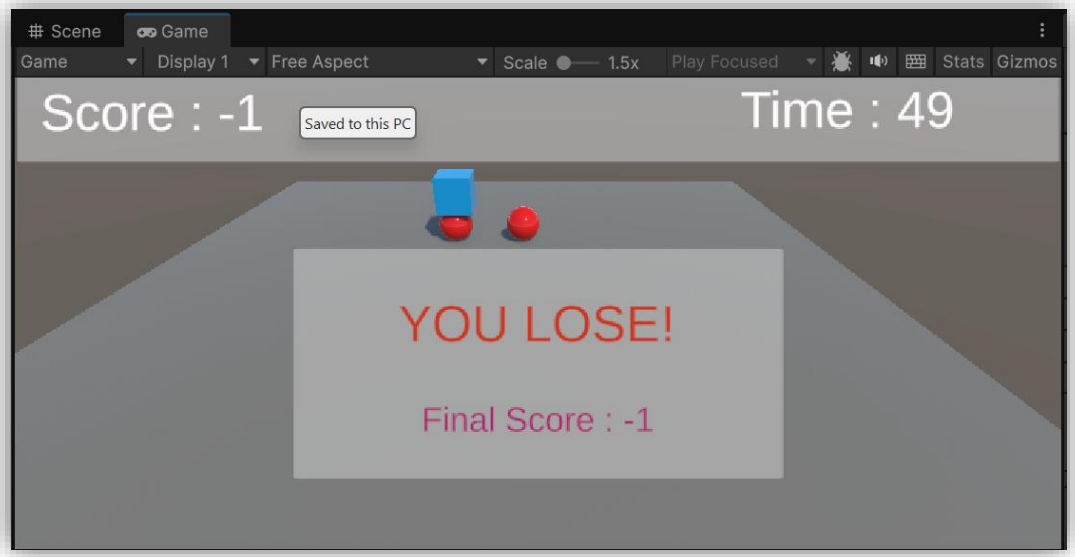
www.gamedevpbru.com

miniGameNo.2

ดนัย เจษฎาฐิติกุล
จารุต บุศราทิจ
DcG.IT@PBRU


Scene
Game
Game
Display 1
Free Aspect
Scale
1.5x
Play Focused
Stats
Gizmos
Score : 4
Time : 37


Scene
Game
Game
Display 1
Free Aspect
Scale
1.5x
Play Focused
Stats
Gizmos
Score : 40
Time : 0
YOU WIN!
Final Score : 40


Scene
Game
Game
Display 1
Free Aspect
Scale
1.5x
Play Focused
Stats
Gizmos
Score : -1
Saved to this PC
Time : 49
YOU LOSE!
Final Score : -1


Scene
Game
Game
Display 1
Free Aspect
Scale
1.5x
Play Focused
Stats
Gizmos
Score : 31
Time : 0
TIME UP
Final Score : 31

โครงงานนี้มีวัตถุประสงค์เพื่อศึกษาและประยุกต์ใช้หลักการพัฒนาเกมสามมิติ (3D Game Development) โดยมุ่งเน้นการออกแบบระบบเกมเชิงโครงสร้าง (Systematic Game Design) และการจัดการสถานะของเกมด้วยแนวคิด **Finite State Machine** (FSM)

เกมที่พัฒนาขึ้นคือเกม “**ตีตัวตุ่น**” (Whack-a-Mole) ซึ่งเป็นเกมประเภท Reaction Game ผู้เล่นต้องอาศัยความรวดเร็วและการตัดสินใจภายในระยะเวลาจำกัด โดยมีทั้งวัตถุที่ให้รางวัลและวัตถุที่ให้โทษ เพื่อเพิ่มความท้าทายและความสมดุลของเกม

ประเภทเกม

3D Reaction / Reflex Game

มุมมองกล้องแบบ Top-down หรือ Slight Isometric

วัตถุประสงค์

1. ออกแบบระบบเกมที่มีโครงสร้างชัดเจน
2. แยกความรับผิดชอบของแต่ละระบบอย่างเป็นระบบ
3. สามารถขยายหรือปรับปรุงได้ในอนาคต

ขอบเขตและเงื่อนไขของเกม (Game Specification)

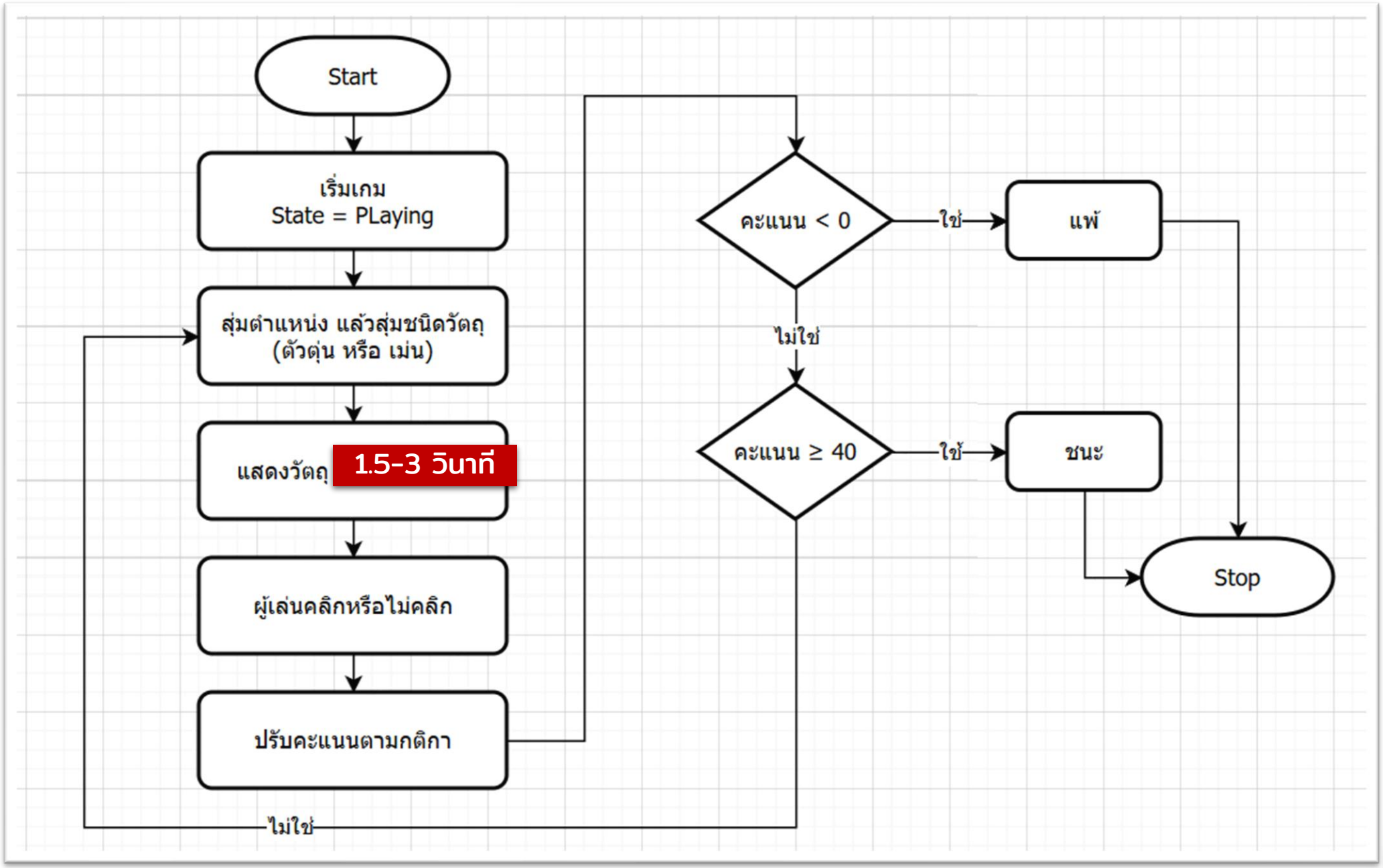
เกมตีตัวตุ่นสามมิตินี้มีเงื่อนไขและกติกาการเล่นดังต่อไปนี้

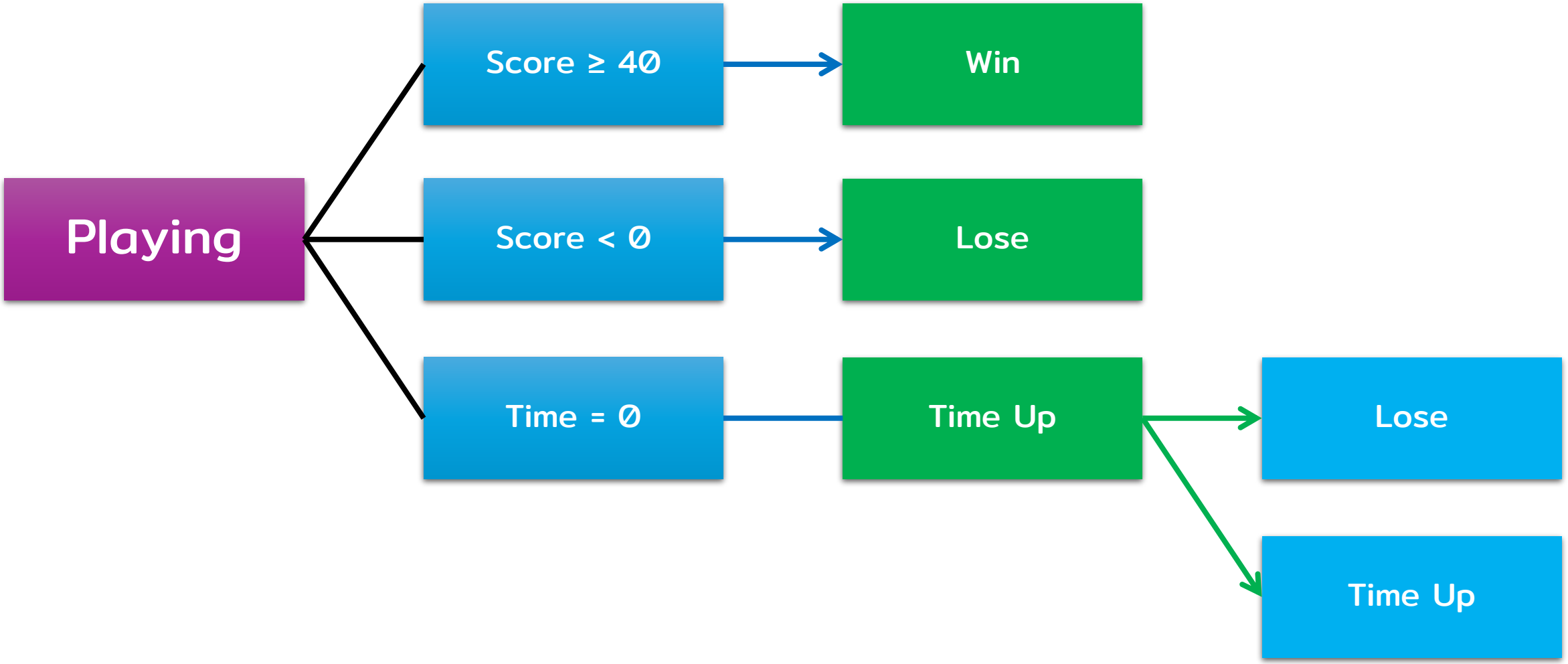
1. ใช้ Primitive ของ Unity แทนวัตถุในเกม เพื่อลดความซับซ้อนด้านกราฟิก
    1. ค้อน: Cube สีน้ำเงิน
    2. ตัวตุ่น: Cube สีเขียว
    3. เม่น: Sphere สีแดง
2. พื้นที่เล่นเป็นตารางขนาด 5×5 รวมทั้งหมด 25 ช่อง
3. ตัวตุ่นและเม่นจะแสดงผลในตำแหน่งแบบสุ่ม ครั้งละหนึ่งตำแหน่ง

4. ระยะเวลาแสดงผลของวัตถุอยู่ในช่วง 1.5 – 3.0 วินาที
5. ระบบคะแนน
    1. คลิกโดนตัวตุ่น: +1 คะแนน
    2. ตัวตุ่นหายไปโดยไม่ถูกคลิก: -1 คะแนน
    3. คลิกโดนเม่น: -1 คะแนน
    4. เม่นหายไปเอง: ไม่เปลี่ยนคะแนน

6. ระบบปรับระดับความยาก (Dynamic Difficulty)
    1. คะแนน ≥ 10: โอกาสเกิดเม่น 60%
    2. คะแนน ≥ 20: โอกาสเกิดเม่น 70%
    3. คะแนน ≥ 30: โอกาสเกิดเม่น 80%
7. เงื่อนไขแพ้–ชนะ
    1. คะแนน ≥ 40: ชนะเกม
    2. คะแนน < 0: แพ้เกม
8. การจัดการสถานะเกมใช้ Finite State Machine (FSM)

Game Flow


Start
เริ่มเกม
State = PLaying
สุ่มตำแหน่ง แล้วสุ่มชนิดวัตถุ
(ตัวตุ่น หรือ เม่น)
แสดงวัตถุ
1.5-3 วินาที
ผู้เล่นคลิกหรือไม่คลิก
ปรับคะแนนตามกติกา
ไม่ใช่
คะแนน < 0
ใช่
แพ้
ไม่ใช่
คะแนน ≥ 40
ใช่
ชนะ
Stop


Playing
Score ≥ 40
Win
Score < 0
Lose
Time = 0
Time Up
Lose
Time Up

การออกแบบโครงสร้างเกม
(Game Architecture Design)

ระบบเกมถูกแบ่งออกเป็นโมดูลหลัก 4 ส่วน

- Game State Management
- Object Spawning System
- Player Interaction System
- Score & Time Management System

การแบ่งระบบเช่นนี้ช่วยลดการพึ่งพาซึ่งกันและกัน (Loose Coupling) และเพิ่มความสามารถในการบำรุงรักษา (Maintainability)

GameManager

- ควบคุม Game State (FSM)
- จัดการคะแนน เวลา และเงื่อนไขชนะ/แพ้

GridManager

- สร้างตาราง 5x5
- ควบคุมการสุ่มเกิด “ตัวตุ่น / เม่น”

Spawnable Object

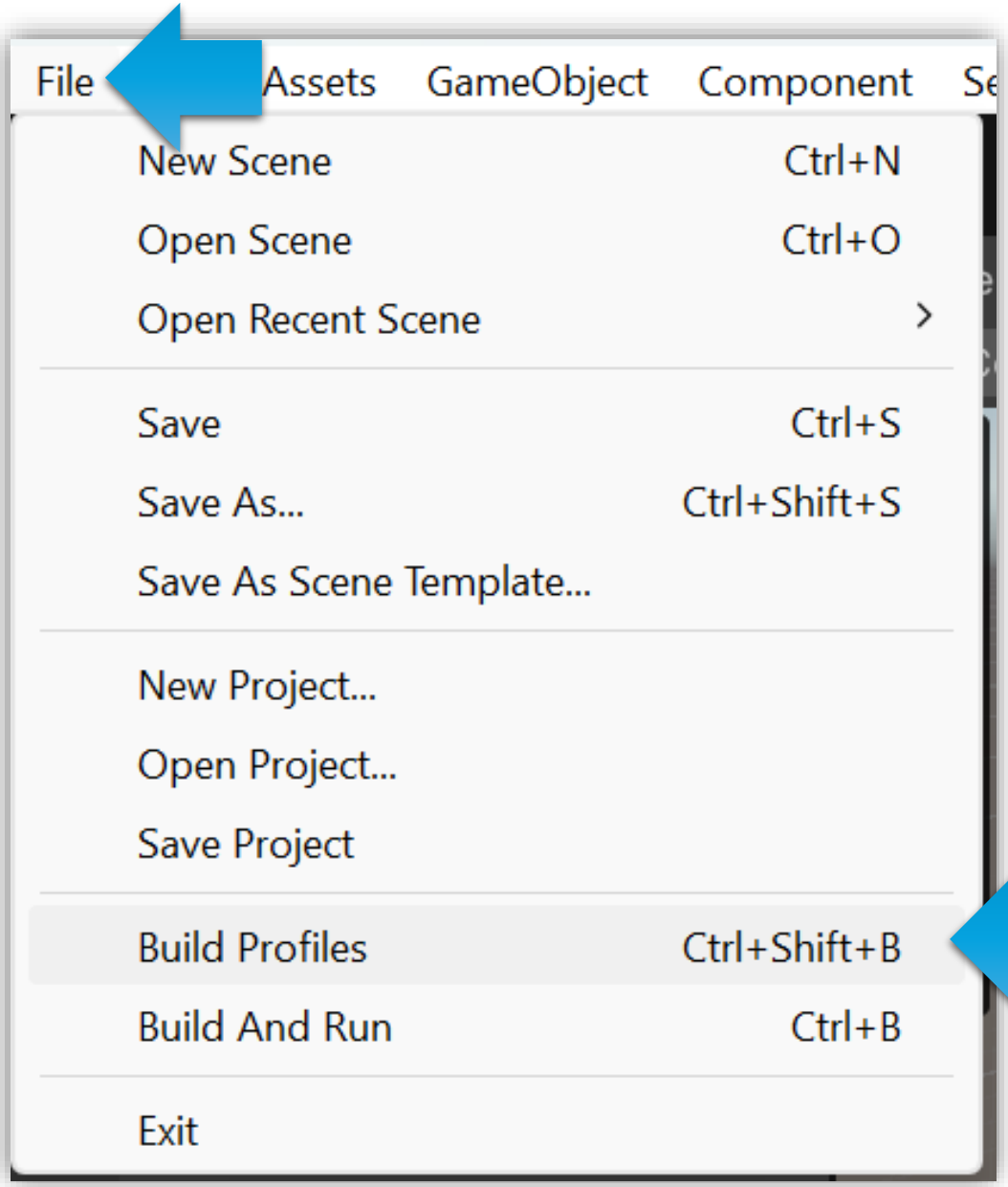
- Mole / Hedgehog
  - Cube สีเขียว = ตัวตุ่น
  - Sphere สีแดง = เม่น
- มีอายุการแสดงผล 1.5-3.0 วินาที

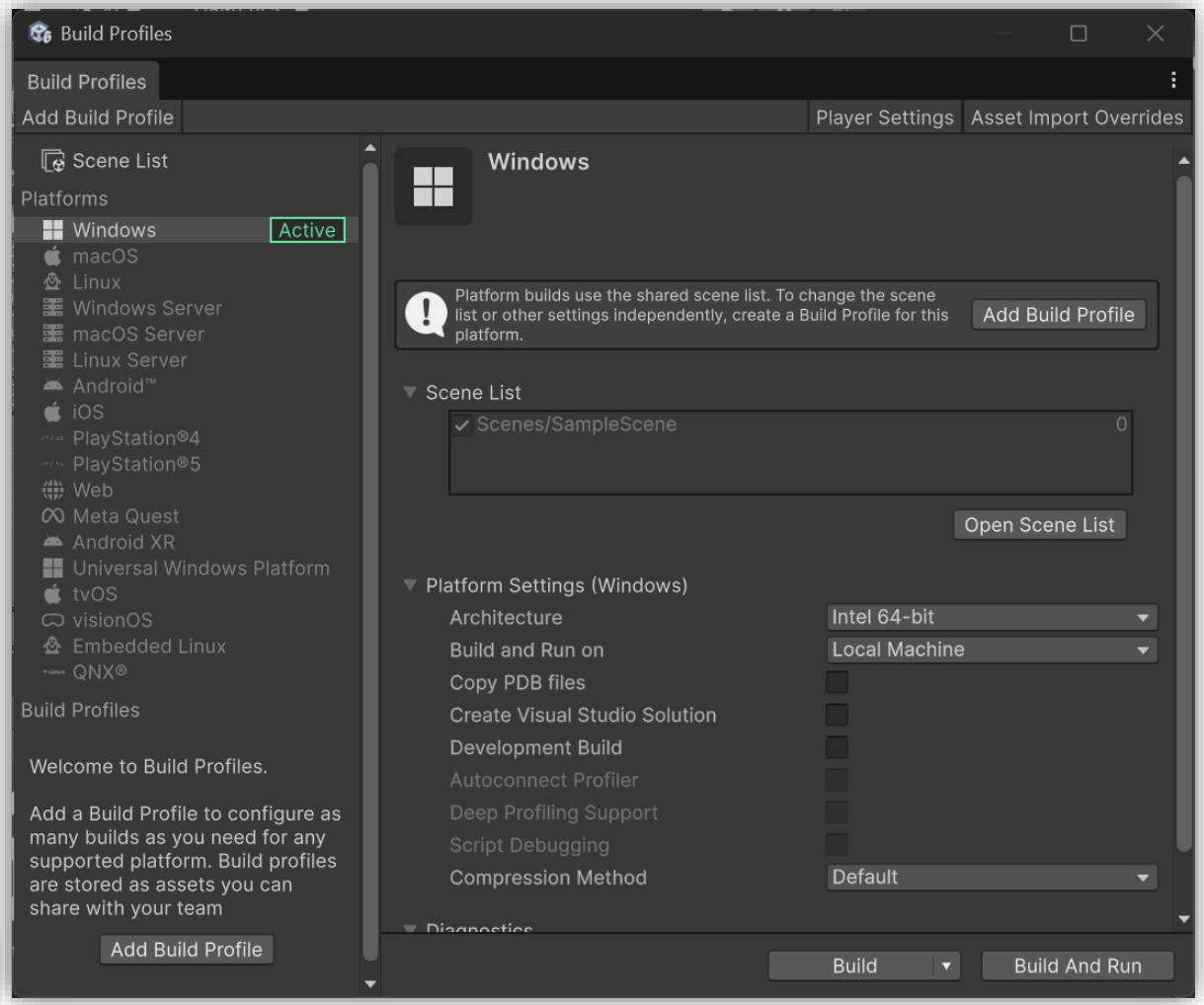
Player Interaction

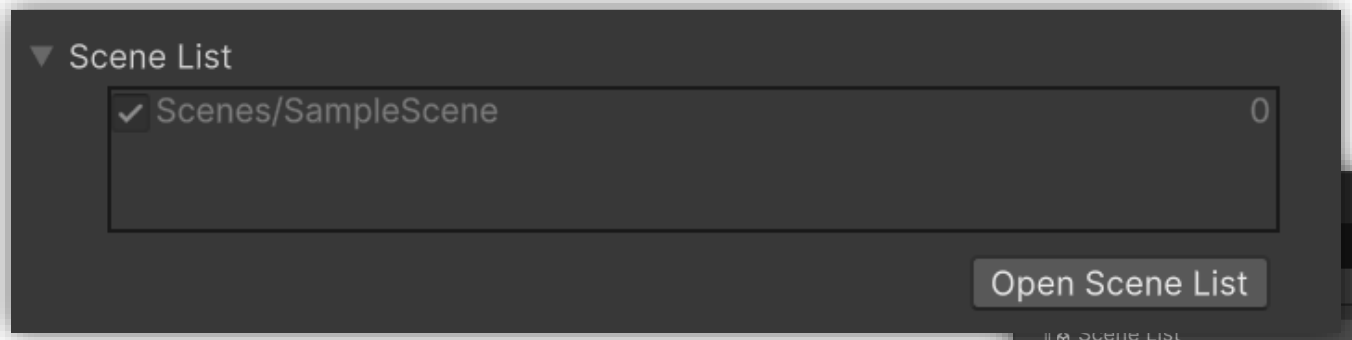
- ค้อน (Cube สีน้ำเงิน)
- ตรวจจับการคลิกด้วย Raycast

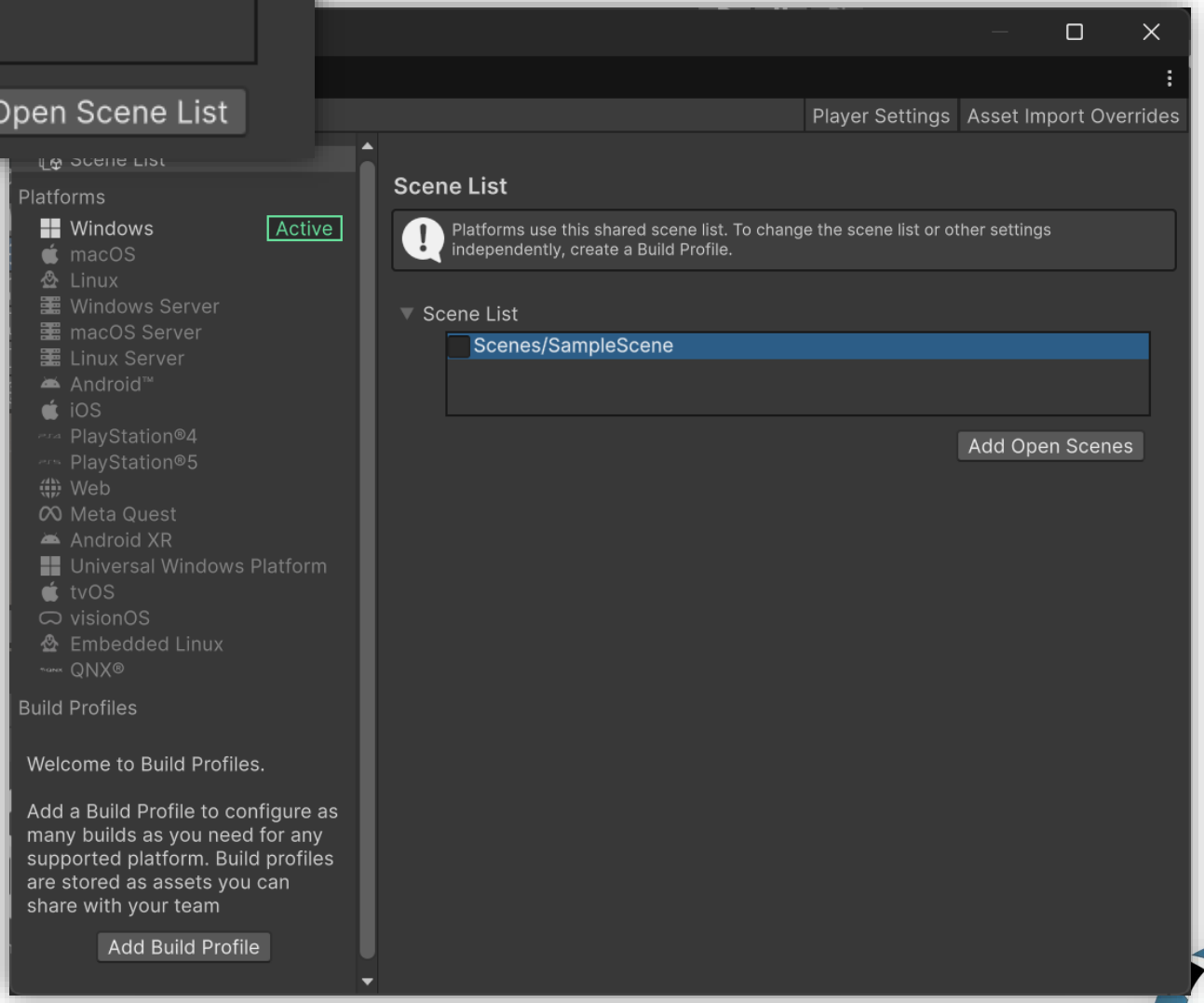
เตรียม Scenes

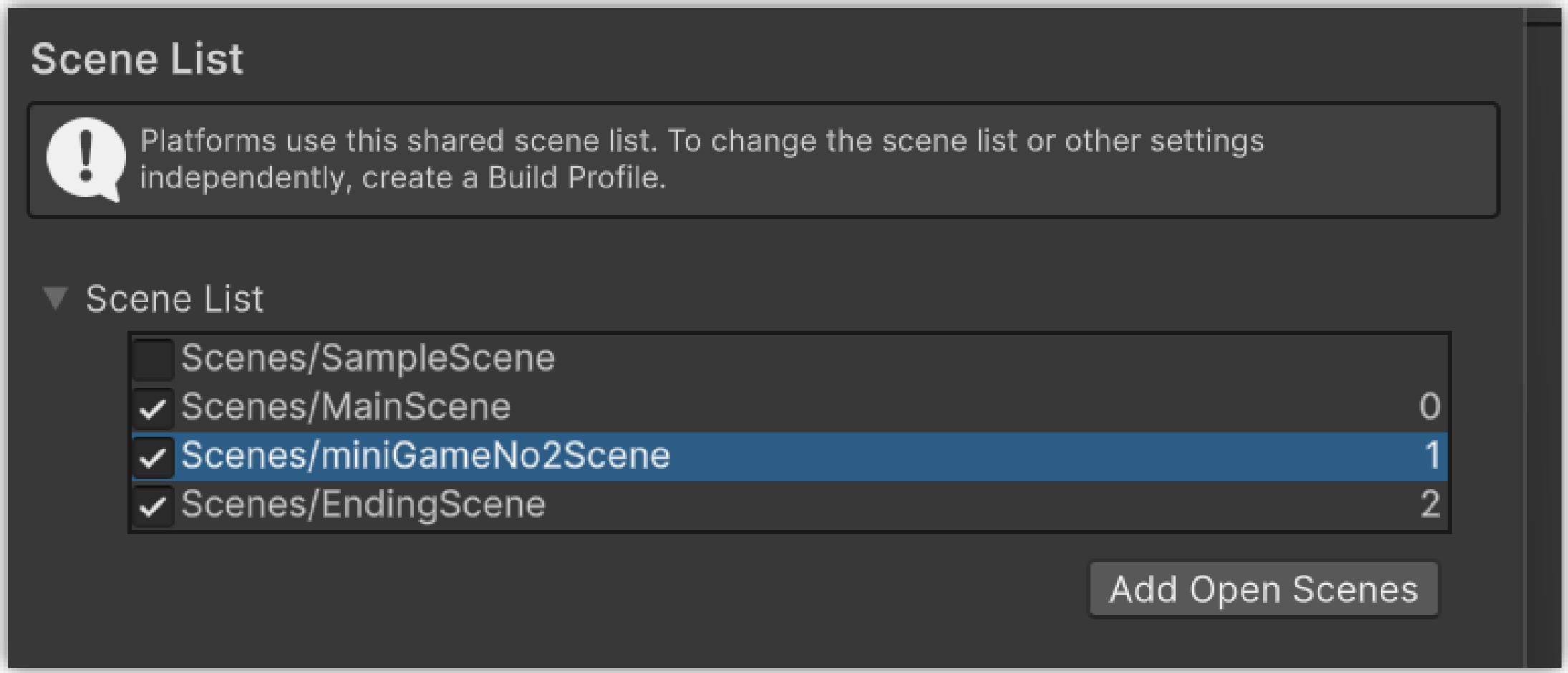

Assets > Scenes
EndingScene
MainScene
miniGameNo2Scene
SampleScene


File
Assets
GameObject
Component
New Scene Ctrl+N
Open Scene Ctrl+O
Open Recent Scene
Save Ctrl+S
Save As... Ctrl+Shift+S
Save As Scene Template...
New Project...
Open Project...
Save Project
Build Profiles Ctrl+Shift+B
Build And Run Ctrl+B
Exit

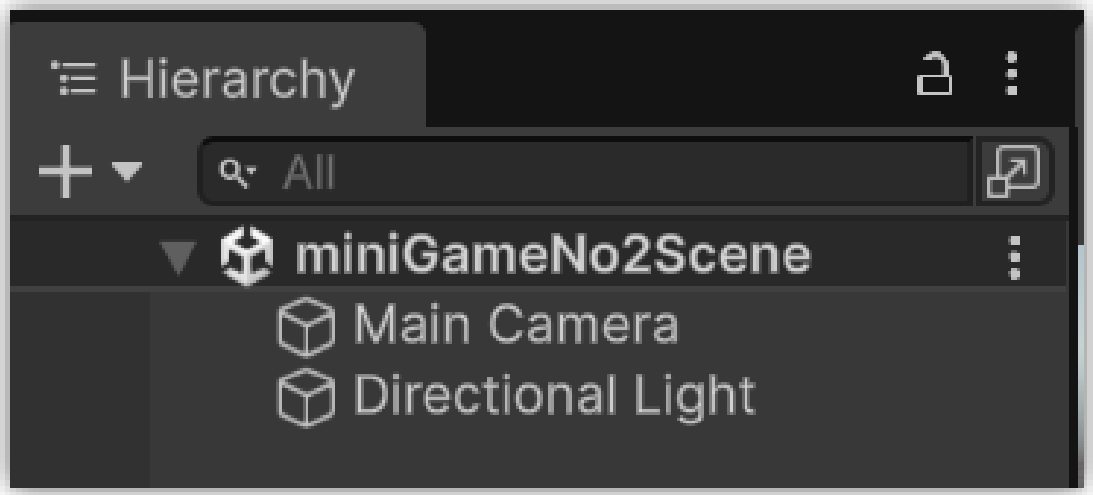

Build Profiles
Build Profiles
Add Build Profile
Player Settings
Asset Import Overrides
Scene List
Platforms
Windows
Active
macOS
Linux
Windows Server
macOS Server
Linux Server
Android™
iOS
PlayStation®4
PlayStation®5
Web
Meta Quest
Android XR
Universal Windows Platform
tvOS
visionOS
Embedded Linux
QNX®
Build Profiles
Welcome to Build Profiles.
Add a Build Profile to configure as many builds as you need for any supported platform. Build profiles are stored as assets you can share with your team
Add Build Profile
Windows
Platform builds use the shared scene list. To change the scene list or other settings independently, create a Build Profile for this platform.
Add Build Profile
Scene List
Scenes/SampleScene 0
Open Scene List
Platform Settings (Windows)
Architecture Intel 64-bit
Build and Run on Local Machine
Copy PDB files
Create Visual Studio Solution
Development Build
Autoconnect Profiler
Deep Profiling Support
Script Debugging
Compression Method Default
Build
Build And Run

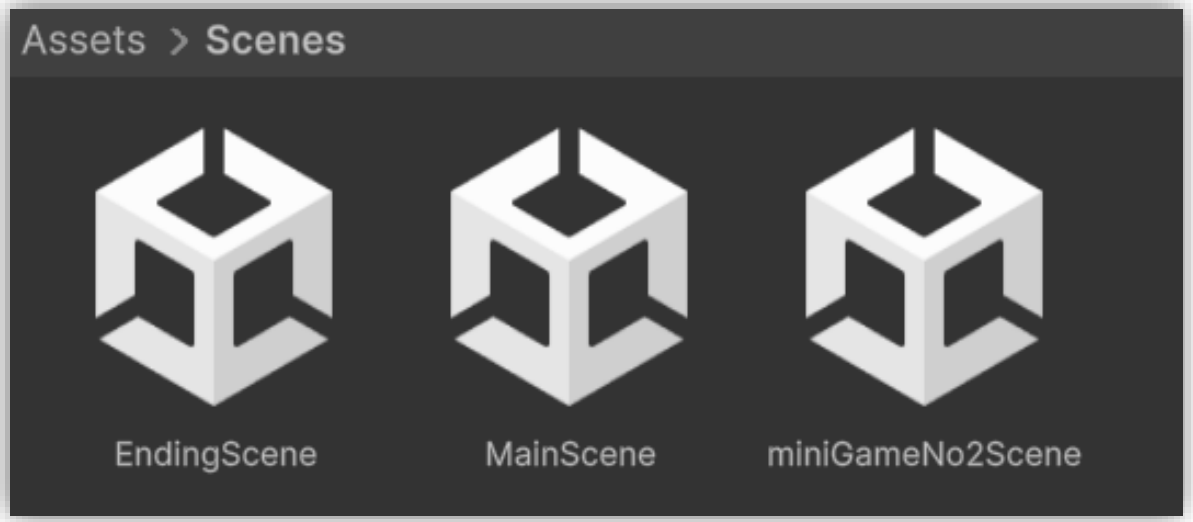

Scene List
Scenes/SampleScene
0
Open Scene List


Player Settings
Asset Import Overrides
Platforms
Windows
Active
macOS
Linux
Windows Server
macOS Server
Linux Server
Android™
iOS
PlayStation®4
PlayStation®5
Web
Meta Quest
Android XR
Universal Windows Platform
tvOS
visionOS
Embedded Linux
QNX®
Build Profiles
Welcome to Build Profiles.
Add a Build Profile to configure as many builds as you need for any supported platform. Build profiles are stored as assets you can share with your team
Add Build Profile
Scene List
Platforms use this shared scene list. To change the scene list or other settings independently, create a Build Profile.
Scene List
Scenes/SampleScene
Add Open Scenes

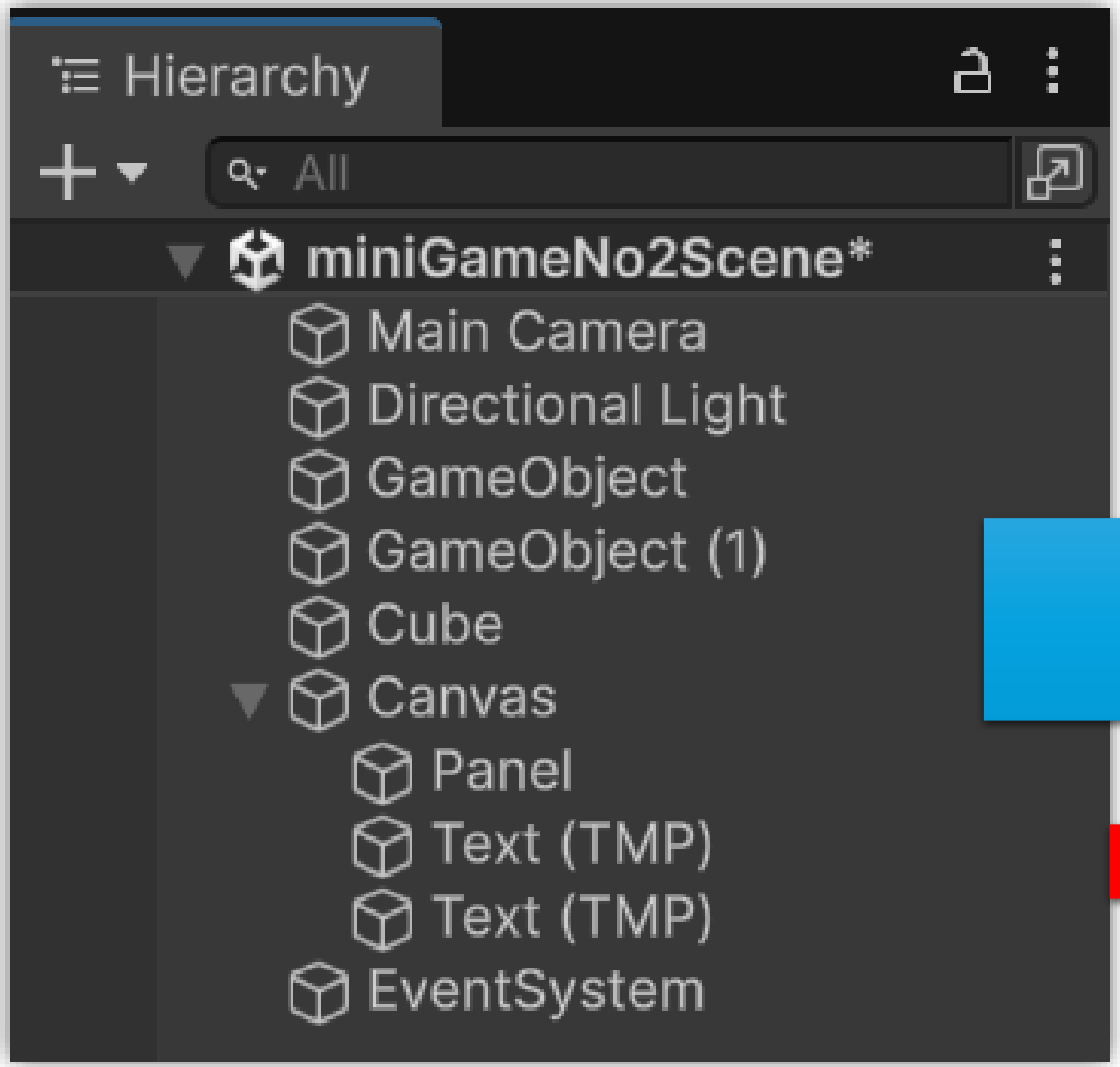


ให้เปิดแต่ละ Scene แล้วเพิ่มเข้า Scene List หลังจากนั้นคลิกเอาเครื่องหมายออกจาก Sample Scene แล้วจัดลำดับของฉาก

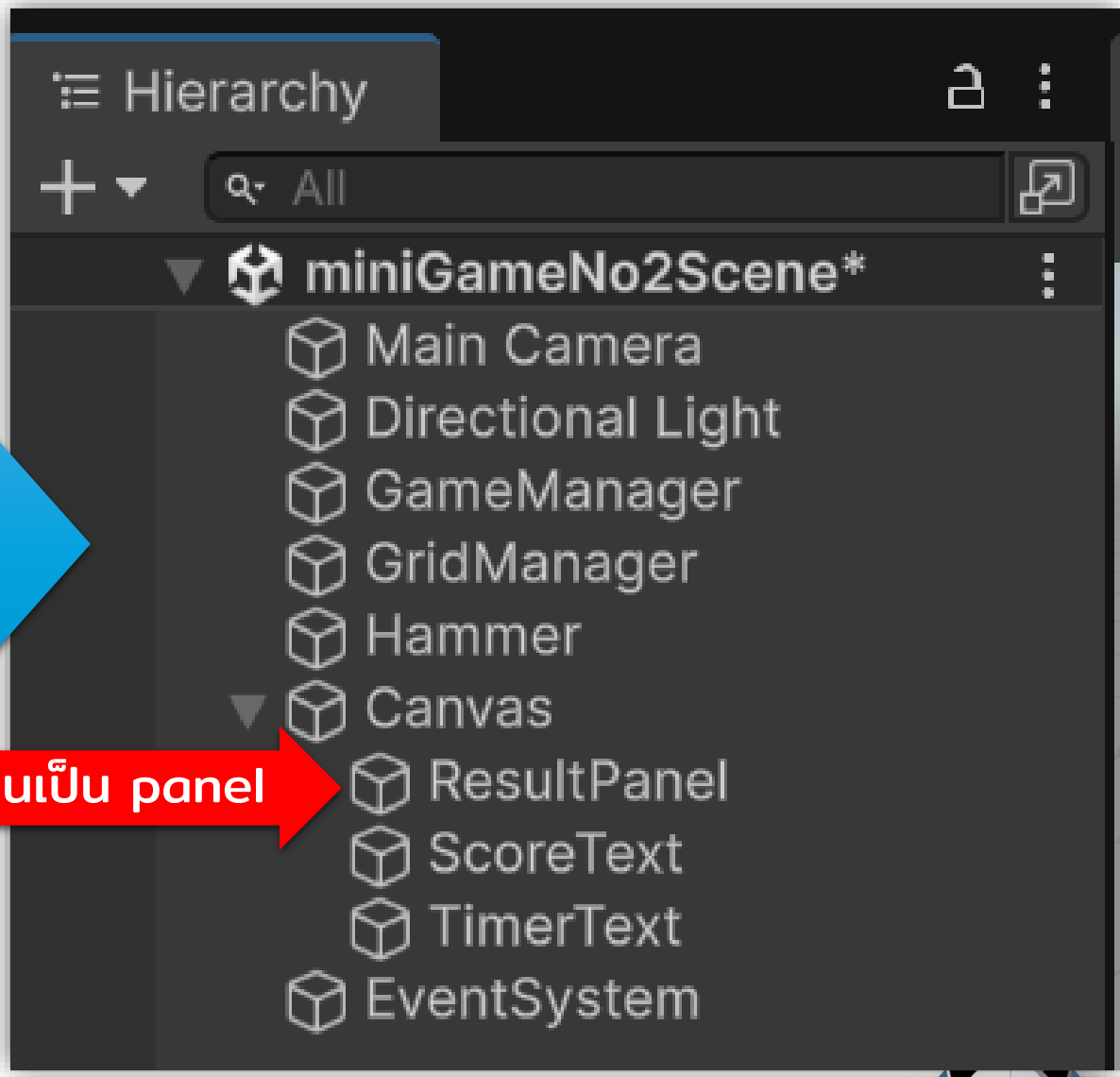


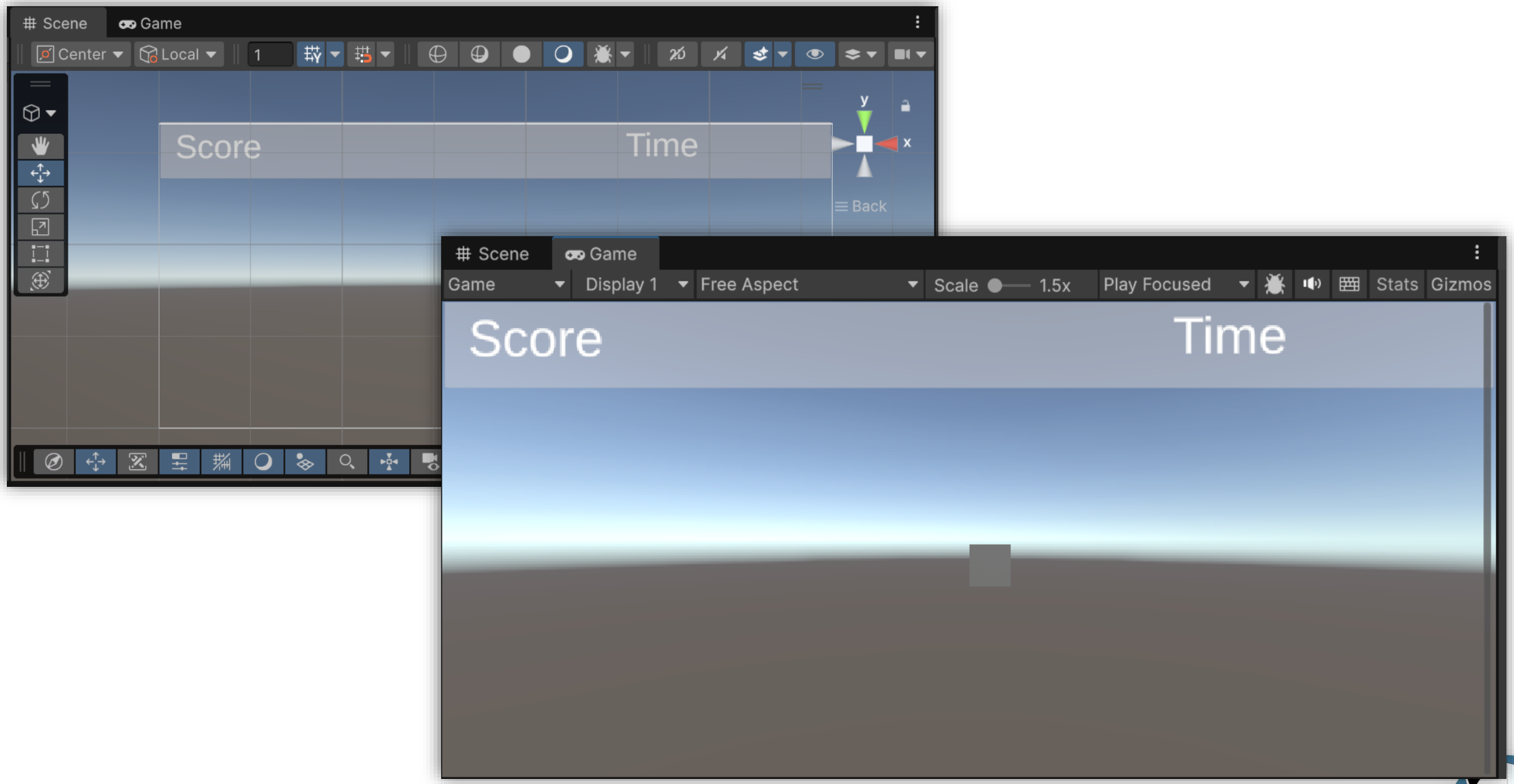


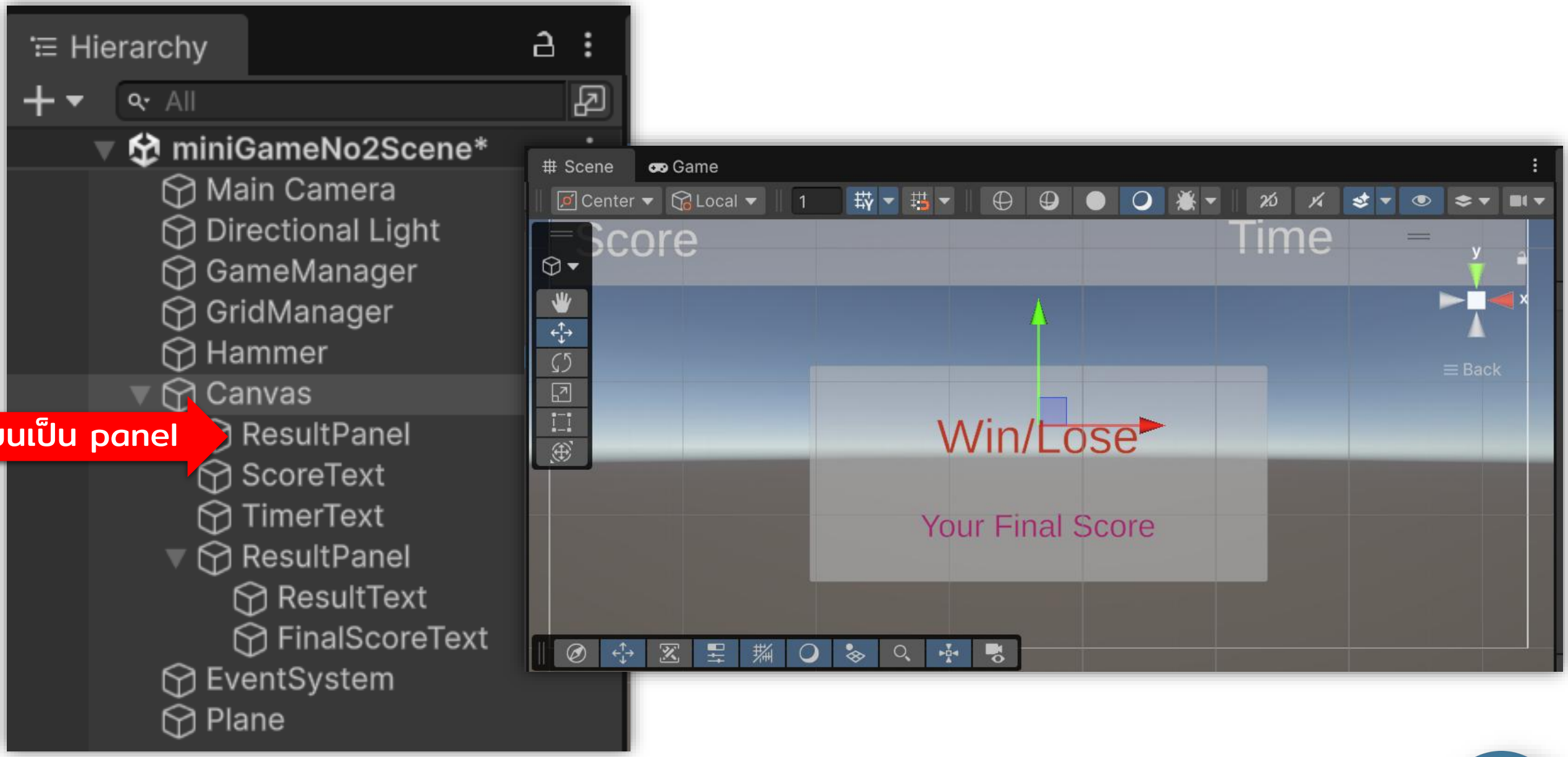
ลบ Sample Scene ออก และเปิด miniGameNo2Scene

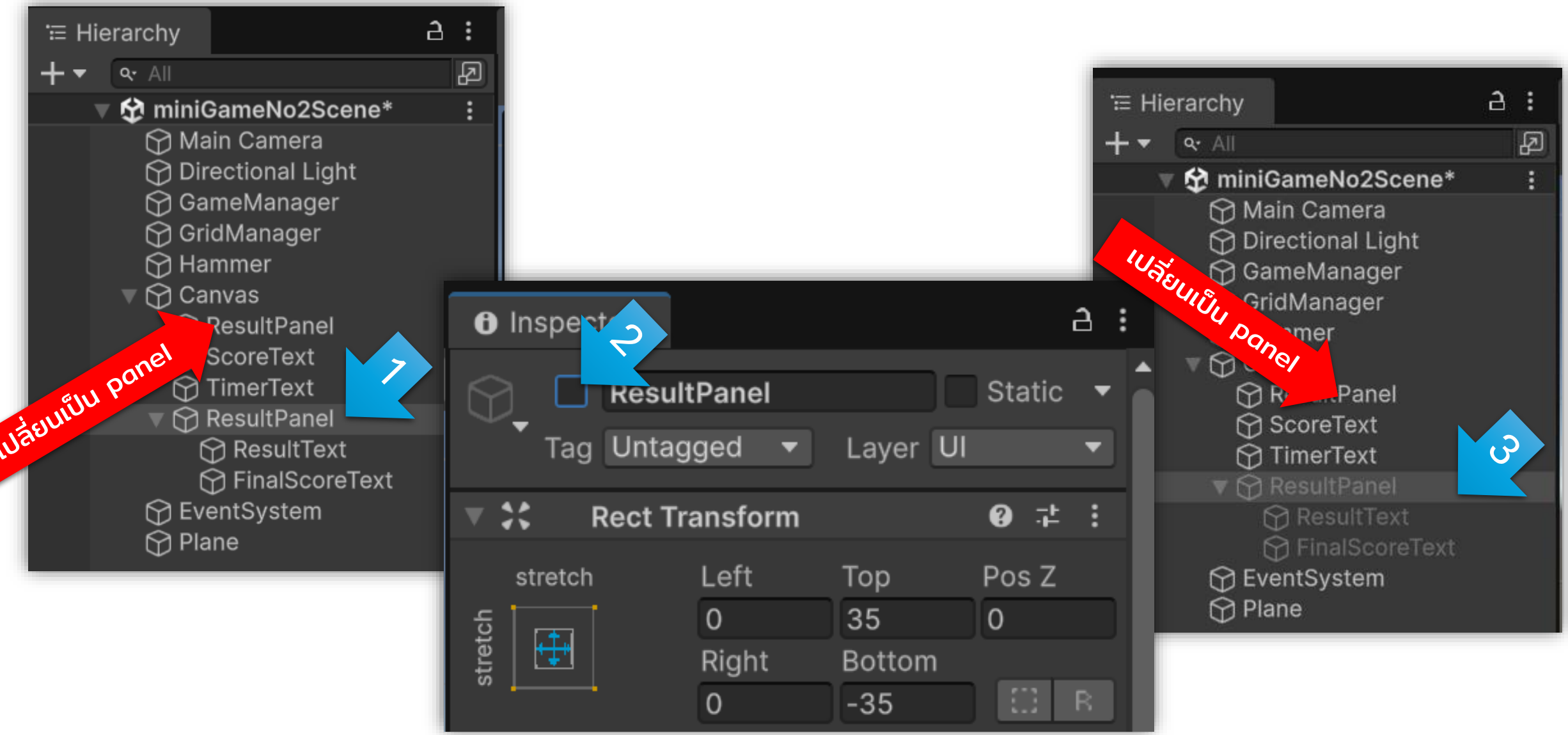


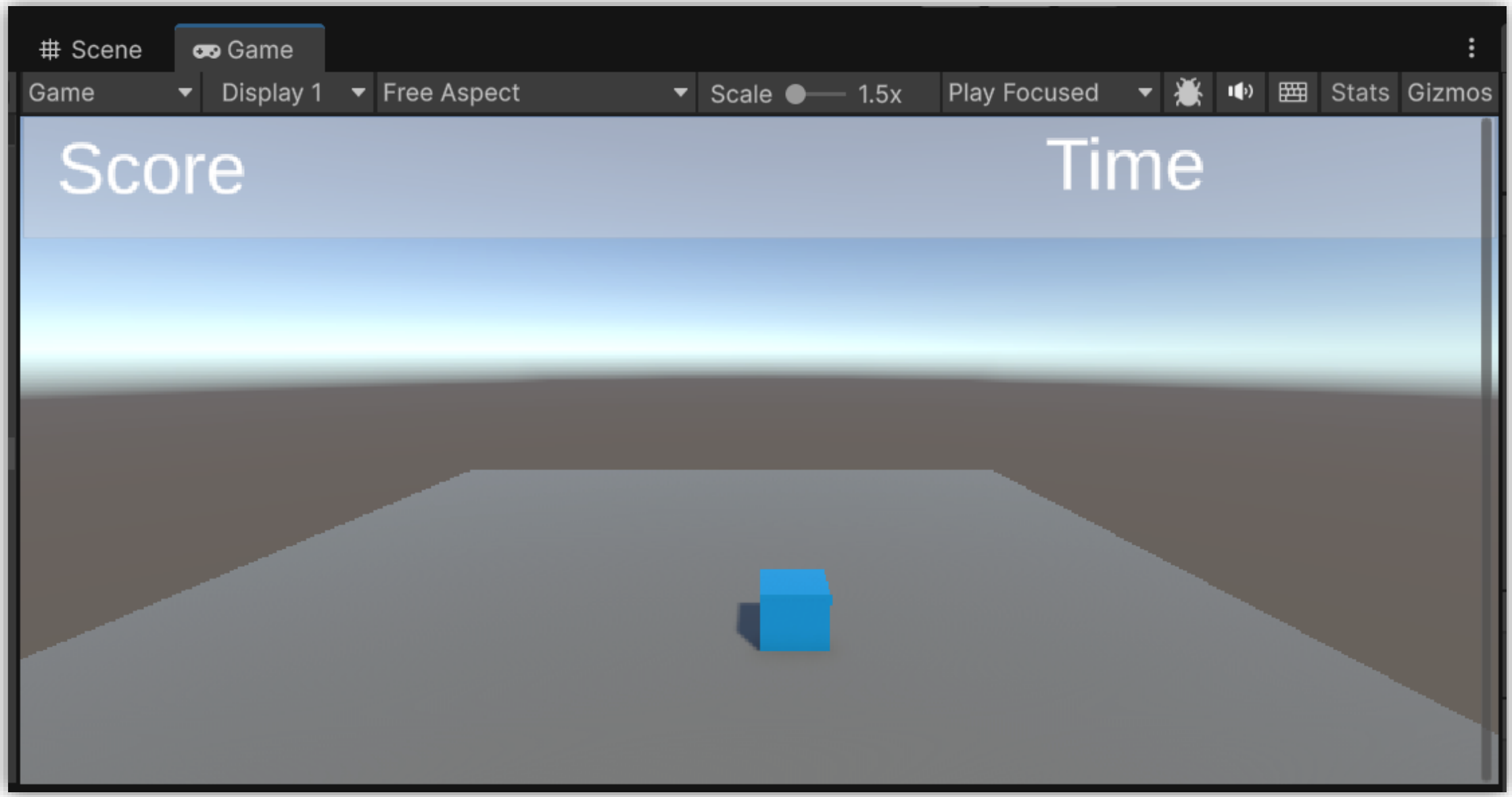
เปลี่ยนเป็น panel




Scene
Game
Center
Local
1
Score
Time
y
x
Back
Scene
Game
Game
Display 1
Free Aspect
Scale
1.5x
Play Focused
Stats
Gizmos
Score
Time


Hierarchy
All
miniGameNo2Scene*
Main Camera
Directional Light
GameManager
GridManager
Hammer
Canvas
ResultPanel
ScoreText
TimerText
ResultPanel
ResultText
FinalScoreText
EventSystem
Plane
ยนเป็น panel
Scene
Game
Center
Local
Score
Time
Back
Win/Lose
Your Final Score


Hierarchy
All
miniGameNo2Scene*
Main Camera
Directional Light
GameManager
GridManager
Hammer
Canvas
ResultPanel
ScoreText
TimerText
ResultPanel
ResultText
FinalScoreText
EventSystem
Plane
เปลี่ยนเป็น panel
1
2
Inspector
ResultPanel
Static
Tag
Untagged
Layer
UI
Rect Transform
stretch
Left
Top
Pos Z
0
35
0
Right
Bottom
0
-35
เปลี่ยนเป็น panel
3


Scene
Game
Game
Display 1
Free Aspect
Scale
1.5x
Play Focused
Stats
Gizmos
Score
Time

สถานะของเกม
(Game States)

แนวคิด Finite State Machine

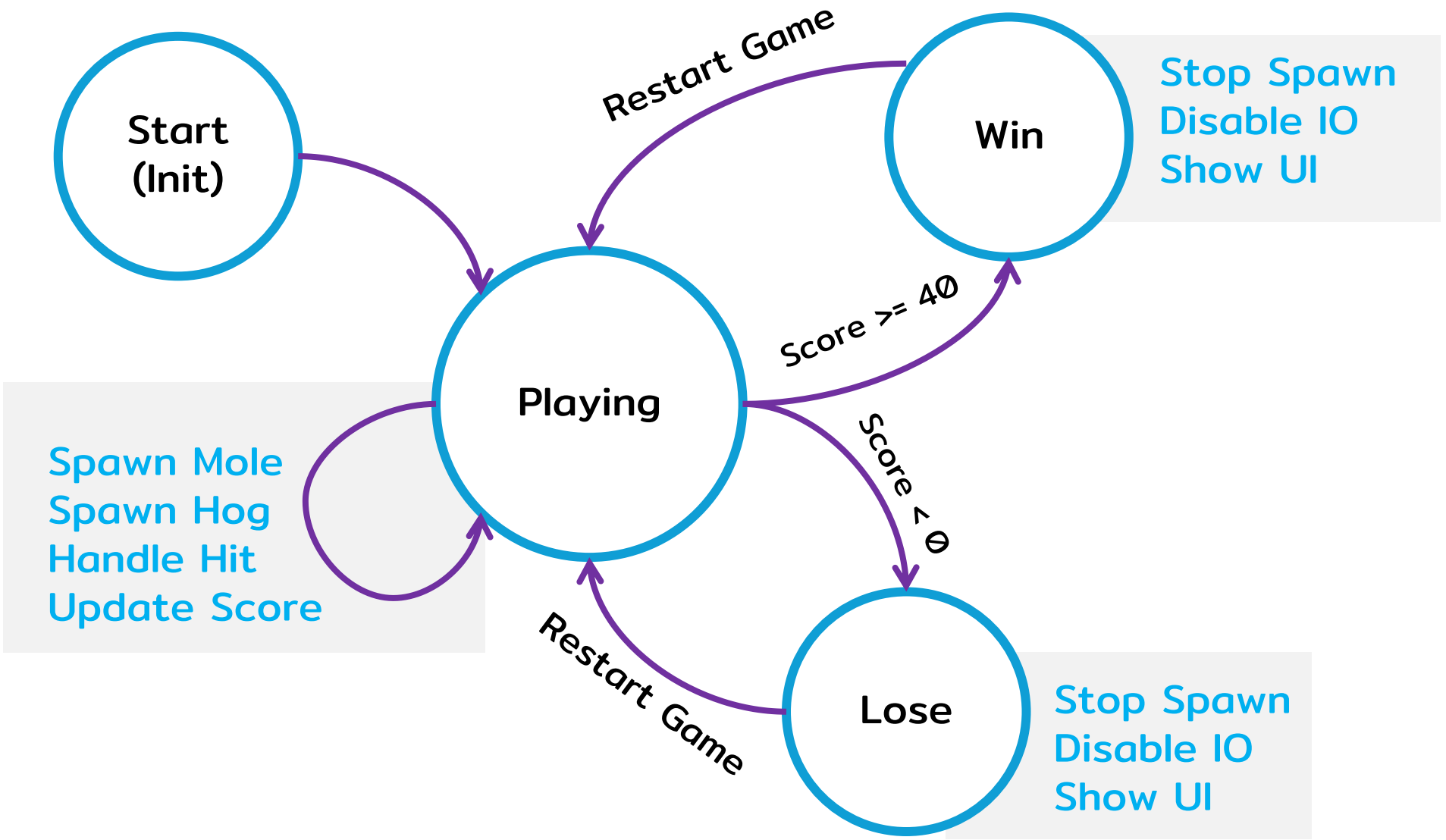
- Finite State Machine (FSM) คือ รูปแบบการออกแบบเชิงพฤติกรรม (Behavioral Design Pattern) ที่ใช้ควบคุมการทำงานของระบบโดยแบ่งออกเป็นสถานะ (State) และการเปลี่ยนสถานะ (Transition) ตามเงื่อนไขที่กำหนด
- งานวิจัยและแนวปฏิบัติด้าน Game Development นิยมใช้ Finite State Machine (FSM) ในการควบคุมพฤติกรรมของเกม ในเกมนี้ FSM ถูกนำมาใช้เพื่อควบคุมลำดับการทำงานของเกม ลดความซับซ้อนของเงื่อนไข และทำให้โครงสร้างโปรแกรมมีความชัดเจน

นิยาม

Finite State Machine คือแบบจำลองเชิงคณิตศาสตร์ที่ประกอบด้วย

1. ชุดของสถานะ (States)
2. การเปลี่ยนสถานะ (Transitions)
3. เงื่อนไขการเปลี่ยนสถานะ (Conditions)

FSM


Start
(Init)
Restart Game
Win
Stop Spawn
Disable IO
Show UI
Playing
Score >= 40
Spawn Mole
Spawn Hog
Handle Hit
Update Score
Score < 0
Restart Game
Lose
Stop Spawn
Disable IO
Show UI

Playing State

- เป็นสถานะหลักของเกม
- ระบบจะสุ่มตำแหน่งและชนิดของวัตถุ
- รับ Input จากผู้เล่นและปรับคะแนนแบบ Real-time

Win State

- เข้าสู่สถานะนี้เมื่อคะแนน ≥ 40
- หยุดการเกิดวัตถุทั้งหมด
- ปิดการรับ Input จากผู้เล่น
- แสดงข้อความแสดงผลการชนะเกม

Lose State

- เข้าสู่สถานะนี้เมื่อคะแนน < 0
- หยุดการทำงานของระบบ Spawn
- ปิด Input และแสดงข้อความ Game Over

การกำหนดสถานะของเกม

เกมถูกกำหนดสถานะหลักไว้ดังนี้

- Ready เกมอยู่ในสภาพพร้อมเริ่ม
- Playing ผู้เล่นสามารถโต้ตอบกับเกมได้
- Win ผู้เล่นชนะเกม
- Lose ผู้เล่นแพ้จากคะแนนติดลบ
- TimeUp หมดเวลาการเล่น

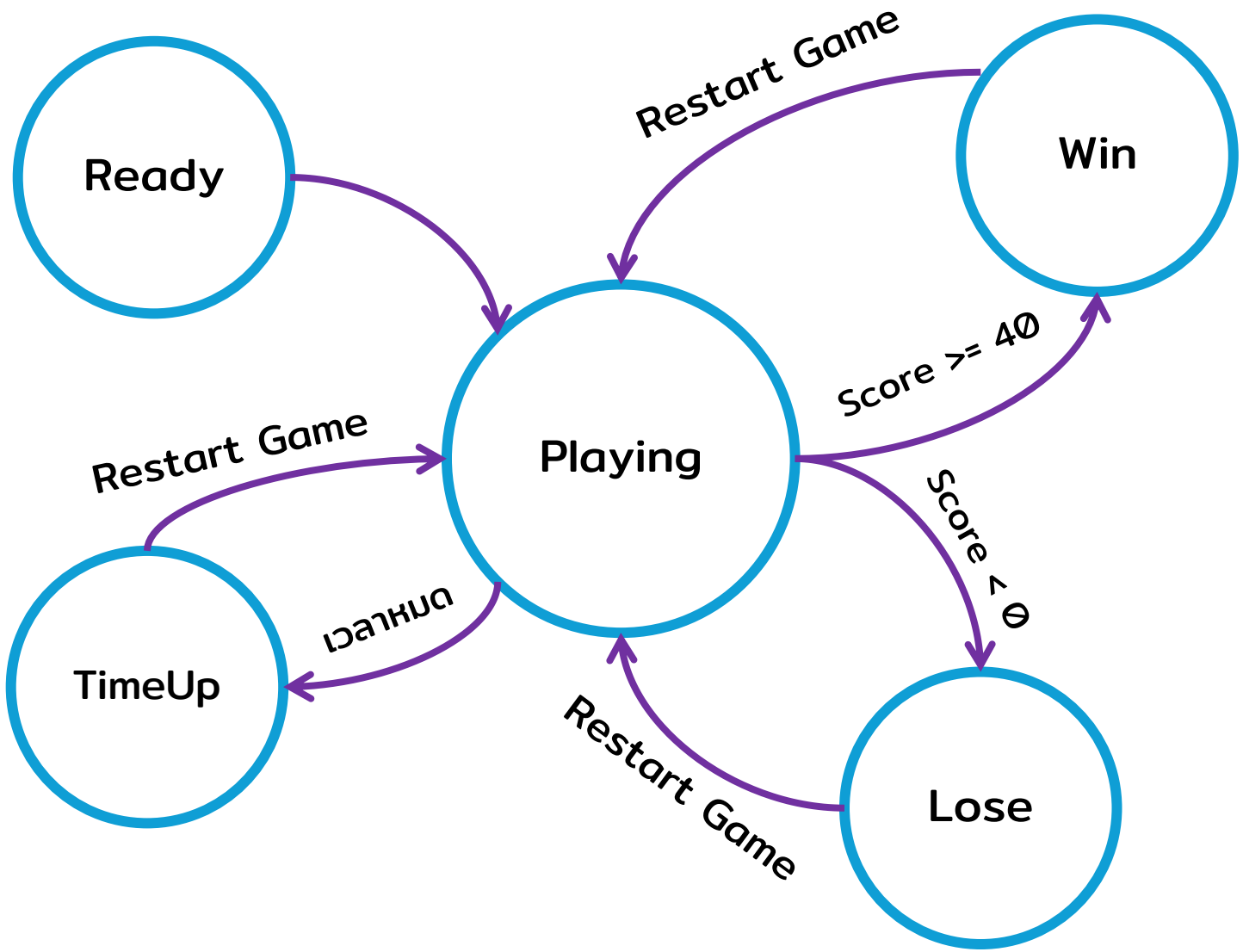
การเปลี่ยนสถานะ
(State Transition)

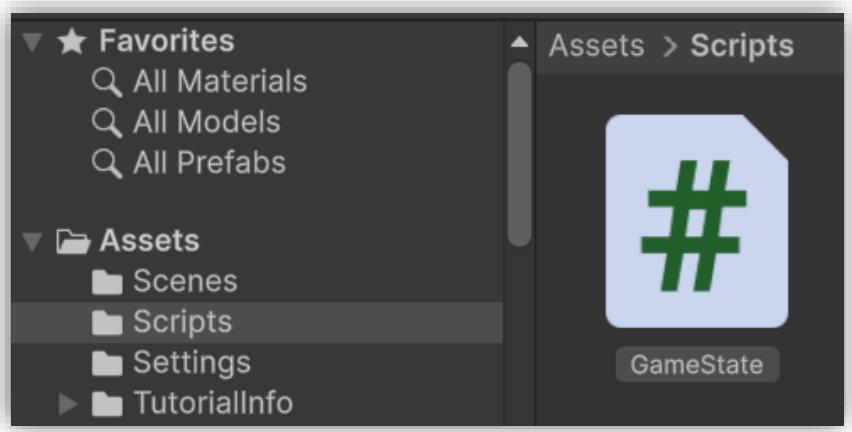
การเปลี่ยนสถานะเกิดจากเงื่อนไขที่ชัดเจน เช่น

- Ready เป็น Playing เมื่อ ผู้เล่นเริ่มเกม
- Playing เป็น Win เมื่อ คะแนนถึง 40 คะแนน
- Playing เป็น Lose เมื่อ คะแนนต่ำกว่า 0
- Playing เป็น TimeUp เมื่อ เวลาเล่นหมด

การควบคุมด้วย FSM ช่วยป้องกันปัญหาทางตรรกะ เช่น การเพิ่มคะแนนหลังเกมจบแล้ว

FSM


Ready
Restart Game
Win
Restart Game
Playing
Score >= 40
Score < 0
TimeUp
เวลาหมด
Restart Game
Lose



นิยามสถานะของเกม (FSM)

GameState.cs

Assembly-CSharp — GameState

```
0 references
1  public enum GameState
2  {
3      Ready,
4      Playing,
5      Win,
6      Lose,
7      TimeUp
8  }
```

ระบบการเกิดตัวละคร
(Object Spawning System)

ระบบการเกิดวัตถุและการให้คะแนน

- การเกิดของตัวตุ่นและเม่นใช้การสุ่มตำแหน่งจาก Grid 5×5 และสุ่มชนิดวัตถุตามค่าความน่าจะเป็น (Probability) ที่ขึ้นกับคะแนนปัจจุบันของผู้เล่น
- แนวคิดนี้ช่วยให้เกมมีความไม่แน่นอน (Uncertainty) และเพิ่มความท้าทายเมื่อผู้เล่นมีคะแนนสูงขึ้น
- ระบบคะแนนถูกออกแบบให้ผู้เล่นต้องตัดสินใจอย่างรอบคอบ ไม่ใช่เพียงคลิกให้เร็วที่สุด แต่ต้องแยกแยะชนิดของวัตถุที่ปรากฏ

โครงสร้างพื้นที่เกม

พื้นที่เล่นถูกออกแบบเป็น ตารางขนาด 5x5 ซึ่งทำหน้าที่เป็นจุดกำเนิด (Spawn Point) ของวัตถุ ลักษณะนี้มีข้อดีคือ

- ควบคุมตำแหน่งวัตถุได้ง่าย
- ลดความซับซ้อนขง Collision Detection
- ปรับสมดุลเกมได้สะดวก

Prefab

Prefab	รูปร่าง	สี
MolePrefab	Cube	เขียว
HedgehogPrefab	Sphere	แดง
Hammer	Cube	น้ำเงิน

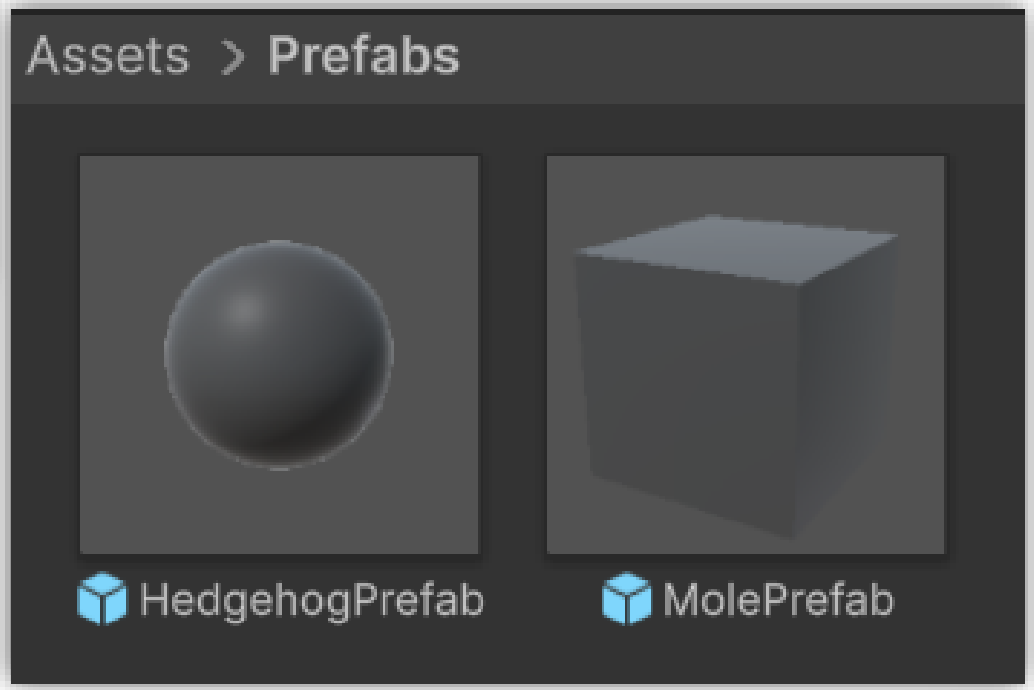
อายุการแสดงผลของวัตถุ

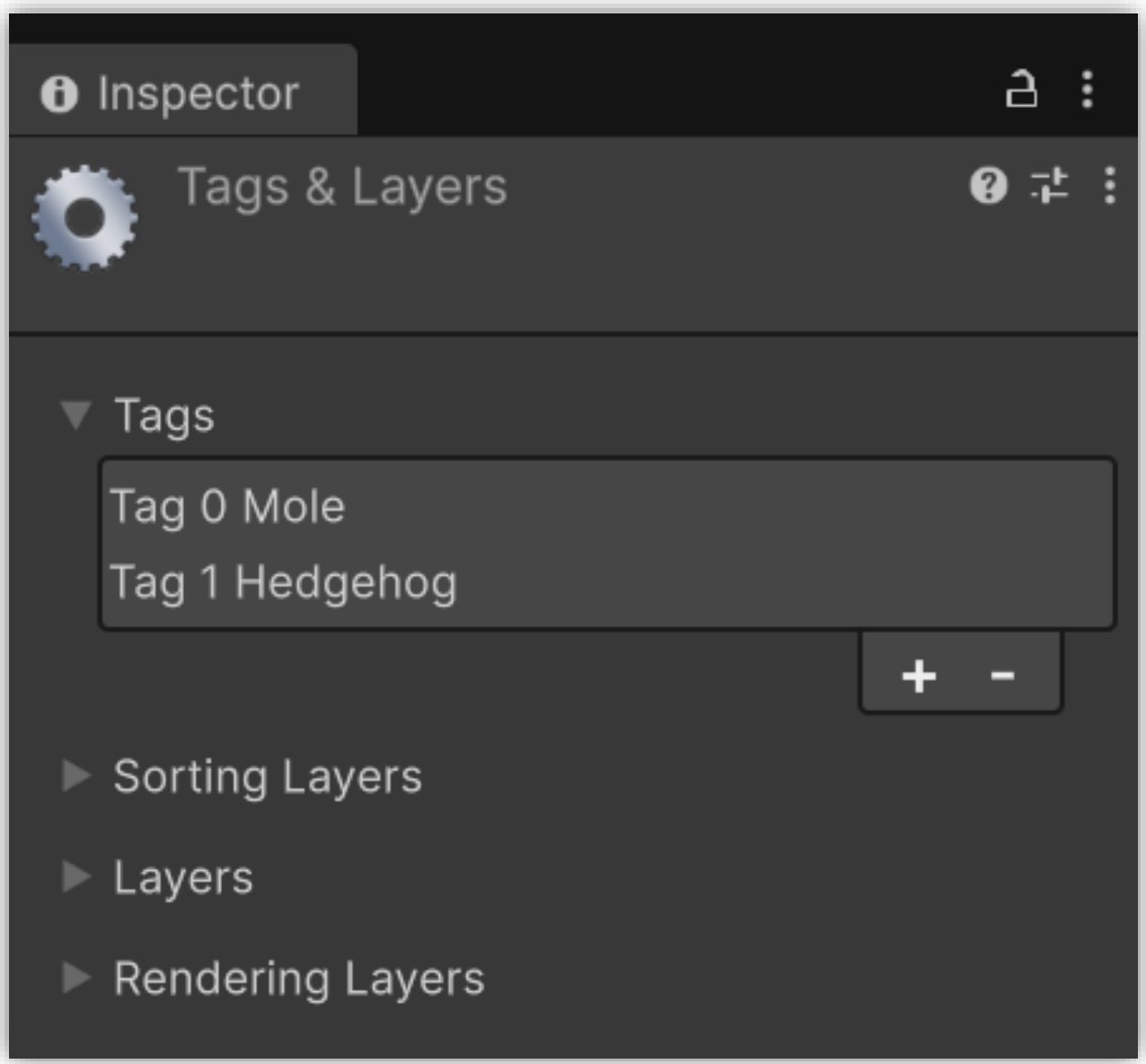
วัตถุแต่ละชนิดจะปรากฏบนหน้าจอเป็นเวลาสุ่มระหว่าง 0.5 – 1 วินาที

- ตัวตุ่น หากหายไปโดยไม่ถูกตี จะถูกหัก 1 คะแนน
- เม่น หากหายไปเอง จะไม่ส่งผลต่อคะแนน

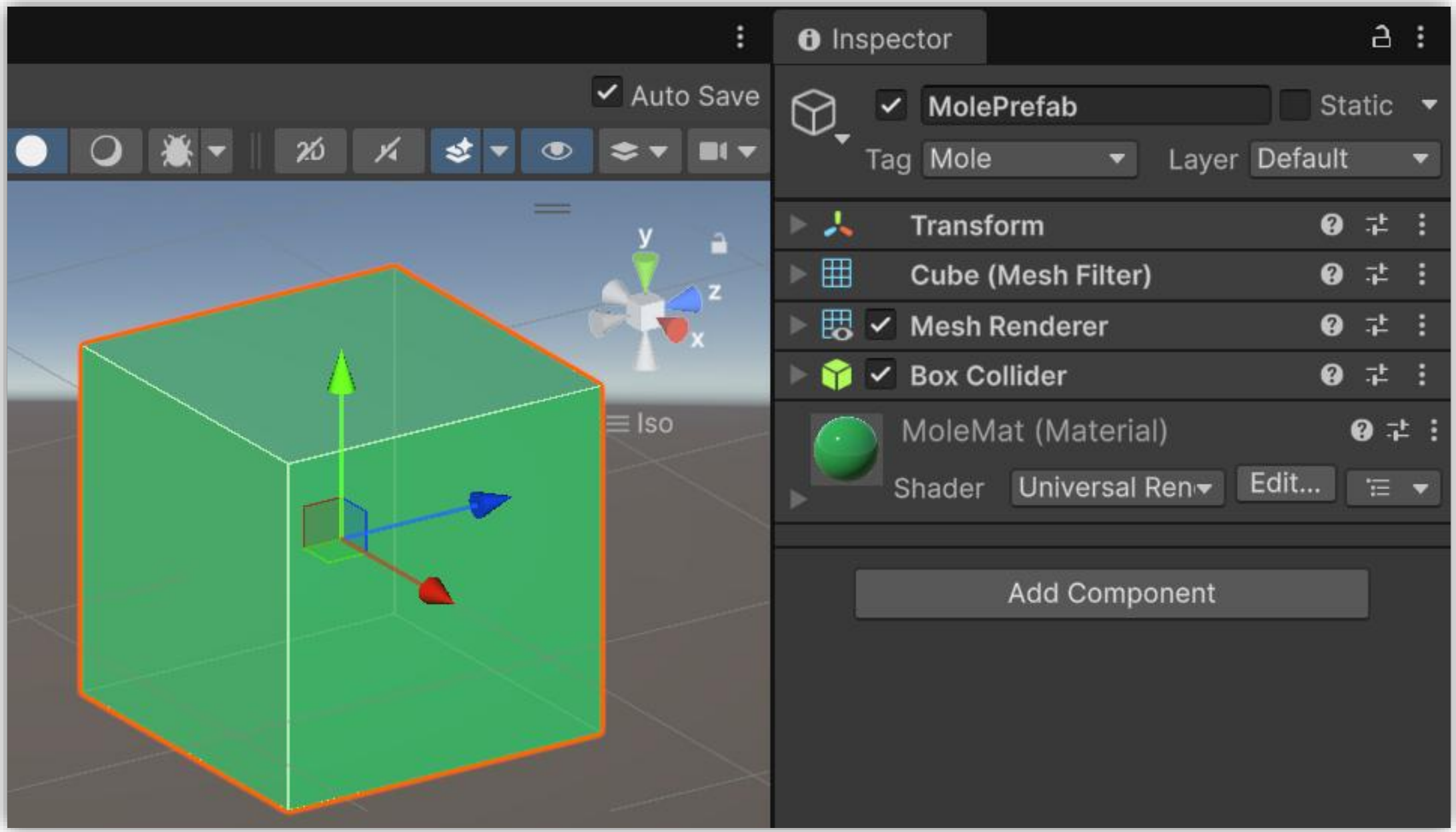
แนวคิดนี้สร้างแรงกดดันเชิงเวลาแก่ผู้เล่น

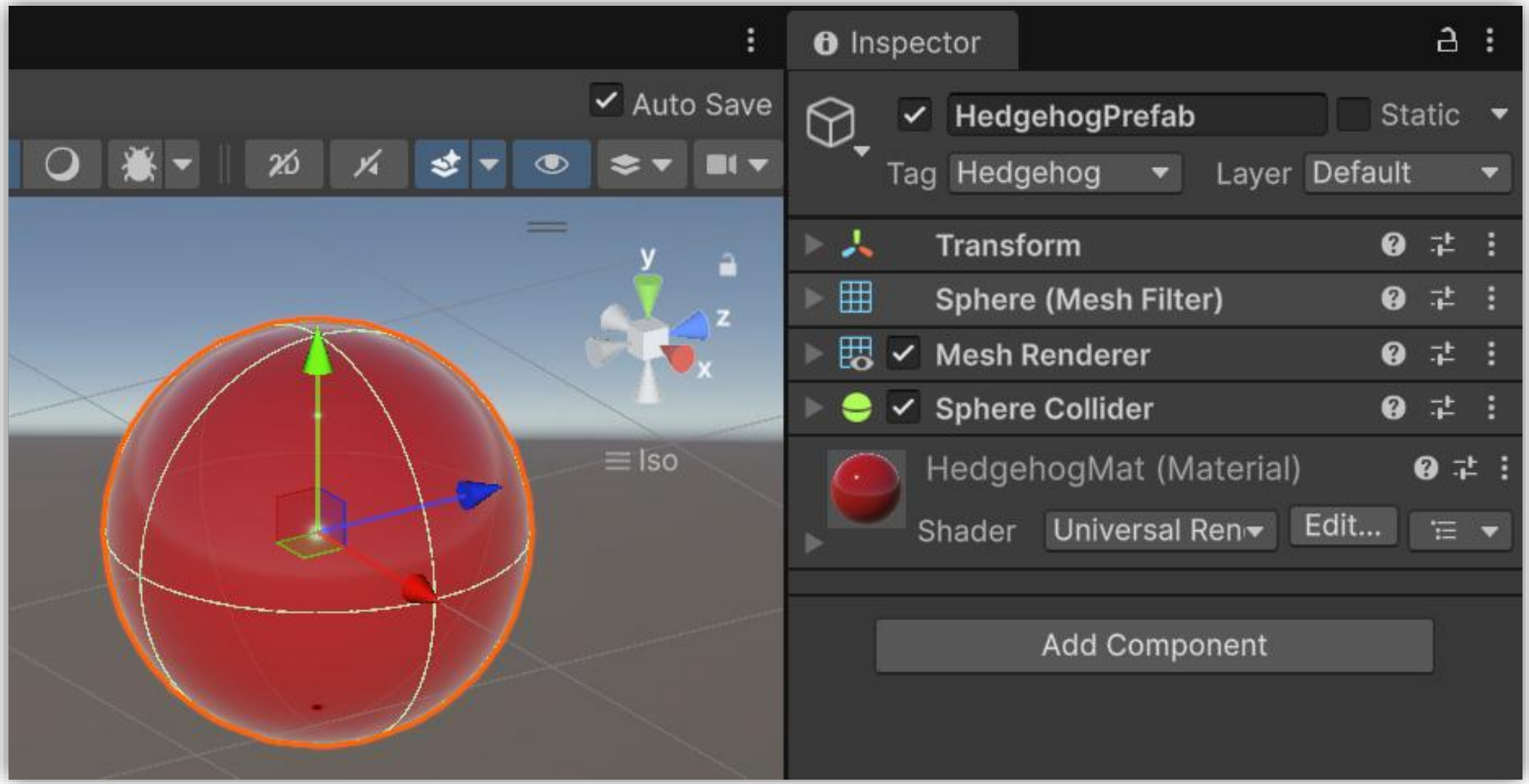
เตรียม material และ prefabs





เตรียม Tag ชื่อ Mole และ Hedgehog


Auto Save
Inspector
MolePrefab
Static
Tag
Mole
Layer
Default
Transform
Cube (Mesh Filter)
Mesh Renderer
Box Collider
MoleMat (Material)
Shader
Universal Ren
Edit...
Add Component
y
z
x
Iso


Auto Save
2D
y
z
x
Iso
Inspector
HedgehogPrefab
Static
Tag
Hedgehog
Layer
Default
Transform
Sphere (Mesh Filter)
Mesh Renderer
Sphere Collider
HedgehogMat (Material)
Shader
Universal Ren
Edit...
Add Component

ระบบความน่าจะเป็นและการปรับระดับความยาก (Dynamic Difficulty Adjustment)

หลักการปรับความยากตามคะแนน

เมื่อคะแนนเพิ่มขึ้น โอกาสที่เม่น (วัตถุเชิงลบ) จะเกิดก็เพิ่มขึ้นตามลำดับ การออกแบบลักษณะนี้ช่วยให้เกมไม่ง่ายเกินไปเมื่อผู้เล่นเก่งขึ้น และรักษาความท้าทายจนถึงช่วงท้ายเกม

ปรับปรุงความน่าจะเป็นการเกิดเม่น?

เหตุการณ์	ผลคะแนน
Score < 10	Hedgehog 20%
Score ≥ 10	Hedgehog 60%
Score ≥ 20	Hedgehog 70%
Score ≥ 30	Hedgehog 80%

ทำงานเกี่ยวกับ FSM, Score และ Timer โดยต้องฝังเข้ากับ GameObject ชื่อ GameManager ที่สร้างไว้ในฉาก

การทำงานร่วมกับ FSM

- UI จะอัปเดต เฉพาะตอน Playing
- เมื่อ State เปลี่ยนเป็น
  - Win / Lose / TimeUp
  - เวลาและคะแนนจะ “ค้าง” อัตโนมัติ
- สามารถใช้ OnStateChanged ต่อกับ Result Panel ได้ทันที

```
1   using System;
2   using UnityEngine;
3   using TMPro;
4
    Unity Script (1 asset reference) | 13 references
5   public class GameManager : MonoBehaviour
6   {
7       public static GameManager Instance;
8
9       [Header("Game State")]
        4 references
10      public GameState CurrentState { get; private set; }
11      public event Action<GameState> OnStateChanged;
12
13      [Header("Score & Time")]
        9 references
14      public int Score { get; private set; }
15      public float TimeLeft = 50f;
16
```

GameManager.cs

```
17      [Header("UI References")]
18      public TMP_Text scoreText;
19      public TMP_Text timerText;
20
```

```
21      private void Awake()
22      {
23          if (Instance == null)
24              Instance = this;
25          else
26              Destroy(gameObject);
27
28          ChangeState(GameState.Ready);
29          UpdateScoreUI();
30          UpdateTimerUI();
31      }
32
```

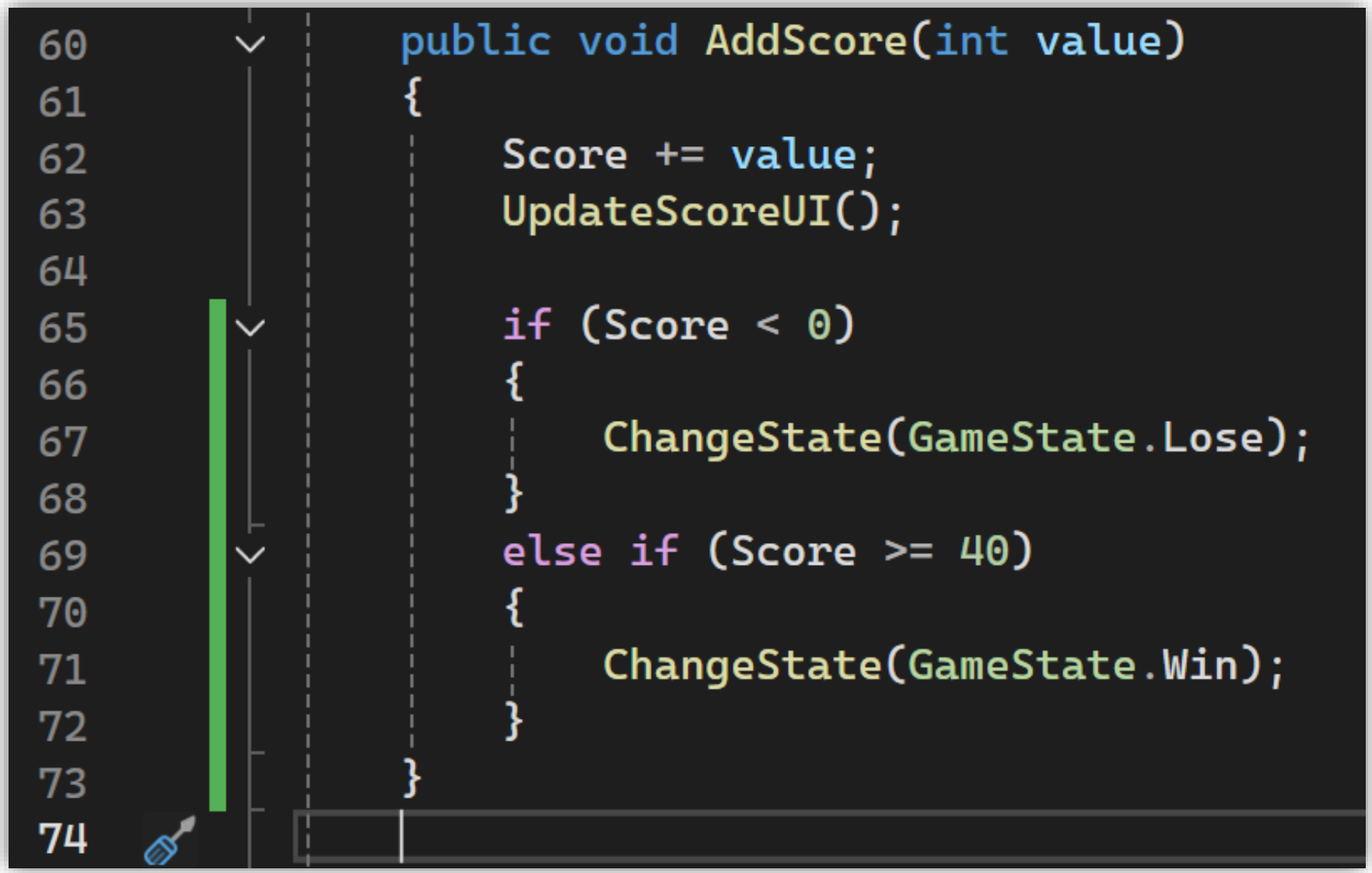
GameManager.cs

```
33      private void Start()
34      {
35          StartGame();
36      }
```

```
37      private void Update()
38      {
39          if (CurrentState != GameState.Playing) return;
40
41          TimeLeft -= Time.deltaTime;
42
43          if (TimeLeft <= 0)
44          {
45              TimeLeft = 0;
46              ChangeState(GameState.TimeUp);
47          }
48          UpdateTimerUI();
49      }
50
```

GameManager.cs

```
51      public void StartGame()
52      {
53          Score = 0;
54          TimeLeft = 50f;
55          ChangeState(GameState.Playing);
56          UpdateScoreUI();
57          UpdateTimerUI();
58      }
59
```



GameManager.cs

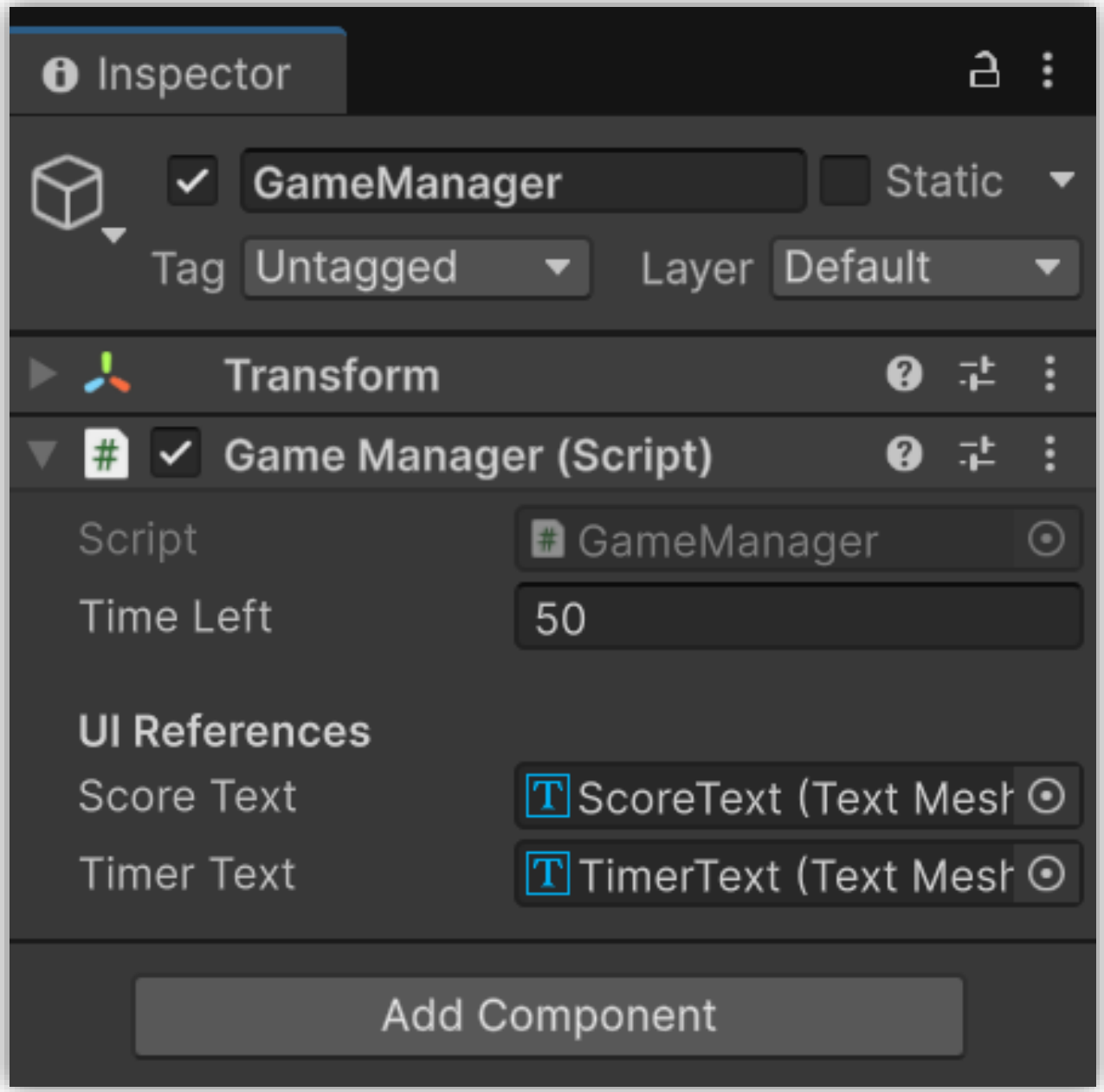
```
75      public void ChangeState(GameState newState)
76      {
77          CurrentState = newState;
78          OnStateChanged?.Invoke(newState);
79      }
80
```

```
81      public float GetHedgehogChance()
82      {
83          if (Score >= 30) return 0.8f;
84          if (Score >= 20) return 0.7f;
85          if (Score >= 10) return 0.6f;
86          return 0.3f;
87      }
88
```

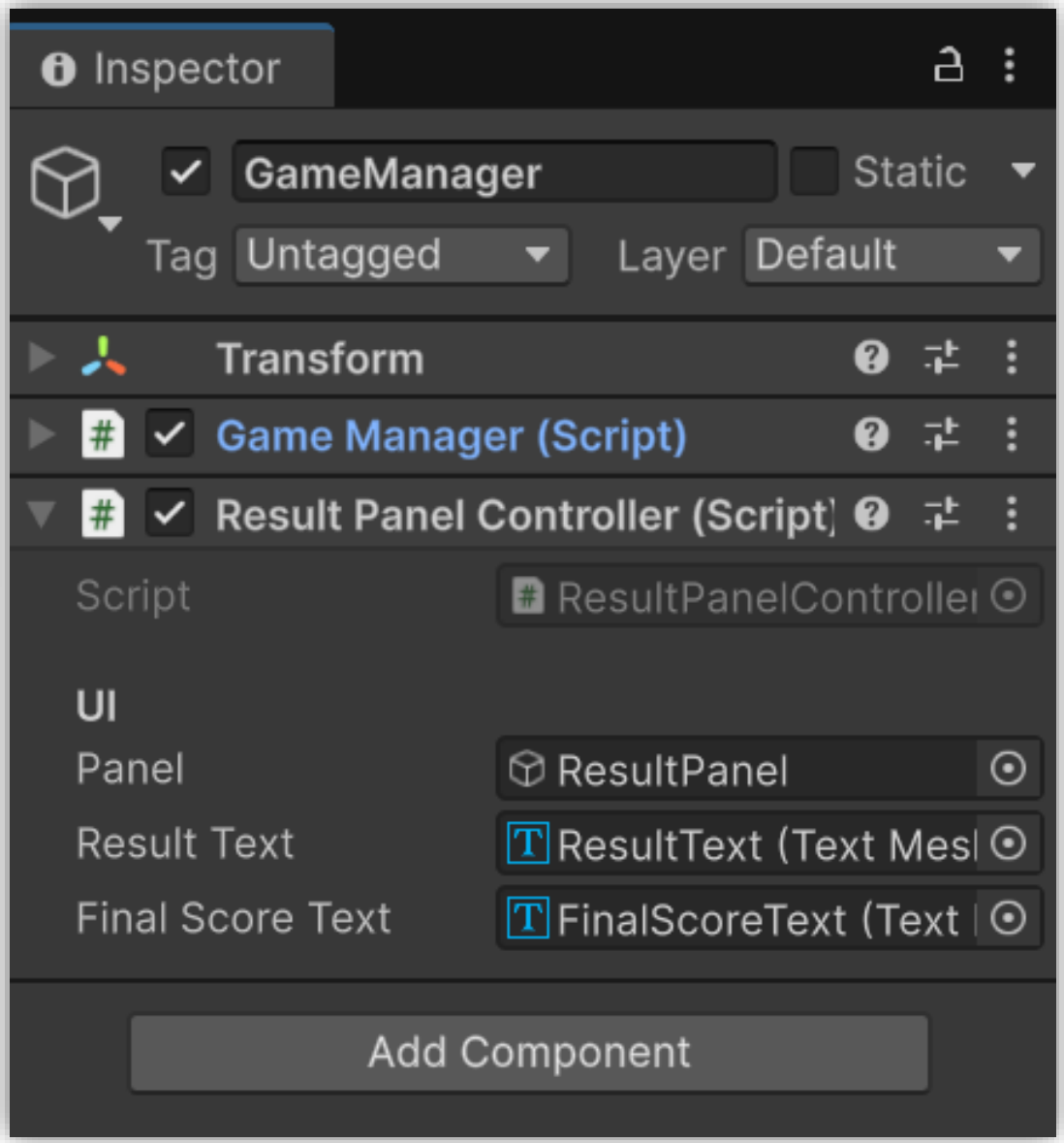
```
89      void UpdateScoreUI()
90      {
91          if (scoreText != null)
92              scoreText.text = $"Score : {Score}";
93      }
94
```

```
95      void UpdateTimerUI()
96      {
97          if (timerText != null)
98              timerText.text = $"Time : {Mathf.CeilToInt(TimeLeft)}";
99      }
100 }
```

GameManager.cs



นำ ScoreText และ TimerText มาวางให้กับสคริปต์



เชื่อม ResultPanel เข้ากับ OnStateChanged ของ GameManager และนำไปผสานเข้ากับ GameObject ที่ชื่อ GameManager

การทำงานร่วมกับ FSM (ลำดับเหตุการณ์)

1. เกมอยู่ใน Playing
2. เงื่อนไขใดเงื่อนไขหนึ่งเกิดขึ้น
    1. คะแนน ≥ 40
    2. คะแนน < 0
    3. เวลา = 0
3. GameManager.ChangeState(...)
4. OnStateChanged ถูกเรียก
5. ResultPanelController รับ Event
6. แสดง ResultPanel + ข้อความ + คะแนน

```
1   using UnityEngine;
2   using TMPro;
3
    Unity Script (1 asset reference) | 0 references
4   public class ResultPanelController : MonoBehaviour
5   {
6       [Header("UI")]
7       public GameObject panel;
8       public TMP_Text resultText;
9       public TMP_Text finalScoreText;
10
```

ResultPanelController.cs

```
11      private void Start()
12      {
13          panel.SetActive(false);
14
15          // Subscribe Event
16          GameManager.Instance.OnStateChanged += HandleGameStateChanged;
17      }
18
```

ResultPanelController.cs

```
19      private void OnDestroy()
20      {
21          // Unsubscribe ป้องกัน memory leak
22          if (GameManager.Instance != null)
23              GameManager.Instance.OnStateChanged -= HandleGameStateChanged;
24      }
25
```

ResultPanelController.cs

```
26      void HandleGameStateChanged(GameState state)
27      {
28          switch (state)
29          {
30              case GameState.Win:
31                  Time.timeScale = 0.0f;
32                  ShowResult("YOU WIN!");
33                  break;
34
```

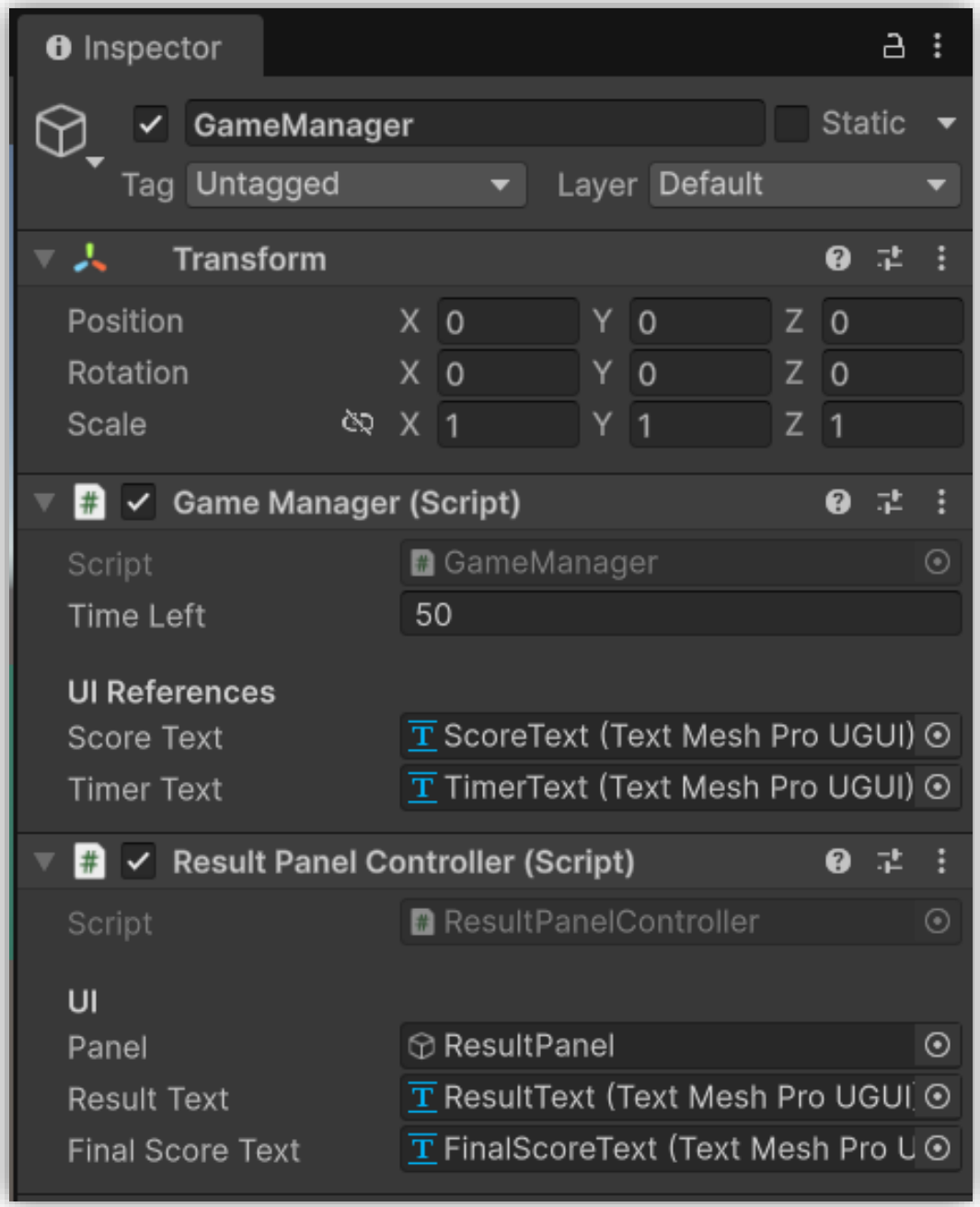
ResultPanelController.cs

```
35              case GameState.Lose:
36                  Time.timeScale = 0.0f;
37                  ShowResult("YOU LOSE!");
38                  break;
39
40              case GameState.TimeUp:
41                  Time.timeScale = 0.0f;
42                  ShowResult("TIME UP");
43                  break;
44
45              default:
46                  Time.timeScale = 1.0f;
47                  panel.SetActive(false);
48                  break;
49          }
50      }
51
```

ResultPanelController.cs

```
52      void ShowResult(string message)
53      {
54          panel.SetActive(true);
55          resultText.text = message;
56          finalScoreText.text = $"Final Score : {GameManager.Instance.Score}";
57      }
58  }
59
```

ResultPanelController.cs



ResultPanelController.cs

Coroutine

ในการพัฒนาเกมหรือแอปพลิเคชันแบบเรียลไทม์ด้วย Unity นักพัฒนามักต้องควบคุมพฤติกรรมที่ “**ใช้เวลา**” (time-dependent behavior) เช่น การรอเวลา การเคลื่อนไหวเป็นช่วง ๆ หรือการแสดงผลชั่วคราว หากใช้โครงสร้างคำสั่งแบบลำดับปกติ (Sequential Execution) จะทำให้โปรแกรมหยุดทำงานระหว่างรอ (Blocking) ซึ่งไม่เหมาะสมกับระบบเกมที่ต้องอัปเดตภาพและรับอินพุตอย่างต่อเนื่อง

Unity แก้ปัญหานี้ด้วยกลไกที่เรียกว่า Coroutine ซึ่งเป็นรูปแบบของการทำงานแบบไม่บล็อก (Non-blocking) ภายในเฟรมเวิร์กของ Unity

Coroutine

- Coroutine คือ ฟังก์ชันพิเศษใน Unity ที่สามารถ “หยุดพักการทำงานชั่วคราว” และกลับมาทำงานต่อในภายหลังได้ โดยไม่ทำให้เกมหรือโปรแกรมหยุดการทำงานทั้งหมด
- ลักษณะสำคัญของ Coroutine คือ ทำงานร่วมกับ Game Loop ของ Unity, ไม่ทำงานใน Thread แยก (ไม่ใช่ Multi-thread), ถูกเรียกและควบคุมโดย Unity Engine หรือ ใช้คำสั่ง yield return เพื่อระบุจุดพักการทำงาน

Coroutine กับ Game Loop

Unity ใช้โครงสร้าง Game Loop ซึ่งประกอบด้วยการเรียกเมธอดสำคัญ เช่น

- Update() เรียกทุกเฟรม
- FixedUpdate() เรียกตามอัตราฟิสิกส์
- LateUpdate() เรียกหลัง Update

Coroutine จะถูก “แทรก” เข้าไปใน Game Loop โดย Unity จะเรียก Coroutine ต่อเนื่องทุกเฟรมจนกว่าจะสิ้นสุด หรือจนกว่าจะเจอคำสั่ง yield return ที่บอกให้พัก

กล่าวอีกนัยหนึ่ง Coroutine คือกระบวนการที่ Unity สลับไปมาระหว่างเฟรมเพื่อไปทำ Coroutine กับงานอื่น ๆ จนกว่าจะเสร็จ หรือเจอคำสั่ง yield return เพื่อบอกให้พัก

โครงสร้างพื้นฐานของ Coroutine

1. รูปแบบของฟังก์ชัน Coroutine ต้องมีชนิดข้อมูลคืนค่าเป็น IEnumerator

```
IEnumerator ExampleCoroutine()
{       // คำสั่ง
        yield return null;
}
```

2. การเริ่มต้น Coroutine ต้องเรียกผ่านคำสั่ง

```
StartCoroutine(ExampleCoroutine())
```

โดยสามารถเรียกได้จากคลาสที่สืบทอด MonoBehaviour เท่านั้น

คำสั่ง yield return

คำสั่ง yield return เป็นหัวใจของ Coroutine ใช้บอก Unity ว่า “หยุดทำงานไว้ก่อน แล้วกลับมาทำต่อในภายหลัง”

รูปแบบที่ใช้บ่อย

- yield return null
  รอ 1 เฟรม
- yield return new WaitForSeconds(t)
  รอ t วินาที
- yield return new WaitUntil(condition)
  รอจนกว่าเงื่อนไขจะเป็นจริง
- yield return new WaitWhile(condition)
  รอขณะที่เงื่อนไขเป็นจริง

ตัวอย่างการหน่วงเวลา

แสดงข้อความหลังจากรอ 2 วินาที โดยไม่ทำให้เกมหยุด หากเขียนแบบจะไม่สามารถหน่วงเวลาได้โดยไม่บล็อกการทำงาน

```
IEnumerator DelayMessage()
{
    yield return new WaitForSeconds(2f);
    Debug.Log("เวลาผ่านไป 2 วินาที");
}
```

ตัวอย่างการเคลื่อนไหวแบบใช้เวลา (Time-based Motion)

ต้องการให้วัตถุเคลื่อนที่จากตำแหน่งหนึ่งไปยังอีกตำแหน่งหนึ่งอย่างนุ่มนวลภายในเวลาที่กำหนด ทั้งนี้ Coroutine สามารถควบคุมการเปลี่ยนแปลงทีละเฟรมได้อย่างเป็นระบบ

```
IEnumerator MoveObject(Vector3 start, Vector3 end, float duration)
{
    float elapsed = 0f;

    while (elapsed < duration)
    {
        elapsed += Time.deltaTime;
        float t = elapsed / duration;

        transform.position = Vector3.Lerp(start, end, t);
        yield return null;
    }

    transform.position = end;
}
```

การหยุด Coroutine

หยุด Coroutine ที่กำลังทำงาน

```
StopCoroutine(coroutineReference);
```

หรือ

```
StopAllCoroutines();
```

โดยปกติ Coroutine จะหยุดเองเมื่อฟังก์ชันทำงานจนจบ, GameObject ถูกทำลาย (Destroy) หรือสคริปต์ถูก Disable

ตัวอย่างการใช้งานในเกมตีตัวตุ่น

ในเกมตีตัวตุ่น Coroutine ถูกใช้สำหรับ

- ควบคุมเวลาแสดงผลของตัวตุ่น
- การยุบและเด้งกลับของค้อน
- การหน่วงเวลาระหว่างการ Spawn

ซึ่งเป็นพฤติกรรมแบบ “เกิด ไปหา รอ ไปหา หาย” ที่เหมาะสมกับการทำงานด้วย Coroutine

Coroutine เป็นกลไกสำคัญของ Unity สำหรับการจัดการพฤติกรรมที่ขึ้นกับเวลาโดยไม่รบกวนการทำงานของระบบหลัก การเข้าใจ Coroutine อย่างถูกต้องช่วยให้นักพัฒนาสามารถออกแบบโค้ดที่ชัดเจน มีโครงสร้าง และง่ายต่อการบำรุงรักษา โดยเฉพาะในเกมที่มีลำดับเหตุการณ์จำนวนมาก

Grid 5x5

- สร้าง Empty GameObject ชื่อ GridManager
- สร้างตำแหน่ง (Transform) 25 จุด ในรูปแบบ 5x5
- เก็บตำแหน่งทั้งหมดใน List<Transform>

ทำงานเกี่ยวกับ ตาราง 5x5 + การสุ่มโดยต้องฝังเข้ากับ GameObject ชื่อ GridManager ที่สร้างไว้ในฉาก

การทำงานร่วมกับ FSM (ลำดับเหตุการณ์)

1. GameManager เปลี่ยน State เป็น Playing
2. GridManager เริ่ม SpawnLoop
3. สุ่มเกิด Mole / Hedgehog
4. ผู้เล่นตีหรือปล่อยให้หายไป
5. เมื่อ State เป็น Win / Lose / TimeUp
    1. SpawnLoop หยุดทันที
    2. ไม่มีวัตถุใหม่เกิดอีก

```
1   using UnityEngine;
2   using System.Collections;
3
    Unity Script (1 asset reference) | 0 references
4   public class GridManager : MonoBehaviour
5   {
6       [Header("Grid Settings")]
7       public int gridSize = 5;
8       public float cellSpacing = 2f;
9       public Vector3 gridOrigin = Vector3.zero;
10
11      [Header("Spawn Settings")]
12      public GameObject molePrefab;
13      public GameObject hedgehogPrefab;
14      public float minLifeTime = 1.5f;
15      public float maxLifeTime = 3.0f;
16      public float spawnInterval = 0.5f;
17
18      private Coroutine spawnRoutine;
19
```

GridManager.cs

```
20      void Start()
21      {
22          // ฟังการเปลี่ยน State ของเกม
23          GameManager.Instance.OnStateChanged += HandleGameStateChanged;
24      }
25
```

```
void Start()
{
    // ย้ายจุดกึ่งกลางของจุดเกิดไปไว้ที่ (-2f*cellSpacing, 0f, -2f*cellSpacing)
    gridOrigin = new Vector3(-2f*cellSpacing, 0f, -2f*cellSpacing);

    GameManager.Instance.OnStateChanged += HandleGameStateChanged;
}
```

GridManager.cs

```
26      void OnDestroy()
27      {
28          if (GameManager.Instance != null)
29              GameManager.Instance.OnStateChanged -= HandleGameStateChanged;
30      }
31
```

GridManager.cs

```
32      void HandleGameStateChanged(GameState state)
33      {
34          if (state == GameState.Playing)
35          {
36              spawnRoutine = StartCoroutine(SpawnLoop());
37          }
38          else
39          {
40              if (spawnRoutine != null)
41                  StopCoroutine(spawnRoutine);
42          }
43      }
44
```

GridManager.cs

```
45      IEnumerator SpawnLoop()
46      {
47          while (GameManager.Instance.CurrentState == GameState.Playing)
48          {
49              SpawnOne();
50              yield return new WaitForSeconds(spawnInterval);
51          }
52      }
53
```

GridManager.cs

```
54      void SpawnOne()
55      {
56          Vector3 spawnPos = GetRandomGridPosition();
57          bool isHedgehog =
58              Random.value < GameManager.Instance.GetHedgehogChance();
59          GameObject prefab =
60              isHedgehog ? hedgehogPrefab : molePrefab;
61          GameObject obj = Instantiate(prefab, spawnPos, Quaternion.identity);
62          float lifeTime = Random.Range(minLifeTime, maxLifeTime);
63
64          StartCoroutine(LifeTimer(obj, isHedgehog, lifeTime));
65      }
66
```

GridManager.cs

```
67      IEnumerator LifeTimer(GameObject obj, bool isHedgehog, float time)
68      {
69          yield return new WaitForSeconds(time);
70
71          if (obj != null)
72          {
73              // ตัวตุ่นหายไปเอง = โดนหักคะแนน
74              if (!isHedgehog &&
75                  GameManager.Instance.CurrentState == GameState.Playing)
76              {
77                  // GameManager.Instance.AddScore(-1);
78              }
79
80              Destroy(obj);
81          }
82      }
83
```

GridManager.cs

```
84      Vector3 GetRandomGridPosition()
85      {
86          int x = Random.Range(0, gridSize);
87          int z = Random.Range(0, gridSize);
88
89          return gridOrigin + new Vector3(
90              x * cellSpacing,
91              0.5f,
92              z * cellSpacing
93          );
94      }
95  }
96
```

GridManager.cs

Inspector

GridManager — Static

Tag: Untagged — Layer: Default

Transform

Grid Manager (Script)

Script: GridManager

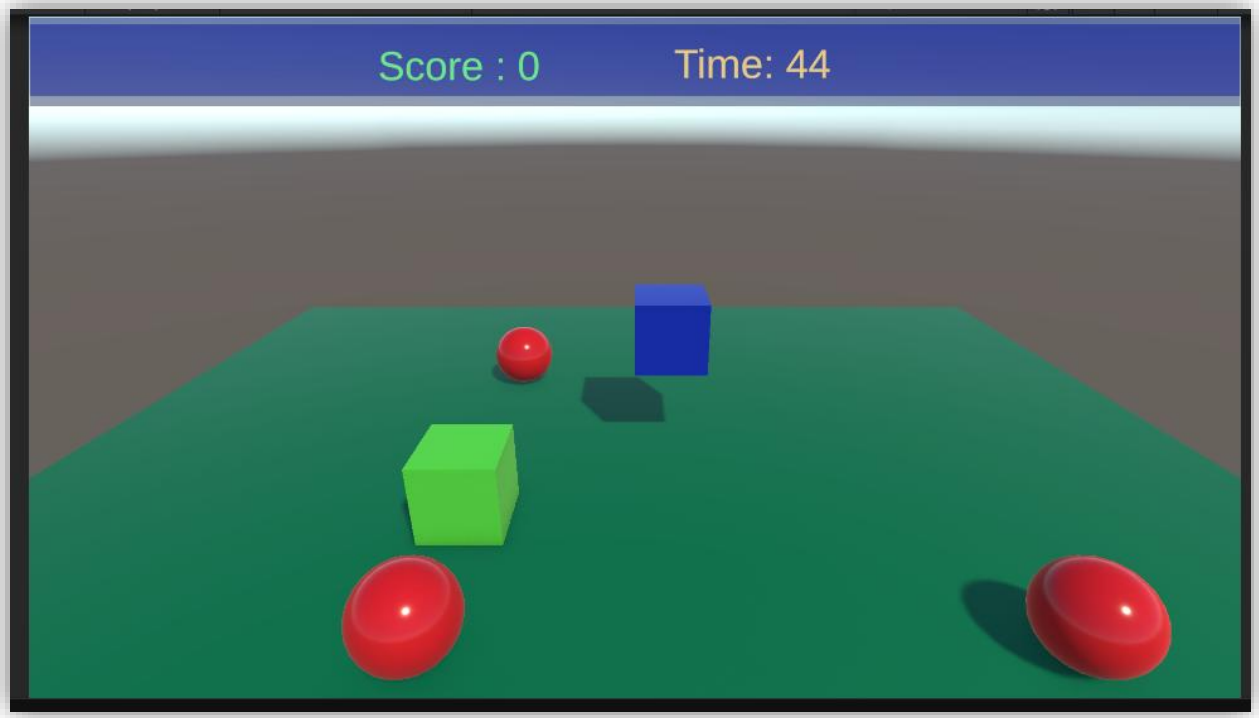
Grid Settings

Grid Size	5
Cell Spacing	2
Grid Origin	X 0 Y 0 Z 0

Spawn Settings

Mole Prefab	MolePrefab
Hedgehog Prefab	HedgehogPrefab
Min Life Time	1.5
Max Life Time	3
Spawn Interval	0.5

Add Component



ระบบการโต้ตอบของผู้เล่น
(Player Interaction System)

ผู้เล่นใช้ ค้อน (Cube สีน้ำเงิน)

เป็นสัญลักษณ์ของการกระทำ การโต้ตอบเกิดจาก

- การคลิกเมาส์
- การตรวจสอบว่าวัตถุที่ถูกคลิกเป็นตัวตุ่นหรือเม่น

ผลลัพธ์ของการโต้ตอบจะเชื่อมโยงโดยตรงกับระบบคะแนน

ระบบคะแนนและกติกา (Scoring Rules)

เหตุการณ์	ผลคะแนน
ตีโดนตุ่น	+1
ตุ่นหายไปเอง	-1
ตีโดนเม่น	-1
เม่นหายไปเอง	0
คะแนน < 0	แพ้
คะแนน ≥ 40	ชนะ

RayCasting

Ray

Ray ใน Unity เป็นเส้นตรงที่ยิงออกไปจากจุดเริ่มต้น (origin) ในทิศทางที่กำหนด (direction) เพื่อตรวจสอบว่าชนกับวัตถุอื่นหรือไม่ มักใช้ในการตรวจจับการชน, การเล็งเป้าหมาย และการสร้างอินเทอร์แอคชั่นต่าง ๆ ในเกม

การใช้งาน Ray

1. Physics.Raycast ฟังก์ชันหลักที่ใช้ในการยิงรังสีและตรวจสอบการชน
2. RaycastHit โครงสร้างข้อมูลที่เก็บข้อมูลเกี่ยวกับการชน เช่น จุดที่ชน วัตถุที่ชน และอื่นๆ

```
1   using UnityEngine;
2
3   public class RaycastExample : MonoBehaviour
4   {
5       public float rayLength = 5f; // ความยาวของรังสี
6       void Update()
7       {
8           // สร้างรังสี
9           Ray ray = new Ray(transform.position, transform.forward);
10          // ตรวจสอบการชน
11          RaycastHit hit;
12          if (Physics.Raycast(ray, out hit, rayLength))
13          {
14              Debug.Log("ชนกับวัตถุ " + hit.collider.name);
15              // ทำอะไรบางอย่างเมื่อชน เช่น ยิงกระสุน
16          }
17      }
18  }
```

RaycastExample.cs

อธิบาย

1. Ray ray = new Ray(transform.position, transform.forward);: สร้างรังสีใหม่ โดยกำหนดจุดเริ่มต้นที่ตำแหน่งของ GameObject และทิศทางให้ตรงไปข้างหน้า
2. Physics.Raycast(ray, out hit, rayLength): ยิงรังสีและตรวจสอบการชน
    2.1 ray: รังที่จะยิง
    2.2 out hit: ตัวแปรชนิด RaycastHit ที่จะเก็บข้อมูลการชน
    2.3 rayLength: ความยาวสูงสุดของรังสี
3. if (Physics.Raycast(ray, out hit, rayLength)): ตรวจสอบว่ารังสีชนกับวัตถุหรือไม่ ถ้าชนจะเข้าไปในบล็อก if

พารามิเตอร์

1. Origin: จุดเริ่มต้นของรังสี สามารถเป็นตำแหน่งของกล้อง ตัวละคร หรือจุดใดๆ ในฉาก
2. Direction: ทิศทางของรังสี มักจะกำหนดโดยเวกเตอร์ที่ชี้จากจุดเริ่มต้นไปยังเป้าหมาย
3. Distance: ระยะทางสูงสุดที่รังสีจะยิงออกไป
4. LayerMask: กำหนด Layer ของวัตถุที่ต้องการตรวจสอบการชน
5. QueryTriggerInteraction: กำหนดวิธีการตรวจสอบการชนกับ Collider ที่มี Trigger
6. RaycastHit: โครงสร้างข้อมูลที่เก็บข้อมูลเกี่ยวกับการชน เช่น จุดที่ชน วัตถุที่ชน และอื่นๆ

ตัวอย่าง: ยิงรังสีจากปลายกระบอกปืน

- จุดเริ่มต้น เปลี่ยนค่าของ transform.position ในการสร้าง Ray
- ทิศทาง ใช้ transform.forward, transform.up, transform.right หรือสร้างเวกเตอร์ทิศทางขึ้นมาเอง

```
Ray ray = new Ray(gunTip.position, gunTip.forward);
```

ตัวอย่าง: ตรวจสอบการชนในระยะ 10 หน่วย

- เปลี่ยนค่า rayLength ในฟังก์ชัน Physics.Raycast

```
if (Physics.Raycast(ray, out hit, 10f))
{
}
```

ตัวอย่าง: ตรวจสอบการชนเฉพาะกับวัตถุที่มี Layer เป็น "Enemy"

- ใช้ LayerMask.GetMask("LayerName") เพื่อสร้าง LayerMask

```
int layerMask = LayerMask.GetMask("Enemy");
if (Physics.Raycast(ray, out hit, rayLength, layerMask))
{
}
```

ตัวอย่าง: ตรวจสอบการชนกับ Trigger

- ใช้ QueryTriggerInteraction.Collide เพื่อตรวจสอบการชนกับ Trigger

```
if (Physics.Raycast(ray, out hit, rayLength, layerMask, QueryTriggerInteraction.Collide))
{
}
```

การดึงข้อมูลจาก RaycastHit

- hit.point จุดที่รังสีชน
- hit.normal เวกเตอร์ตั้งฉากกับพื้นผิวที่ชน
- hit.collider Collider ที่ถูกชน

```
Debug.Log("ชนกับวัตถุ: " + hit.collider.name);
```

```
1   using UnityEngine;
2
3   public class RaycastExample2 : MonoBehaviour
4   {
5       public float rayLength = 10f;
6       void Update()  {
7           // สร้างรังสี
8           Ray ray = new Ray(transform.position, transform.forward);
9           // กำหนด LayerMask
10          int layerMask = LayerMask.GetMask("Enemy");
11          // ตรวจสอบการชน
12          RaycastHit hit;
13          if (Physics.Raycast(ray, out hit, rayLength, layerMask)) {
14              Debug.Log("ชนกับศัตรู: " + hit.collider.name);
15          }
16      }
17  }
```

RaycastExample2.cs

อธิบาย

ตัวอย่างการปรับแต่ง Raycast เพื่อตรวจสอบการชนกับวัตถุที่มี Tag เป็น "Enemy" ในระยะ 10 หน่วย

1. Physics.SphereCast ยิงรังสีเป็นทรงกลม เหมาะสำหรับตรวจสอบการชนในพื้นที่กว้าง
2. Physics.BoxCast ยิงรังสีเป็นทรงกล่อง เหมาะสำหรับตรวจสอบการชนกับวัตถุที่มีรูปทรงซับซ้อน
3. Physics.OverlapSphere ตรวจสอบว่ามี Collider ใดบ้างอยู่ในรัศมีที่กำหนด

การใช้ RaycastHit

- เมื่อรังสี (Ray) ที่ยิงออกไปไปชนกับวัตถุใด ๆ ในฉาก Unity จะเก็บข้อมูลเกี่ยวกับการชนนั้นไว้ในโครงสร้างข้อมูลที่เรียกว่า **RaycastHit**
- ข้อมูลเหล่านี้มีประโยชน์อย่างมากในการสร้างเกมและแอปพลิเคชันต่าง ๆ เช่น การตรวจสอบว่าชนกับวัตถุประเภทใด การคำนวณแรงกระทำ หรือการสร้างเอฟเฟกต์ เป็นต้น

ข้อมูลที่เก็บอยู่ใน RaycastHit

- point: จุดที่รังสีชนกับวัตถุในโลกจริง
- normal: เวกเตอร์ตั้งฉากกับพื้นผิวที่ชน
- collider: Collider ของวัตถุที่ถูกชน
- rigidbody: Rigidbody ของวัตถุที่ถูกชน (ถ้ามี)
- distance: ระยะห่างจากจุดเริ่มต้นของรังสีถึงจุดที่ชน
- triangleIndex: (สำหรับ Mesh Collider) ดัชนีของสามเหลี่ยมที่ถูกชน
- textureCoord: พิกัด UV บน Texture ที่ถูกชน

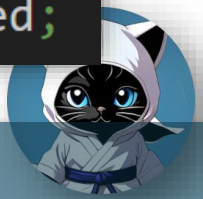
ข้อมูลที่เก็บอยู่ใน RaycastHit

- point: จุดที่รังสีชนกับวัตถุในโลกจริง
- normal: เวกเตอร์ตั้งฉากกับพื้นผิวที่ชน
- collider: Collider ของวัตถุที่ถูกชน
- rigidbody: Rigidbody ของวัตถุที่ถูกชน (ถ้ามี)
- distance: ระยะห่างจากจุดเริ่มต้นของรังสีถึงจุดที่ชน
- triangleIndex: (สำหรับ Mesh Collider) ดัชนีของสามเหลี่ยมที่ถูกชน
- textureCoord: พิกัด UV บน Texture ที่ถูกชน

```
1   using UnityEngine;
2
3   public class RaycastExample3 : MonoBehaviour  {
4       public float rayLength = 5f;
5
6       void Update() {
7           Ray ray = new Ray(transform.position, transform.forward);
8           RaycastHit hit;
9           if (Physics.Raycast(ray, out hit, rayLength)) {
10              // ดึงข้อมูลจาก RaycastHit
11              Debug.Log("ชนกับวัตถุ: " + hit.collider.name);
12              Debug.Log("จุดที่ชน: " + hit.point);
13              Debug.Log("เวกเตอร์ตั้งฉาก: " + hit.normal);
14              // ทำอะไรบางอย่างกับวัตถุที่ชน เช่น เปลี่ยนสี
15              hit.collider.GetComponent<Renderer>().material.color = Color.red;
16          }
17      }
18  }
```

RaycastExample3.cs

```
1   using UnityEngine;
2
3   public class RaycastExample4 : MonoBehaviour
4   {
5       public float rayLength = 5f;
6
7       void Update()
8       {
9           Ray ray = new Ray(transform.position, transform.forward);
10          RaycastHit hit;
11
12          if (Physics.Raycast(ray, out hit, rayLength)) {
13              // ดึงข้อมูลจาก RaycastHit
14              Debug.Log("ชนกับวัตถุ: " + hit.collider.name);
15              Debug.Log("จุดที่ชน: " + hit.point);
16              Debug.Log("เวกเตอร์ตั้งฉาก: " + hit.normal);
17              // ทำอะไรบางอย่างกับวัตถุที่ชน เช่น เปลี่ยนสี
18              hit.collider.GetComponent<Renderer>().material.color = Color.red;
```



RaycastExample4.cs

```
20                  if (hit.collider.CompareTag("Enemy"))
21                  {
22                      // ถ้าชนกับศัตรู ให้สร้างเอฟเฟกต์การระเบิดที่จุดที่ชน
23                      GameObject explosion = Instantiate(
24                                                 explosionPrefab,
25                                                 hit.point,
26                                                 Quaternion.identity);
27                      Destroy(hit.collider.gameObject); // ทำลายศัตรู
28                  }
29              }
30          }
31      }
```

RaycastExample4.cs

ข้อควรจำ

1. Raycast จะตรวจพบเฉพาะ Collider ที่ติดอยู่กับ GameObject เท่านั้น
2. สามารถปรับแต่ง Layer ที่จะตรวจสอบการชนได้
3. มีฟังก์ชันอื่นๆ ที่เกี่ยวข้องกับ Ray เช่น Physics.SphereCast, Physics.BoxCast
4. LayerMask ใช้เพื่อจำกัดการตรวจสอบการชนเฉพาะวัตถุใน Layer ที่ต้องการ
5. QueryTriggerInteraction ใช้เพื่อกำหนดวิธีการตรวจสอบการชนกับ Collider ที่มี Trigger
6. Physics.SphereCast และ Physics.BoxCast ใช้สำหรับการตรวจสอบการชนในพื้นที่ที่กว้างกว่า

ทำงานเกี่ยวกับการเคลื่อนตำแหน่งค้อนตามเมาส์ การทุบ การคลิกตรวจจับโดยใช้ Raycast โดยต้องฝังเข้ากับ Hammer ที่สร้างไว้ในฉาก

พฤติกรรม “ทุบ”

ประกอบด้วย 3 ช่วง

1. ตำแหน่งปกติ (Idle Height)
2. ยุบลง (Strike Down) ด้วยการลดค่า Y อย่างรวดเร็ว
3. เด้งกลับ (Recover) โดยให้กลับสู่ค่า Y เดิม

ป้องกันการกดซ้ำ

ใช้ตัวแปร isStriking ป้องกันการกดซ้ำ และไม่ให้ตำแหน่ง Y ถูกเขียนทับผิดจังหวะ

การเลื่อนตำแหน่งตามเมาส์

ในเกม 3 มิติ เมาส์อยู่ใน Screen Space (2D) แต่ค้อนอยู่ใน World Space (3D) ดังนั้นจำเป็นต้อง

1. แปลงตำแหน่งเมาส์จาก Screen Space
2. ฉายตำแหน่งลงบนระนาบ (Plane) ในโลก 3 มิติ
3. ขยับค้อนให้ไปยังตำแหน่งที่ฉายได้

แนวคิดนี้เรียกว่า Screen-to-World Projection ผ่าน Ray–Plane Intersection

```
1   using UnityEngine;
2   using UnityEngine.InputSystem;
3   using System.Collections;
4
    Unity Script (1 asset reference) | 0 references
5   public class HammerController : MonoBehaviour
6   {
7       [Header("Movement Settings")]
8       public float heightOffset = 1.5f;       // ความสูงของค้อนจากพื้น
9       public float moveSpeed = 15f;            // ความเร็วการเคลื่อนที่
10
11      private Plane groundPlane;
12
13      [Header("Strike Settings")]
14      public float strikeDepth = 0.7f;        // ระยะยุบลงในแกน Y
15      public float strikeSpeed = 0.05f;       // เวลาลง
16      public float recoverSpeed = 0.08f;      // เวลากลับ
17
18      private bool isStriking = false;
19
```

HammerController.cs

```
20      void Start()
21      {
22          // สร้างระนาบพื้น (แกน Y)
23          groundPlane = new Plane(Vector3.up, Vector3.zero);
24      }
```

HammerController.cs

```
25      void Update()
26      {
27          if (Mouse.current == null) return;
28
29          FollowMouse();
30
31          if (Mouse.current.leftButton.wasPressedThisFrame && !isStriking)
32          {
33              HandleHit();
34          }
35      }
36
```

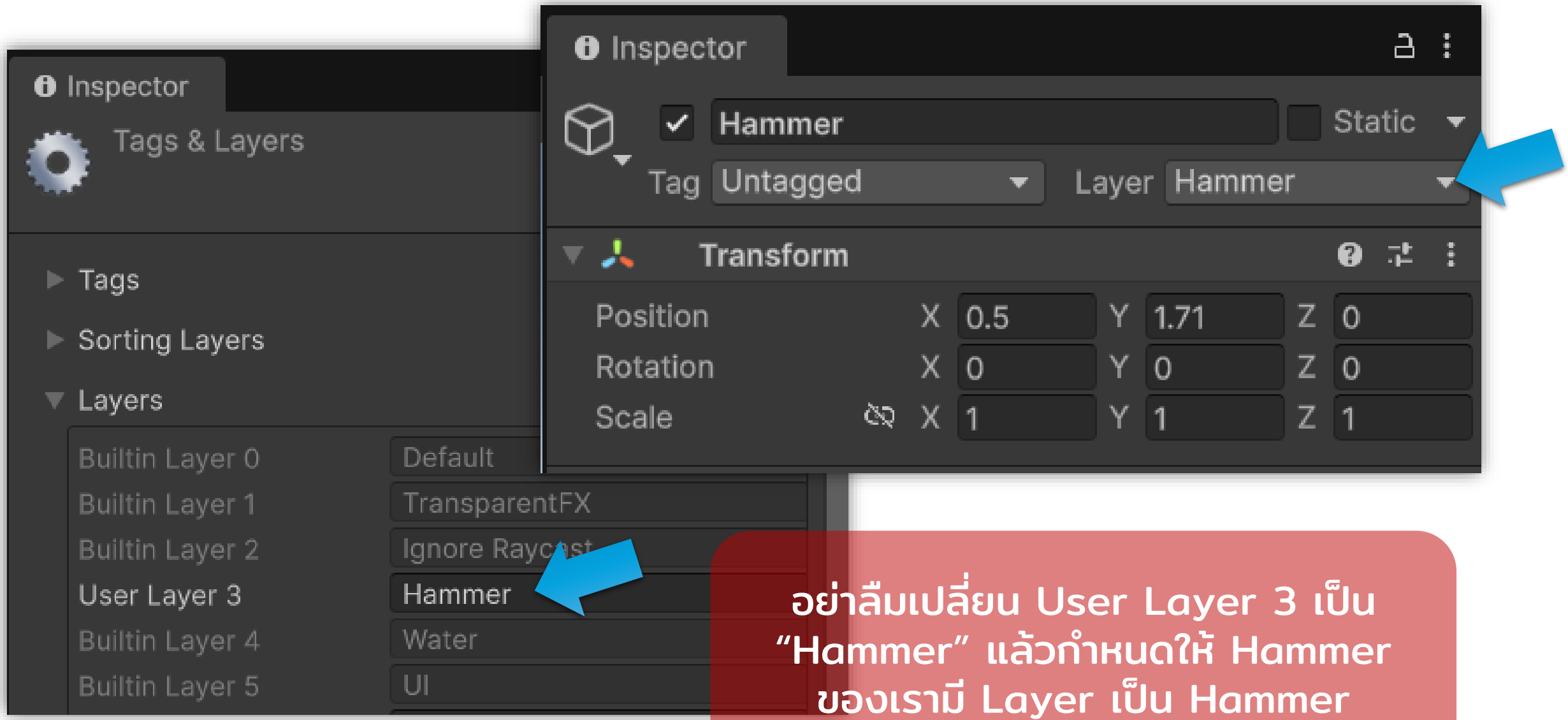
HammerController.cs

```
37      IEnumerator StrikeAnimation()
38      {
39          isStriking = true;
40          Vector3 startPos = transform.position;
41          Vector3 downPos = startPos + Vector3.down * strikeDepth;
42
43          // ยุบลง
44          float t = 0;
45          while (t < 0.2f)
46          {
47              t += Time.deltaTime / strikeSpeed;
48              transform.position = Vector3.Lerp(startPos, downPos, t);
49              yield return null;
50          }
51
```

HammerController.cs

```
52              // เด้งกลับ
53              t = 0;
54              while (t < 0.2f)
55              {
56                  t += Time.deltaTime / recoverSpeed;
57                  transform.position = Vector3.Lerp(downPos, startPos, t);
58                  yield return null;
59              }
60
61              isStriking = false;
62          }
63
```

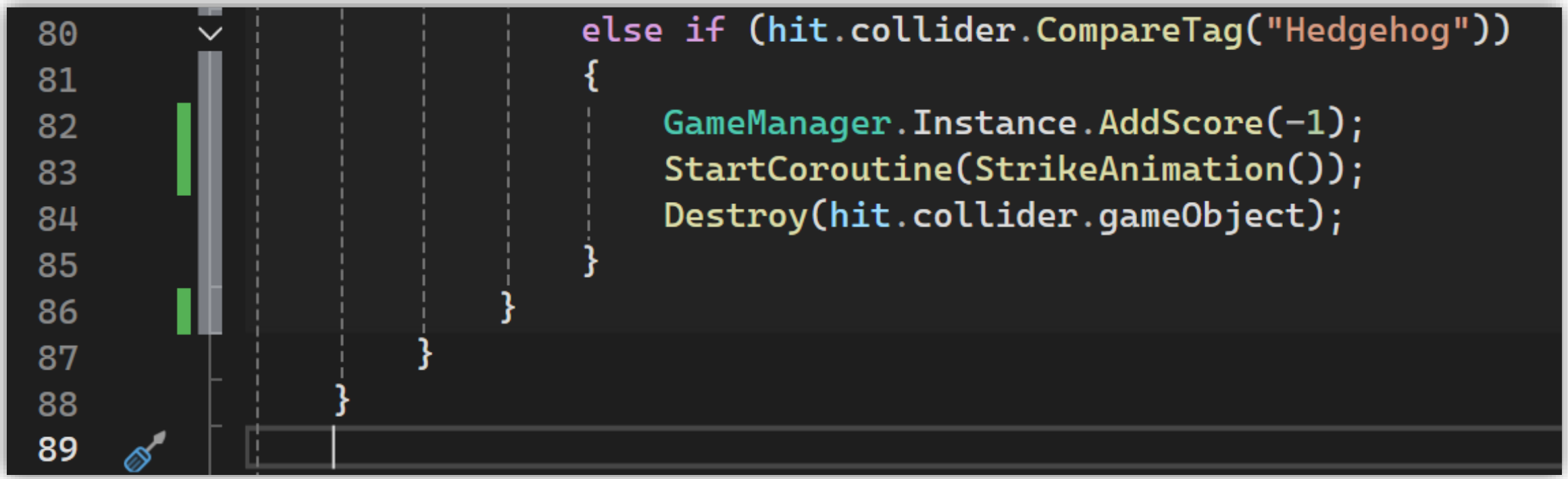
HammerController.cs



อย่าลืมเปลี่ยน User Layer 3 เป็น “Hammer” แล้วกำหนดให้ Hammer ของเรามี Layer เป็น Hammer

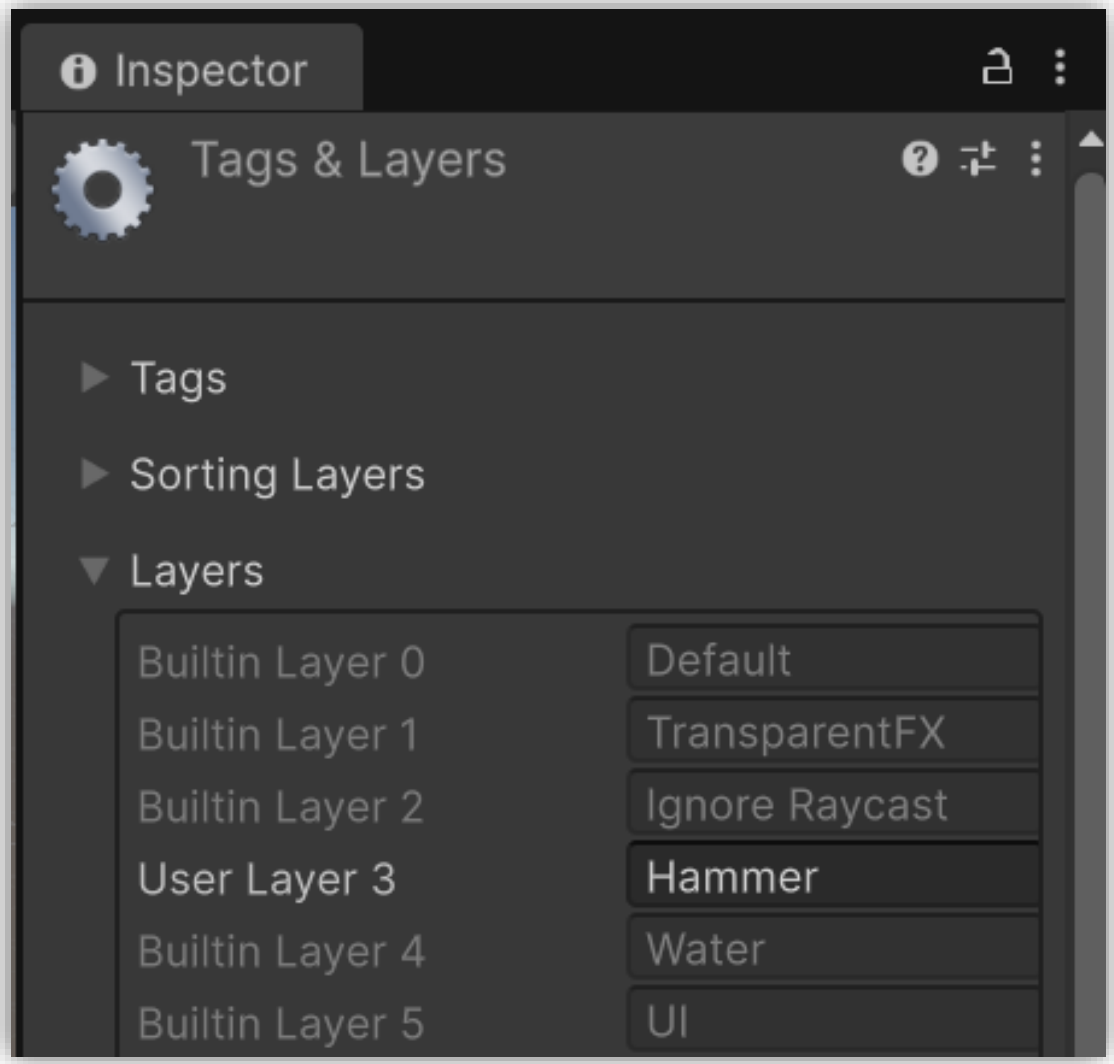
```
64      void HandleHit()
65      {
66          if (Mouse.current.leftButton.wasPressedThisFrame && !isStriking)
67          {
68              Ray ray = Camera.main.ScreenPointToRay(
69                                  Mouse.current.position.ReadValue());
70              int layerMask = ~LayerMask.GetMask("Hammer");
71
72              if (Physics.Raycast(ray, out RaycastHit hit, 100f, layerMask))
73              {
74                  if (hit.collider.CompareTag("Mole"))
75                  {
76                      GameManager.Instance.AddScore(1);
77                      StartCoroutine(StrikeAnimation());
78                      Destroy(hit.collider.gameObject);
79                  }
```

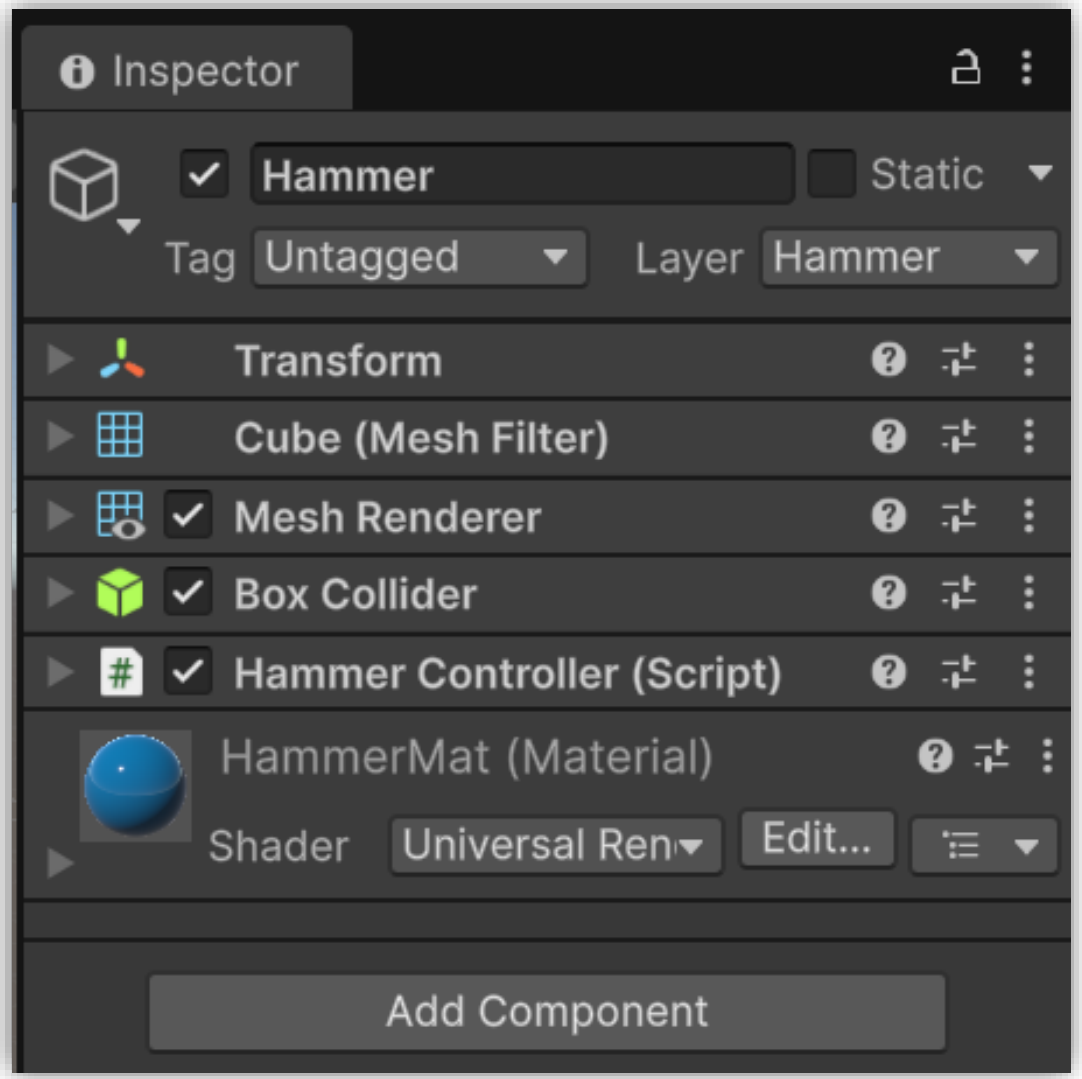
HammerController.cs


80 else if (hit.collider.CompareTag("Hedgehog"))
81 {
82 GameManager.Instance.AddScore(-1);
83 StartCoroutine(StrikeAnimation());
84 Destroy(hit.collider.gameObject);
85 }
86 }
87 }
88 }
89

```
90      void FollowMouse()
91      {
92          Ray ray = Camera.main.ScreenPointToRay(
93                          Mouse.current.position.ReadValue() );
94
95          if (groundPlane.Raycast(ray, out float distance))
96          {
97              Vector3 hitPoint = ray.GetPoint(distance);
98              hitPoint.y += heightOffset;
99
100             // เคลื่อนที่แบบ Smooth
101             transform.position = Vector3.Lerp(
102                 transform.position,
103                 hitPoint,
104                 moveSpeed * Time.deltaTime
105             );
106         }
107     }
108 }
```

HammerController.cs


Inspector
Tags & Layers
Tags
Sorting Layers
Layers
Builtin Layer 0
Default
Builtin Layer 1
TransparentFX
Builtin Layer 2
Ignore Raycast
User Layer 3
Hammer
Builtin Layer 4
Water
Builtin Layer 5
UI


Inspector
Hammer
Static
Tag
Untagged
Layer
Hammer
Transform
Cube (Mesh Filter)
Mesh Renderer
Box Collider
Hammer Controller (Script)
HammerMat (Material)
Shader
Universal Ren
Edit...
Add Component

ระบบเวลา
(Time Management)

ระบบเวลา

- เกมถูกจำกัดเวลาการเล่นที่ 50 วินาที
- เวลาแสดงผลแบบนับถอยหลัง
- เมื่อเวลาหมด จะเปลี่ยนสถานะเป็น TimeUp
- แสดงผลสรุปคะแนนที่ผู้เล่นทำได้
- ระบบเวลาทำหน้าที่เป็นกลไกบังคับจบเกม (Hard Stop Condition)

การแสดงผลและสรุปผลลัพธ์
(Result Presentation)

การแสดงผลและสรุปผล

- เมื่อเกมสิ้นสุด (Win / Lose / TimeUp)
- ปิดการโต้ตอบของผู้เล่น
- แสดงผลลัพธ์และคะแนนสุดท้าย
- ป้องกันการเกิดวัตถุใหม่ในฉาก
- แนวคิดนี้ช่วยแยก “การเล่น” ออกจาก “การแสดงผลลัพธ์”

สรุป
(Conclusion)

การประยุกต์ใช้เชิงวิชาการ

จากการพัฒนาเกมนี้ สามารถสรุปแนวคิดเชิงวิชาการที่เกี่ยวข้องได้ดังนี้

- การใช้ Finite State Machine ในการควบคุมลำดับการทำงานของเกม
- การออกแบบระบบแบบ Object-Oriented Programming (OOP)
- การใช้ Polymorphism ในการจัดการพฤติกรรมของวัตถุหลายชนิด
- การปรับระดับความยากของเกมแบบ Dynamic Difficulty Adjustment

แนวคิดเหล่านี้เป็นพื้นฐานสำคัญในการพัฒนาเกมและซอฟต์แวร์เชิงโต้ตอบในระดับที่สูงขึ้น

สรุป

- โครงงานเกมตีตัวตุ่นสามมิแสดงให้เห็นถึงการบูรณาการระหว่างแนวคิดการออกแบบเกมและหลักการเขียนโปรแกรมเชิงโครงสร้าง โดยเฉพาะการใช้ Finite State Machine เพื่อจัดการสถานะของเกมอย่างมีประสิทธิภาพ
- ผลลัพธ์ที่ได้คือเกมที่มีโครงสร้างชัดเจน เข้าใจง่าย และสามารถนำไปต่อยอดหรือขยายระบบเพิ่มเติมได้ในอนาคต เช่น การเพิ่มระบบจับเวลา ระบบ Pause หรือการบันทึกคะแนนสูงสุด เป็นต้น
- โครงงานนี้จึงเหมาะสมที่จะใช้เป็นตัวอย่างการเรียนรู้ด้านการพัฒนาเกม

Q&A