物件導向程式設計期末專案報告

Unity 3D 傳送門解謎遊戲

課程名稱： 物件導向程式設計 (Object-Oriented Programming)
專案名稱： Puzzle Game - Portal Mechanics
小組編號： Group 13
提交日期： 2025年12月29日

目錄

---

1. 專案概述

1.1 專案簡介

本專案開發了一款基於Unity引擎的3D第一人稱解謎遊戲，核心玩法靈感來源於經典遊戲《傳送門 (Portal)》。玩家可以在場景中發射兩個相互連接的傳送門，通過傳送門在空間中移動，並利用物理機制解決各種謎題。

1.2 專案目標

- **學習目標：** 實踐物件導向程式設計的核心概念（封裝、繼承、多型、抽象）
- **技術目標：** 掌握Unity遊戲開發框架與C#程式設計
- **設計目標：** 建立可擴展、可維護的遊戲架構

1.3 核心功能

- **傳送門系統：** 雙向傳送門的創建、連接與物件傳送
- **角色控制：** 第一人稱視角移動、跳躍與相機控制
- **物體互動：** 拾取、放置與投擲可互動物體
- **雷射機關：** 動態雷射發射與反射系統
- **關卡管理：** 目標點檢測與關卡重置機制

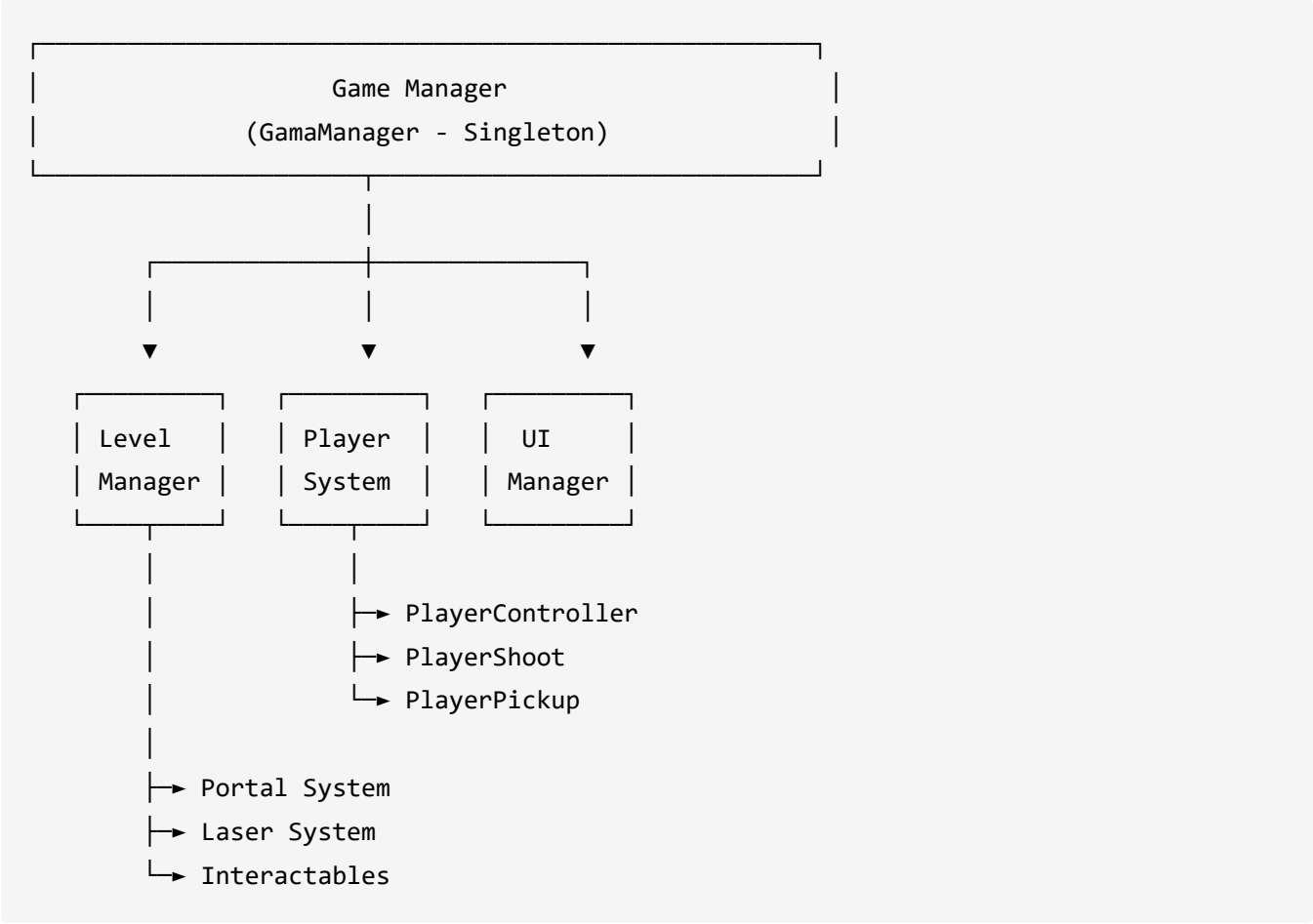
1.4 開發環境

- **遊戲引擎：** Unity 2022.x
- **程式語言：** C# (.NET Framework)
- **開發工具：** Visual Studio 2022
- **版本控制：** Git / GitHub

---

2. 系統架構

2.1 整體架構圖



2.2 類別層次結構

```
MonoBehaviour (Unity基類)
│
├─► Singleton<T>
│   ├─► GamaManager
│   ├─► LevelManager
│   ├─► PlayerUIManager
│   ├─► PlayerShoot
│   └─► PlayerPickup
│
├─► PortalTravellerSingleton<T> : Singleton<T>
│   └─► PlayerController
│
├─► PortalTraveller
│   ├─► PlayerController
│   └─► RigidbodyTraveller
│
├─► Pickupable
│   └─► (可擴展特殊物品類別)
│
├─► Portal
├─► Laser
├─► LaserEmitter
└─► GoalPoint
```

---

3. 核心物件導向設計

3.1 封裝 (Encapsulation)

3.1.1 單例模式基類

```
public class Singleton<T> : MonoBehaviour where T : MonoBehaviour
{
    private static T _instance;

    public static T instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = FindObjectOfType<T>();
                if (_instance == null)
                {
                    Debug.LogError("Cannot find " + typeof(T) + "!");
                }
            }
            return _instance;
        }
    }
}
```

設計理念:

- 使用泛型類別實現通用單例模式
- 私有靜態變數確保唯一實例
- 公開屬性提供全域存取點
- 延遲初始化 (Lazy Initialization) 優化效能

3.1.2 資料封裝範例

```
public class PlayerController : PortalTravellerSingleton<PlayerController>
{
    [Header("Movement")]
    public float walkSpeed = 4f;       // 公開可調整參數
    private Vector3 velocity;          // 私有內部狀態
    private CharacterController characterController;  // 私有組件引用

    // 公開方法提供受控存取
    public void ResetPosition() { ... }
}
```

封裝優勢：

- 隱藏內部實作細節
- 提供清晰的公開介面
- 防止不當存取與修改
- 便於Unity編輯器調整參數

3.2 繼承 (Inheritance)

3.2.1 傳送門旅行者繼承鏈

```
// 基礎類別：定義傳送門穿越行為
public class PortalTraveller : MonoBehaviour
{
    public GameObject graphicsObject;
    public GameObject graphicsClone { get; set; }

    public virtual void Teleport(Transform fromPortal, Transform toPortal,
                                 Vector3 pos, Quaternion rot)
    {
        transform.position = pos;
        transform.rotation = rot;
    }

    public virtual void EnterPortalTrigger() { ... }
    public virtual void ExitPortalTrigger() { ... }
}

// 衍生類別：剛體物件的特殊處理
public class RigidbodyTraveller : PortalTraveller
{
    public override void Teleport(Transform fromPortal, Transform toPortal,
                                  Vector3 pos, Quaternion rot)
    {
        base.Teleport(fromPortal, toPortal, pos, rot);
        // 額外處理速度向量的轉換
        Rigidbody rb = GetComponent<Rigidbody>();
        rb.velocity = TransformVelocity(...);
    }
}

// 玩家控制器：結合單例與傳送
public class PlayerController : PortalTravellerSingleton<PlayerController>
{
    // 整合角色控制與傳送門機制
}
```

繼承優勢：

- 程式碼重用減少冗餘
- 建立清晰的類別階層
- 支援多態行為
- 便於擴展新型可傳送物件

3.2.2 可拾取物品繼承體系

```
public class Pickupable : MonoBehaviour
{
    protected bool isPicked = false;

    public virtual void OnPickup(PlayerPickup holder) { ... }
    public virtual void OnDrop(bool thrown) { ... }
    protected virtual void FixedUpdate() { ... }
}

// 未來可擴展：
// public class WeightedBox : Pickupable { ... }
// public class ExplosiveBarrel : Pickupable { ... }
```

3.3 多型 (Polymorphism)

3.3.1 虛擬方法覆寫

```
// PortalTraveller基類定義虛擬方法
public virtual void Teleport(Transform fromPortal, Transform toPortal,
                             Vector3 pos, Quaternion rot)
{
    transform.position = pos;
    transform.rotation = rot;
}

// PlayerController覆寫以處理角色控制器
public override void Teleport(...)
{
    base.Teleport(...);
    characterController.enabled = false;
    // 特殊處理
    characterController.enabled = true;
}

// RigidbodyTraveller覆寫以處理物理
public override void Teleport(...)
{
    base.Teleport(...);
    Rigidbody.velocity = TransformVelocity(...);
}
```

多型優勢:

- 統一介面處理不同物件
- Portal類別無需關心具體傳送對象類型
- 新增傳送物件類型無需修改現有程式碼
- 符合開放封閉原則 (Open-Closed Principle)

3.3.2 接口統一性

```
// Portal.cs 中使用統一介面
List<PortalTraveller> trackedTravellers;

void UpdateTravellers()
{
    foreach (var traveller in trackedTravellers)
    {
        // 多型調用：根據實際類型執行不同行為
        traveller.Teleport(fromPortal, toPortal, newPos, newRot);
    }
}
```

3.4 抽象 (Abstraction)

3.4.1 管理器抽象層

```
// LevelManager：抽象關卡管理細節
public class LevelManager : Singleton<LevelManager>
{
    // 提供高層抽象介面
    public void OnPlayerArriveAtGoal() { ... }
    public void ResetPlayerPosition(Transform player) { ... }
    public void LoadLevel(int level) { ... }

    // 隱藏內部實作
    private void InitializePortals() { ... }
    private void ConfigurePhysicsLayers() { ... }
}
```

抽象優勢：

- 簡化複雜系統的使用
- 降低系統間耦合度
- 提升程式碼可讀性
- 便於系統維護與擴展

3.4.2 工具抽象

```
// CameraUtility: 抽象相機計算細節
public static class CameraUtility
{
    public static bool SegmentQuad(Vector3 p1, Vector3 p2, Transform quad)
    {
        // 複雜的幾何計算被抽象為簡單方法
        // 使用者無需理解內部數學原理
    }
}
```

---

4. 主要功能模組

4.1 傳送門系統

4.1.1 Portal類別設計

```
public class Portal : MonoBehaviour
{
    [Header("Main Settings")]
    public Portal linkedPortal;           // 連接的另一個傳送門
    public Collider screenCollider;       // 傳送門屏幕碰撞器
    public MeshRenderer screen;           // 渲染表面
    public int recursionLimit = 3;        // 遞迴渲染深度

    RenderTexture viewTexture;            // 渲染紋理
    Camera portalCamera;                  // 傳送門相機
    List<PortalTraveller> trackedTravellers;  // 追蹤的穿越物件

    // 核心方法
    public static Portal SpawnPortal(...)  // 生成傳送門
    public void Render()                    // 渲染視圖
    public void Teleport(PortalTraveller traveller)  // 執行傳送
}
```

4.1.2 傳送機制實作

關鍵技術點:

1. **坐標轉換:** 使用Matrix4x4進行空間座標變換
2. **視角渲染:** 動態相機定位與RenderTexture
3. **碰撞檢測:** OnTriggerEnter/Exit追蹤物件進出
4. **無縫傳送:** 克隆物件實現平滑過渡

```
// 坐標變換示例
Matrix4x4 m = linkedPortal.transform.localToWorldMatrix *
              Matrix4x4.Rotate(Quaternion.Euler(0f, 180f, 0f)) *
              transform.worldToLocalMatrix;
Vector3 newPos = m.MultiplyPoint3x4(traveller.transform.position);
Quaternion newRot = m.rotation * traveller.transform.rotation;
```

4.1.3 遞迴渲染

```
void Render()
{
    if (!linkedPortal) return;

    // 設定相機位置與方向
    SetupPortalCamera();

    // 遞迴渲染另一側傳送門視圖
    if (recursionDepth < recursionLimit)
    {
        linkedPortal.Render(recursionDepth + 1);
    }

    portalCamera.targetTexture = viewTexture;
    portalCamera.Render();
}
```

4.2 玩家控制系統

4.2.1 PlayerController

功能特性:

- 第一人稱角色控制器 (CharacterController)
- 滑鼠視角控制（Pitch/Yaw）
- 移動、跑步、跳躍
- 重力與地面檢測
- 傳送門穿越整合

關鍵程式碼:

```
void Update()
{
    // 視角控制
    float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity;
    float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity;

    yaw += mouseX;
    pitch -= mouseY;
    pitch = Mathf.Clamp(pitch, minPitch, maxPitch);

    // 移動輸入
    float horizontal = Input.GetAxisRaw("Horizontal");
    float vertical = Input.GetAxisRaw("Vertical");

    Vector3 moveDirection = new Vector3(horizontal, 0, vertical);
    moveDirection = transform.TransformDirection(moveDirection);

    // 應用重力
    velocity.y += gravity * Time.deltaTime;

    // 執行移動
    characterController.Move((moveDirection * speed + velocity) * Time.deltaTime);
}
```

4.2.2 PlayerShoot

射擊傳送門機制:

- 滑鼠左鍵/右鍵發射不同傳送門
- Raycast檢測有效表面
- 動態生成與管理傳送門實例

```
void PerformShoot(int button)
{
    int layerMask = ~LayerMask.GetMask("Ignore Raycast", "Portal", ...);

    if (Physics.Raycast(eyeTransform.position, eyeTransform.forward,
                        out RaycastHit hit, maxShootDistance, layerMask))
    {
        if (button == 0)
            portal1 = Portal.SpawnPortal(portalPrefab, portal2, hit, ...);
        else
            portal2 = Portal.SpawnPortal(portalPrefab, portal1, hit, ...);
    }
}
```

4.2.3 PlayerPickup

拾取系統特性:

- E鍵拾取/放下物品
- Q鍵投擲物品
- HoldPoint追蹤與傳送門整合
- 物理力施加

```
void TryPickup()
{
    int layerMask = LayerMask.GetMask("Pickupable");
    if (Physics.Raycast(eyeTransform.position, eyeTransform.forward,
                        out RaycastHit hit, pickupRange, layerMask))
    {
        Pickupable obj = hit.collider.GetComponent<Pickupable>();
        if (obj != null)
        {
            obj.OnPickup(this);
            heldObject = obj;
        }
    }
}
```

4.3 可互動物體系統

4.3.1 Pickupable基類

```
public class Pickupable : MonoBehaviour
{
    public Rigidbody rigid;
    protected bool isPicked = false;

    // 拾取時的物理調整
    public virtual void OnPickup(PlayerPickup holder)
    {
        isPicked = true;
        rigid.drag = pickedDrag;
        rigid.angularDrag = pickedAngularDrag;
        // 設定追蹤點
        holdAnchor = holder.holdPoint;
    }

    // 物理更新：保持在HoldPoint附近
    protected virtual void FixedUpdate()
    {
        if (isPicked)
        {
            float distance = Vector3.Distance(transform.position, holdAnchor.position);
            if (distance > moveThreshold)
            {
                Vector3 direction = holdAnchor.position - transform.position;
                rigid.AddForce(direction * pickupForce);
            }
        }
    }
}
```

設計亮點:

- 使用物理力而非直接設定位置（更自然）
- 支援傳送門下的HoldPoint切換
- 虛擬方法便於擴展特殊物品

4.4 雷射系統

4.4.1 LaserEmitter

功能:

- 發射雷射光束
- 支援反射（通過Portal或鏡面）
- 動態碰撞檢測
- 鏈式雷射段管理

```
public void EmitLaser()
{
    float remainLength = maxLength;

    for (int i = 0; i < lasers.Count; i++)
    {
        if (lasers[i].gameObject.activeSelf)
        {
            float usedLength = LaserRaycast(i, remainLength);
            remainLength -= usedLength;
            if (remainLength <= 0) break;
        }
    }
}

float LaserRaycast(int index, float length)
{
    // 發射射線
    // 調整雷射段的縮放與位置
    // 處理碰撞（Player、Portal等）
    // 返回使用的長度
}
```

4.4.2 Laser觸發器

```
public class Laser : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            LevelManager.instance.ResetPlayerPosition(other.transform);
        }
        GetComponentInParent<LaserEmitter>().EmitLaser();
    }
}
```

4.5 關卡管理系統

4.5.1 LevelManager

```
public class LevelManager : Singleton<LevelManager>
{
    public Transform goalPoint;
    public int currentLevel = 1;
    public List<Portal> portals;

    void Start()
    {
        // 設定物理層碰撞規則
        Physics.IgnoreLayerCollision(
            LayerMask.NameToLayer("Portal Traveller"),
            LayerMask.NameToLayer("Portal"), true);

        portals = new List<Portal>(FindObjectsOfType<Portal>());
    }

    public void OnPlayerArriveAtGoal()
    {
        Debug.Log("Player has reached the goal!");
        // 關卡完成邏輯
    }

    public void ResetPlayerPosition(Transform player)
    {
        player.position = Vector3.zero;
        player.rotation = Quaternion.identity;
    }
}
```

4.5.2 GoalPoint

```
public class GoalPoint : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            LevelManager.instance.OnPlayerArriveAtGoal();
        }
    }
}
```

---

5. OOP設計模式應用

5.1 Singleton Pattern（單例模式）

應用場景：

- GamaManager（遊戲總管理器）
- LevelManager（關卡管理器）
- PlayerController（玩家控制器）
- PlayerShoot（射擊系統）
- PlayerPickup（拾取系統）
- PlayerUIManager（UI管理器）

優勢：

- 確保唯一實例
- 提供全域存取點
- 避免重複初始化

實作細節：

```
public class Singleton<T> : MonoBehaviour where T : MonoBehaviour
{
    private static T _instance;

    public static T instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = FindObjectOfType<T>();
                if (_instance == null)
                {
                    Debug.LogError("Cannot find " + typeof(T) + "!");
                }
            }
            return _instance;
        }
    }
}
```

5.2 Template Method Pattern（模板方法模式）

應用場景： PortalTraveller繼承體系

```
public class PortalTraveller : MonoBehaviour
{
    // 模板方法：定義傳送流程
    public void PerformTeleport(Portal from, Portal to)
    {
        // 1. 準備階段
        PreTeleport();

        // 2. 執行傳送（可覆寫）
        Teleport(from.transform, to.transform, newPos, newRot);

        // 3. 後處理
        PostTeleport();
    }

    // 可覆寫的虛擬方法
    public virtual void Teleport(...) { }
    protected virtual void PreTeleport() { }
    protected virtual void PostTeleport() { }
}
```

5.3 Observer Pattern（觀察者模式）

應用場景： 事件系統

```
// 目標點檢測
public class GoalPoint : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            // 通知LevelManager（觀察者）
            LevelManager.instance.OnPlayerArriveAtGoal();
        }
    }
}

// LevelManager作為觀察者響應事件
public class LevelManager : Singleton<LevelManager>
{
    public void OnPlayerArriveAtGoal()
    {
        // 處理關卡完成事件
        Debug.Log("Level Complete!");
        // 可擴展：UI更新、音效播放、統計記錄等
    }
}
```

5.4 Object Pool Pattern（物件池模式）

應用場景： 雷射段管理

```
public class LaserEmitter : MonoBehaviour
{
    private List<Laser> lasers;  // 物件池

    public Laser LaserInit(int index, Vector3 position, Quaternion rotation)
    {
        // 重用現有物件
        if (lasers.Count <= index)
        {
            lasers.Add(Instantiate(laserPrefab, ...).GetComponent<Laser>());
        }

        // 啟用並配置
        lasers[index].gameObject.SetActive(true);
        lasers[index].startPosition = position;
        return lasers[index];
    }

    public void EmitLaser()
    {
        // 停用未使用的物件
        for (int i = 1; i < lasers.Count; i++)
        {
            lasers[i].gameObject.SetActive(false);
        }
        // 按需啟用
    }
}
```

優勢:

- 減少Instantiate/Destroy開銷
- 提升效能
- 降低垃圾回收壓力

5.5 Strategy Pattern（策略模式）

潛在應用: 不同傳送門行為

```
// 可擴展設計
public interface IPortalBehavior
{
    void OnEnter(PortalTraveller traveller);
    void OnExit(PortalTraveller traveller);
}

public class StandardPortalBehavior : IPortalBehavior { ... }
public class GravityFlipPortalBehavior : IPortalBehavior { ... }

public class Portal : MonoBehaviour
{
    public IPortalBehavior behavior = new StandardPortalBehavior();

    void ProcessTraveller(PortalTraveller t)
    {
        behavior.OnEnter(t);
    }
}
```

---

6. 技術實作細節

6.1 物理系統整合

6.1.1 層級碰撞矩陣

```
void Start()
{
    // 防止傳送門與穿越者碰撞
    Physics.IgnoreLayerCollision(
        LayerMask.NameToLayer("Portal Traveller"),
        LayerMask.NameToLayer("Portal"), true);

    Physics.IgnoreLayerCollision(
        LayerMask.NameToLayer("Clone Traveller"),
        LayerMask.NameToLayer("Portal"), true);
}
```

層級設計：

- `Portal Traveller`：可傳送物件
- `Clone Traveller`：克隆物件（視覺用）
- `Portal`：傳送門碰撞器
- `Portal Frame`：傳送門邊框
- `Player`：玩家
- `Pickupable`：可拾取物品

6.1.2 CharacterController vs Rigidbody

PlayerController使用CharacterController：

- 優勢：更好的角色控制、內建階梯攀爬
- 挑戰：傳送時需要特殊處理

```
public override void Teleport(...)
{
    characterController.enabled = false;  // 臨時禁用
    base.Teleport(fromPortal, toPortal, pos, rot);
    characterController.enabled = true;   // 重新啟用
}
```

可拾取物品使用Rigidbody：

- 完整物理模擬
- 支援力與扭矩
- 傳送時需轉換速度向量

6.2 渲染技術

6.2.1 RenderTexture動態視圖

```
void CreateViewTexture()
{
    if (viewTexture == null || viewTexture.width != Screen.width)
    {
        if (viewTexture != null)
            viewTexture.Release();

        viewTexture = new RenderTexture(Screen.width, Screen.height, 24);
    }
    screen.material.mainTexture = viewTexture;
}
```

6.2.2 相機定位計算

```
void SetupPortalCamera()
{
    // 計算玩家相機相對於當前傳送門的位置
    Matrix4x4 m = linkedPortal.transform.localToWorldMatrix *
                  Matrix4x4.Rotate(Quaternion.Euler(0f, 180f, 0f)) *
                  transform.worldToLocalMatrix *
                  playerCamera.transform.localToWorldMatrix;

    portalCamera.transform.SetPositionAndRotation(
        m.GetPosition(),
        m.rotation);
}
```

6.2.3 遞迴渲染優化

```
public void Render(int recursionDepth = 0)
{
    if (recursionDepth >= recursionLimit) return;

    // 先渲染下一層
    if (linkedPortal)
        linkedPortal.Render(recursionDepth + 1);

    // 再渲染當前層
    portalCamera.Render();
}
```

6.3 數學與幾何計算

6.3.1 坐標空間轉換

```
// 世界空間 → 本地空間 → 連接傳送門本地空間 → 連接傳送門世界空間
Matrix4x4 m = linkedPortal.transform.localToWorldMatrix *
              Matrix4x4.Rotate(Quaternion.Euler(0f, 180f, 0f)) *
              transform.worldToLocalMatrix;

Vector3 newPosition = m.MultiplyPoint3x4(oldPosition);
Quaternion newRotation = m.rotation * oldRotation;
Vector3 newVelocity = m.MultiplyVector(oldVelocity);
```

6.3.2 傳送門放置驗證

```
static bool CanSpawnPortal(Vector3 position, Vector3 forward, Vector3 up,
                           Portal ignorePortal, out Vector3 newPosition)
{
    // 檢查四邊與四角是否有足夠空間
    Vector3 right = Vector3.Cross(up, forward);
    float halfWidth = defaultScale.x * 0.5f;
    float halfHeight = defaultScale.y * 0.5f;

    // 邊緣檢測
    foreach (var edge in edgePermutations)
    {
        Vector3 checkPoint = position + right * edge.x * halfWidth +
                             up * edge.y * halfHeight;

        if (Physics.CheckBox(checkPoint, ...))
            return false;  // 空間不足
    }

    return true;
}
```

6.3.3 線段與平面相交檢測

```
public static bool SegmentQuad(Vector3 p1, Vector3 p2, Transform quad)
{
    // 計算線段與四邊形平面的交點
    // 判斷交點是否在四邊形內
    // 用於檢測物體是否穿過傳送門
}
```

6.4 性能優化策略

6.4.1 幀率限制

```
public class FPSLimit : MonoBehaviour
{
    void Start()
    {
        QualitySettings.vSyncCount = 0;
        Application.targetFrameRate = 60;
    }
}
```

6.4.2 條件渲染

```
void Update()
{
    // 僅在傳送門可見時渲染
    if (IsVisibleToCamera(playerCamera))
    {
        Render();
    }
}
```

6.4.3 物件池管理

- 雷射段重用避免頻繁Instantiate
- 克隆物件的啟用/停用而非創建/銷毀

7. 開發心得與反思

7.1 OOP概念的實踐體會

7.1.1 封裝的重要性

在開發過程中，我們深刻體會到良好封裝的價值：

- **資料保護：** 私有變數防止了意外修改，避免了許多潛在bug
- **介面清晰：** 公開方法提供明確的使用方式，降低團隊協作成本
- **易於調試：** 封裝使得問題定位更加容易

案例： PlayerController的velocity變數設為private，所有速度修改都通過Update()方法進行，確保了物理計算的一致性。

7.1.2 繼承的雙面性

優勢體驗：

- `PortalTraveller` 基類大幅減少了程式碼重複
- 新增可傳送物件類型非常方便

遇到的挑戰：

- 繼承鏈過深可能導致理解困難
- 多重繼承限制（C#單繼承）

解決方案：

- 保持繼承層次淺（最多3層）
- 優先使用組合而非繼承（Component-based）

7.1.3 多型的威力

印象最深的應用：

```
// Portal.cs 中的通用處理
foreach (var traveller in trackedTravellers)
{
    traveller.Teleport(...);  // 根據實際類型調用不同實現
}
```

這段程式碼優雅地處理了玩家、物品、克隆體等不同物件的傳送，無需任何類型判斷。

7.1.4 抽象的藝術

良好抽象的體現：

- `LevelManager` 抽象了關卡管理細節
- `CameraUtility` 抽象了複雜的幾何計算
- 使用者只需調用簡單方法，無需理解內部實作

7.2 Unity與OOP的結合

7.2.1 Component-Based架構

Unity的Component系統本身就是優秀的OOP實踐：

- **組合優於繼承：** GameObject通過添加Component組合功能
- **介面分離：** 每個Component負責單一職責
- **依賴注入：** 通過Inspector注入依賴

我們的應用：

```
[RequireComponent(typeof(PlayerController))]
public class PlayerShoot : Singleton<PlayerShoot>
{
    PlayerController playerController;

    void Awake()
    {
        playerController = PlayerController.instance;
    }
}
```

7.2.2 生命週期方法

Unity的MonoBehaviour提供了明確的生命週期：

- `Awake()`：初始化
- `Start()`：開始前設置
- `Update()`：每幀更新
- `FixedUpdate()`：固定物理更新
- `OnDestroy()`：清理資源

最佳實踐：

- 初始化在Awake/Start完成
- 邏輯更新在Update
- 物理操作在FixedUpdate

7.3 遇到的技術挑戰

7.3.1 傳送門穿越的平滑性

問題： 物件穿越傳送門時出現抖動或卡頓

解決方案：

1. 使用克隆物件實現視覺連續性
2. 精確計算穿越點
3. 調整碰撞層級避免干擾

```
public virtual void EnterPortalTrigger()
{
    graphicsClone = Instantiate(graphicsObject);
    // 克隆體在傳送門另一側顯示
}
```

7.3.2 物理與傳送的衝突

問題： CharacterController在傳送時產生異常

解決： 臨時禁用再啟用

```
characterController.enabled = false;
transform.SetPositionAndRotation(newPos, newRot);
characterController.enabled = true;
```

7.3.3 渲染性能優化

問題： 遞迴渲染導致幀率下降

優化措施：

1. 限制遞迴深度（recursionLimit = 3）
2. 條件渲染（僅渲染可見傳送門）
3. 降低RenderTexture解析度

7.3.4 拾取系統的複雜性

挑戰： 拾取的物品需要在傳送門間正確追蹤

實作：

- 雙HoldPoint系統（holdPoint + holdPointTP）
- 動態選擇距離最近的HoldPoint
- 傳送時同步更新HoldPoint位置

7.4 團隊協作經驗

7.4.1 程式碼規範

建立的規範：

- 類別命名：PascalCase
- 變數命名：camelCase
- 私有變數：_下劃線前綴（部分）
- 註解：重要邏輯必須註解

7.4.2 版本控制

Git使用經驗：

- 功能分支開發
- 提交前本地測試
- 清晰的commit訊息

7.4.3 任務分工

- **成員A：** 傳送門核心系統
- **成員B：** 玩家控制與射擊
- **成員C：** 拾取系統與物品
- **成員D：** 雷射系統與關卡管理

7.5 收穫與成長

7.5.1 技術能力提升

- **C#程式設計：** 深入掌握泛型、委派、事件
- **Unity引擎：** 理解物理系統、渲染管線
- **數學應用：** 向量、矩陣、四元數的實際運用

7.5.2 設計思維進步

- **先設計後編碼：** 類別圖與UML的重要性
- **可擴展性考量：** 為未來功能預留介面
- **性能意識：** 時刻考慮優化

7.5.3 問題解決能力

- **調試技巧：** 善用Debug.Log、斷點、Gizmos
- **文檔查閱：** Unity官方文檔、社群資源
- **思維方式：** 分解問題、逐步驗證

7.6 未來改進方向

7.6.1 功能擴展

- [ ] 更多傳送門類型（單向、延時、縮放）
- [ ] 關卡編輯器
- [ ] 存檔系統

- [ ] 音效與音樂
- [ ] 更豐富的謎題元素

7.6.2 架構優化

- [ ] 引入事件系統（UnityEvent或自定義）
- [ ] 狀態機管理（玩家狀態、遊戲狀態）
- [ ] 資源管理系統（AssetBundle）
- [ ] 網路多人（Mirror或Netcode）

7.6.3 程式碼品質

- [ ] 單元測試（Unity Test Framework）
- [ ] 程式碼審查流程
- [ ] 性能分析（Profiler）
- [ ] 記憶體優化

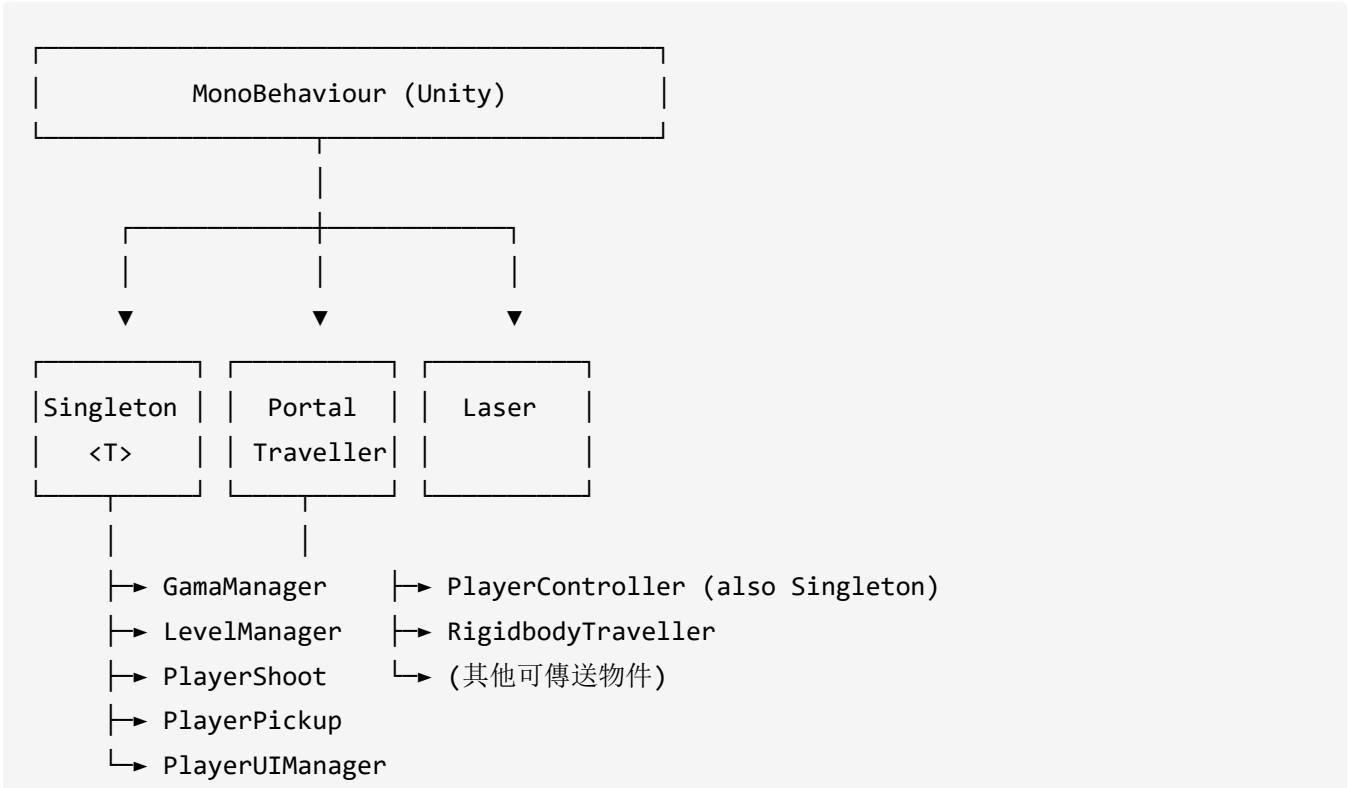
---

8. 附錄

8.1 專案結構

```
Puzzle Game/
├── Assets/
│   ├── Scenes/              # 遊戲場景
│   │   └── SampleScene.unity
│   ├── Scripts/             # C#腳本
│   │   ├── Managers/        # 管理器類別
│   │   │   ├── GamaManager.cs
│   │   │   ├── LevelManager.cs
│   │   │   └── FPSLimit.cs
│   │   ├── Player/          # 玩家系統
│   │   │   ├── PlayerController.cs
│   │   │   ├── PlayerShoot.cs
│   │   │   ├── PlayerPickup.cs
│   │   │   └── MainCamera.cs
│   │   ├── Tools/           # 工具與道具
│   │   │   ├── Portal.cs
│   │   │   ├── PortalGun.cs (空)
│   │   │   ├── Laser.cs
│   │   │   └── LaserEmitter.cs
│   │   ├── Interactables/   # 可互動物體
│   │   │   ├── Pickupable.cs
│   │   │   └── RigidbodyTraveller.cs
│   │   ├── UI/              # 使用者介面
│   │   │   ├── PlayerUIManager.cs
│   │   │   └── FPSCounter.cs
│   │   ├── Singleton.cs     # 單例基類
│   │   ├── PortalTraveller.cs  # 傳送者基類
│   │   ├── GoalPoint.cs     # 目標點
│   │   └── CameraUtility.cs # 相機工具
│   ├── Materials/           # 材質資源
│   ├── Resources/           # 資源文件
│   └── Settings/            # 項目設定
├── ProjectSettings/         # Unity專案設定
├── Packages/                # 套件依賴
└── docs/                    # 文檔資料
    └── sprint-artifacts/    # 敏捷開發文件
```

8.2 類別關係圖



8.3 核心類別說明表

類別名稱	類型	主要職責	關鍵方法
`Singleton<T>`	基類	單例模式實現	`instance`
`GamaManager`	管理器	遊戲總控制	`Start()`
`LevelManager`	管理器	關卡管理	`OnPlayerArriveAtGoal()`, `ResetPlayerPosition()`
`PlayerController`	控制器	玩家移動	`Update()`, `Teleport()`
`PlayerShoot`	控制器	射擊傳送門	`PerformShoot()`
`PlayerPickup`	控制器	拾取物品	`TryPickup()`, `Drop()`
`Portal`	工具	傳送門邏輯	`SpawnPortal()`, `Render()`, `Teleport()`
`PortalTraveller`	基類	傳送行為	`Teleport()`, `EnterPortalTrigger()`

類別名稱	類型	主要職責	關鍵方法
`Pickupable`	互動物	可拾取物品	`OnPickup()`, `OnDrop()`
`LaserEmitter`	工具	雷射發射	`EmitLaser()`, `LaserRaycast()`
`Laser`	互動物	雷射碰撞	`OnTriggerEnter()`
`GoalPoint`	互動物	目標檢測	`OnTriggerEnter()`

8.4 OOP概念應用總結

OOP概念	應用範例	效益
封裝	`Singleton<T>` 的私有 `_instance`	保護資料、提供受控存取
繼承	`PortalTraveller` → `PlayerController`	程式碼重用、建立階層
多型	`Teleport()` 虛擬方法覆寫	統一介面、不同實現
抽象	`LevelManager` 隱藏複雜邏輯	簡化使用、降低耦合
泛型	`Singleton<T>`	類型安全、通用性
介面	（潛在） `IPortalBehavior`	解耦、可替換

8.5 設計模式應用總結

設計模式	應用位置	解決問題
Singleton	各Manager類別	全域存取、唯一實例
Template Method	`PortalTraveller.Teleport()`	定義算法框架
Observer	`GoalPoint` → `LevelManager`	事件通知
Object Pool	`LaserEmitter.lasers`	物件重用、性能優化
Strategy	（未完全實現）Portal行為	可替換算法

8.6 Unity特性應用

Unity特性	使用方式	目的
`[Header]`	`[Header("Movement")]`	編輯器分組
`[SerializeField]`	私有變數序列化	編輯器可見
`[RequireComponent]`	`[RequireComponent(typeof(...))]`	依賴檢查
`MonoBehaviour`	所有遊戲腳本基類	Unity生命週期
`RenderTexture`	Portal視圖	動態渲染
`LayerMask`	碰撞過濾	物理控制

8.7 參考資料

8.7.1 官方文檔

- Unity Documentation: https://docs.unity3d.com/
- C# Programming Guide: https://docs.microsoft.com/en-us/dotnet/csharp/
- Unity Physics: https://docs.unity3d.com/Manual/PhysicsSection.html

8.7.2 設計模式

- Game Programming Patterns by Robert Nystrom
- Design Patterns: Elements of Reusable Object-Oriented Software

8.7.3 靈感來源

- Portal (Valve Corporation)
- Unity官方教程與範例專案

8.8 專案統計

項目	數量
C#腳本文件	22

項目	數量
程式碼行數（估計）	~3000
核心類別數	15
單例類別數	6
繼承層級深度	3
Unity場景	1

8.9 開發時程（估計）

階段	任務	時間
第1週	需求分析、系統設計	-
第2週	基礎架構（Singleton、Managers）	-
第3週	玩家控制系統	-
第4週	傳送門系統核心	-
第5週	拾取與互動系統	-
第6週	雷射系統	-
第7週	整合測試與優化	-
第8週	文檔撰寫	-

結語

本專案成功實踐了物件導向程式設計的核心概念，通過Unity遊戲開發的實際場景，深入理解了封裝、繼承、多型與抽象的威力。我們不僅完成了一款可玩的解謎遊戲，更重要的是建立了可擴展、可維護的程式碼架構。

在開發過程中，我們遭遇了許多技術挑戰，從物理系統的複雜性到渲染優化的困難，但通過團隊

協作與不斷學習，我們逐一克服了這些障礙。這些經驗將成為我們未來軟體開發生涯的寶貴財富。

OOP的核心價值在於：

- **模組化：** 清晰的職責劃分
- **重用性：** 減少重複程式碼
- **可擴展：** 便於添加新功能
- **可維護：** 易於理解與修改

我們相信，這個專案不僅展示了技術能力，更體現了我們對軟體工程原則的理解與應用。未來，我們將繼續深化這些概念，追求更優雅的程式碼設計。

小組成員簽名：

- _____
- _____
- _____
- _____

指導教授： _____

日期： 2025年12月29日