

輔仁大學資訊工程學系
畢業專題報告
B03 組

Rish Engine - 2D 遊戲引擎

從零開始自幹遊戲引擎

成員

406262515 鍾秉桓 me@roy4801.tw
406262084 梁博全 me@icejj.tw
406262319 黃育皓 suntalk1224@gmail.com
406262163 黃品翰 william31212@gmail.com

報告編號: CS109-PR-B03

指導教授
鄭進和

December 9, 2020

Contents

1 摘要	3
1.1 背景簡介	3
1.2 問題說明	3
1.3 實作結果	3
2 緒論	4
2.1 動機	4
2.2 問題描述	4
2.2.1 歷史介紹	4
2.2.2 遊戲是什麼	4
2.2.3 遊戲引擎是什麼	4
2.2.4 遊戲引擎具備的功能	5
3 系統說明	7
3.1 使用技術	7
3.1.1 使用語言	7
4 系統實作結果	9
4.1 引擎介紹	9
4.1.1 架構	9
4.1.2 Entity Component System	9
4.1.3 Batch Rendering	23
4.1.4 Particle System	27
4.1.5 2D Lighting	34
4.1.6 Scriptable Entity	40
4.1.7 Physics Engine	45
4.2 編輯器介紹	72
4.2.1 版面配置與操作介紹	72
4.2.2 架構與未來展望	78
5 結語	82
5.1 心得	82
5.1.1 鍾秉桓 Roy	82
5.1.2 梁博全 ICEJJ	83
5.1.3 黃育皓 SunTalk	83
5.1.4 黃品翰 Halloworld	84

List of Figures

4.1 組件圖	11
4.2 ECS 架構圖	12
4.3 不同 Rendering 方法效能比較	26
4.4 Particle System 模擬	34
4.5 光衰減	35
4.7 對物體畫出陰影	37
4.9 ScriptableEntity 介面圖	40
4.10 ScriptableEntity 架構圖	40
4.11 NativeScript 流程圖	43
4.12 加入 NativeScript 示意圖	43
4.13 NativeScript 介面圖	44
4.14 編輯器整體介面	72
4.15 主選單介面	73
4.16 Hierarchy	74
4.17 工具欄介面	75
4.18 Scene View	76
4.19 Gizmo 的不同模式	77
4.20 Game View	77
4.21 除錯訊息視窗 Log	78
4.22 訊息視窗 Status Bar	78
4.23 Editor 架構	80
4.24 引擎實體檢視介面	81

List of Tables

4.1 不同 Rendering 方法下 FPS 差別	25
---------------------------------------	----

Chapter 1

摘要

1.1 背景簡介

本專題是以 C++ 開發之 2D 遊戲引擎，開發者可以使用引擎提供之編輯器 (RishEditor) 編輯遊戲場景 (Scene)，並且可以對遊戲物件 (Entity) 附加遊戲邏輯 (使用 C++ 撰寫，並和編輯器一同編譯)，並支援 Batch Rendering¹、2D Lighting (Point Light, Ambient Light), Particle System, Constrain-based Physics (支援圓形、多邊形等)

1.2 問題說明

以往我們寫遊戲大部分都使用較為成熟的遊戲引擎，但往往都不能了解裡面的實作和內部的原理。透過本次專題，能夠讓開發者清楚了解到內部的實作，以及透過遊戲引擎來加速整個遊戲開發流程。

1.3 實作結果

成員們透過撰寫遊戲以及參考大型遊戲引擎設計，來發掘和打磨引擎內部功能，並學習各種程式設計架構，使可讀性增加便於維護。除此之外，組員們從中了解組件其中之原理並留下紀錄，供後人想了解遊戲引擎背後的原理參考。

¹一種 Rendering 技巧，可支援同屏幕高達 100000 個 sprites

Chapter 2

緒論

2.1 動機

隨著遊戲的發展，現代的遊戲越來越趨於複雜，從數人的小型獨立製作，到數百人的大型 3A 級遊戲，遊戲的規模與以前不可同日而語，現今一個獨立開發工作室製作的 2D 遊戲，在十多年前要達到同樣的規模可能要數十人的團隊才可能達到，而這些節省下來的時間成本，就是多虧了遊戲引擎的強大之處。現代遊戲引擎的複雜度，先進的圖學技術，複雜的物理模擬，是需要一個具有規模的團隊來開發的，我們嘗試以此為目標，試著實作了一個具有一定規模的 2D 遊戲引擎。

2.2 問題描述

2.2.1 歷史介紹

最初並沒有所謂的遊戲引擎，遊戲常常是重頭開始建構 (from scratch) 的，這個概念被眾人所知最早是由 Jhon Carmack 發揚光大，他為 Doom 以及 Quake 系列遊戲開發的 3D 遊戲引擎，深深地影響了遊戲業界，為早期 (1990 末) 的遊戲業界標準，並促成引擎授權的商業模式，使得遊戲軟體的規模上升。

2.2.2 遊戲是什麼

電腦遊戲的技術本質是實時 (real-time) 可交互 (interactive) 的程式，遊戲程式會模擬出不精確但足以表現的遊戲世界，並且根據玩家的輸入做出相對的輸出 (例如操作搖桿控制角色等)，通常遊戲會實作遊戲循環 (Game Loop)，更新遊戲邏輯、物理模擬、更新 AI 等。而通常遊戲要維持在每秒更新 60 幀才能保證流暢運行 (低於 60 fps 通常會感覺卡頓)，也就是要在 16 毫秒內做完所有的遊戲更新並渲染到螢幕上頭，更何況在 VR 上要維持 90 FPS 才不會感覺暈眩，這也是為什麼遊戲會如此要求效能。

2.2.3 遊戲引擎是什麼

遊戲引擎 (Game Engine) 從字面上解釋是驅動遊戲的基礎程式，因此遊戲引擎須具備了窗口管理、輸出入管理、渲染 (Rendering) 系統、物理 (Physics) 系統…等子系統，

除了基礎程式之外，要是一個遊戲引擎還必須要有讓遊戲開發者使用之工具（可視化、非可視化），可能是單個工具或一整個工具鏈（Toolchain），使得遊戲開發者可以用該工具開發遊戲（Developing）、進行測試（Debugging）、封裝發布（Shipping）等，否則就只能被稱為遊戲框架（Game Framework）。遊戲引擎會讀入自訂的資源（Asset）格式¹，並且有工具支援設計師將素材（貼圖、音效、模型等）轉換成引擎的格式，或是引擎工具可以直接產生，因此遊戲引擎可以說是專門開發遊戲的開發環境。

2.2.4 遊戲引擎具備的功能

TODO

¹常常是為了效能才會這樣做

I.6. Runtime Engine Architecture

29

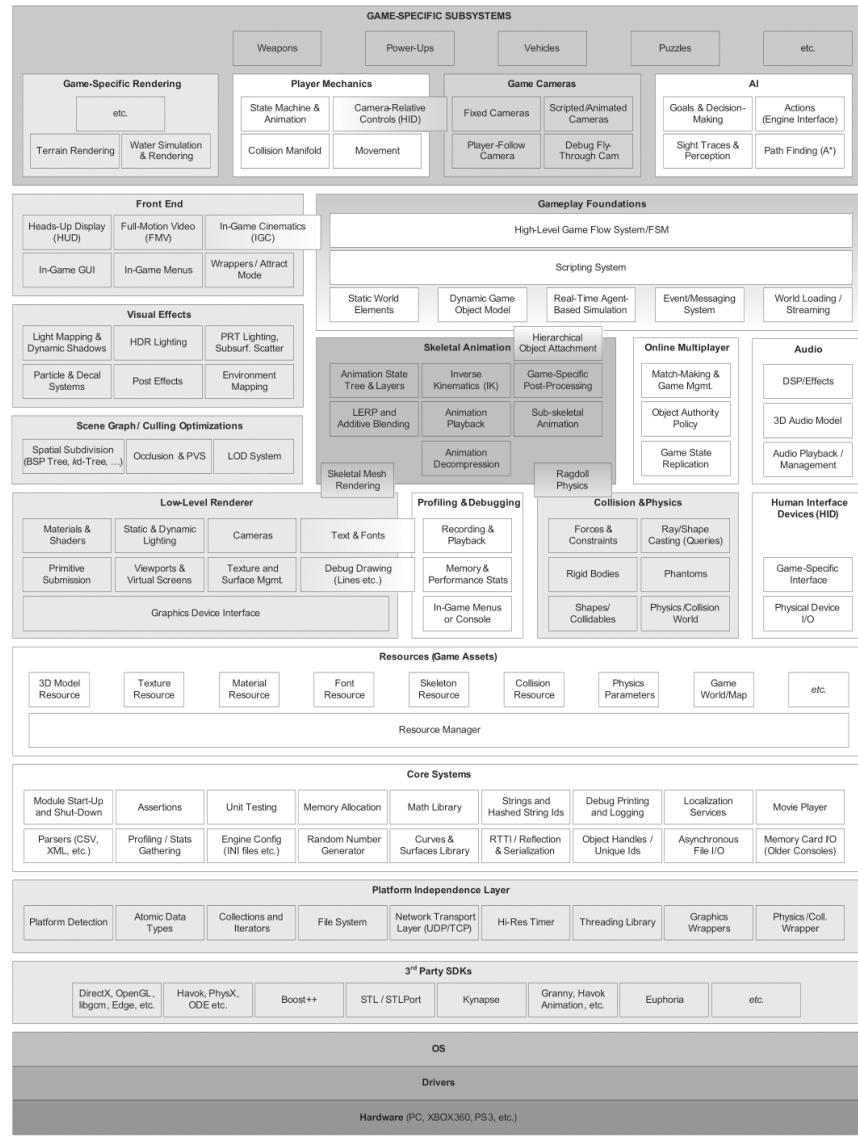


Figure I.11. Runtime game engine architecture.

Chapter 3

系統說明

3.1 使用技術

3.1.1 使用語言

使用 C++ 11 作為主要開發語言，使用 msys2 作為套件管理器¹，CMake 建置專案，使用 doxygen 工具將註解轉換 documentation, cppcheck C++ 的靜態程式碼分析器

- Slack

團隊協作軟體，用於溝通紀錄

- Trello

看板軟體，用於工作進度管理

- git, github

版本控制

為何選 C++ 做為開發引擎之語言？

因為 C++ 讓開發者可以自由掌控記憶體，讓開發者可以精確的控制變數的生命週期，沒有垃圾回收 (Garbage Collection)，這對於效能注重的遊戲引擎來說相當重要；比 C 來得高階但效能卻沒損失很多；C++ 歷史悠久且有許多精良的函式庫可以使用。[\[4\]](#)

使用函式庫

- [SFML](#)

一個 C++ 的跨平台用於遊戲、多媒體程式開發的函式庫，於此專案中用於 Input 及窗口操作等

- OpenGL + [glad](#)

一個跨平台的 API ，使用 glad 載入器

¹在 Windows 上，沒有套件管理器會非常痛苦

- [imgui](#)

一個輕量、快速的 Immediate Mode GUI 函式庫，常在遊戲開發、工具開發等使用，於此專案中用於編輯器的 UI 開發。

- [spdlog](#)

一個輕量、快速的 Logging 函式庫，於此專案中用於處理除錯訊息。

- [nativefiledialog](#)

小型的 Open File Dialog 函式庫，於此專案中用於處理開啟檔案視窗。

- [IconFontCppHeaders](#)

字體的 Helper Headers，於此專案中用於引入自訂字體。

- [fmt](#)

補足了 C++ 標準庫缺少的格式化輸出輸入。

- [glm](#)

OpenGL 數學函式庫，於此專案中用於處理向量、投影等相關數學函式。

- [cereal](#)

C++ 缺少的序列化函式庫，於此專案中用於序列化儲存資源。

- [re2](#)

來自 Google 的 Regular Expression 函式庫，標準庫的有夠慢 (確信)。

Chapter 4

系統實作結果

4.1 引擎介紹

4.1.1 架構

流程圖

架構圖

TODO

4.1.2 Entity Component System

ECS 是甚麼

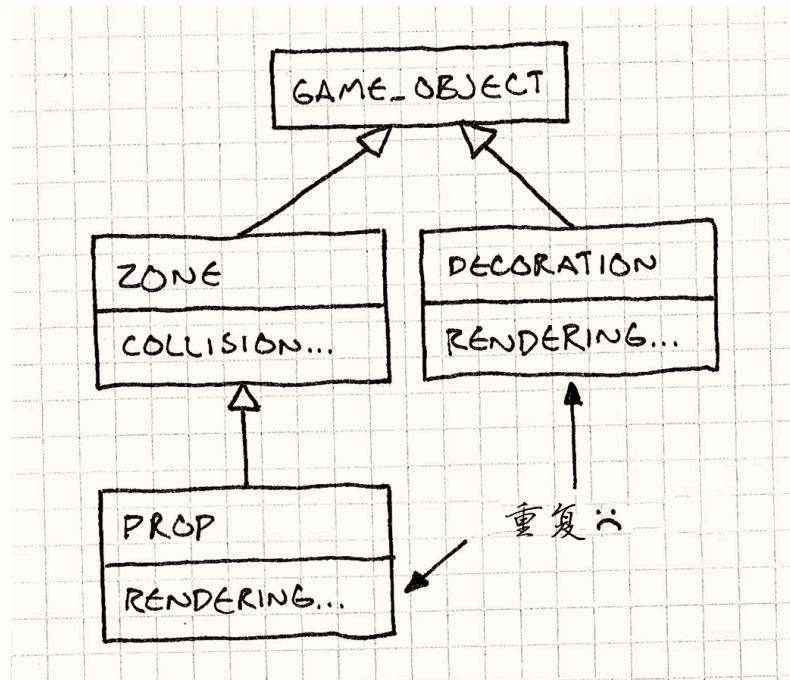
遊戲的本質其實就是大量物件 (Entity) 的行為以及它們之間的交互 (Manager)。但顯然遊戲裡的物件不會只有少少幾個。隨著物件邏輯的增加，我們會將相同邏輯的部分進行拆分，由繁化簡，而拆分的方法有很多種。

繼承

最直接的寫法就是繼承，當我們的實體需要哪些邏輯的部分，我們就繼承那個部分。例如遊戲中地圖上有許多物品，這些物品 (Item) 分成裝飾和物件 (Prop)，裝飾是地圖上沒有物理、可以看見的實體；物件是可以互動、可以看見的實體；而地圖會有區域 (Zone)，切割地圖的區域和作為觸發 (Trigger) 使用。所有的遊戲物件都繼承自 GameObject，裏頭包含了位置等每個遊戲物體都有的資訊，Zone 內部實作碰撞，Prop 繼承自 Zone 實作了渲染，Decoration 也實作了渲染。可以發現到 Prop 跟 Decoration 只差在一個有碰撞，另一個沒有。如果我們嘗試將 Prop 改成繼承自 Zone 與 Decoration 則會導致菱形繼承，而這不是我們想要的。因此使用繼承並不能很好得解決這個問題。

組件 Component Pattern

比起繼承，組件的靈活度更高。當我們需要那部分邏輯時，我們不再繼承他，而是讓物件擁有他。這樣不但沒有宏偉的繼承樹，要讓各個實體溝通也就方便許多。



繼承圖，取自 Game Programming Patterns

組件 (Component) 的優勢，讓我們只要將 Component 插入我們所要的對象，就能建構複雜且具有豐富行為的實體 (Entity)。由於單一個實體 (Entity) 跨越了多個領域 (物理、渲染等等)，為了保持領域之間的相互分離，將每部分的邏輯放進各自的組件 (Component)。這意味著實體 (Entity) 被簡化成組件的容器。簡單的 ‘Component Pattern’ 程式碼的遊戲流程會如下所示

```
class Entity {
public:
    void update() {
        physis->update();
        transform->update();
        graphic->update();
    }
private:
    Physis* physics;
    Transform* transform;
    Graphic* graphic;
};

/* In Game loop when it need update */
while(running) {
    for(int i = 0 ; i < MAX_ENTITES ; i++)
        entityList->update();
}
```

組件雖好，但這種方法對緩存不友好，CPU 在抓取資料時會一次抓取一組資料進行處理，稱為 cache line，每當 CPU 處理完當前任務，會先從 cache line 尋找下個任務，如果找到，就不再需要去記憶體裡面找下個任務。我們的遊戲存取了每個實體的指標，而實體又儲存了每個組件的指標。每當要更新實體，我們得去遍歷他，又因為存取的

是指標，因此會造成 cache miss，浪費寶貴的時間。每個實體又會更新每個組件，組件存的也是指標，又一次的 cache miss。一個遊戲注重的，除了遊戲本身好不好玩，另外一個就是效能了，Entity Component System 解決了這個問題。

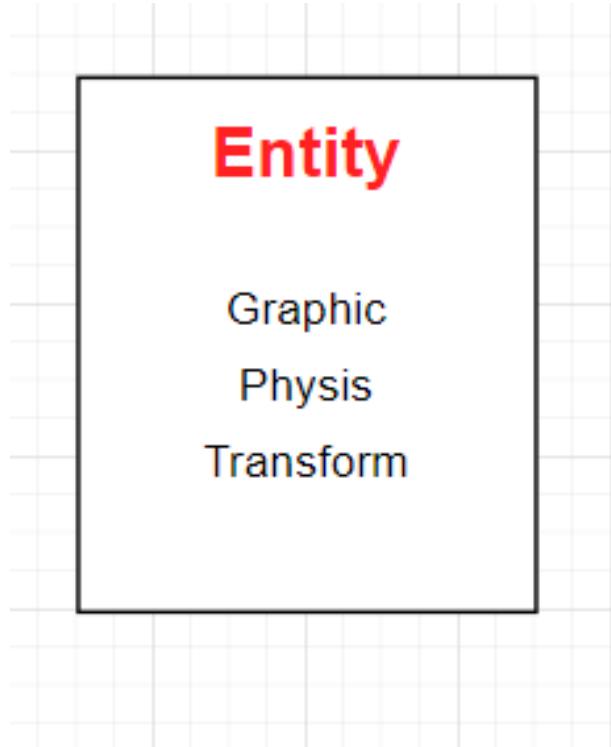


Figure 4.1: 組件圖

Entity Component System

Entity Component System，簡稱 ECS，這裡是指我們會有 3 個東西。Entity、Component 以及 System。

ECS 與 Component Pattern 不同的地方在於，ECS 的 Component 不再擁有任何的邏輯，Component 所擁有的僅僅是 Data，而邏輯的部分全部移駕到對應的 System。這樣的好處是，System 可以自己決定要操縱哪些 Data。

- 特徵
 - System 是唯一擁有邏輯的部分
 - Component 只擁有 Data
 - Entity 是多個 Component 的橋樑，用於表示這些 Component 是屬於這個 Entity 的。一般來說，Entity 僅僅只是一個 int
- 好處
 - 由於 Data 被拆散了，不容易出現讀入整個對象卻只使用其中一個屬性的情況，有利於 cache
 - 狀態和邏輯分離，適合同步邏輯

- 所有同樣的 Component，我們使用緊密的數組來儲存，因此我們能批量的連續進行處理，這在理想的情況能夠大量減少 cache miss
- 由於 Component 只剩下 Data，我們能夠很好的序列化

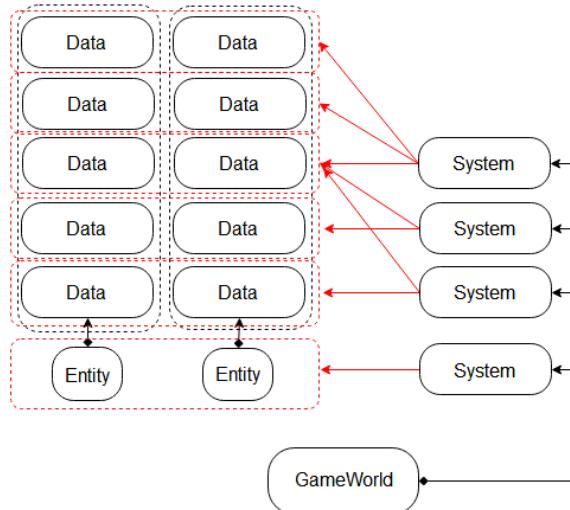


Figure 4.2: ECS 架構圖

實作

- ECS 將遊戲物件以及其行為拆成以下三樣東西
 - Entity
 - Component(Data)
 - System(Logic, behavior)

Entity Entity 僅是一個簡單的 ID，他不會真的存著 Component 或任何其他東西，事實上他只會用來作為 Component 陣列的索引而已

```
using Entity = std::uint32_t;
const Entity MAX_ENTITIES = 10000;
```

Component Component 只會存著需要的 Data，不會擁有任何的邏輯

```
// 角色的位置資訊
struct Transform {
    vec3 position;
    vec3 size;
    float rotate;
};

// 材質
struct Graphic {
    Texture texture_;
};
```

每種 Component 也會需要一個 ID。

```
using ComponentType = std::uint8_t;
const ComponentType MAX_COMPONENT = 32;
```

我們希望能夠追蹤這個 Entity 擁有哪些 Component，同時也必須追蹤這個 System 會有哪些 Component 參與其中。為了方便追蹤，我們會給 Entity 及 System 一個 Signature，上面登記著它擁有了哪些 Component。

`std::bitset` 很適合達到我們的要求，前面我們幫每種 Component 訂了 ID，意思是指要此物件或系統擁有這物件，就只要設定那個 ID 的 bit。

舉個例子，`Transform` 是 0，`Graphic` 是 1，`RigidBody` 是 2，擁有 `Transform` 和 `Rigid-Body` 的物件，它的 Signature 會被設定成 0b101(bit 0, 2 會被設定)。

Entity Manager Entity Manager 是負責分配及追蹤 Entity，記錄著哪些 Entity ID 已經被使用用最簡單的 `std::queue` 就可以達成，當產生一個 Entity，我們就返回一個 Entity ID，當 Entity 被銷毀，我們再將此 ID push 回 queue

```
class EntityManager {
public:
    Entity createEntity();
    void destroyEntity(Entity entity);
    void setSignature(Entity entity, Signature signature);
    Signature getSignature(Entity entity);
private:
    // Array of signatures where the index is the Entity
    std::array<Signature, MAX_ENTITIES> Signatures_;
    // ALL Unused Entities
    std::queue<Entity> Entites_;
    // Total living Entities
    uint32_t EntitiesCounts_ = 0;
};

EntityManager::EntityManager() {
    // Unused Entites
    for(Entity entity = 0 ; entity < MAX_ENTITIES ; entity++)
        Entites_.push(entity);
}

Entity EntityManager::createEntity() {
    assert(EntitiesCounts_ < MAX_ENTITIES && "Too Many Entities");
    Entity id = Entites_.front();
    Entites_.pop();
    ++EntitiesCounts_;

    return id;
}

void EntityManager::destroyEntity(Entity entity) {
    assert(entity < MAX_ENTITIES && "Entity Out Of Range");
    Signatures_[entity].reset();
    Entites_.push(entity);
    --EntitiesCounts_;
```

```

}

void EntityManager::setSignature(Entity entity, Signature signature) {
    assert(entity < MAX_ENTITIES && "Entity Out Of Range");
    Signatures_[entity] = signature;
}

Signature EntityManager::getSignature(Entity entity) {
    assert(entity < MAX_ENTITIES && "Entity Out Of Range");
    return Signatures_[entity];
}

```

Component Array 在這裡，我們必須實作一種簡單的陣列，但必須永遠是連續的，代表說它是不會有空洞出現的。因為我們的 Entity 只是 ID，要獲得跟 Entity 相關的 Component 是很簡單的，但是當 Entity 被銷毀時，對於陣列來說此 Index(Entity ID) 已經失效，我們不希望在遍歷陣列時出現失效的 Entity。

但是事實上當 Entity 被銷毀時，它仍然是存在在陣列裡的。在這裡，我們維護了兩個 map，存著 Entity ID 與 Index 之間的關係。當要使用 Entity 時，利用 Entity ID 去尋找在陣列裡實際的位置。當 Entity 被移除時，我們將陣列裡最後一個尚未失效的 Entity 移動到被移除的位置上，接著只要更新 map，一切就完成。在這裡做個簡單的演示。

1. 我們假設 MAX_ENTITES 是 5，陣列一開始是空的，map 也是空的

Array	0:	1:	2:	3:	4:	5:
Entity->Index						
Index->Entity						
Size	0					

2. 接著我們加入 Entity A，它的 Entity ID 為 0，在陣列中的位置是 0，所以將 map 的對應關係寫好

Array	0:	1:	2:	3:	4:	5:
Entity->Index	0:0					
Index->Entity	0:0					
Size	1					

3. 接著加入 B

Array	0:	1:	2:	3:	4:	5:
Entity->Index	0:0	1:1				
Index->Entity	0:0	1:1				
Size	2					

4. 加入 C

Array	0:	1:	2:	3:	4:	5:
Entity->Index	0:0	1:1	2:2			
Index->Entity	0:0	1:1	2:2			
Size	3					

5. 加入 D -> 到這裡陣列保持連續，接著我們要刪除 B，也就是 Entity 1，為了保持連續，我們將 D 覆蓋 B 的位置然後更新 map

Array	0:	1:	2:	3:	4:	5:
Entity->Index	0:0	1:1	2:2	3:3		
Index->Entity	0:0	1:1	2:2	3:3		
Size	4					

6. 刪除 B

Array	0:	1:	2:	3:	4:	5:
Entity->Index	0:0	3:1	2:2			
Index->Entity	0:0	1:3	2:2			
Size	3					

7. 接著刪除 Entity 3，也就是 D

Array	0:	1:	2:	3:	4:	5:
Entity->Index	0:0	2:1				
Index->Entity	0:0	1:2				
Size	2					

8. 然後我們加入 E，也就是 Entity 4

Array	0:	1:	2:	3:	4:	5:
Entity->Index	0:0	4:2				
Index->Entity	0:0	2:4				
Size	3					

```
class IComponentArray {
public:
    virtual ~IComponentArray() = default;
    virtual void entityDestroyed(Entity entity) = 0;
};

template<typename T>
class ComponentArray: public IComponentArray {
public:
    void insertData(Entity entity, T component) {
        assert(EntityToIndex.find(entity) == EntityToIndex.end() &&
               "Components added to the same Entity more than once");
        size_t newIndex = size_;
        EntityToIndex[entity] = newIndex;
        IndexToEntity[newIndex] = entity;
    }
};
```

```

        componentArray_[newIndex] = component;
        size_++;
    }

    void removeData(Entity entity) {
        assert(EntityToIndex.find(entity) != EntityToIndex.end() &&
               "Entity does not exist");

        size_t remove = EntityToIndex[entity];
        size_t lastElement = size_-1;
        componentArray_[remove] = componentArray_[lastElement];
        Entity lastEntity = IndexToEntity[lastElement];
        EntityToIndex[lastEntity] = remove;
        IndexToEntity[remove] = lastEntity;
        EntityToIndex.erase(entity);
        IndexToEntity.erase(lastElement);
        size_--;
    }

    T& getComponent(Entity entity) {
        assert(EntityToIndex.find(entity) != EntityToIndex.end() &&
               "retrieving none exist component");
        return componentArray_[EntityToIndex[entity]];
    }

    void entityDestroyed(Entity entity) override {
        if(EntityToIndex.find(entity) != EntityToIndex.end())
            removeData(entity);
    }
private:
    std::array<T, MAX_ENTITIES> componentArray_;
    std::unordered_map<Entity, size_t> EntityToIndex;
    std::unordered_map<size_t, Entity> IndexToEntity;
    size_t size_ = 0;
};

1

```

Component Manager Component Manager 是負責管理所有的 Component Array 的，並不是讓他們之間溝通，而是通知他們我註冊了哪些 Component，或是該刪除哪些 Component

```

class ComponentManager {
public:
    template<typename T>
    void registerComponent() {
        const char* typeName = typeid(T).name();
        assert(componentTypes_.find(typeName) == componentTypes_.end() &&
               "Registering component type more than once");
        componentTypes_.insert({typeName, nextComponentType_});
    }
};

```

¹這層抽象層是必要的，我們會有很多的 Component array，並且用一個 list 存著他們，每當有 Entity 被銷毀時，我們必須逐一地去通知他們有 Entity 被銷毀了。能讓我們存各個不同 type 的 Component 的辦法就是有一層抽象層了。

```

        componentArray_.insert({typeName,
            std::make_shared<ComponentArray<T>>()});
        ++nextComponentType_;
    }

template<typename T>
ComponentType getComponentType() {
    const char* typeName = typeid(T).name();
    assert(componentTypes_.find(typeName) != componentTypes_.end() &&
        "Component did not register");
    return componentTypes_[typeName];
}

template<typename T>
void addComponent(Entity entity, T component) {
    getComponentArray<T>()->insertData(entity, component);
}

template<typename T>
void removeComponent(Entity entity) {
    getComponentArray<T>()->removeData(entity);
}

template<typename T>
T& getComponent(Entity entity) {
    return getComponentArray<T>()->getComponent(entity);
}

void entityDestroyed(Entity entity) {
    for(auto const& pair: componentArray_) {
        auto const& component = pair.second;
        component->entityDestroyed(entity);
    }
}
}

private:
std::unordered_map<const char*, ComponentType> componentTypes_;
std::unordered_map<const char*, std::shared_ptr<IComponentArray>>
componentArray_;
ComponentType nextComponentType_ = 0;

template<typename T>
std::shared_ptr<ComponentArray<T>> getComponentArray() {
    const char* typeName = typeid(T).name();
    assert(componentTypes_.find(typeName) != componentTypes_.end() &&
        "Component did not register");

    return
        std::static_pointer_cast<ComponentArray<T>>(componentArray_[typeName]);
}
};

```

System

System 是行為存在的地方。每個 System 都放著存著 Entity 的 list

```
class System {
public:
    std::set<Entity> Entities_;
};
```

當我們需要實作系統就時繼承它

```
class PhysicSystem : public System {
public:
    void update(float dt);
};

class GraphicSystem : public System {
public:
    void update();
    void render(sf::RenderTarget& target);
};
```

而 System 的更新方法，會將此系統所在乎的 Component(Data)，取出，並進行更新。

```
void PhysicSystem::update(float dt) {
    for(auto const& entity : Entities_) {
        auto& transform = ecs.getComponent<Transform>(entity);
        auto& rigidBody = ecs.getComponent<RigidBody>(entity);

        transform.y_ += rigidBody.v_ * dt;

        rigidBody.v_ += 10 * dt;
    }
}
```

[2](#)

System Manager System Manager 是負責管理所有的 System 及他們的 Signature。每個 System 的 Signature 代表此 System 會用到的 Component，以便將合適的 Entity 分配給它。當 Entity 被銷毀，System 的 list 也得跟著更新。

```
class SystemManager {
public:
    template<typename T>
    std::shared_ptr<T> registerSystem() {
        const char* typeName = typeid(T).name();
        assert(system_.find(typeName) == system_.end() && "Registering
            system more than once");
        auto system = std::make_shared<T>();
        system_.insert({typeName, system});
        return system;
    }
};
```

²物理系統關心實體的 Transform 以及 RigidBody，因此沒有這兩個 Component 的實體，系統是不會去更新他們。

```

    }

    template<typename T>
    void setSignature(Signature signature) {
        const char* typeName = typeid(T).name();
        assert(system_.find(typeName) != system_.end() && "System did not
               register");
        signatures_.insert({typeName, signature});
    }

    void entityDestroyed(Entity entity) {
        for(auto const& pair : system_) {
            auto const& system = pair.second;
            system->Entities_.erase(entity);
        }
    }

    void entitySignatureChanged(Entity entity, Signature entitySignature) {
        for(auto const& pair: system_){
            auto const& type = pair.first;
            auto const& system = pair.second;
            auto const& systemSignature = signatures_[type];

            if((entitySignature & systemSignature) == systemSignature)
                system->Entities_.insert(entity);
            else system->Entities_.erase(entity);
        }
    }
private:
    std::unordered_map<const char*, Signature> signatures_;
    std::unordered_map<const char*, std::shared_ptr<System>> system_;
};


```

ECS 擁有管理 Entity 的 Entity Manager、管理 Component 的 Component Manager 以及管理 System 的 System Manager 之後，最後就是將三個 Manager 封裝起來。

```

class ECS {
public:
    ECS() = default;
    void init();
    Entity createEntity();
    void destroyEntity(Entity entity);

    template<typename T>
    void registerComponent() {
        componentManager_->registerComponent<T>();
    }

    template<typename T>
    void addComponent(Entity entity, T component) {
        componentManager_->addComponent<T>(entity, component);
        auto signature = entityManager_->getSignature(entity);
        signature.set(componentManager_->getComponentType<T>(), true);
        entityManager_->setSignature(entity, signature);
    }
};


```

```

        systemManager_->entitySignatureChanged(entity, signature);
    }

    template<typename T>
    void removeComponent(Entity entity){
        componentManager_->removeComponent<T>(entity);
        auto signature = entityManager_->getSignature(entity);
        signature.set(componentManager_->getComponentType<T>(), false);
        entityManager_->setSignature(entity, signature);
        systemManager_->entitySignatureChanged(entity, signature);
    }

    template<typename T>
    T& getComponent(Entity entity) {
        return componentManager_->getComponent<T>(entity);
    }

    template<typename T>
    ComponentType getComponentType() {
        return componentManager_->getComponentType<T>();
    }

    template<typename T>
    std::shared_ptr<T> registerSystem() {
        return systemManager_->registerSystem<T>();
    }

    template<typename T>
    void setSystemSignature(Signature signature) {
        systemManager_->setSignature<T>(signature);
    }
private:
    std::unique_ptr<EntityManager> entityManager_;
    std::unique_ptr<ComponentManager> componentManager_;
    std::unique_ptr<SystemManager> systemManager_;
};

void ECS::init() {
    entityManager_ = std::make_unique<EntityManager>();
    componentManager_ = std::make_unique<ComponentManager>();
    systemManager_ = std::make_unique<SystemManager>();
}

Entity ECS::createEntity() {
    return entityManager_->createEntity();
}

void ECS::destroyEntity(Entity entity) {
    entityManager_->destroyEntity(entity);
    componentManager_->entityDestroyed(entity);
    systemManager_->entityDestroyed(entity);
}

```

使用方法 假設擁有有三個 Component，儲存角色資訊的 Transform 、儲存物理資訊的 RigidBody 、以及儲存圖像的 Graphic 。

```

struct Transform {
public:
    float x_;
    float y_;
};

struct RigidBody {
public:
    float v_;
};

struct Graphic {
public:
    Texture texture;
};

```

實作 Graphic System，並且撰寫渲染圖像的邏輯。

```

class GraphicSystem : public System {
public:
    void render(RenderTarget& target);
};

void GraphicSystem::render() {
    for(auto const& entity : Entities_) {
        auto& graphic = ecs.getComponent<Graphic>(entity);
        auto& transform = ecs.getComponent<Transform>(entity);

        Renderer::Draw(transform.x, transform.y, graphic.texture);
    }
}

```

實作 PhysicSystem，處理物理模擬方面的邏輯。

```

class PhysicSystem : public System {
public:
    void update(float dt);
};

void PhysicSystem::update(float dt) {
    for(auto const& entity : Entities_) {
        auto& transform = ecs.getComponent<Transform>(entity);
        auto& rigidBody = ecs.getComponent<RigidBody>(entity);

        transform.y_ += rigidBody.v_ * dt;

        rigidBody.v_ += 10 * dt;
    }
}

```

初始化

```

ECS::ECS ecs;

ecs.init();

```

```
ecs.registerComponent<ECS::Transform>();
ecs.registerComponent<ECS::RigidBody>();
ecs.registerComponent<ECS::Graphic>();
```

註冊 System 以及設定 signature

```
auto graphicSystem = ecs.registerSystem<ECS::GraphicSystem>();
ECS::Signature signature;
signature.set(ecs.getComponentType<ECS::Graphic>());
signature.set(ecs.getComponentType<ECS::Transform>());
ecs.setSystemSignature<ECS::GraphicSystem>(signature);

auto physicSystem = ecs.registerSystem<ECS::PhysicSystem>();
signature.set(ecs.getComponentType<ECS::Transform>());
signature.set(ecs.getComponentType<ECS::RigidBody>());
ecs.setSystemSignature<ECS::PhysicSystem>(signature);
```

產生 Entity

```
std::vector<ECS::Entity> entites(ECS::MAX_ENTITIES);

for(auto& entity: entites) {
    entity = ecs.createEntity();
    ecs.addComponent(
        entity,
        ECS::Transform{randPositionX(generator), randPositionY(generator),
                      p});
    ecs.addComponent(
        entity,
        ECS::RigidBody{20});
    ecs.addComponent(
        entity,
        ECS::Graphic{texture});
}
```

4.1.3 Batch Rendering

研究背景

傳統上 OpenGL 要畫圖形到螢幕上時，會建立 VAO(Vertex Array Object) 用來儲存 VBO (Vertex Buffer Object) 和 IBO (Index Buffer Object)，其中 VBO 儲存圖形的點座標、IBO 儲存 index 座標，最後用 glDrawElements() 讓 OpenGL 將圖形經過 Rendering Pipeline 畫在 Target 上 (通常是螢幕或是 Framebuffer)。

```
InitializeRenderer();

for( /* 場景的所有物體 */ )
{
    Render();
}
```

但是遊戲引擎在畫場景中的每個物體時，如果每個物體都要重新建構 VBO 和 IBO 時，則會耗費大量時間在從 CPU 傳輸資料到 GPU 中，同時也必須減少 OpenGL 的 State 變換，因此將渲染前先將多種圖形的頂點收集起來，再一次進行繪製，減少 Draw Call 進而增加可以繪製的圖形數量。

```
InitializeRenderer();
size_t vertex_count = 0;

while( /* 場景還有物體尚未渲染 */ )
{
    CollectVertices();
    vertex_count += size;
    // 超過單次 Draw Call 的上限，先渲染
    if(vertex_count > MAX_VERTEX_COUNT_PER_DRAWCALL)
    {
        Render();
        ResetRenderer();
        vertex_count = 0;
    }
}
```

實作

以畫矩形作為例子，首先要有辦法操作頂點，因此要有頂點的 class

```
struct QuadVertex
{
    vec3 position;
    vec4 color;
    vec2 texCoord;
    float texIndex;
};
```

四個頂點構成一個矩形，所以接著宣告矩形的 class，重載小於運算子是為了要能排序，這在要開啟深度測試時很重要。³

³由於 Depth Test 是為了去除掉覆蓋的問題，也就是 z 比較大的物體會蓋過 z 比較小的物體，因此

```

struct QuadShape
{
    QuadVertex p[4];

    friend bool operator<(const QuadShape &lhs, const QuadShape &rhs)
    {
        return lhs.p[0].position.z < rhs.p[0].position.z;
    }
};

```

接著可以開始著手進行 Renderer 的撰寫:

```

struct QuadRenderer
{
    void init()
    {
        vao = CreateVertexArray();

        VBO *vbo = CreateVertexBuffer(MaxQuadVertexCount *
            sizeof(QuadVertex));
        vbo->SetVertexBufferLayout();
        vao->SetVertexBuffer(vbo);

        IBO *ibo = CreateIndexBuffer(MaxQuadIndexCount);
        ibo->SetIndexBuffer( /* 根據 primitive type 的不同 */ );
        vao->SetIndexBuffer(ibo);

        // 載入 Shader
        shader = LoadShader();
        shader->initTextureSlots();
    }

    void submit(const vec4 position[4], const vec4 &color, const vec2
        texCoords[4], float texIndex)
    {
        QuadShape submitQuad{};
        // Add vertices of a quad to buffer
        for(int i = 0; i < 4; i++)
        {
            submitQuad.p[i].position = position[i];
            submitQuad.p[i].color = color;
            submitQuad.p[i].texCoord = texCoords[i];
            submitQuad.p[i].texIndex = texIndex;
        }
        //
        quadIndexCount += 6;
        //
        quadShapeList.push_back(submitQuad);
    }

    void draw(bool DepthTest, const mat4 &ViewProjMatrix)
    {
        if(DepthTest)

```

在渲染時便不用渲染 z 比較小的物體，但在開啟 Blend Mode 時，必須照 z 軸由小畫到大，透明的物體才會是正常的。

```

        sort(quadShapeList.begin(), quadShapeList.end());

        // 設定 頂點資料
        vao->setVertexBufferData(quadShapeList);

        // 設定 Shader uniform
        shader->setUniformMat4("u_ViewProjection", ViewProjMatrix);

        // 渲染
        DrawElement(vao);
    }

VAO *vao = nullptr;
Shader *shader = nullptr;
std::vector<QuadShape> quadShapeList;
uint32_t quadIndexCount = 0;
};


```

init() 負責初始化 Renderer 的狀態以及準備 VBO, IBO 和 Shader 的初始化，Renderer 初始化結束後，便可以開始提交頂點到 Renderer 上，submit() 可以提交要畫的頂點，接著 draw() 在每個 Batch 的最後將頂點渲染，結束這一次的渲染。

結論

加入 Batch Rendering 後，RishEngine 的 Renderer 獲得了顯著的提升：

Number of Sprites	Modes					
	Debug Non-batch	Release Non-batch	Debug Batch	Release Batch		
100	1000	1055	1900	2789		
1000	167	158	568	1379		
10000	18	17	82	247		
100000	< 1	< 1	9	28		

Table 4.1: 不同 Rendering 方法下 FPS 差別

可以發現到：加入 Batch Rendering 後，Renderer 的效率獲得了巨大的提升，在小規模數量的 Sprites 時，FPS 的增加相當可觀；但數量一大起來時，增加的幅度就變小了，由此可知在數量大時渲染瓶頸在其他地方。

未來展望



- 將 Renderer 變成 Multi-thread 架構

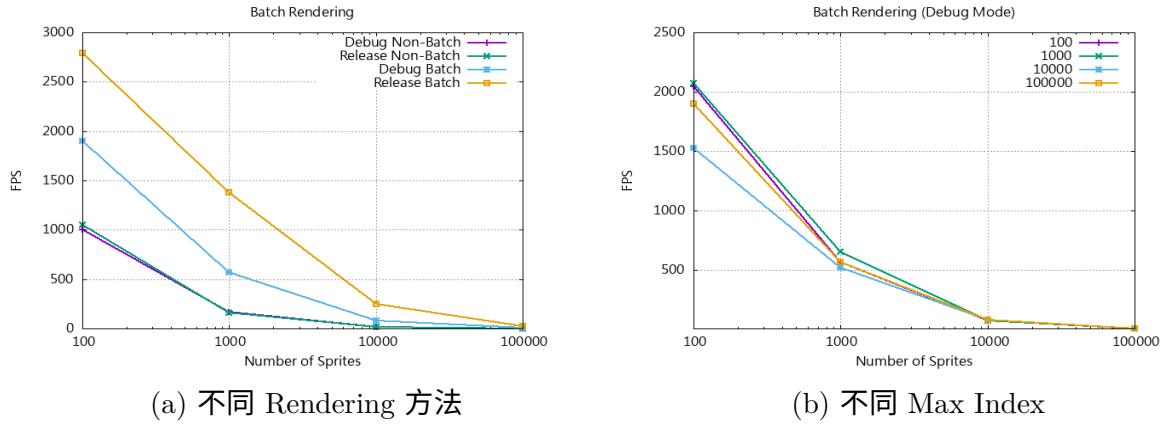


Figure 4.3: 不同 Rendering 方法效能比較

- 由於 OpenGL 在設計當初並沒有考慮到多線程，因此如果要在 OpenGL 使用多線程架構則要用 C++ 模擬，相較於其他現代的 API，就沒有此限制 (Vulkan, DirectX)。
- 將程式分成 Main Thread 與 Rendering Thread
 - Main Thread 主要負責遊戲循環，如果要畫東西時，會將 Render 資訊打包成一個 Task，接著加到共用的 Rendering Queue 中 (Message Queue)。
 - Rendering Thread 負責消化 Rendering Queue 中的 Task，向 GPU 發起 Draw Call。
 - 在原本單線程的 Renderer 中，主要的瓶頸在 Submit Draw Call，繪製時必須等待繪製好了 (Blocking)，改成多線程則可以解決這個問題。

4.1.4 Particle System

研究背景

一般 2D 遊戲是利用 sprite 將一些靜態元素表現在螢幕上，若要做出角色移動這類的動畫效果，就會將很多 Sprite 組成 Sprite Sheet，接著去變換角色的 Sprite 就能達到效果，像是這樣：



若需要一些特效，火焰、煙霧、煙火…等等，每個 Sprite 都需要請人畫，當東西複雜起來時這不會是個很好的方法。因此我們需要一個能夠讓我們模擬各種不同效果的系統，而且只需要調參數就能讓我們快速的模擬各種不同的效果。

實作

這裡實作都以虛擬碼呈現

Particle

Particle 是粒子效果的基本單位，我們必須調整每個 particle 的位置、顏色、大小等等屬性以達到我們所要的效果

```
struct Particle
{
    glm::vec2 pos;
    glm::vec2 startVel, endVel;
    glm::vec2 currentVel;
    float currentSize, startSize, endSize;
    float angle;
    float startRotSpeed;
    float currentRotSpeed;
    glm::vec4 startColor;
    glm::vec4 currentColor;
    glm::vec4 endColor;
    float t; // Timestep
    uint32_t startLife;
    uint32_t life;
};
```

- pos
 - particle 目前的位置
- startVel
 - particle 產生時的速度
- currentVel
 - particle 目前的速度
- endVel
 - particle 死亡時的速度
- startSize
 - particle 產生時的大小
- currentSize
 - particle 目前的大小
- endSize
 - particle 死亡時的大小
- angle
 - Particle Rotate 的角度
- startRotSpeed
 - Particle 起始旋轉速度
- currentRotSpeed
 - Particle 目前旋轉速度
- startColor
 - Particle 產生時的顏色
- currentColor
 - Particle 目前的顏色
- endColor
 - Particle 死亡的顏色
- t
 - 插值所用的 timestep
- startLife
 - Particle 的生命週期
- life

- 目前生下的生命

我們能夠設定每個 Particle 的初始、死亡的速度、大小、顏色等等，以達到 Particle 由快轉慢、由大轉小等效果、顏色由淺轉深等效果。

Emitter

Emitter 是定義 Particle 行為的地方，意思是說，如果我希望我的效果會讓粒子從紫色變黃色，我就要告訴 Emitter 我希望我的起始顏色是紫色，結束顏色是黃色。接著當遊戲循環開始時，Emitter 就會根據數據，將他給予到每個 Particle，讓他們模擬。

有了 Emitter，當我們需要產生不同的效果，只需產生多個 Emitter，給予不同的參數，就能有多個效果。Emitter 擁有許多與 Particle 類似的數據。

```
struct ParticleComponent
{
    glm::vec2 angleRange = {0.f, 360.f};
    float startSpeed = 0.f;
    float endSpeed = 0.f;
    float startSize = 0.f;
    float endSize = 0.f;
    float rotateSpeed = 0.f;
    int emissionRate = 0;
    uint32_t emitNumber = 0;
    uint32_t emitVariance = 0;
    uint32_t maxParticleLife = 0;
    uint32_t maxParticlesPerFrame = 0;
    int poolSize;
    bool active = false;
    float life = -1;
    float sleepTime;
    Timer lifeTimer;
    Timer sleepTimer;
    glm::vec4 startColor = {0, 0, 0, 0};
    glm::vec4 endColor = {0, 0, 0, 0};
    glm::vec2 rotSpeedRand = {0.f, 0.f};
    glm::vec2 startSpeedRand = {0.f, 0.f};
    glm::vec2 endSpeedRand = {0.f, 0.f};
    glm::vec2 emitVarianceRand = {0.f, 0.f};
    glm::vec2 startSizeRand = {0.f, 0.f};
    glm::vec2 endSizeRand = {0.f, 0.f};
    glm::vec2 lifeRand = {0.f, 0.f};
    float disX = 0.f;
    float disY = 0.f;
    int lastUnusedParticle = 0;
    std::shared_ptr<Texture2D> texture;
    std::vector<Particle> particles;
};
```

- angleRange
 - Particle 會在哪個角度反為裡排放
- emissionRate

- 每個 frame 產生 Particle 的數量，由 emitNumber 跟 emitVariance 算出
- emitNumber
 - 每個 frame 保底產生 Particle 的數量
- emitVariance
 - 讓每個 frame 產生的數量不固定。此數會乘上一個介於 (0, 1) 範圍的值再加上 emitNumber 來讓每個 frame 產生的數量不固定。
- maxParticleLife
 - particle 的生命週期
- maxParticlesPerFrame
 - 用來計算 pool size
- poolSize
 - particle pool 的大小
- active
 - 這個 Emitter 是否活著
- life
 - Emitter 的生命週期。-1 代表永久活著
- sleepTime
 - 每 sleepTime 產生一次 Particle
- XXXRand
 - 代表 XX 的 random 範圍，讓 Particle 看起來不會那麼整齊
- disX/Y
 - disX, disY 用來決定 paritcle 產生的位置範圍
- vector<Particle>
 - particle pool，這個 Emitter 所模擬的 particle

EmitData

我們會將所有的數據放在外部，這樣更改數據之後並不需要重新編譯，也能重新選取檔案來快速切換效果。

```
{
  "value0": {
    "angleRange": {
      "x": 250.0,
      "y": 280.0
    },
    "startSpeed": 200.0,
    "endSpeed": 300.0
  }
}
```

```

        "endSpeed": 200.0,
        "startSize": 0.0,
        "endSize": 80.0,
        "rotateSpeed": 0.0,
        "emitNumber": 3,
        ...
    }
}

```

Particle System

Particle System 會模擬 Particle 的物理，以及幫助繪製粒子。

```

class ParticleSystem
{
public:
    void update(Time dt, Emitter& emitter);
    void render(Emitter& emitter);
    void init(EmitData& data, Emitter& emitter);
}

```

選取想要的效果檔案後，能夠將讀取的資料傳給 Particle System，並將資料輸入給 Emitter

```

void ParticleSystem::initEmitter(EmitData& data, Emitter& emitter) {
    emitter.angleRange = data.angleRange;
    emitter.startSpeed = data.startSpeed;
    emitter.endSpeed = data.endSpeed;
    emitter.startSize = data.startSize;
    emitter.endSize = data.endSize;
    emitter.rotateSpeed = data.rotateSpeed;
    /* ... */
}

```

Emitter 擁有資料之後，Particle System 就能夠開始模擬例子效果了。

```

void ParticleSystem::update(Time dt, Emitter& emitter) {
    if (emitter.active) {
        emitter.emissionRate = (int)(emitter.emitNumber +
            emitter.emitVariance *
            randFloat(emitter.emitVarianceRand.x,
                      emitter.emitVarianceRand.y));
        for(int i = 0 ; i < emitter.emissionRate ; i++) {
            int unusedParticle =
                firstUnusedParticle(emitter.lastUnusedParticle,
                                     emitter.particlePool);
            respawnParticle(emitter.particlePool, unusedParticle);
        }
    }
}

```

更新的時候，首先確認此 Emitter 是否存活 (emitter.active) 如果存活，計算出這一個 frame 要產生多少個 particle(emitter.emissionRate)，然後激活 emissionRate 數量的粒子。

激活粒子方法呢，會在我們的 Particle pool 中，找到未被使用的粒子並將它激活。

```
uint32_t ParticleSystem::firstUnusedParticle(uint32_t &lastUnusedParticle,
    vector<Particle>& particlePool) {
    for(int i = lastUnusedParticle ; i < particlePool.size() ; i++) {
        if(particlePool[i].life <= 0) {
            lastUnusedParticle = i;
            return i;
        }
    }

    for(int i = 0 ; i < lastUnusedParticle ; i++) {
        if(particlePool[i].life <= 0) {
            lastUnusedParticle = i;
            return i;
        }
    }
    lastUnusedParticle = 0;
    return lastUnusedParticle;
}
```

激活就是將 Emitter 的數據給予每個 Partcle 當每個 Particle 都擁有了數據以及生命之後，就能來更新他的數據了。

```
/* .... */

for(int i = 0 ; i < emitter.poolSize ; i++) {
    auto &particle = emitter.particles[i];
    if(particle.life > 0.f) {
        particle.currentSize = interpolateBetweenRange(particle.startSize,
            particle.t, particle.endSize);
        particle.currentVel.x =
            interpolateBetweenRange(particle.startVel.x, particle.t,
            particle.endVel.x);
        particle.currentVel.y =
            interpolateBetweenRange(particle.startVel.y, particle.t,
            particle.endVel.y);
        particle.currentColor = RGBAinterpolation(particle.startColor,
            particle.t, particle.endColor);

        particle.pos.x += particle.currentVel.x * dt;
        particle.pos.y += particle.currentVel.y * dt;
        particle.life--;

        particle.t += (1.0f/(float)particle.startLife);
        if(particle.t >= 1.f)
            particle.t = 0.f;

        particle.currentRotSpeed += particle.startRotSpeed;
        particle.angle += dt * particle.currentRotSpeed;
        particle.angle = fmod(particle.angle, 360.f);
    }
}
```

如果那顆粒子是活的，我們就更新它。利用插值法更新速度、大小以及顏色。然後根

據速度以及這一個 frame 經過的時間更新 Particle 的位置

Vortex

Vortex，渦流，類似一種像漩渦般不斷旋轉的空氣，而我們的 Particle 會被這個旋轉的空氣影響，進而影響行走的路徑。可以想像有個隱形的颱風在螢幕上，而每個直走的 Particle 會因為經過颱風而往右邊或左邊走。

Particle 會受 Vortex 的位置 (pos)、旋轉速度 (speed) 以及大小 (size) 影響。

```
struct Vortex {
    glm::vec2 pos = {0.f, 0.f};
    float speed = 0.f;
    float size = 0.f;
}
```

擁有這些數據後，我們更改 Particle System 裡 Particle 的更新方法。

```
float dx = particle.pos.x - vortex.pos.x;
float dy = particle.pos.y - vortex.pos.y;
float vx = -dy * vortex.turbulence.x;
float vy = dx * vortex.turbulence.y;
factor_ = 1.0f / (1.0f + (dx*dx + dy*dy)/vortex.currentSize*0.1);

particle.pos.x += (vx - particle.currentVel.x) * factor_ +
    particle.currentVel.x * dt;
particle.pos.y += (vy - particle.currentVel.y) * factor_ +
    particle.currentVel.y * dt;
```

這種 Vortex 是固定不動的，如果希望 Vortex 也會跟著移動的話，只需要將 Vortex 當作另一種 Particle 一起排放。

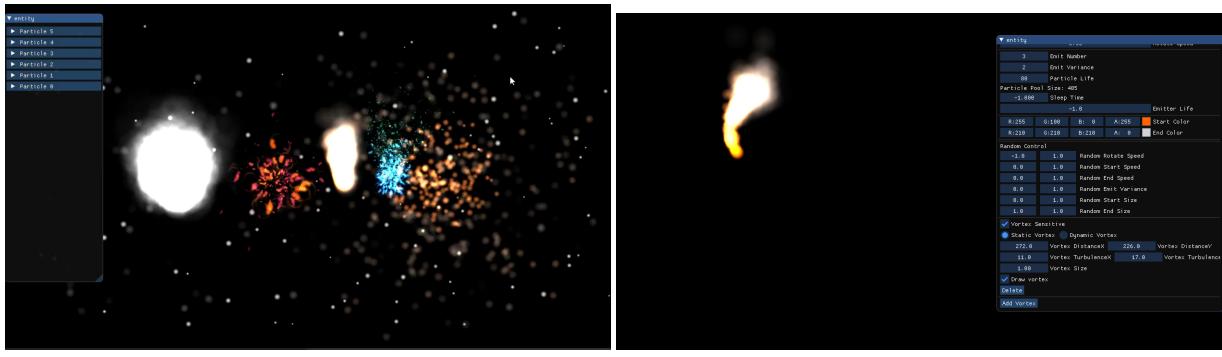
動態的 Vortex 影響 Particle 的方法：

```
factor_ = 1.0f / (1.0f + (dx*dx + dy*dy) / (vortex.currentSize * 0.1));
float lifeFactor = vortex.life / emitter.vortexMaxParticleLife;
factor_ *= (1-lifeFactor) * lifeFactor * 4;
```

Vortex 隨著時間移動，也會隨著時間消失，而在 Vortex 快死亡時對 Particle 的影響逐漸減少。

結論

目前這個 Particle System 能夠做出像是煙霧、煙火、火焰、雪等效果，各種其他特效也能透過條參數來獲得效果，且還能控制排放的間隔，每多少秒才排放一次。未來希望能夠新增更多的 particle 排放的曲線，不只能夠調整參數，能夠有個函數圖形讓使用者們直接選取中意的曲線，來模擬 particle 的路徑。



(a) Particle

(b) Vortex

Figure 4.4: Particle System 模擬

4.1.5 2D Lighting

研究背景

市面上不管 3D 或 2D 遊戲都會有光以及 RayCasting 的效果，而引擎應該要提供類似的效果。在一款 2D 遊戲裡我們希望能有的效果是，有燈光的地方以及玩家在的地方是有視野的，其他地方是黑的，又或者是其他地方比較暗，玩家及其他地方有光的地方比較亮。

效果

而光一般會有以下三種效果

- 光衰減
- 照亮世界
- Ray-Casting

實作

光衰減

我們是在 GPU 計算光的。光在光源的位置是最亮的，之後距離光源越遠光的強度會越來越弱。因此我們會需要知道光源的位置，接著計算每個 pixel 與光源之間的距離，利用這個距離計算出光的衰減。

```
#version 450 core
out vec4 FragColor;

in vec2 v_LightPosition;
in vec4 v_Color;
in float v_Constant;
in float v_Linear;
in float v_Quadratic;

void main()
{
```

```

// 獲取每個pixel
vec2 pixel = vec2(gl_FragCoord.x, gl_FragCoord.y);
// 計算每個pixel到光源的距離
float distance = length(v_LightPosition - pixel.xy);
// 如過大於光半徑則丟棄
if(distance > v_Radius) discard;
// 根據強度以及距離計算強度
float attenuation = 1 / (1 + distance * 1/v_Strength);
// 將光衰減乘上光顏色
vec4 color = vec4(attenuation, attenuation, attenuation,
    pow(attenuation, 3)) * v_Color;

FragColor = color;
}

```



Figure 4.5: 光衰減

獲取當前窗口所有座標，接著將傳入的 `lightPosition` 與每個 `pixel` 與光源計算距離，若距離大於所設定之影響半徑則丟棄，之後帶入光衰減函數去計算。

$$F_{att} = \frac{1.0}{K + D * \frac{1.0}{S}} \quad (4.1)$$

此光衰減函數定義了三個項，分別是：

- 常數項 K (Constant)
 - 通常保持 1.0，目的是保證分母永遠不會比 1 來的小，否則距離越遠強度反而會越強
- 距離 D (Linear)
 - 此距離為每個 `pixel` 與光源的距離
- 強度 S (Quadratic)
 - 影響光的強度

計算好當前 `pixel` 的強度之後，將他乘上光的顏色，就能夠得到當前 `pixel` 的顏色了

照亮世界

在 2D 的世界裡，要將兩張半透明的 Texture 混在一起最簡單的作法是將兩張 Texture 相乘。利用這個想法，我們需要將物體與光分別畫在不同的 Framebuffer 上，之後將這兩個 Framebuffer 合併，這樣就能得到光照在物體上並變亮的效果出來。

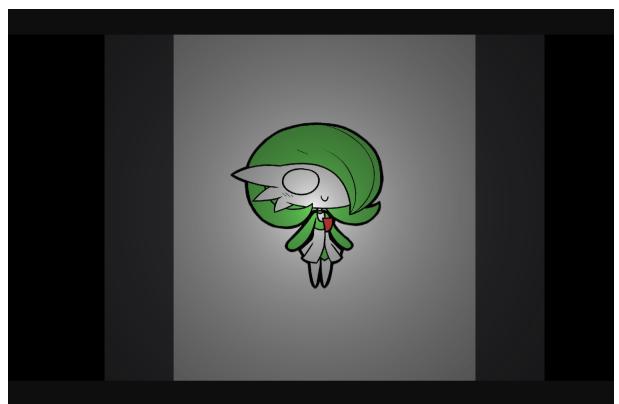
```
{
    worldFBO->bind();
    Renderer::DrawStuff();
    worldFBO->unbind();
    lightFBO->bind();
    Renderer::DrawLight();
    lightFBO->unbind();
    combineFBO(lightFBO, worldFBO);
}
```

將物體與光分別畫在不同的 Framebuffer 裡，接著再將兩張 Texture 用 Shader 將它相乘。

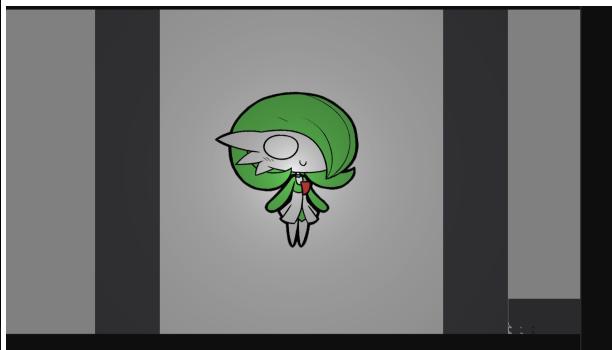
```
#version 430 core
in vec2 v_TexCoord;

out vec4 color;
uniform sampler2D u_Textures;
uniform sampler2D u_Textures2;

void main()
{
    vec4 worldColor = texture(u_Textures, v_TexCoord);
    vec4 lightColor = texture(u_Textures2, v_TexCoord);
    // 分別將兩張Texture乘起來
    color = worldColor * lightColor;
}
```



(a) 照亮世界



(b) Ambient

在光的那張 Texture，光影響不到的地方的 rgb 是 $(0, 0, 0)$ ，而 $(0, 0, 0)$ 不管乘上任何的顏色都會是 $(0, 0, 0)$ ，而有光的地方 rgb 不會是 0，因此乘上世界的物體顏色就能將物體照亮物體，其餘地方將會是黑色

這樣只有有光的地方才能看到物體，沒有光的地方就是一片黑，若希望沒有光的地方不是全暗，仍能看到一些暗暗的背景的話，只需要在光的 Framebuffer 的背景上，畫上一張有顏色的矩形。如此一來，在光的 framebuffer 上我們會先有一個有顏色的背景，接著在這背景上畫上我們的光源，最後將這個與世界物體相乘，這樣沒有被光影響到的地方會乘上背景的顏色，就不會是全黑了。

RayCasting

光線照射到物體上，物體背後會產生陰影。

我們有光源位置以及兩個點 (P_1, P_2)，我們就能夠畫出陰影。點 P_1, P_2 所連成的直線背後是不可視的，因此要在後面畫上陰影 (黑色)。我們取得 P_1 以及 P_2 的座標，利用向量計算光源到點 p_1 的延伸點 P_3 ，以及光源到點 P_2 的延伸點 P_4 ，用這四個點所圍成的四邊形即是陰影的部分。

運用這個道理，我們就能夠將世界物體的陰影給畫出來。在我們的引擎裡面，物體都是四邊形的。我們能夠輕易地找到座標畫出陰影。

若光源與 $P_3 P_4$ 以及 $P_2 P_3$ 畫陰影，會發生陰影蓋住物體的情況發生，因此需要找到距離光源較遠的邊。我們可以利用光到點的向量以及每條邊的法向量來幫助我們。

利用四個點我們能得到 $v_1 v_2 v_3 v_4$ 四條向量，而我們分別找出這四條向量的法向量 $n_1 n_2 n_3 n_4$ ，利用這四條法向量，與光源到點的向量座內積，如果大於零，方向基本相同，夾角在 0° 到 90° 之間則取兩點畫上陰影。

光與點 P_1 所形成的向量與法向量 n_1 做內積，小於 0，則利用光源與點 p_1, p_2 做陰影。接著利用光與點 p_2 所形成的向量語法向量 n_2 做內積，以此類推。

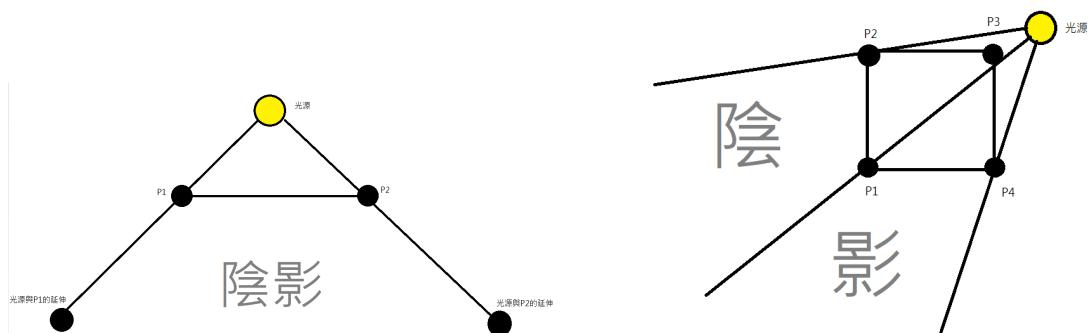


Figure 4.7: 對物體畫出陰影

```
for(auto light : allLight)
{
    for(auto worldEntity : allWorldEntity)
    {
        // 獲取物體的點
        vector<glm::vec2> vertices = worldEntity->getVertices();

        for(int i = 0 ; i < vertices.size() ; i++)
        {
            // 目前的點
            ...
        }
    }
}
```

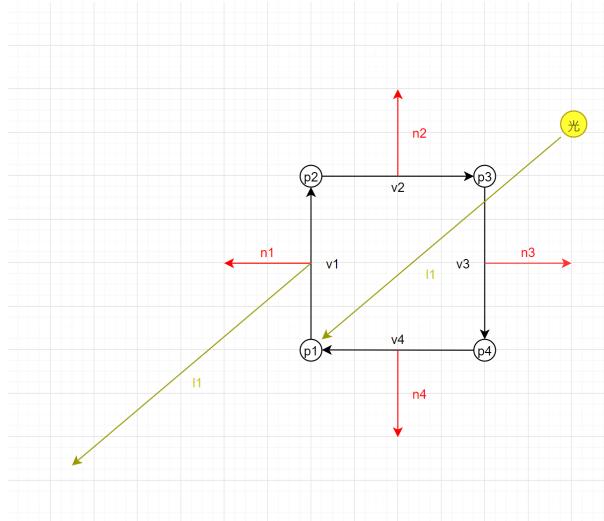
```

glm::vec2 currentVertex = vertices[i];
// 下一個點，與目前的點連成一條線
glm::vec2 nextVertex      = vertices[(i+1)%vertices.size()];
// 計算兩點之向量
glm::vec2 edge             = nextVertex - currentVertex;
// 兩點向量之法向量
glm::vec2 normal           = glm::vec2(-edge.y, edge.x);
// 光源至目前點的向量
glm::vec2 lightToCurrent   = currentVertex - light->pos;

// 計算內積判斷是否畫陰影
if(glm::dot(normal, lightToCurrent) > 0)
{
    Renderer::DrawShadow();
}
}

// Draw Light
}

```



(a) 判斷哪個邊要畫陰影



(b) Ray-Casting

結論

擁有光之後，就能夠讓遊戲畫面更加的豐富，我們能夠調動世界的光暗，讓世界是暗的，只有擁有光源的地方能夠產生亮光。

未來展望

- 提供更多的光
 - 目前只提供點光源
- 進入 3D

- 將光照系統新增提供 3D 環境的光照

4.1.6 Scriptable Entity

前言

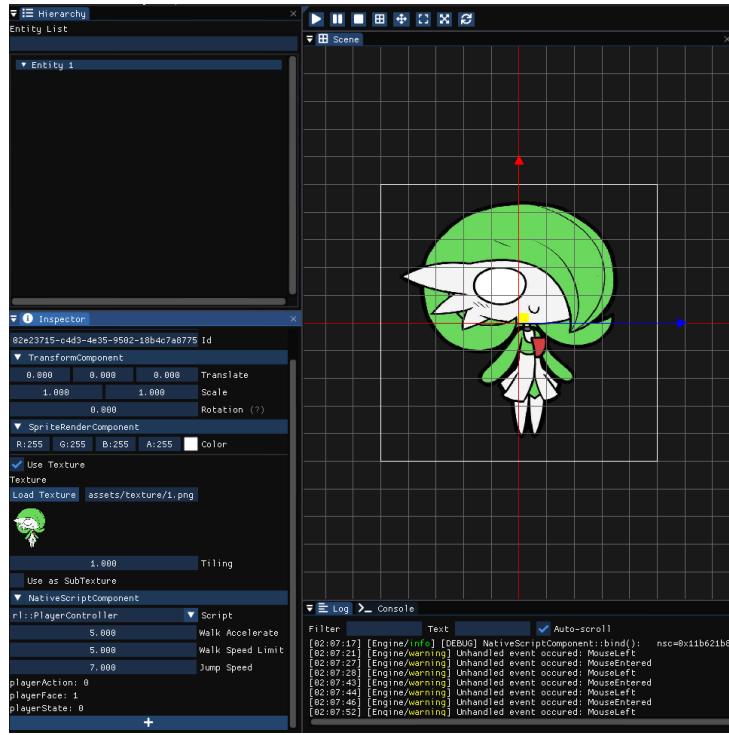


Figure 4.9: ScriptableEntity 介面圖

RishEngine 支援開發者使用引擎提供之 API 撰寫遊戲邏輯，並使用編輯器將寫好的邏輯綁定在多個遊戲物件上，構成可以互動的遊戲場景 (Scene)。

實作

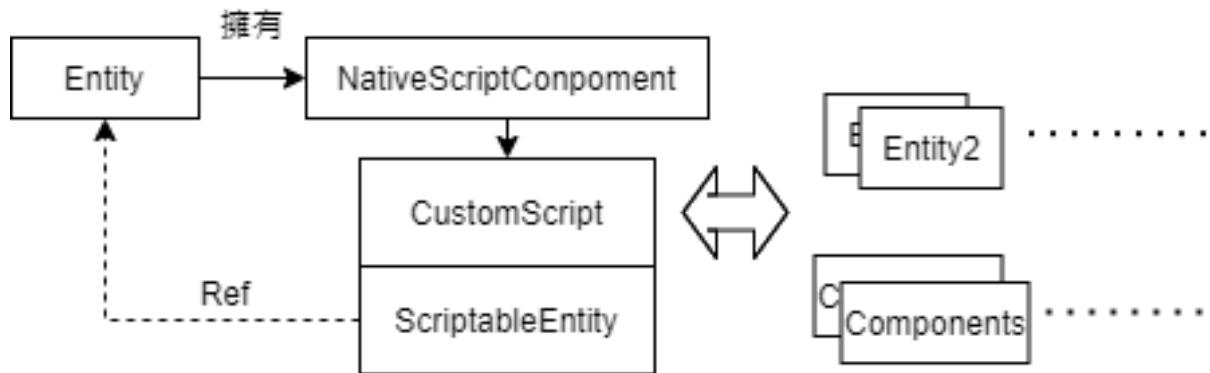


Figure 4.10: ScriptableEntity 架構圖

原本引擎在 ECS 中的 Entity 是代表遊戲物件，而 ScriptableEntity 代表被 Script 所操控的 Entity。而 NativeScriptComponent 擁有 ScriptableEntity 類，接著 Script 得以操作自身或是其他的 Entity 的 Component。

```

class ScriptableEntity
{
public:
    ScriptableEntity() = default;
    virtual ~ScriptableEntity() = default;

    // Main Functions
    virtual void onCreate() {}
    virtual void onDestroy() {}
    virtual void onUpdate(Time dt) = 0;
    virtual void onImGuiRender() = 0;

    // Virtual Constructor Pattern
    template<class Derived>
    Derived* clone() const
    {
        return new Derived// new type Derived
            static_cast<const Derived &>(*this) // cast to derived type
    };
}
private:
    Entity m_entity;
};

```

如果要實作一個 Script 時，要繼承 ScriptableEntity 並實作其提供的介面，onCreate() 會在 Script 建立時被執行、onUpdate() 每 frame 會執行一次、onImGuiRender() 則是用於 Editor 時可以調參數、onDestroy() 則是在 Script 被銷毀前會執行。

```

class ExampleScript : public ScriptableEntity
{
public:
    virtual void onCreate() override
    {
    }
    virtual void onDestroy() override
    {
    }
    virtual void onUpdate(Time dt) override
    {
        /* Update the Entity */
    }
    virtual void onImGuiRender() override
    {
        /* Update UI in Editor */
    }
};

```

但因為我們現在引擎是採用 ECS 架構，所以還要有一個 Component 讓 Entity 擁有。定義 NativeScriptComponent 有三個資料：instance 是 ScriptableEntity 也就是實際邏輯的 Object、scriptName 是 Script 的名稱，預設是 `rl::EmptyScript`、valid 則代表該 Script 是否初始化。

[4](#)

⁴`entt::type_info<T>::name()` 是我們使用的一個 ECS 函式庫提供的 RTTI 的函式，可以拿到一

提供了兩個函式: `bind()` 和 `unbind()` 分別是綁定和解除綁定。在 `bind()` 時會初始化參數和指定正確的 Entity，因為 Script 內可能會去拿當前 Entity 的其他 Component 例如位置、旋轉、速度等，所以必須在初始化時綁定好；而 `unbind()` 就是刪除 ScriptableEntity。

```
struct NativeScriptComponent
{
    ScriptableEntity* instance = nullptr;
    std::string scriptName =
        std::string{entt::type_info<EmptyScript>::name()};
    bool valid
        = false;

    NativeScriptComponent() = default;
    ~NativeScriptComponent() = default;

    template<typename T, typename ... Args>
    void bind(Entity entity, Args&& ... args)
    {
        if(instance)
        {
            delete instance;
            instance = nullptr;
        }
        //
        scriptName = entt::type_info<T>::name();
        instance = new T(std::forward<Args>(args)...);
        instance->m_entity = entity;
    }

    void unbind()
    {
        delete instance;
        instance = nullptr;
    }
};
```

將一個 Entity 加上 NativeScriptComponent 在引擎 API 使用起來像這樣:

```
// 新增一個叫 Player 的 Entity
Entity ent = scene->createEntity("player");
// 加入 NativeScriptComponent 並綁定 ExampleScript
ent.addComponent<NativeScriptComponent>().bind<ExampleScript>();
```

在引擎的流程中，在 `onScenePlay()` 時，會初始化 Scene，以及所有 System 其中就包含了 NativeScriptSystem 其中會呼叫 NativeScript 的 `OnCreate()` 初始化該 Script，接著每次 loop 會呼叫 `OnUpdate()` 執行 Script 的邏輯。在要 Script 要結束時（主動結束或是被動結束）會呼叫 `OnDestroy()` 來清理該 Script。

在 RishEditor 中，可以使用 UI 替一個遊戲物件 Entity 加上 NativeScriptComponent。

個 Type 的名稱

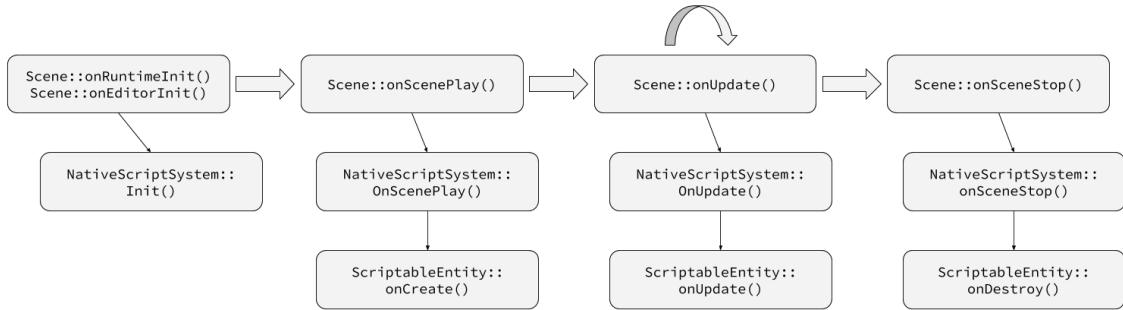


Figure 4.11: NativeScript 流程圖

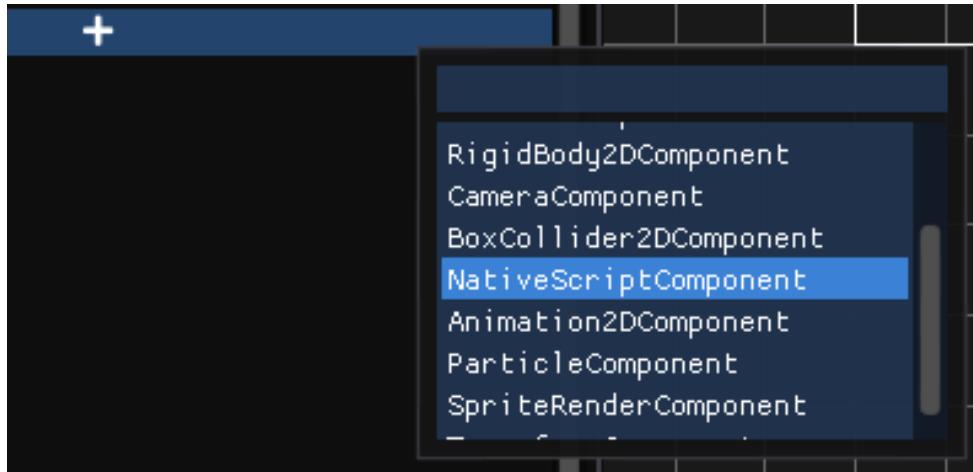


Figure 4.12: 加入 NativeScript 示意圖

Workflow

如何在 RishEngine 上新增自己的 Script 呢？首先要先撰寫一個繼承自 ScriptableEntity 的 class 例如撰寫角色的移動邏輯

```

class PlayerController : public ScriptableEntity
{
public:
    void onUpdate(Time dt) override
    {
        auto &transform = GetComponent<TransformComponent>();
        /* Player movement code */
        if(Input::IsKeyPressed(Keyboard::Left))
            transform.translate.x -= dt * 10.f;
        /* ... */
    }
}
    
```

接著在 ScriptableManager::Init() 註冊，註冊後的 Script 會出現在 Editor 中：

```

void ScriptableManager::Init()
{
```

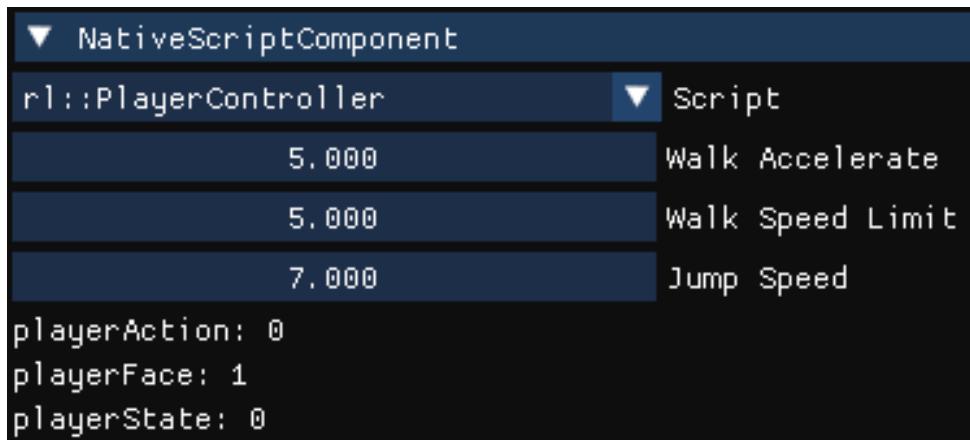


Figure 4.13: NativeScript 介面圖

```
/* Other Scripts */
ScriptableManager::Register<PlayerController>();
}
```

重新編譯並打開 Editor 後可以在 NativeScriptComponent 中選擇剛剛撰寫的 Script 名稱，在 Script 中的 `onImGuiRender()` 可以撰寫 UI 的邏輯（用於 Debug、挑整參數），便會顯示在此處。

未來展望

- 使用 Scripting Language
 - 目前 RishEngine 直接使用 C++ 作為 Script 的語言，優點是可以直接使用引擎的 C++ API，但缺點就是編譯非常花時間
 - 可以使用開源的專案，接著只要提供該語言的引擎 API 之後，便可以用該腳本語言撰寫遊戲邏輯
 - 例如 `mono(C#)`、`sol2(lua)`、`pybind11(python)` 綁定腳本語言
 - 動態語言雖然效能沒有編譯語言好，但動態語言不用等待編譯，這對需要快速迭代的遊戲來說至關重要

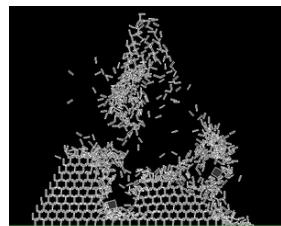
4.1.7 Physics Engine

前言

甚麼是遊戲物理引擎 為了讓遊戲能夠更趨於真實，聰明的遊戲開發者就將物理學的知識與觀念融入到遊戲當中，試著用各種不同的物理參數來真實世界的情況。物理引擎又有兩種常見的類型，一種為實時的物理引擎和高精度的物理引擎。在遊戲物理中，會選擇使用實時的物理引擎，並在程式中想辦法降低演算法的複雜度，或許從中會犧牲少部分的精確度，但在不影響肉眼「看起來」的情況下是可以被接受的。而高精度的物理引擎，通常使用在科學研究（計算物理學）和電腦動畫電影製作上，對於細節比較講究的才會選擇使用高精度的物理引擎。

常見的物理引擎介紹

2D 遊戲物理引擎



- Box2D
 - 一款免費的開源二維物理引擎，由 Erin Catto 使用 C++ 編寫，在 zlib 授權下發布
 - 著名遊戲



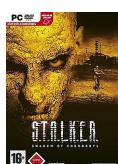
- Angry Bird
- 一個剛體動力學模擬引擎和一個碰撞檢測引擎，支援長方體、球體、圓柱體

3D 遊戲物理引擎



Open Dynamics Engine

- ODE
 - Russ Smith
 - 一個剛體動力學模擬引擎和一個碰撞檢測引擎，支援長方體、球體、圓柱體
 - 著名遊戲



- 浩劫殺陣：車諾比之影



- 一個跨平台的開源物理引擎，支援三維碰撞檢測、柔體動力學和剛體動力學，多用於遊戲開發和電影製作中
- 著名遊戲



- Havok
- 遊戲動力學開發工具包 (Havok Game Dynamics SDK)
 - 一個用於物理（動力學）效應模擬的遊戲引擎，為電子遊戲所設計，注重在遊戲中對於真實世界的模擬
 - 著名遊戲



Note. 以上內容出自維基百科

研究背景

在大學期間，系上的相關撰寫遊戲課程當中，一般來說要運用物理模擬幾乎都會用到Box2D 這個 2D 的物理引擎，但都往往無法深入探究裡面的實作以及其中的原理，頂多只會運用物理引擎的函式庫來時做遊戲。因此，想透過這次專題的機會，來深入探討 Box2D 其中的原理，試著自己實作看看簡單的物理模擬，將所學到的物理知識透過程式來實現出來，並將遊戲物理融合到這次的專案當中。

研究歷程

- 拾回物理的理論
- 搜尋網路資源
 - [Box2d-lite](#)
 - [Google Code Archive Box2d](#)
 - [phenomLi](#)
 - [GDC2014 - Understanding Constraints](#)
 - [GDC2015 - Bend the Physics Engine to Your Will](#)
 - [AllenChou Game Physics series](#)
- 閱讀相關技術文章，結果掉出同樣在研究遊戲物理來我個人網站上留言

 phenom • 4 months ago
看见了你读了我的文章，我的文章通常没什么人读，看来你也是研究物理引擎的同行？有空可以交流一下。
2 ⤵ | ⤴ • Reply • Share >

 halloworld Mod → phenom • 4 months ago
感謝大大的文章，讓我對box2d-lite的實作有一點理解，雖然還沒到完全理解，很高興大大寒舍光顧，由衷感謝，期望能與您有更多討論。
^ | ⤴ • Reply • Share >

 phenom → halloworld • 4 months ago
过奖了，我不是大大，只是一个普通学生而已💡，不知道你对box2d-lite的研究到哪一个程度呢？
其实我个人感受，如果要学习物理引擎，看matter.js的源码比较好，box2d-lite比较简陋，而box2d又太过复杂，matter.js在他们之间，功能足够，写得比较有条理也易懂。但是contact solver部分还是参考box2d-lite比较好，因为matter.js用了一种不是很主流的方法。
^ | ⤴ • Reply • Share >

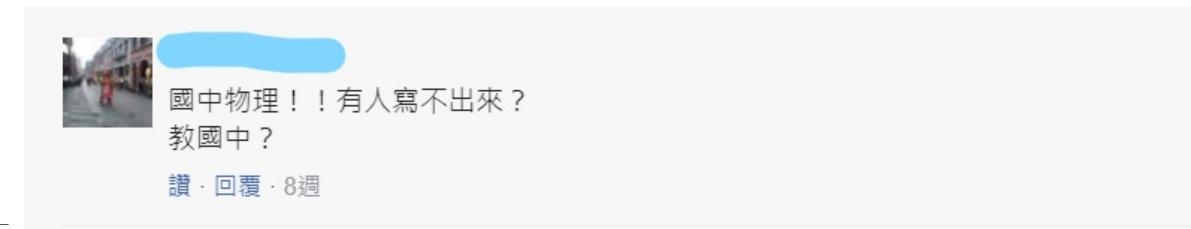
 halloworld Mod → phenom • 4 months ago
我大概看完(記得在box2d-lite裡是會return 兩兩物體的接觸點，不過細節的數學部分我有部分不太懂)，應該就contact solver的部分，然後我有看到您的文章，也有提到matter.js的部分，我還沒去研究它，我近期會來研究。如果有遇到問題，是否能夠請教你。
^ | ⤴ • Reply • Share >

 phenom → halloworld • 4 months ago
可以的，我尽量解答，因为我有一些地方也没有完全搞懂（比如约束部分）
^ | ⤴ • Reply • Share >

- SITCON2020 - 你說這隻溫馴的繩嗎？
 - 有資源可以詢問
 - 藉此拿到遊戲物理界大老 Allen Chou 系列影片
- 證明 Box2d 裡的一些物理公式，藉此驗證以及知其所以然
- 融合進 RishEngine 當中
- 參考其他開源專案的實作
 - Impulse Engine
 - 了解圓、多邊形的 Contact Point 找法
 - 藉此能夠實作 RishEngine 的圓、多邊形的碰撞

懷疑與困難

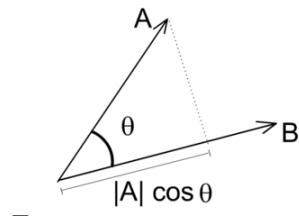
- 重造輪子的問題
 - 在這個成熟的 box2D 函式庫，其中已被許多人參與過此專案，經過多年的維護以及各方的腦力激盪，程式碼早難以閱讀
 - 看到周圍用 Unity 的同學，通常只要新增一個 Component 就能夠使用，常常會懷疑自己在幹嘛，何必研究這個東西，其他人都寫好了
- 我不是物理系的
 - 雖然裡面主要的實作幾乎都只用到高中物理，但已經離高中時期有一段時間，所以也花了時間來複習以前的高中物理
- 資源鮮少
 - 大部分都是英文文獻，以及套用函式庫的教學
 - google、youtube 搜尋「物理引擎」，幾乎都是教「怎麼用」，而不是「裡面在幹嘛」
- 3D 太難
 - 牽涉到空間的物理模擬對於短時間來說要實做出來太過於困難（畢業專題從開始製作總共也就幾個月而已）
 - 因此便放棄研究 3D
- 網路上的酸言酸語



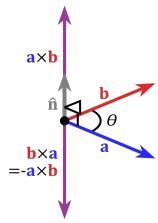
核心觀念

數學

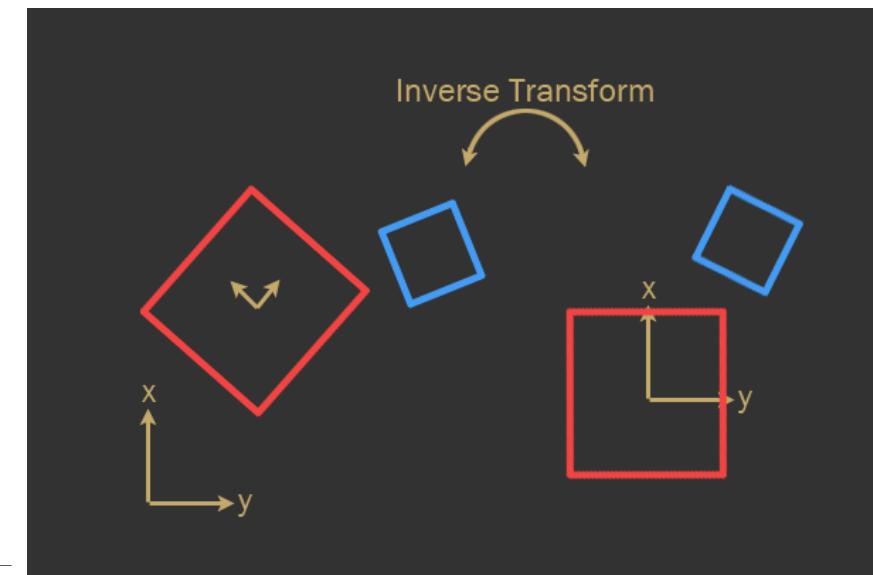
- 純量
 - 只有大小、沒有方向、可用實數表示的一個量
- 向量
 - 內積
 - 內積在物理引擎模擬時通常都是用在投影上，代表一條向量到另外一條向量的投影量
 - 外積
 - 外積通常擔任的身分都是旋轉居多，在三維空間的外積通常表示兩向量維出紙面或入紙面的旋轉，在二維則為順時針旋轉跟順時針旋轉居多



- 座標轉換
 - 利用線性代數座標空間觀念來方便我們做運算
 - A 為旋轉矩陣
 - $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$
 - $A^{-1} \cdot p$
 - 將 world space 轉換到 p 的 local space
 - $A^T \cdot p$
 - 將 p 的 ‘local space’ 轉換到 ‘world space’



- 座標轉換
 - 利用線性代數座標空間觀念來方便我們做運算
 - A 為旋轉矩陣
 - $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$
 - $A^{-1} \cdot p$
 - 將 world space 轉換到 p 的 local space
 - $A^T \cdot p$
 - 將 p 的 ‘local space’ 轉換到 ‘world space’



Note. 左下角為 worldSpace 的座標，紅框裡的則為各個物體的 localspace 座標

物理

線性相關

- 牛頓力學
 - 定律一
 - 物體在無外力的作用下，會傾向於維持靜止不動，或繼續固定的速度在一直線上運動，這就是慣行的觀念
 - 定律二
 - 物體的加速度與作用於該物體上的力成正比，加速度方向與作用力方向相同
 - $F = m \cdot a$
 - 定律三
 - 對於作用於物體上的力，都有一個大小相同方向相反的反作用力，且作用力與反作用力位在一條線上
 - 物理參數
 - 質量 m
 - 位置 x
 - 速度 v
 - $x = v \cdot \Delta t$
 - 加速度 a
 - 牛二 - $F = m \cdot a$

- 動量

- 向量，意義是物體在其運動方向上保持運動的趨勢

- $P = m \cdot v$

- 衝量

- 單位時間內的動量變化量

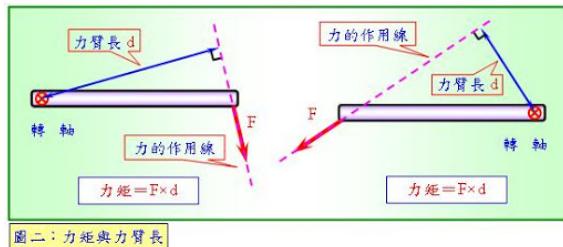
- $J = \Delta P = m \cdot \Delta v = F \cdot \Delta t$

角度相關

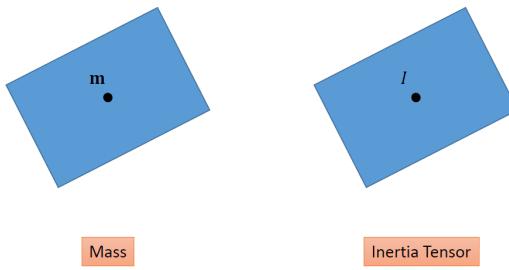
- 力矩 τ

- $\tau = F \cdot d$

- d 為受力點與質心的距離



- 轉動慣量 (慣性矩) I



Note. 既然線性中有質量，在旋轉上也就有所謂的轉動慣量

- 定義

- $I = \sum m_i r_i^2 = \int_V \rho r^2 dV$

- 拿同樣質量的物體做旋轉時，因參考點的不同，導致有慣性的力量出現

- e.g. 筆拿不同邊，甩的時候力量不一樣，拿邊邊要用比較多力，拿中間甩起來比較輕鬆

- 2D 矩形的質心慣性矩

- 質量為 m ，寬度為 w ，高度為 h 的二維

- $I = \frac{m(h^2 + w^2)}{12}$

- 圓形的質心慣性矩
 - 質量為 M ，半徑為 r
 - 二維的轉動慣量為單一值，三維則是 3×3 的矩陣

Note. 各種不同的慣性矩如下: https://en.wikipedia.org/wiki/List_of_moments_of_inertia
牛二旋轉版，角加速度 α

- 線性有 $F = ma$ ，物體的角加速度 (α) 與物體所受的轉矩 (τ) 成正比，其方向跟轉矩的方向相同

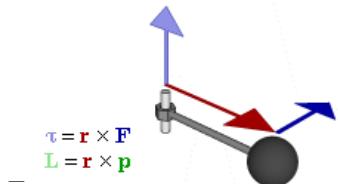
- $\tau = I\alpha, \alpha = \frac{\tau}{I}$
- α 為角加速度

角速度

- 算出物體旋轉的角度
 - $\omega = \alpha \cdot \Delta t$
 - $\theta = \omega \cdot \Delta t$

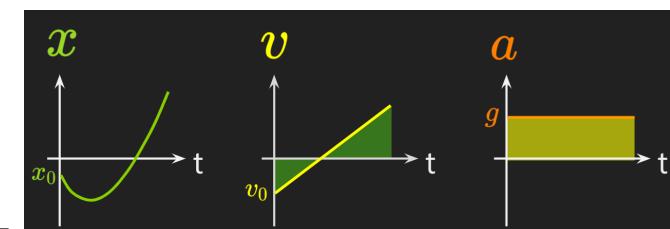
角動量

- 物體的角動量是物體的位置向量和動量的叉積，通常寫做 L
 - $L = r \times p$



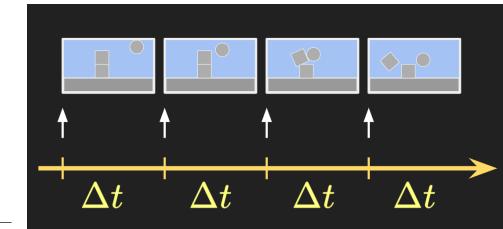
微分、積分

- 位置、速度、加速度
 - 位置 \rightarrow 速度 \rightarrow 加速度 ($\frac{d}{dt}$) 微分
 - 位置 \leftarrow 速度 \leftarrow 加速度 ($\int dt$) 積分



- 那，不就找到積分函數就好了？
 - 因為我無法知道下一刻的加速度長甚麼樣子，函數圖形有可能會漲跌

- 而只要有力，加速度就會有變化 (碰撞、拉扯、浮力 (x))
- 因此只能透過數值積分來解決



- Euler-Method(歐拉法數值積分)
 - 用前一刻的值，算出下一刻的值是多少
 - 程式寫起來可能通常長這樣

```
v += a * dt;
x += v * dt;
```

Note. 圖片出自: SITCON2020 - 你說這隻溫馴的繩嗎

Runge-Kutta *Runge – Kutta*方法

- 以 3rd Runge-Kutta 舉例

```
void timeStep(body)
{
    auto f = [=](StateStep &step, float2 vel, float t) {
        vel += gravity * t;
        step.velocity = vel;
    };

    StateStep F0, F1, F2, F3, F4;
    F0.velocity = float2(0,0);
    f(F1, F0.velocity, 0);
    f(F2, F1.velocity / 2, deltaTime / 2);
    f(F3, F2.velocity / 2, deltaTime / 2);
    f(F4, F3.velocity, deltaTime);

    auto v0 = body->vel + body->force * deltaTime /
        body->mass;
    body->vel = v0 + (F1.velocity + 2 * F2.velocity + 2 *
        F3.velocity + F4.velocity) / 6;
    body->pos += body->vel * deltaTime;
}
```

遊戲物理名詞定義

Rigid Bodies

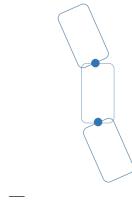
- 鋼體，能夠忽略形變的物體



Note. 鋼體與柔體的比較，鋼體例子如鐵盒，柔體例子如黏土

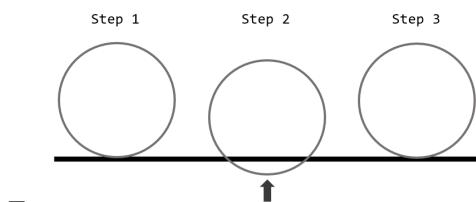
Joints

- 關節點，能夠連接兩個對象在一個單一的支點



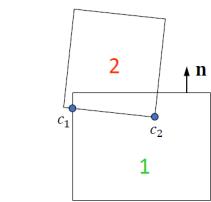
Constraint

- 中文翻譯為約束
 - 在遊戲物理的意義，則是讓物體符合遊戲物理世界的規定，像是物體不能穿過地板



Contact points

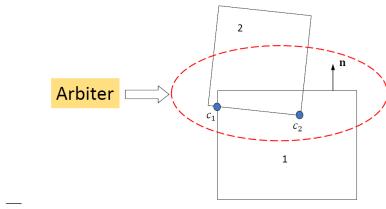
- 在此示意圖中， c_1, c_2 為接觸點
 - 在兩物體重疊之後，會計算出接觸點，透過接觸點來算出如何解掉約束



Arbiters

- 英文原意，仲裁者

- 每個 Arbiter 都會有兩個 contact-point，以及兩個 RigidBody(一個攻、一個受)，計算出該 point 應該要修正的衝量大小



實作



$$\Delta \mathbf{v} = \frac{\mathbf{P}}{m}$$

前言

- 依據 Box2D 作者描述，使用衝量模擬有以下優點
 - Most people don't hate impulses
 - 大家不討厭衝量**
 - The math is almost understandable
 - 數學總是好懂得**
 - Intuition often works
 - 較直覺去實現**
 - Impulses can be robust
 - 衝量可以較為穩定**

Note. 內容出自 Box2d Lite 投影片

模擬流程

分析受力 ComputeForce

- ComputeForce
 - 透過受到的力，算出物體的速度及加速度
 - 若 $mass = 0$ ，則不更新
 - $v += \Delta_t * (g + \frac{F}{m})$
 - $\omega += \Delta_t * \frac{\tau}{I}$

```

if (invMass == 0.0f)
    return;
else
{
    velocity += delta_t * (gravity + invMass * force);
    angularVelocity += delta_t * invI * torque;
}

```

更新速度 *UpdateVelocity*

- Integrate Velocities
 - 對速度、速度做積分，能知道 position 與 rotation 實際的值
 - $pos += \Delta_t * v$
 - $rot += \Delta_t * \omega$

```

for (int i = 0; i < (int)bodies.size(); ++i)
{
    Body* b = bodies[i];

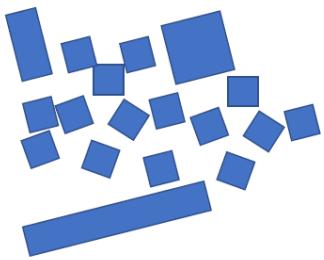
    b->position += dt * b->velocity;
    b->rotation += dt * b->angularVelocity;

    b->force.Set(0.0f, 0.0f);
    b->torque = 0.0f;
}

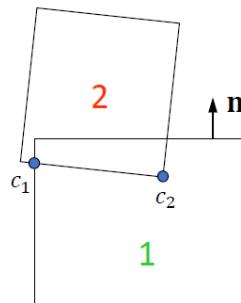
```

碰撞檢測、找出接觸點 *CollisionDetection*

broad-phase



narrow-phase



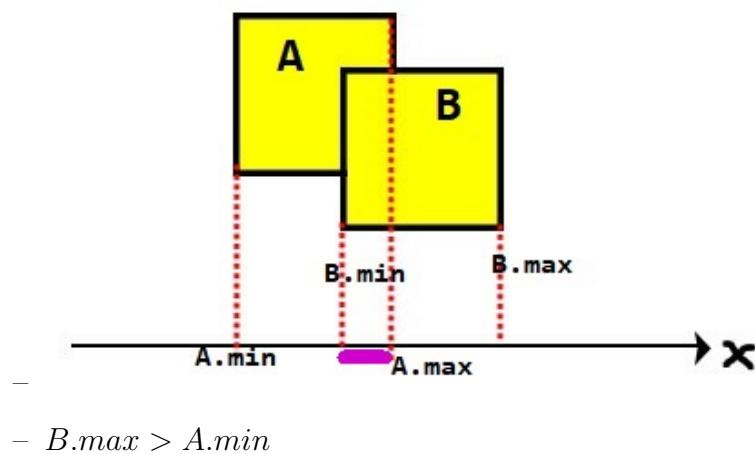
Broad-phase大略檢測

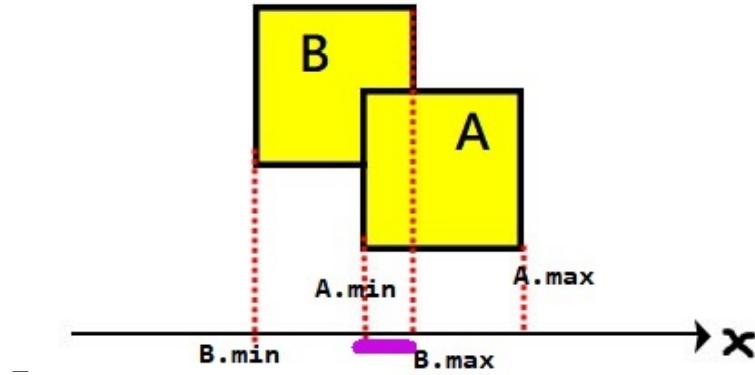
- 暴力法 *BruteForce*
 - 運用迴圈偵測每兩兩個物體，並判斷是否有無碰撞

- Box2D Lite 使用的是 $O(N^2)$ 算法，在實際的 Box2D 引擎中，採用的是 AABB Tree, grid 等等的優化演算法，以利於加速整個引擎的碰撞判斷
- 四叉樹 *QuadTree*
 - 四叉樹 (QuadTree) 是一種劃分 2D 區域的樹狀資料結構，類似一般的二元樹，不過子節點為 4 個
 - 可以用來加速碰撞的粗估判斷
 - QuadTree 細節

Narrow-Phase (實際檢測) - 碰撞演算法

- AABB
 - 簡介
 - 為了方邊物體之間進行碰撞檢測運算，通常會對物體創建一個長方形將其包圍，AABB 包圍盒也被稱為軸對齊包圍盒
 - 以矩形包圍物體
 - 矩形的每條邊，皆與坐標系的軸垂直
 - 缺點
 - 當物體旋轉時就無法檢查
 - 只能檢查矩形問題
 - 實作 (需同時滿足兩個條件)
 - $A.max > B.min$
 - $B.max > A.min$





- Example Code

```

bool CheckCollision(Shape &a, Shape &b) // AABB - AABB
    collision
{
    // x-axis
    bool collisionX = a.Position.x + a.Size.x >=
        b.Position.x &&
        b.Position.x + b.Size.x >= a.Position.x;
    // y-axis
    bool collisionY = a.Position.y + a.Size.y >=
        b.Position.y &&
        b.Position.y + b.Size.y >= a.Position.y;
    return collisionX && collisionY;
}

```

Note. 圖片參考自: <http://davidhsu666.com/archives/gamecollisiondetection/>

SAT SeperatedAxisTheorem

- 優點
 - 可以判斷旋轉時的物體之碰撞
- 缺點
 - 無法檢查凹多邊形，但是能透過將多個凸多邊形組合成凹多邊形的形狀，來作碰撞檢測 SAT 必須檢查所有的法向量，來確保物體之間沒有分離線，越多的物體發生碰撞，效率也就越低
 - 演算法細節請參考以下連結

Note. <https://hackmd.io/bo-sYg9TQhyGgpWfJoWC2Q?view>

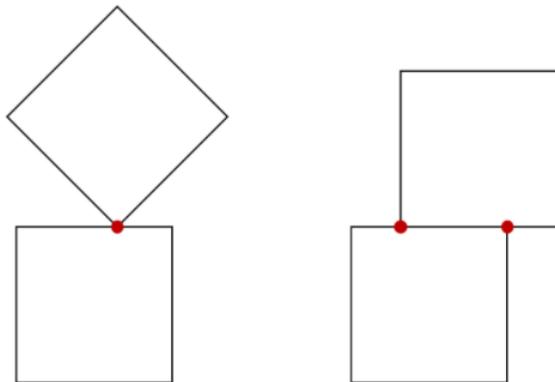
GJK Gilbert – Johnson – Keerthi EPA Expanding Polytope Algorithm

- 演算法細節請參考以下連結

Note. <https://hackmd.io/WOPle5lbRzuBzUhgFZ3h5Q>

尋找接觸點和最小穿透軸 *FindContactPoints, MinimumpenetrationAxis*

- 在解決約束之前，我們必須要先找到適合的點以及能夠將兩物體分離的向量
- 接觸點

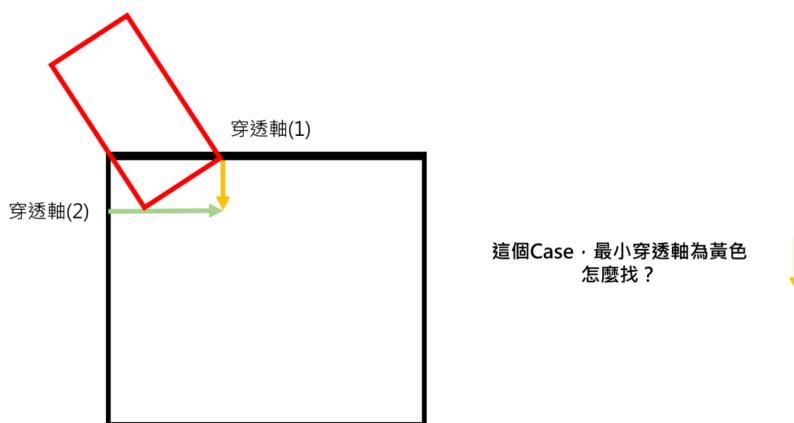


– 碰撞點是處理碰撞的關鍵，因為這取決了碰撞之後求解器計算出的衝量要應用於物體的哪個位置，在物體的不同位置應用衝量往往會產生截然不同的效果

– 舉例，若將接觸點訂在質心，東西則會筆直的向外飛出，而在非質心點的位置，則可能造就物體的旋轉

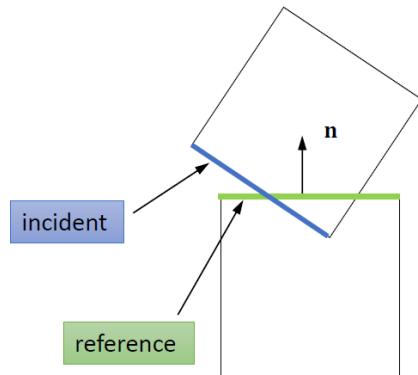
- 最小穿透軸 *MinimumpenetrationAxis*

– 在兩兩物體互相碰撞的狀態下，一定有部分地方邊被插入得比較深，相對的，也會有部分邊被插入得比較淺，或是根本沒有被碰撞到。我們可以透過 SAT *SeperatedAxisTheorem* 演算法，找出到底哪個地方的邊是被插入最少的，並透過最小穿透軸的法向量，找出最能有效將兩物體分離之向量



- Reference Face Incident Face

- Identify reference face

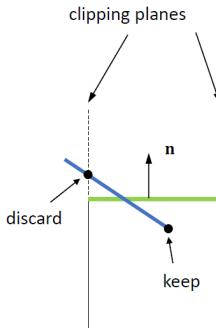


- Identify incident face

- 找出最小穿透軸之後，我們會決定 incident face 及 reference face，reference 代表被插入的邊，incident face 則是插入的邊
- V-Clip 演算法

Box-Box Clipping

- Clip incident face against side planes



- Discard points above reference face

- 透過 V-Clip 演算法，能夠切除在接觸面以外的接觸點，過濾不必要的接觸點，讓模擬更加穩定及合理， n 為最小穿透軸的法向量，綠色為 reference face，藍色為 incident face

- 詳細算法可以參照以下連結

– https://www.slideshare.net/slideshow/embed_code/key/5R0vx2057BpeRL

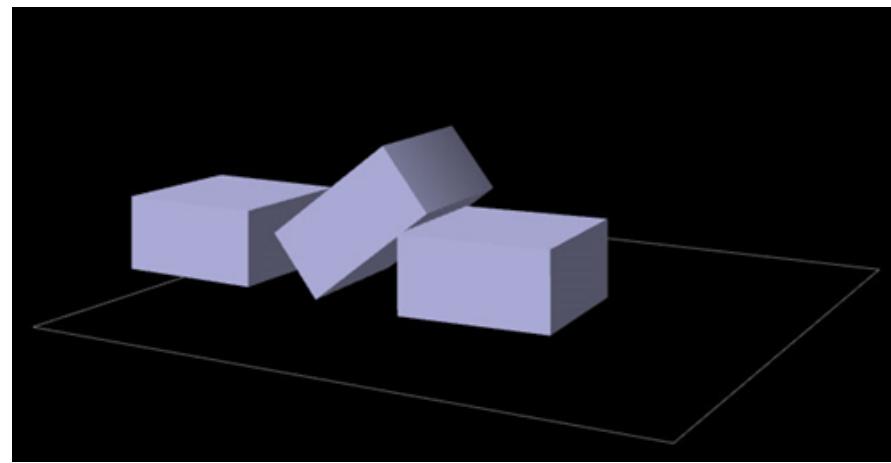
解決約束 SolveConstraints

- 約束

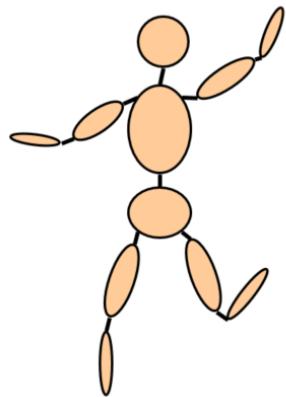
– 可以想像成一種在物理世界訂定的規定，遊戲設計者可以依據不同的鋼體物理元素賦予不一樣的規定，讓鋼體有不同的結果。在前一個階段我們計算出接觸點，目的就是藉此用來解決約束

- 約束的例子

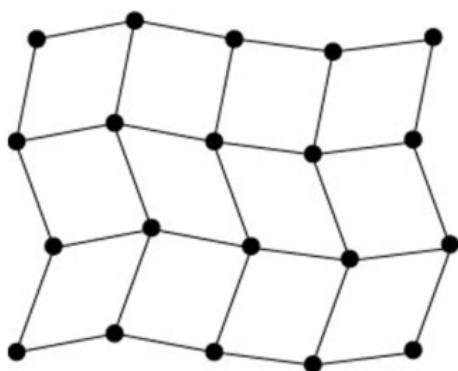
– Contact and Friction



- Ragdolls



- Particles and cloth

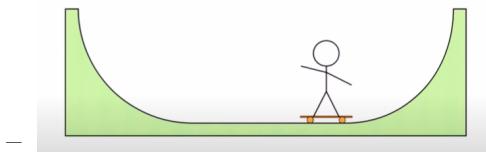


Note. 圖一是接觸與摩擦力約束，圖二是布娃娃的物理系統，詳情可以參考，圖三是用物理粒子連結成一塊布，以上都是約束的例子，圖片出自 Erin Catto 2008 Physics PPT

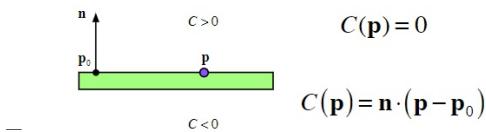
位置約束

- 位置約束

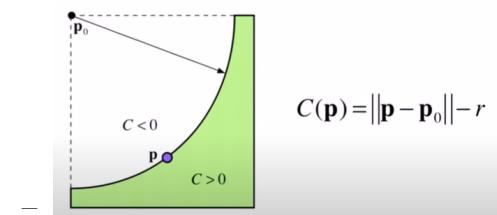
- 舉一個簡單的例子，一個人在溜滑板，滑板被限制在坡道上，使人能夠在坡道上安信的划著滑板



- 我們可以將滑板當作一個質點，就可以簡單的求出滑板在彎道及平面的約束式子， C 可以當成一個函數，當 $C > 0$ ，代表滑板在平面或曲面之上，換句話說，一旦我們發現這個函數並不等於 0 的時候，程式就必須介入來解決約束



Note. 平面上的約束



Note. 曲面上的約束

Ground Constraint 地面約束

- 約束條件，我們不讓物件掉到地下
 - 約束條件 $y > 0$
 - 對應的位置約束: if $y \leq 0$ 則 $y = 0$
 - 對應的速度約束: $V_y = 0$?
 - 若直接對速度直接設定為 0，結果會是我碰到地板之後立即停下，但並沒有任何的緩衝
 - 解決方式：Baumgarte Stabilization

Baumgarte Stabilization 邦加特修正

- 透過 y 的位置誤差，修正 V_y
 - $V_y = 0 \rightarrow V_y + \frac{\beta}{\Delta t} = 0$
 - β 是 0 1 之間小數
 - beta 值越小，不立即修正位置誤差

- beta 值越大，立即修正位置誤差
- 精確的求法，牽涉到更嚴格的磨擦係數

地面約束實作

- 透過 y 的位置誤差，修正 V_y

```
float dt = delta_time;
float y = pos.y;
vy += Gravity(重力加速度) * dt;

if (y <= 0.0f)
{
    vy = -(Beta / dt) * y;
}
y += vy * dt;
```

Generalization 通則化

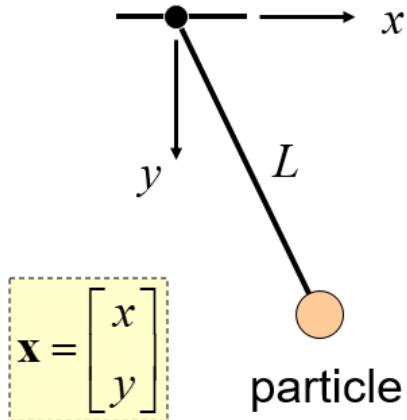
- 我們可以把約束，通則成以下這個矩陣式子
 - $JV + b = 0$
 - $\begin{bmatrix} J_{v1} & J_{v2} \end{bmatrix} \begin{bmatrix} V_x \\ V_y \end{bmatrix} + b = 0$
 - J: Jacobian Matrix
 - V: Velocity matrix 速度矩陣
 - 線性速度分量
 - b: bias 偏重
 - 速度修正項
- 當地面約束套入通則化
 - $JV + b = 0$
 - $V_y + \frac{\beta}{\Delta t} = 0$
 - $J = [0 \ 1]$
 - x 分量的修正為 0， y 分量的修正係數為 1
 - $b = \frac{\beta}{\Delta t}y$
 - 偏重修正

解析速度約束

- 1. $JV + b = 0$

- 此式這就是一個通則，能夠將各個約束套入這個式子，這個過程叫做 Generalization
- J 就把他想成修正項，對原先速度的 V 修正，再加上偏差項
- 在多維度也會適用
- 2 . $M(V_2 - V_1) = L\lambda$
 - M 質量矩陣
 - $\begin{bmatrix} M_1 & 0 \\ 0 & M_2 \end{bmatrix}$
 - V_1 : 一開始違反約束的速度
 - V_2 : 是修正後的速度
 - 因此 $V_2 - V_1$ 為速度的修正量
 - L : 向量，修正衝量方向
 - λ : 純量，修正衝量大小
- 我們要算出 L (修正衝量之方向)， λ (修正衝量之大小) 是多少?
 - 3 . $L = J^T - J^T$ 為 J 的轉置矩陣
- 將 3 帶入 2
 - 4 $V_2 = V_1 + M^{-1}J^T\lambda$
- 再將 4 帶入 1 ，我們知道修正後的速度 V_2 帶入 Jacobian 的那條式子，一定為 0
 - $J(V_1 + M^{-1}J^T\lambda) + b = 0$
- 移項後可得
 - $(JM^{-1}J^T)\lambda = -(JV_1 + b)$
- 求得 λ
 - 5 $\lambda = \frac{-(JV_1 + b)}{JM^{-1}J^T}$
- 賦予定義
 - M_{eff} 有效質量 EffectiveMass -
 - 6 $M_{eff} = (JM^{-1}J^T)^{-1}$
 - λ 拉格朗日乘數 LagrangeMultiplier -
 - 7 $\lambda = M_{eff} * [-(JV_1 + b)]$

距離約束



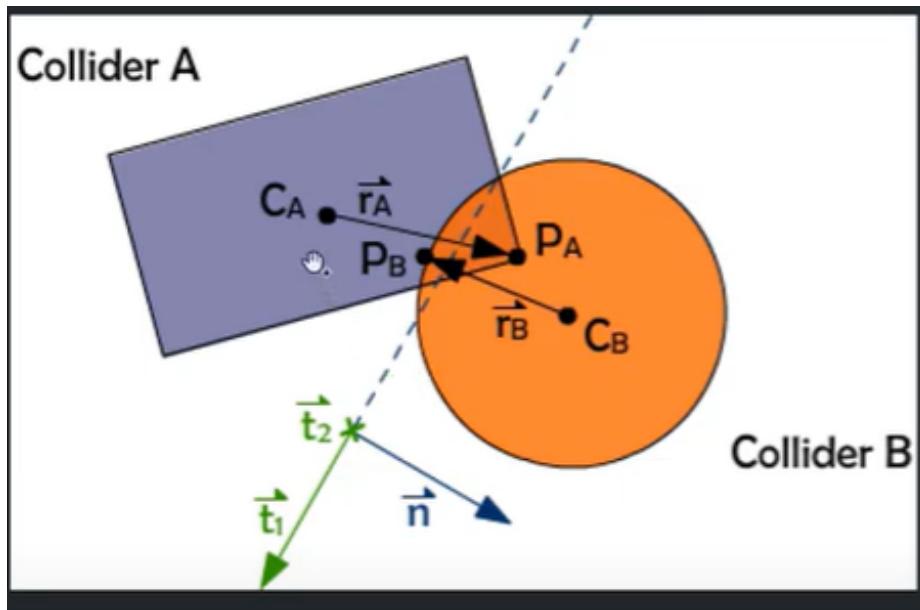
- 他的位置約束的式子長這樣
 - $C = \|x\| - L$
- 我們將它進行微分，則能得到速度約束的式子，從此也可以得知，修正衝量的方向為 J^T

$$C = \|x\| - L \quad (4.2)$$

$$\begin{aligned} \frac{dC}{dt} &= \frac{d}{dt} \left(\sqrt{x^2 + y^2} - L \right) \\ &= \frac{1}{2\sqrt{x^2 + y^2}} \frac{d}{dt} (x^2 + y^2) - \frac{dL}{dt} \\ &= \frac{2(xv_x + yv_y)}{2\sqrt{x^2 + y^2}} - 0 \\ &= \frac{1}{\sqrt{x^2 + y^2}} \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} v_x \\ v_y \end{bmatrix} \end{aligned} \quad (4.3)$$

$$\dot{C} = \frac{x^T}{\|x\|} v \quad (4.4)$$

碰撞回饋



- 質心 C_A, C_B
 - 穿透點 P_A, P_B
 - 向量 \vec{r}_A, \vec{r}_B
 - 兩者能夠最有效分開的法向量為 \vec{n}
 - \vec{n} 的切線向量為 \vec{t}_1, \vec{t}_2
- 需要 Constraints 的條件
 - 位置約束
 - $(P_B - P_A) \cdot n \geq 0$
 - P_A 指向 P_B 的向量投影在 \vec{n} 上
 - 若內積 < 0 ，代表碰撞
 - $(C_B + r_B - C_A - r_A) \cdot \vec{n} \geq 0$
 - 換句話呈現，拆成點與向量
 - 速度約束
 - $(-V_A - \omega_A \times r_A + V_B + \omega_B \times r_B) \cdot n \geq 0$
 - 這兩點的相對速度在法向量 \vec{n} 上，是要能分開的，也就不會越陷越深

- 通則化

$$\begin{aligned}
 \dot{C} &= (v_{p2} - v_{p1}) \cdot n \\
 &= [v_2 + w_2 \times (p - x_2) - v_1 - w_1 \times (p - x_1)] \cdots n \\
 &= \underbrace{\begin{bmatrix} -n \\ -(p - x_1) \times n \\ n \\ (p - x_2) \times n \end{bmatrix}}_J^T \begin{bmatrix} v_1 \\ w_1 \\ v_2 \\ w_2 \end{bmatrix} \quad (4.5)
 \end{aligned}$$

$$A \cdot (B \times C) = C \cdot (A \times B) = B \cdot (A \times C) \quad (4.6)$$

- 通則化結果

$$JV + b \geq 0 \quad v = \begin{bmatrix} V_A \\ w_A \\ V_B \\ w_B \end{bmatrix} \quad (4.7)$$

$$J = [-n^T \quad (-r_A \times n)^T \quad n^T \quad (r_B \times n)^T] \quad (4.8)$$

$$\lambda = \frac{-(JV_1 + b)}{(JM^{-1}J^T)} \quad M = \begin{bmatrix} M_A & 0 & 0 & 0 \\ 0 & I_A & 0 & 0 \\ 0 & 0 & M_B & 0 \\ 0 & 0 & 0 & I_B \end{bmatrix} \quad (4.9)$$

- 位置誤差修正 *bias*

- $JV + b = 0$
- $C : (P_B - P_A) \cdot n$

- 通式與位置約束合併

- $b = \frac{\beta}{\Delta t} C = \frac{\beta}{\Delta t} (P_B - P_A) \cdot n$
- 如果只套位置約束修正，頂多讓他們兩個不越陷越深，並不會彈開

- 必須要加上速度約束

- $b = \frac{\beta}{\Delta t} (P_B - P_A) \cdot n + C_R (-V_A - \omega_A \times r_A + V_B + \omega_B \times R_B) \cdot n$
- C_R 為彈性係數， $C_R = 1$ 會變完全彈性碰撞， $C_R = 0$ 的話會黏上去
- 有效質量的證明方式，因為細節較為繁雜，而提供連結給閱讀者參考
 - <http://www.dyn4j.org/2010/09/distance-constraint/comment-page-1/>

- 結合 M_{eff} 有效質量 (分母) 和修正的速度和方向 (分子)，就能得知修正衝量的結果

- 我們預期設定的 $v_n = 0 \quad P_n \geq 0$

- 求得的結果 $P_n = \max\left(\frac{-\Delta\bar{v} \cdot n}{k_n}, 0\right)$

- 修正衝量算式

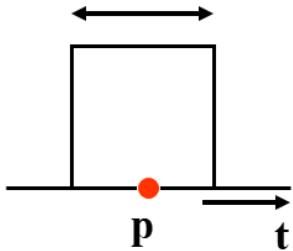
- $\Delta\bar{v} = \bar{v}_2 + \bar{w}_2 \times r_2 - \bar{v}_1 - \bar{w}_1 \times r_1$
- $k_n = \frac{1}{m_1} + \frac{1}{m_2} + [I_1^{-1}(r_1 \times n) \times r_1 + I^{-1}2(r_2 \times n) \times r_2] \cdot n$

- Joint

- 距離約束除了碰撞回饋之外，我們可以將其應用在將兩兩物體相連

摩擦力

- 消除穿透點 PA, PB 與切面方向 t_1 的相對速度
- 根據 Coulomb 理論，摩擦力會箝制在以下式子
 - Coulomb's law $|\lambda_t| \leq \mu\lambda_n$
 - In 2D $-\mu\lambda_n \leq \lambda_t \leq \mu\lambda_n$
- 一樣，我們可以將其通則化



$$\begin{aligned}
 \dot{C} &= v_p \cdot t \\
 &= [v + w \times (p - x)] \cdot t \\
 &= \underbrace{\begin{bmatrix} t \\ (p - x) \times t \end{bmatrix}}_J^T \begin{bmatrix} v \\ w \end{bmatrix}
 \end{aligned} \tag{4.10}$$

$$\begin{aligned}
 J_n &= [-n^T \quad (-r_A \times n)^T \quad n^T \quad (r_B \times n)^T] \quad \lambda_n \geq 0 \\
 J_{t_1} &= [-t_1^T \quad (-r_A \times t_1)^T \quad t_1^T \quad (r_B \times t_1)^T] \quad -\mu\lambda_n \leq \lambda_{t_1} \leq \mu\lambda_n \\
 J_{t_2} &= [-t_2^T \quad (-r_A \times t_2)^T \quad t_2^T \quad (r_B \times t_2)^T] \quad -\mu\lambda_n \leq \lambda_{t_2} \leq \mu\lambda_n
 \end{aligned} \tag{4.11}$$

積分結果 Integralposition

- Example Code

```

for (int i = 0; i < (int)bodies.size(); ++i)
{
    Body* b = bodies[i];
    b->position += timeStep * b->velocity;
    b->rotation += timeStep * b->angularVelocity;
    b->force.Set(0.0f, 0.0f);
    b->torque = 0.0f;
}

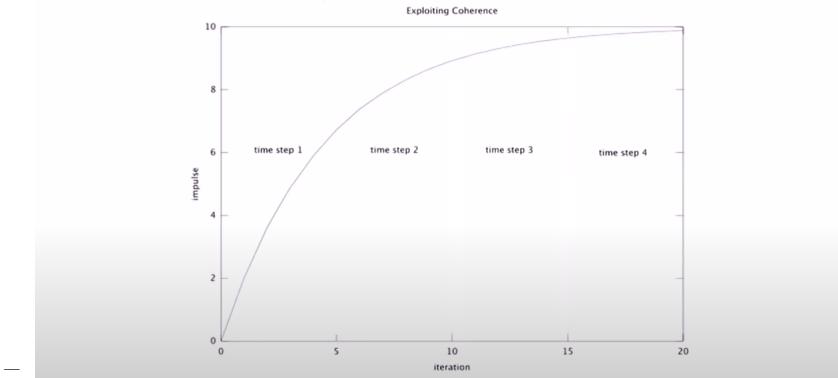
```

提升穩定性

- Warm Starting(暖身)

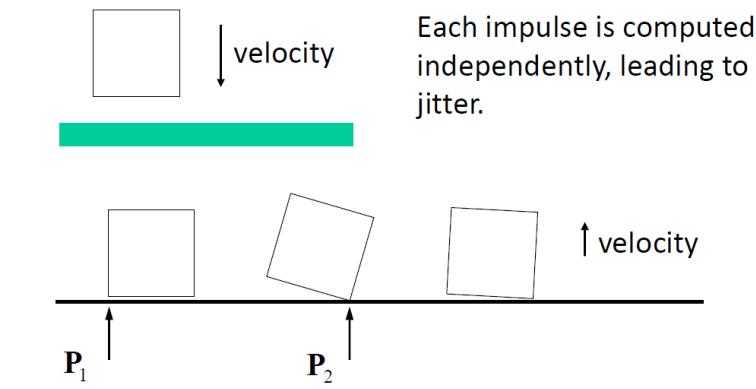
– 為了解決迭代次數過多的問題，我們可以使用暖身（warm starting）的方式來減少迭代次數。在大部分情況下發生碰撞的情況下，每一 frame 的並不會太大，因此我們可以利用連續的 frame 來分攤計算的迭代次數。實作非常簡單，在前一幀計算出修正衝量，直接施加在下一個 frame 上。

Warm starting can boost convergence



- Accumulation Impulse

– 在一般的衝量計算（Box2d 稱他為 Naïve Impulses）中，每個接觸點算出來的衝量都是獨立出來個別計算的，但這樣會導致微幅的抖動。除此之外，也不能保證每次算出來的修正量值都一定為正，這有可能導致物體會越陷越深的問題



– 解決的方法是將每一個 frame 的衝量都記錄下來，存成一個累積值 (accumulation)，並將其修正衝量都限制在 0 以上，就能夠避免越陷越深的問題了，以下使用簡單 code 來說明

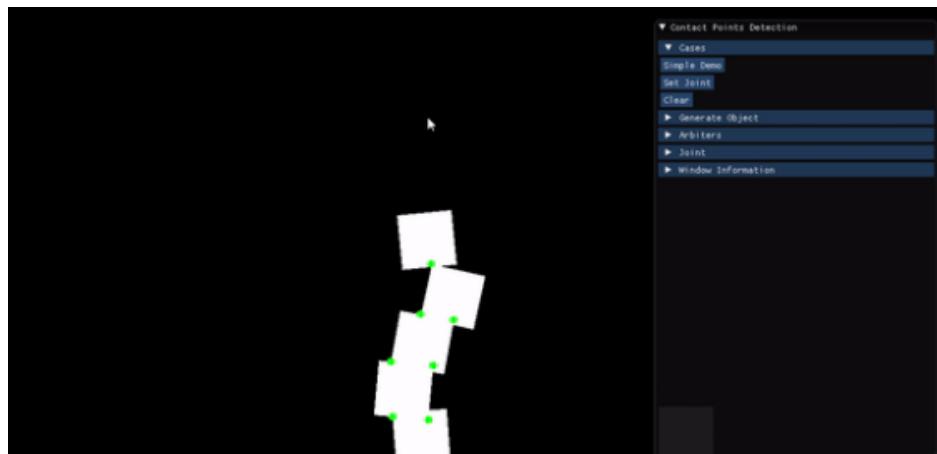
- Normal Clamping
- accmulation_Pn 累積衝量
- Pn 這一個 frame 要修正的衝量

```
temp = accmulation_Pn
accmulation_Pn = max(accmulation_Pn + Pn, 0)
Pn = accmulation_Pn - temp
\caption{}
```

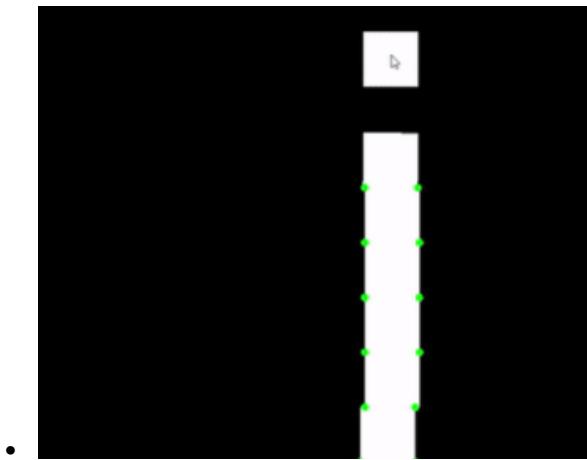
- Friction Clamping
 - accmulation_Pt 摩擦力累積衝量
 - Pt 這一個 frame 要修正的摩擦力衝量
 - u 為摩擦係數

```
temp = accmulation_Pt
accmulation_Pn = Clamp(accmulation_Pt + Pt, -u*accmulation_Pn,
                       u*accmulation_Pn)
Pt = accmulation_Pt - temp
```

- Non WarmStarting & Accumulation



- WarmStarting & Accumulation



問題

- 不支援凸多邊形
 - 因為使用 SAT 演算法來找出穿透點，因此無法支援圖多邊形，雖然看到網路上的教學都說把它切成凹多邊形即可，但似乎沒有甚麼東西能夠參考或是資訊可以知道要怎麼切
- 穿隧問題
 - Δt 的不固定，導致每次做數值積分的時候，位置更新都有部分誤差

未來展望

- 新增軟性約束
 - 軟性的約束可以模擬彈簧的效果
- 擴展成 3D 的遊戲物理引擎
 - 牽扯到 3D，相關運算似乎都要使用一些線性代數中矩陣乘法的一些技巧或是求解器，因為專題時間的關係，無法支援到立體之間的碰撞。

4.2 編輯器介紹

對於一個引擎而言，除了內部的程式操作外，亦需要一個面對於使用者的 GUI 來進行操作。其作用為讓開發者在進行開發時，可以簡化其操作步驟並方便的進行創作，透過對引擎可視化工具的操作來建置與修改遊戲內的場景與物體。

4.2.1 版面配置與操作介紹

打開 RishEditor (RishEngine 的編輯器) 可以看到版面就跟市面上常見的編輯器雷同。大致上可以分成四個區塊：工具欄、主要編輯視窗、遊戲物件 (Entity) 編輯視窗、Log。

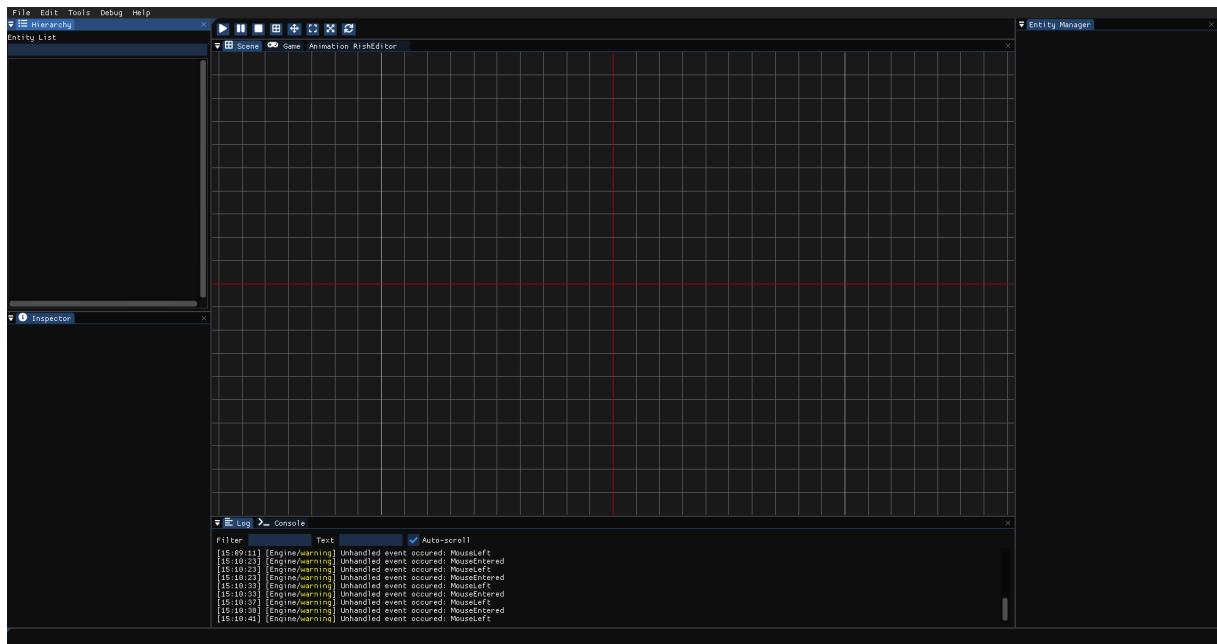


Figure 4.14: 編輯器整體介面

主選單MenuBar

- File
 - 新增、開啟、儲存目前正在編輯的遊戲場景 (Scene)
- Edit
 - 對遊戲物件 (Entity) 之操作: 複製、貼上、刪除
- Tools
 - 對 Editor 進行設定
- Help
 - 引擎介紹與製作團隊說明



Figure 4.15: 主選單介面

引擎實體列表 Hierarchy

- 主要為顯示場景中的 Entity 清單，並可對其進行操作
 - 滑鼠左鍵點擊選取 Entity，按住 Ctrl、Shift 可進行多選
 - 滑鼠右鍵開啟選單對 Entity 進行進一步的操作
 - 新增、複製、貼上、刪除 Entity
 - 將多個 Entity 設定為 Group 一起操作
 - 可將單一 Entity 用滑鼠拖曳進其他 Entity 中，使其成為 Group

引擎實體檢視 Inspector

- 顯示單一 Entity 所擁有的 Component 並加以操作
 - 新增、刪除、修改數值
- Component 的操作介面
 - 新增、刪除、修改數值
- 各個 Component 的操作 詳細參照 [Figure 4.24](#)
 - TagComponent
 - 標示其 Tag 與 ID
 - Tag 可修改，ID 不可修改
 - TransformComponent
 - 標示其位置、大小、旋轉角度

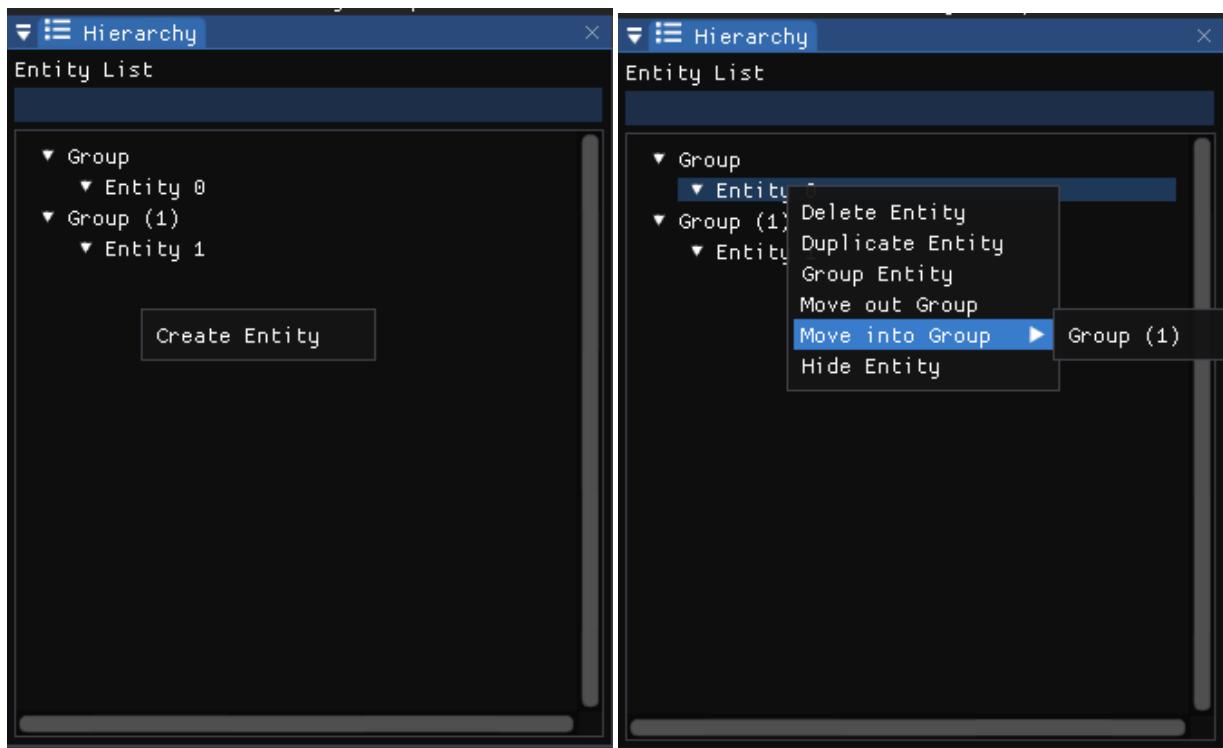


Figure 4.16: Hierarchy

- 皆可直接進行修改
- SpriteRenderComponent
 - 可讓 Entity 顯示圖像，RGB 調整圖像色塊
 - 可使用圖片與 TileMap
- CameraComponent
 - 有該 Component 的 Entity 其範圍內為遊戲進行時顯示的窗口
- NativeScriptComponent
 - 可讓 Entity 進行自主設定之 Script 的操作
- ParticleComponent
 - 粒子效果，能夠使用各種預設效果，像是火焰、雪等等。也能透過調動參數達到想要的效果並且儲存
- RigidBody2DComponent
 - 使物體具有鋼體物理的性質，能透過修改物理參數來控制不同的模擬結果
 - 能夠決定受力位置，模擬物體被推或拉一定的力量
- BoxCollider2DComponent
 - 讓物體能夠加入碰撞判斷，並具有力回饋

- Joint2DComponent
 - 能夠讓兩個剛體物理相互連接，並使兩物體控制在一定的距離之間
- LightComponent
 - 點光源效果，能夠讓世界擁有光源並受光照影響可見度亦會使物體產生陰影
 - 能夠決定受力位置，模擬物體被推或拉一定的力量
- AmbientLightComponent
 - 環境光照效果，能夠讓整個遊戲的可見度受此元件影響

工具欄 ToolBar



Figure 4.17: 工具欄介面

- Start
 - 將場景切換到 Game 介面，並開始進行測試
- Pause
 - 將 Game 介面的動作暫停
- Stop
 - 結束測試返回 Scene 介面
- Grid
 - 顯示/隱藏世界坐標軸
- Move
 - Gizmo 切換成移動模式
- Zoom
 - Gizmo 切換成拉伸模式
- Scale
 - Gizmo 切換成縮放模式
- Rotate
 - Gizmo 切換成旋轉模式

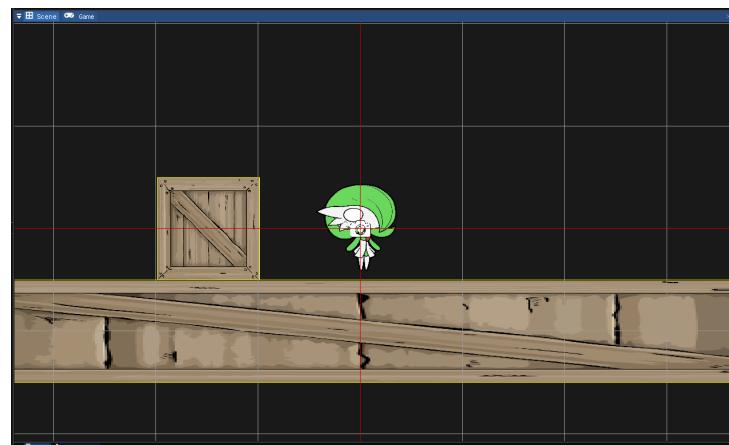


Figure 4.18: Scene View

場景視窗 Scene View

- 顯示所創建的 Entity
 - 滑鼠左鍵點選或圈選 Entity，並根據 Gizmo 的模式對其進行相對應的操作
 - 滑鼠右鍵移動視角
 - 滑鼠滾輪進行縮放
 - 使用快捷鍵對 Entity 進行操作
 - ‘Ctrl+A’ 全選
 - ‘Ctrl+C’ 複製
 - ‘Ctrl+V’ 貼上
 - ‘Ctrl+G’ 將選取的 Entity 放到一 Group 中
 - ‘Delete’ 將選取的 Entity 刪除
 - ‘Esc’ 取消選取 Entity
- Gizmo
 - 可讓使用者直觀的對 Entity 的大小與位置進行調整
 - 移動
 - 滑鼠按壓中間黃色區塊拖曳 Entity 移動位置
 - 按壓藍色箭頭限定水平移動
 - 按壓紅色箭頭限定垂直移動
 - 拉伸
 - 在 Entity 上下左右與角落處共八個區塊
 - 按壓並拖曳各個區塊皆可改變 Entity 的大小

- 縮放
 - 對於 Entity 的長與寬從中心延伸出紅線與藍線
 - 在尾端有黃色區塊可按壓拖曳改變 Entity 的長與寬
- 旋轉
 - 可將 Entity 進行旋轉，會有一紅線表示現 Entity 的旋轉角度

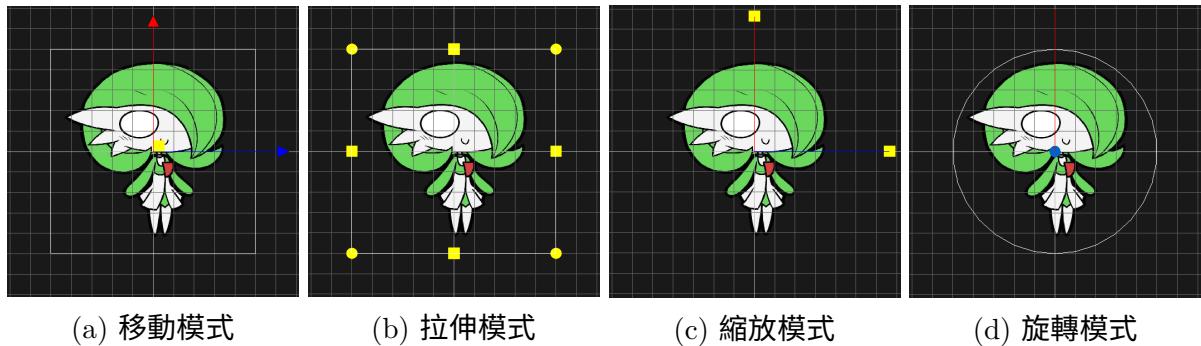


Figure 4.19: Gizmo 的不同模式

遊戲視窗 Game View

在測試期間將會顯示標示為 primary 的 CameraComponent 的視角並運行各個 Component 的動作

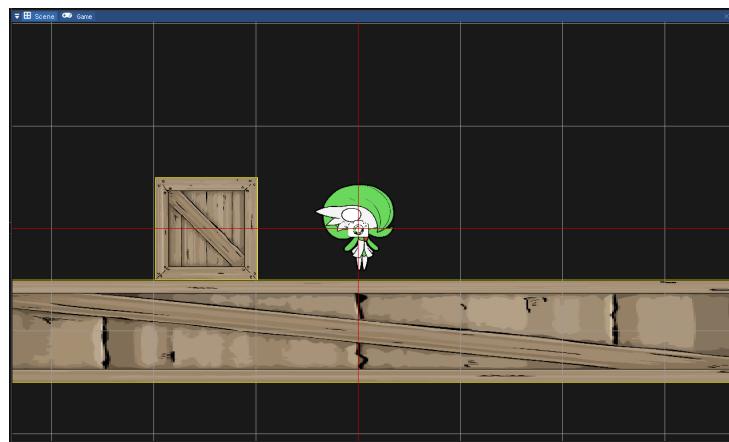


Figure 4.20: Game View

除錯訊息視窗 Log

- 顯示開發人員輸出或引擎輸出之訊息，並將其分級顯示
 - 依照錯誤級別分成: TRACE, INFO, WARN, ERROR, CRITICAL
- 可對 Log 進行過濾，方便遊戲開發者在眾多訊息中定位

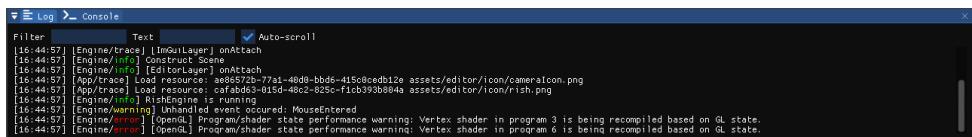


Figure 4.21: 除錯訊息視窗 Log

訊息視窗 Status Bar

顯示目前 Editor 的狀態

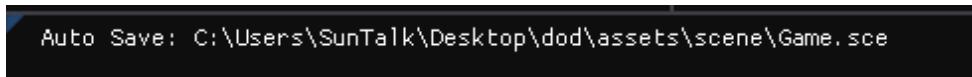


Figure 4.22: 訊息視窗 Status Bar

4.2.2 架構與未來展望

架構

- Editor
 - 在 Editor 底下利用 EditorController 管理著 Scene, Panel, Action
 - Scene 為遊戲世界的畫面管理
 - Panel 為 Editor 的介面管理
 - Action 為 Editor 中的各式操作
- EditorController
 - Grid
 - Event
 - 接收滑鼠與鍵盤的操作
 - Gizmo
 - 移動模式
 - 根據滑鼠點擊的位置來決定 Entity 移動時的權重
 - 根據滑鼠拖拉的距離去乘上權重來改變 Entity 的位置
 - 拉伸模式
 - 根據滑鼠點擊的位置來決定 Entity 移動時的權重與其大小的權重
 - 根據滑鼠拖拉的距離去分別乘上各自的權重來改變其位置與大小
 - 縮放模式
 - 根據滑鼠點擊的位置來決定要改變 Entity 的長或寬
 - 根據滑鼠拖拉的向量與要改變的 Entity 方向去計算其投影向量

- 根據該投影向量來計算需增加的長或寬
- 旋轉模式
 - 根據滑鼠拖拉時的起始點與最終點分別與 Entity 的中心點作為向量
 - 計算二向量的夾角來改變 Entity 的 rotate
- Scene
 - currentScene
 - editorScene
 - 在 Editor 模式下所使用的 Scene
 - runtimeScene
 - 在遊戲模式下所使用的 Scene
- Panel
 - mainPanel 主要顯示的視窗
 - MenuBar
 - 管理 simplePanel 以及與 Editor 相關之 API
 - Hierarchy
 - 從 currentScene 中拿到其所有的 Entity 並顯示其 tag
 - 標記部分 Entity 為選取狀態並對其進行操作
 - ComponentEditor
 - 對於所選取之 Entity 的 Component 進行操作
 - 使用各個 Component 的 API 來進行操作
 - 選取視窗顯示所有可新增的 Component，點選後將其新增到 Entity 上
 - StatusBar
 - 接收 EditorController 發送的訊息並顯示
 - simplePanel 簡易的彈跳式視窗
 - SettingPanel
 - 讀取並修改 Editor 的設定檔
 - HelpPanel
 - AboutPanel
- Action
 - shortCut
 - 設定各個快捷鍵的操控

- setting
- 設定 Editor
- autoSave
- 每過一段時間自動將 Editor 的 Scene 儲存起來

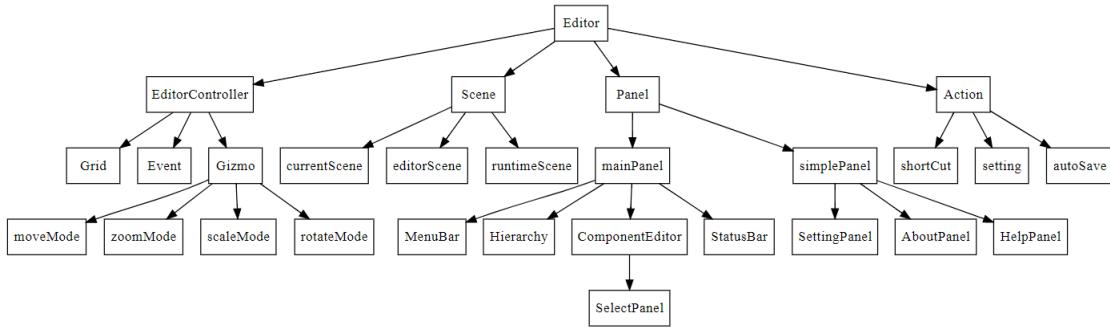


Figure 4.23: Editor 架構

未來展望

- Undo & Redo System
 - 讓使用者可以回復其所做的上一步操作
- 支援 3D 顯示與操作
 - 在引擎的各項功能提供 3D 操作後，讓 Editor 亦可供其顯示與操作

CHAPTER 4. 系統實作結果

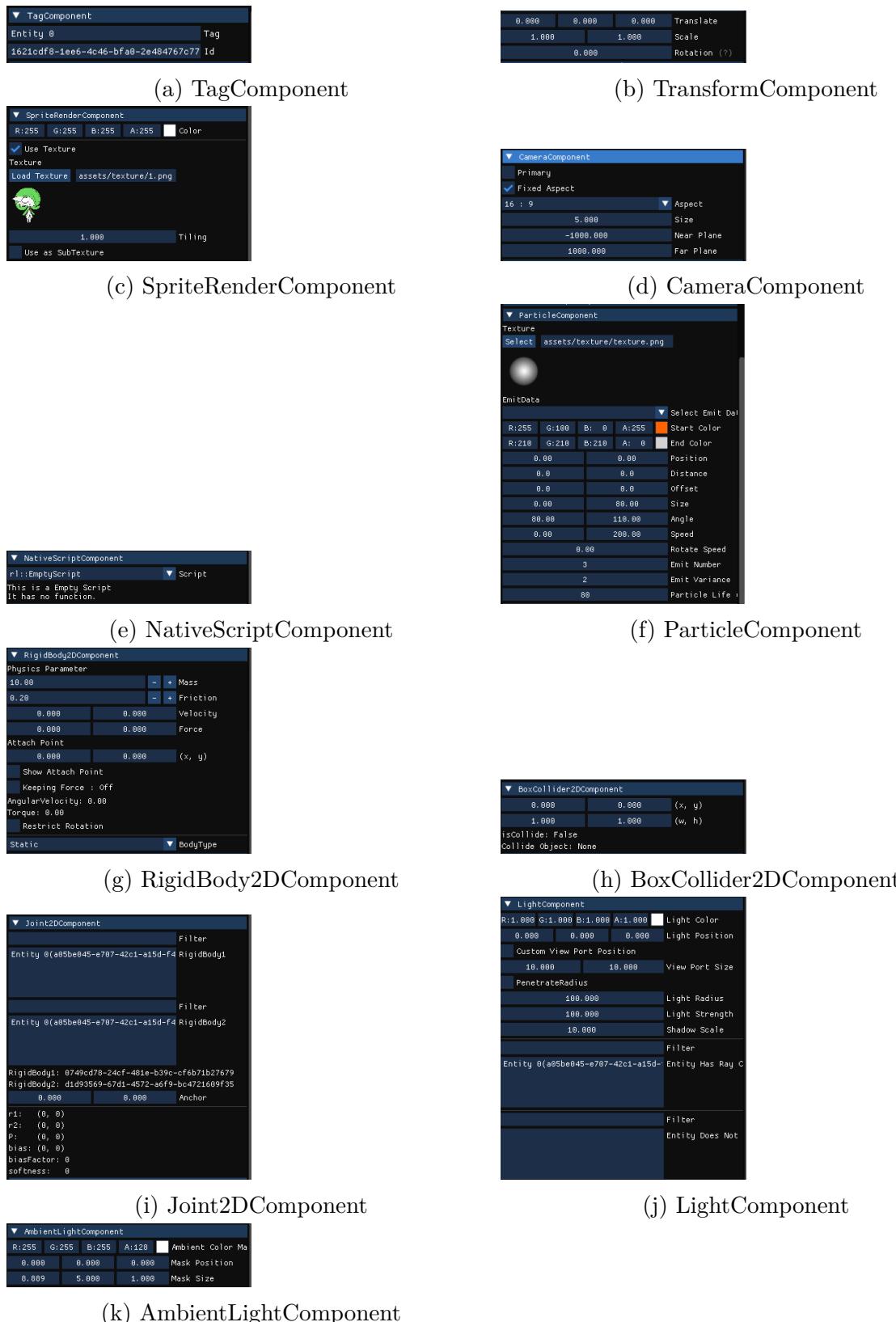


Figure 4.24: 引擎實體檢視介面

Chapter 5

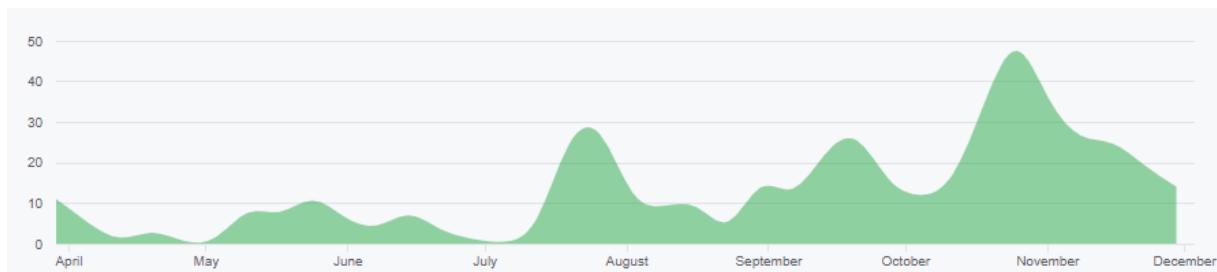
結語

5.1 心得

5.1.1 鍾秉桓 Roy

其實從高中初次接觸程式設計以來，我一直想做的事情便是寫一個自己的遊戲引擎，從高中時初次嘗試撰寫 ADV 類型的遊戲引擎（但受限當時的程式能力，並沒有完成），而到了大學時，我從很早（大概大二）便找好組員，並且也在系上課程的多個專案中不斷地練習和磨合我們之間的合作，我想做的事便是寫一個屬於自己的遊戲引擎。

從三年級下學期開始，經過了九個月的開發，坐在電腦前數百小時的奮鬥，我們 RISH 終於將 RishEngine 的功能告一個段落，儘管有些功能是因為專案時程以及目前能力而有缺憾，但即便如此我們還是完成了這個專案，我們開發了一款 2D 的遊戲引擎，具有 ECS、有引擎編輯器、有 Batch Rendering、有 2D Lighting、有物理引擎、有 Particle System 等。



RishEngine Commit Graph

而在開發每項功能前，都花了一到兩個星期在做資料搜尋、自學，接著做出最小可行的 Demo，僅僅做出功能還不夠，因為我們做得是遊戲引擎，所以還得將 API 打磨，不斷得去想、擴充功能，因為引擎就是得提供許多功能給遊戲開發者，而在引擎開發階段時，每個功能寫出來之後，還得切換身分到遊戲開發者，試圖用自己做出的引擎來打造遊戲，透過這個過程來除錯和發想可以增加的新功能，一直不斷的迭代，而這個過程是相當漫長和痛苦的，常常在和組員討論時一個功能的規格 (spec) 時，往往最後都覺得很迷惘，總是覺得這個功能不夠好，但還是得在現實與理想中取捨。

在這次的專案中我也學到了很多東西，像是如何有效與人溝通和合作，表達自己的想法給對方清楚理解，當然過程中勢必有磨合的時期，因為我們是只有四個人的小組，所以作為組長的我理所當然會處理許多工作，也必須擔當起管理的責任，畢竟繁雜的工作量由一個人扛起也過於不切實際，清楚了解到除了程式之外管理和與人相處溝通之道也相當重要。

這個畢業專題算是完成了我其中的一個心願，自幹一個遊戲引擎，也是我目前處理過最多行的程式，算是我的一個小小的里程碑。

5.1.2 梁博全 ICEJJ

會開始接觸程式，是因為自己很喜歡玩遊戲，而自己也想開發遊戲，所以想當個遊戲開發者，但隨著深入學習這塊領域，發現市面上遊戲引擎已經很成熟，開發遊戲門檻降低，很多人都能用熱門引擎寫出遊戲。作為開發者，我們能用現有引擎輕鬆地拉出一款遊戲，我們從中學習到的僅僅只是引擎的使用方法。若未來需要對引擎底層進行優化，但對底層的架構、邏輯都不了解，那要如何修改如何優化。開發遊戲引擎不只能讓我們更了解遊戲架構，也能更加瞭解一款遊戲的圖形渲染、物理模擬等等是如何運作，我想這會是在眾多遊戲開發者內脫穎而出的關鍵。

在開發引擎功能的，基本上都是從無開始自學，每當被分派到一個模塊時，基本上就是花大量的時間找資料，接著花大量的時間寫 Demo，寫 Demo 的期間還會遇到各種不同的問題，而做完因為是需要將模塊打入引擎裡面，因此還得將 API 想好，整個架構也得配合著引擎，最後在移植到引擎裡面之後，新的問題又會出現。

這次的專案除了深入了解整個遊戲架構、更加熟悉 C++，也知道溝通的重要性。初期常常因為溝通不清楚而導致東西做出來不合預期，因此在被分發工作時，將內容以及目標講清楚，中途遇到問題發現功能無法做出來或是需要更改 API 也需要及時提出，避免最後浪費太多時間。

雖然引擎與一開始預想的有差，但能做出這些算是不錯了，自己也很開心。

5.1.3 黃育皓 SunTalk

在剛開始接觸程式時，最直覺的就是想寫遊戲，想將腦中所想的變成一款真實的遊戲，並為此打磨自己的能力。在學習的過程中，試著寫了幾個遊戲後，發覺自己所寫出來的與市面上所存在的遊戲都有很大的差異，才發現說現如今已經不是純粹靠著自己 coding 來寫遊戲了，大多數的遊戲開發者都是使用已經打磨好的遊戲引擎來進行開發，藉由完善的遊戲引擎來解決繁瑣的底層架構，大大的提升了遊戲開發的效率與效能。也因此讓我們從想寫個遊戲轉變成想試著寫個遊戲引擎出來。

在開發的過程中，由於每個人所負責的模塊都不同，我覺得最主要的是要清楚的理解引擎中的架構，不僅僅是整體的架構，其內每個模塊的架構都要有一定程度的理解，在這的基礎上去進行編輯器的撰寫與整理，在編輯器中除了設計出讓使用者方便使用的各種功能，亦要去正確的對照到引擎中的各個功能才能使得其正常運作。

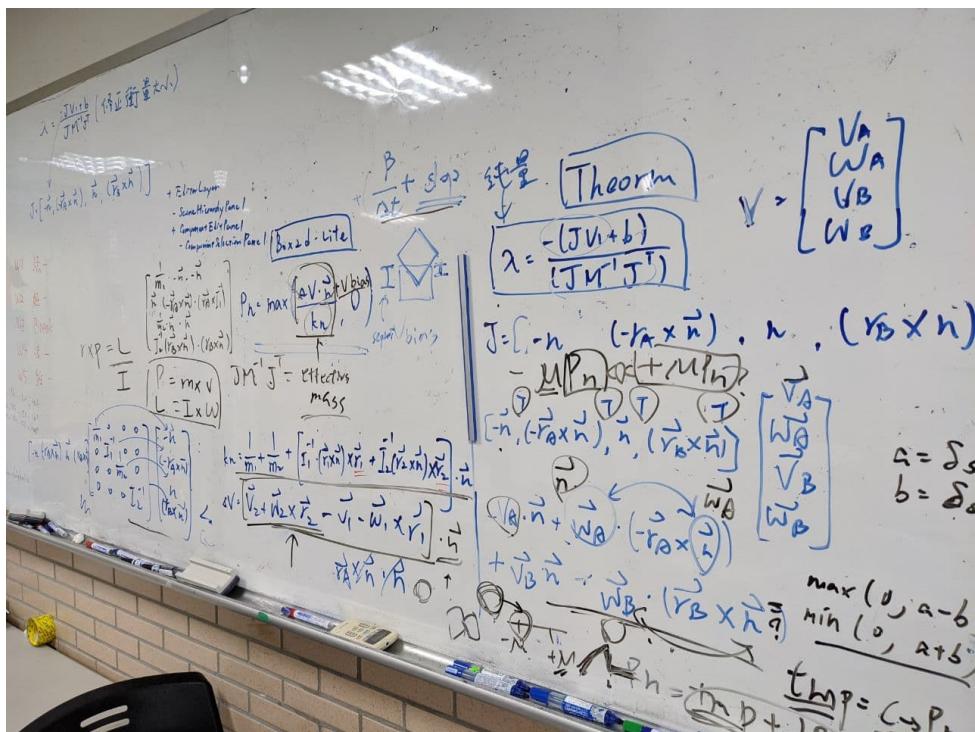
在這次的專案中，我學習到了許多的事情，除了瞭解引擎與遊戲上的架構外，最重要的就是與隊友中的溝通了。像是在理解對方所寫出來的 code 時，只是看著程式碼並不能有效的理解到該區塊的意義，以及如何去使用，在經過有效的溝通後才可以快速的去諧調出 API 與協助 debug 的進行。

能夠以遊戲引擎這一個主題來作為畢業專題，並且做出這摸一個雛形，雖然還有很多可以新增與進步的空間，但也讓我感到相當的開心。

5.1.4 黃品翰 Halloworld

在整個遊戲開發的歷程，我們從一開始試著用現有的遊戲框架 (SFML)，每個人都試著實作一個簡單的遊戲。透過寫遊戲的這段過程，來構思我們的遊戲引擎該如何設計。當初想要寫遊戲引擎，一方面我自己覺得是前人沒有試著這樣做過，另一個方面是想說透過畢業專題的機會，詳細了解遊戲引擎的架構與內部的實作。我們也在網路上找了許多資源，同時也參考現有的遊戲引擎做為參考對象，想辦法設計出一套可以拿來寫遊戲的遊戲引擎。

在物理引擎的學習上，較為細節的研究歷程我都放在報告裡面，心得這邊我就純粹只分享我自己的感想。研究遊戲物理這塊，其實真的很少人在做，中文文獻也是少得可憐，大部分人以及其他遊戲相關課程，通常也只注重在使用函式庫。在最一開始，我只拿到一份真正成功將 2D 物理引擎實作的範例，但作者寫的註解也是少得可憐，於是我不花了不少個禮拜，在研究他的程式碼在寫些甚麼。研究的路途中，我後來發現在研究的路上並不是只有我一個人，感謝有來我個人網頁的來自中國大陸的網友，趁機的與他交換聯絡資訊，跟他來回書信不斷地順順看自己的邏輯，互相腦力激盪出當時撰寫這份物理引擎的想法。另外，我參加了學生計算機年會 (SITCON)，很高興我花我暑假的期間，參加了用遊戲物理自幹一條繩子的議程，議程中的剛好可以跟我學的內容呼應，同時也很厚臉皮的，跟他交換了聯絡方式。我想最值得的，就是這一步步都在突破我自己的舒適圈，透過各種不同的方法找尋資源，想辦法把自己心中的那個為甚麼——地解開。



被我寫滿物理證明的實驗室白板

除了與網路上無緣無故交到的筆友和議程上的講者，我也透過講者的關係下，拿到遊

戲物理界大老的系列教學影片。那短短兩三個小時幾部的影片讓我真正了解到比較深層的理論。如果以程式語言的學習歷程來比喻，在前期自己看程式以及與網路筆友的討論，像是學習高階語言一樣，大概能知道淺層的物理原理，但是再透過那系列的教學影片之後，又知道了有些東西的係數是這樣推倒而來，就如同學習到組合語言一樣，能跟精確地知道整個程式運作的來龍去脈。除此之外，我也試著自己證明遊戲物理引擎的物理參數，看看真的是不是如程式寫得一樣，並自己在試著實作一次。此外，覺得我自己還想更精進的部分，是希望以後能夠自己實作出 3D 的遊戲引擎，以及實作空氣阻力、彈簧... 等更精細的物理模擬。

實作的過程中，很常被 C++ 的語法及語言特性坑殺。舉個最經典的例子，最常遇到的應該就是循環引入的問題，編譯器卻沒告訴你哪裡出了問題，導致我花了很多時間來找 bug。我自認為我在實作遊戲物理邏輯中的部分，跟解決 C++ 錯誤所花的時間比起來，修 C++ BUG 的時間真的佔了我大部分的時間。不過也從中學習到很多技巧。這些東西可能是課堂學不到的，大部分都得靠網路或是同學之間討論，才能夠把這些複雜又惱人的 BUG 解掉。此外，在整個專案當中，隨著時間增長的程式碼，漸漸地讓編譯速度變得相當緩慢，裡面的 Warning 也越來越多，這也是很頭痛的部分。到專案後期，常常花的時間都不是在改程式碼，而是在等編譯。包括我現在寫心得的此時此刻，都是在等待的期間撰寫出來的。

另外，與團隊的交流也很重要，過程中有相當多的磨合以及很多的取捨，不過目的其實都是為了整個專案更好。比較愧疚的部分，是這次沒有參與到遊戲引擎整個開發的部分，引擎內部裡面的細節我沒有仔細的接觸過。主要因為光花那些時間證明物理以及實作就踩了許多坑也耗費了很多時間和冤枉路，加上我自己的資質鴦鈍和我的英文能力，以及，真正研究過遊戲物理的人真的很少，資源也相對地難找，也就導致我沒有參與到整個引擎最主要的開發部分。最後，我想感謝我的組員們，雖然引擎不是最完美的，但我相信從開發過程應該都學到不少的東西以及經驗，也很感謝實驗室的學長和同學，陪我們度過一次又一次的 Bug。



Bibliography

- [1] Scott Pakin. *The comprehensive LaTeX symbol list*. 2020. URL: <http://tug.ctan.org/info/symbols/comprehensive/symbols-a4.pdf>.
- [2] Nir Benty, Kai-Hwa Yao, Petrik Clarberg, Lucy Chen, Simon Kallweit, Tim Foley, Matthew Oakes, Conor Lavelle, and Chris Wyman. *The Falcor Rendering Framework*. <https://github.com/NVIDIAGameWorks/Falcor>. Aug. 2020. URL: <https://github.com/NVIDIAGameWorks/Falcor>.
- [3] Matthias Wloka. *"Batch, Batch, Batch": What Does It Really Mean?* 2003. URL: <https://www.nvidia.com/docs/I0/8230/BatchBatchBatch.pdf>.
- [4] Valmond. *Why is C++ used for game engines? How about its future in game engines?* <https://gamedev.stackexchange.com/questions/38011/why-is-c-used-for-game-engines-how-about-its-future-in-game-engines>. Oct. 2012 (page 7).