Ⅰ. 게임제작 기초 지식

**게임엔진**은 그래픽 출력 렌더링, 물리 엔진, 사운드, 애니메이션, 등 게임에 필요한 요소가 미리 갖춰져 있는 소프트웨어입니다. 대표적으로 Unity, Unreal, Cocos2dx가 있습니다. Unity는 다양한 플랫폼과 2D, 3D를 가리지 않고 범용적으로 많이 사용되고 있는 게임 엔진입니다.

게임을 만들 때 가장 중요한 것은 당연히 기획입니다. 그러나 프로그래밍 파트인 우리는 뛰어난 기획을 실제로 구현할 수 있는 능력을 필요로 합니다. 실제 구현은 크게 3가지 구성요소를 가집니다. 리소스(아트, 사운드 등), 리소스가 어떻게 동작하는지 명시하는 코드, 실제 리소스 배치. 해당 문서에서는 2번째를 중심적으로 다루게 됩니다.

시작하기에 앞서 잊지 말아야 할 점은평생 1인 개발만 할 게 아니라면 **협업**은 게임개발에서 필수불가결합니다. 따라서 승급조건에 팀프로젝트가 있으니 참고해주세요.

Ⅱ. 코드 작성 – C# 기초

**1. 유니티 엔진이 사용하는 언어**

유니티는 내부적으로는 C++로 구성되어 있지만 사용자 스크립트는 다른 설정을 하지 않는 이상C#으로 작성하게 됩니다.

**2. C#의 컴파일 방식과 배경**

C#은 C++의 확장형이 아닌 다른 종류의 언어가 됐습니다. 그리고 그 모체가 되는 언어는 JAVA입니다. C와 C++은 전처리, 컴파일, 링킹을 거쳐 실행 파일을 만들어야 했습니다. 이런 컴파일 방식은 OS마다 이진 파일의 해석이 다르므로 같은 코드라도 OS가 다르면 다른 실행 파일이 나온다는 점이 문제였습니다.

JAVA와 C#은 이런 고전적인 컴파일 방식을 버리고 **JIT**(Just-in-time)라는 컴파일 방식을 사용합니다. JIT방식은 우선 소스코드들을 ‘클래스 단위’로 바이너리 파일(.dll)로 컴파일합니다. 이를 C#에서는 **CIL**(Common Intermediate Language)라고 합니다. 이렇게 만들어진 파일은 **가상머신**이 필요한 라이브러리만 링킹을 동적으로 하게 됩니다. 따라서 처리 속도는 비록 떨어졌지만 컴파일의 결과가 OS에 무관하게 동일하며, 메모리 관리도 효율적으로 변했습니다. 이 가상머신을 C#에서는 **CLR**(Common Language Runtime)이라고 합니다. 한편, C++의 STL과 같이 C#도 라이브러리를 제공해주는데, 이는 FCL(Framework Class Library)라고 하며, CLR과 FCL을 통틀어서 **.NET Framework**(닷넷 프레임워크)라고 합니다.

C#을 개발한 마이크로소프트는 CLR을 윈도우 플랫폼에서만 작동되도록 만들었습니다. 이에 불만을 가진 자마린이란 기업이 C#을 다양한 플랫폼에서도 이용할 수 있도록 오픈소스 프로젝트를 진행됐었는데 이를 **Mono**라고 합니다. 모노 덕분에 윈도우뿐만 아니라 리눅스, 맥 등에서도 C#이 작동되게 되었습니다. 유니티는 이런 모노 닷넷을 기반으로 채택한 이후로 다양한 플랫폼 호환성을 갖춘 이후부터 전성기를 맞이하게 됩니다. 하지만 모노가 오픈소스인 탓에 유니티로 만든 게임은 전부 보안에 취약했습니다. 이 때문에 보안을 강화시킨 IL2CPP라는 빌드방법이 등장하기도 했습니다.

**3. 포인터를 쓰지 않는 코딩**

C#은 포인터를 위험한 자료형으로 정의하고 기본적으로 이를 쓰지 못하도록 막았습니다. 문법이 다음과 같이 변하여 포인터로부터 자유로워졌습니다.

1. 가장 기본단위는 클래스로, 모든 함수와 변수는 클래스 내부에 존재해야 합니다. 정확히는 바이너리 파일이 만들어지는 단위가 클래스입니다. 즉, 모든 내용은 클래스 내부에 작성되어야 합니다.

2. 클래스의 객체 자료형은 별도의 표기 없이 레퍼런스 자료형으로 취급됩니다.

3. new를 한 객체는 delete 해줄 필요 없이 가상머신이 필요할 때 메모리를 정리합니다.

4. this 포인터는 this 레퍼런스로 변경됐습니다.

**4. 가비지 컬렉션**

가비지 컬렉션(GC, Garbage Collection)는 위에서 언급한 스스로 판단하여 delete 처리를 하는 기능을 의미하고 가상머신의 기능 중 하나입니다. 메모리 해제에 대한 모든 권한을 가상머신이 가지기 때문에 메모리 삭제에 대해서는 자유가 뺏기게 됩니다. 필요하다고 호출할 수는 있지만 언제 호출되는지 알 수 없습니다.

이런 가비지 컬렉션은 메모리 해제와 관련된 문제를 고민할 필요가 없다는 장점을 가지지만, 언제 메모리 해제가 일어날 지 모르기 때문에 사용한 메모리를 다시 사용해야 할 때 NULL체크에 주의해야 합니다. 또한 GC가 처리 중일 때는 프로그램이 일시적으로 중단되기 때문에 많은 힙메모리가 쌓여 있다면 예상치 못한 순간에 프로그램이 중단될 수 있습니다. 이는 게임할 때 흔히 겪는 프레임 드롭 현상의 원인 중 하나입니다.

**5. 모든 클래스의 부모 클래스 Object**

어떤 클래스를 만들더라도 해당 클래스는 명시하지 않아도 Object 클래스의 자식 클래스가 됩니다. 이 덕분에 업/다운 캐스팅으로 자료형 변환이 자유로워졌습니다. 이를 이용한 것은 이후 소개할 컬렉션이 있습니다. 또한 클래스가 공통적으로 가져야 하는 특성을 Object가 가지고 있기 때문에 클래스 관리의 편리성이 높아졌습니다.

대표적인 예시로는 string클래스가 있습니다. string 클래스의 객체가 같다는 것은 객체의 레퍼런스가 같다는 것이 아닌 문자열이 같다는 것을 의미할 것입니다. 따라서 String 클래스에서 Object 클래스의 Euals() 함수를 오버라이딩하여 문자열값이 일치한다면 같은 객체로 판단합니다.

한편, Object를 상속한 모든 객체는 해시코드라는 것을 가집니다. 이는 해시 테이블을 기반으로 하는 컬렉션에서 key로 사용될 수 있는 값으로 실제 컬렉션 사용 시 변경될 수도 있는 값입니다. 해시테이블의 특성 상, 다른 객체라도 같은 해시코드를 가질 수도 있으며 때에 따라 key의 위치가 변할 수도 있기 때문에 GetHashCode()를 이용해서 객체가 같은지 판단하는 것은 피해야 합니다. 많이 사용하는 경우는 아니지만, 사용자 정의 클래스에서 string클래스의 Euals() 경우처럼 다른 객체라도 같은 것으로 판단해야 할 땐 오버라이딩을 해야 합니다.

**6. C#의 구조체와 클래스는 다르다**

C++에서는 구조체와 클래스의 개념 구분이 모호했습니다. 그러나 C#에서는 구조체와 클래스는 엄연히 다른 특징을 가집니다. **구조체**는 기본적으로 **값형태**로 데이터가 저장됩니다. 그러나 **클래스**는 **참조형태**로 데이터가 저장됩니다.

첫째로, 함수의 매개변수로 활용할 때 큰 차이를 보입니다. 구조체의 내용을 직접 바꾸고 싶을 때는 ref 등 매개변수를 참조형태로 준다는 것을 명시해야 하지만 클래스는 별도의 표기가 없어도 참조형태로 주게 됩니다.

둘째로, 변수 자체를 사용할 때 큰 차이를 보입니다. 구조체의 경우 a = b;를 하면 값복사가 일어나서 a, b는 별도의 것으로 취급되지만 클래스의 경우 a = b;를 하면 참조복사가 일어나서 a와 b는 같은 객체를 가리키게 됩니다.

**7. 제네릭(Generic)**

제네릭은 C++의 템플릿과 유사한 기능입니다. 그러나 제네릭은 템플릿과는 다르게 런타임 시 자료형이 결정됩니다. 특징으로는 자료형이 값형식일 때는 내부적으로 제네릭의 형식을 새로 만들지만 참조형식일 때는 1개만 만들게 됩니다. 예를 들어 Stack<int>, Stack<float>는 각각의 제네릭 형태가 따로 만들어지지만 Stack<Transform>, Stack<Object> 등 클래스를 매개변수로 하면 참조형식이므로 1개의 제네릭 형태만 만들어집니다. 즉, 내부적으로 코드가 간결해집니다.

**8. 제네릭 컬렉션(Generic Collection)**

제네릭 컬렉션은 C++의 STL과 유사한 기능입니다. 매개변수로 Object를 이용하는 컬렉션도 존재하지만 후술할 박싱/언박싱 문제 때문에 현재는 사용하지 않는 게 권장됩니다. 기본 컬렉션은 호환성 문제로 남아있는 라이브러리기 때문에 꼭 제네릭 컬렉션을 사용해주세요.

List<T>: 동적배열 기반 리스트 (ArrayList는 컬렉션)

LinkedList<T>: 이중연결리스트 기반 연결리스트

SortedList<K, V>: **순서배열** 기반 정렬리스트 (주의: 이진탐색트리X)

Stack<T>: 동적배열 기반 스택

Queue<T>: 동적배열 기반 큐

Dictionary<K, V>: 해시테이블 기반 사전 (HashTable은 컬렉션)

SortedDictionary<K, V>: 레드블랙트리 기반 사전

HashSet<T>: 해시테이블 기반 집합

SortedSet<T>: 레드블랙트리 기반 집합

**9. 박싱과 언박싱**

값형식을 참조형식으로 변환하는 것을 박싱(boxing), 참조형식을 값형식으로 변환하는 것을 언박싱(unboxing)이라고 합니다. 이때, 그냥 참조변수를 사용하는 것이 아닌 new를 통해 새로운 메모리를 할당하게 됩니다. 대부분 박싱과 언박싱은 묵시적으로 일어나기 때문에 원치 않게 힙메모리가 증가하는 원인이 될 수 있습니다. GC의 존재로 바로 삭제가 불가능하기에 박싱과 언박싱이 일어나는 경우가 어떨 때인지 판단하는 것은 중요합니다.

예를 들어 함수 중에 Object가 모든 클래스의 부모 클래스란 점을 이용해서 이를 매개변수로 받는 함수에 int, float 등의 기본자료형 또는 구조체를 넘겨줄 때 문제가 됩니다. 따라서 매개변수로 Object로 받는 함수를 쓸 땐 구조체나 기본 자료형을 쓴다면 다른 함수를 고려하거나 개선 방법이 없는지 모색해야 합니다.

\*참고로 C#에서는 int, float 등의 기본자료형들도 사실은 Object 클래스를 상속받는 구조체입니다.

**10. 프로퍼티(property)**

C++에서는 클래스의 private 접근지정자를 변경하기 위해서 getter, setter 함수를 작성해야 했습니다. C#에서는 프로퍼티라는 문법이 생겨서 get, set을 대신하게 됩니다. 일반 변수와 구분하기 위해 함수처럼 파스칼 표기를 하는 게 일반적이고 실제로 작동도 함수이므로 오버헤드가 발생합니다.

public string Name { get; set; } = string.Empty; //초기화한 변수와 동일한 역할

private string name; public string Name { get=>name; set=>name=value;} //직접구현

public string Name { get; private set; } //클래스 내부에서만 접근 가능

**11. 새로운 접근 지정자**

새로운 접근 지정자 internal이 생겼습니다. 역할은 자바의 default와 똑같습니다. 동일한 패키지(어셈블리파일)라면 접근 가능한 지정자입니다. 유니티의 스크립트는 전부 동일 패키지지만 유니티 특성 때문에 종종 쓰기도 합니다.

**12. 클래스의 상속 범위 변경**

자바처럼 클래스는 오직 1개만 상속받을 수 있게 변경됐습니다. 하지만 인터페이스는 여러 개 상속이 가능합니다.

**13. 가상함수와 추상함수**

C++에서는 아무 함수나 오버라이드 가능했지만 C#에서는 virtual 또는 abstract 예약어를 쓰는 함수만이 오버라이드 가능합니다. 만약 클래스 내부에 abstract를 쓰는 함수가 있다면 해당 클래스는 abstract 키워드를 붙여줘야 하며 추상클래스로 판별됩니다.

**14. 오버라이드 문법**

명시적으로 오버라이드된 함수임을 나타내기 위해 override 라는 문법이 생겼습니다. 해당 예약어를 쓰지 않으면 오버라이드 할 수 없습니다.

**15. sealed**

자바에서 final을 쓴 클래스와 동일한 역할로, 해당 예약어를 쓴 클래스는 상속이 불가능합니다.

**16. base**

base는 부모 접근자로, 자바의 super와 유사합니다. 부모 클래스의 함수를 그대로 쓸 때 사용합니다.

**17. 인터페이스 문법**

명시적으로 인터페이스임을 나타내기 위해 interface라는 문법이 생겼습니다. 해당 예약어를 쓰지 않으면 인터페이스로 만들 수 없습니다.

**18. 문자열 표현**

C#의 문자열은 {index} 로 접근 가능하고 $문자열을 쓰면 {변수명}으로도 접근 가능합니다. 또한 @문자열을 사용하면 큰따옴표를 제외한 이스케이프 시퀀스를 무시할 수 있어 더 보기 편리한 문자열을 쓸 수도 있습니다.

Console.WriteLine(“{0}은 {1}이다.”, name1, name2); //기본 문자열

Console.WriteLine($“{name1}은 {name2}이다.”); //$문자열

Console.WriteLine(@”C:\documents\files\tmp.txt”); //@문자열

**19. 무명 메소드(Anonymous Methods)**

C#에서는 C에서 쓰이는 매크로 함수나 C++에서 쓰이는 인라인 함수가 존재하지 않습니다. 그 대신에 쓰는 문법으로, 함수가 너무 간단하고 자주 쓸 일이 없을 때 사용합니다.

**20. 람다식(Lambda Expressions)**

무명 메소드를 보다 쉽게 쓰기 위해 만들어진 문법입니다. 보통 무명 메소드를 쓸 일이 생긴다면 람다식으로 대체합니다.

(매개변수1, 매개변수 2, …) => 리턴값 //람다식(Expression lambda)

(매개변수 1, 매개변수 2, …) => { 표현 } //람다문(Statement lambda)

**21. 클로저(Closure)**

C#에서의 클로저는 무명 메소드나 람다식이 매개변수 이외의 변수를 다룰 때를 의미합니다. 클로저는 매개변수 이외의 변수를 다루기 위해서 임의의 클래스를 만들고 그곳에 변수를 저장하게 됩니다. 그리고 클로저가 호출될 때 이 클래스가 객체화됩니다. 이때, 해당 클래스는 임시 데이터로, 변수로 가리킬 방법이 없지만 힙메모리에 공간이 할당됩니다. 따라서 GC의 대상이 되며 클로저를 자주 호출할수록 메모리 부하가 있으므로 신중히 사용해야 합니다.

<https://nickname.tistory.com/34>

**22. var**

C++의 auto와 같이 타입추론 변수 지정자인 var 이 생겼습니다. 형식은 컴파일 시 판단됩니다. 보통 타입을 명시해주는 게 가독성이 좋기 때문에 단순 반복할 때, 타입이 복잡할 때, 쿼리할 때 사용합니다.

**23. foreach**

foreach는 사용이 편리하고 의미를 명확하게 한다는 장점이 있습니다. 그러나 for에 비해 더 느리기 때문에 의미의 명확성과 속도 사이에 trade-off를 해야 합니다. 하지만 대부분의 경우 미미한 차이기 때문에 foreach를 적절히 쓰는 게 좋습니다.

**24. readonly**

const는 C커리큘럼에서 배웠다시피 컴파일 시 메모리의 유저 영역 중 데이터 부분에 저장됩니다. 즉, 복잡한 자료형을 const로 지정할 시 바이트코드가 늘어나게 됩니다. readonly는 런타임 시 값이 고정되기 때문에 const의 역할을 해야 하는 클래스나 구조체 등의 복잡한 자료형은 readonly로 만드는 것이 보편적입니다.

**25. 어트리뷰트(attribute)**

클래스에 메타데이터를 추가할 수 있도록 하는 문법입니다. 내부적으로 리플렉션을 사용합니다. 유니티에서 자주 쓰이는 어트리뷰트가 따로 존재하고 이는 후에 소개하겠습니다.

--------------------------------------------------------------------------------------------------------------

\*이후 문법은 추가 자료로 넘어가셔도 됩니다.

**26. 리플렉션(Reflection)**

리플렉션은 런타임 시 객체를 통해 클래스의 정보를 분석하는 문법입니다. 클래스 내부의 지정자가 private이더라도, 그 어떤 데이터도 가져올 수 있습니다. 보통은 어떤 데이터가 존재하는지 검사할 때 사용합니다. 실제 게임 작동 코드 내부에는 쓸 일이 없고 게임엔진 커스터마이징할 때 사용하곤 합니다.

**27. 대리자(Delegate)**

C언에서의 함수 포인터의 확장 문법이라고 봐도 됩니다. 함수를 변수에 담을 수 있으며, +=, -= 연산자를 통해 함수의 개수를 추가하거나 빼서 대리자를 실행시키면 추가된 함수들이 차례로 처리되게 됩니다.

**28. 이벤트(Event)**

대리자와 역할은 똑같지만 다른 점은 대리자와 달리 객체의 상태변화나 사건의 발생을 알리는 용도로 쓰이기 때문에 인터페이스 내부에 선언이 불가능합니다. 또한 public으로 설정되어 있어도 자신 외부에서는 호출이 불가능합니다.

**29. 참조형식 예약어**

총 3가지로 ref, out, in이 있습니다. ref는 C++의 레퍼런스와 동일합니다. out은 매개변수에만 쓸 수 있는 예약어로 특별하게 변수를 생성함과 동시에 매개변수를 넘겨줄 수 있습니다. 참조형식으로 보냄과 동시에 함수 내에서 무조건 값이 바뀐다는 것을 명시합니다. 유니티에선 주로 Raycast 함수를 쓸 때 사용됩니다. in은 인수가 참조로 전달되고 변수를 생성함과 동시에 매개변수를 넘겨줄 수 있지만 out과는 정반대로 절대로 값이 바뀌지 않는다는 것을 명시합니다.

**30. LINQ(Language-Integrated Query)**

C#에서는 LISP라는 프로그래밍 언어를 기반으로한 LINQ라는 쿼리를 제공합니다. 쿼리란 데이터에서 특정 데이터를 검색하는 것입니다. 문법이 사람이 이해하기 쉽게 되어 편리하고 코드의 길이도 줄여주는 좋은 기능이지만 성능은 일반 구현보다 느리다는 단점이 있습니다.

1. 데이터 소스 가져오기 (**from** num **in** numbers) //num는 변수, numbers는 대상
2. 쿼리 만들기 (**where** (num % 2) == 0) //가져올 데이터 조건
3. 쿼리 실행 (**select** num) //결과값의 원소

반환값은 반복기인 IEnumerable<T>이므로 ToList() 또는 ToArray()로 바꿔 이용하기도 합니다. **orderby** (value) **ascending**을 이용하면 오름차순, **desending**이면 내림차순정렬할 수 있습니다.

\* 튜플은 C# 버전에 따라 사용 불가능할 수 있어서 생략하겠습니다.  
\* nullable, is, as 등의 새로운 자료형 및 연산자 또한 C# 버전에 따라 사용 불가능해 생략합니다.

Ⅲ. 코드 작성 - 유니티 기초

**1. double보단 float**

컴퓨터가 사용하는 소수는 기본적으로 부동소수점입니다. float와 double이 존재하는데 float는 4바이트, double은 8바이트를 사용합니다. 게임개발에서 미세한 소수점까지 고려해야 하는 정밀한 계산은 필요하지 않기 때문에 메모리 최적화를 위해 기본적으로 모든 소수는 float를 사용합니다. 이 점을 왜 주의해야하냐면 꼭 숫자에 f를 붙여야 하기 때문입니다.

new Vector(1.1, 2.2, 3.3); //1.1, 2.2, 3.3은 double로 해석. 내부적으로 float 형변환.

new Vector(1, 2, 3); //1, 2, 3은 int로 해석. 내부적으로 float 형변환.

new Vector(1f, 2f, 3f); //정상적으로 float값으로 매개변수가 전달.

**2. 스크립트 작성 기초와 라이프사이클(Life Cycle)**

앞으로 작성하는 모든 코드는 **Monobehaviour 클래스**를 상속받는 클래스를 작성하게 됩니다. 이 작성한 스크립트를 **게임오브젝트**에 부착하게 됩니다. 게임오브젝트는 게임을 실제로 구성하는 가장 기본 요소이고, 앞으로 작성할 클래스를 게임오브젝트에 부착함으로써 어떤 일을 할 지 명령을 내리게 됩니다. 실제 게임이 이뤄지는 곳은 **씬(Scene)**이며 씬은 여러 개의 게임 오브젝트로 이뤄집니다. 그리고 게임은 여러 개의 씬으로 완성되게 됩니다.

Monobehaviour 클래스에는 미리 내장된 함수들이 존재합니다. 그리고 일부 내장된 이벤트 함수들은 게임에서 일정한 순서를 가지고 실행되게 됩니다. 이 순서를 **라이프 사이클**이라고 합니다.

팁: Awake()에서는 Get, Find, new 등 값을 할당하는 초기화를 하고 Start()에서는 다른 클래스의 값이나 함수를 이용하는 등의 초기화를 진행하면 초기화 순서에 따른 오류 걱정이 사라집니다.

<https://docs.unity3d.com/kr/2019.4/Manual/ExecutionOrder.html>

**3. 코루틴(Coroutine) 함수**

스레드(Thread)는 한 프로그램에서 실행되는 흐름의 단위를 의미합니다. 즉, 여러 개의 스레드를 가진다면 한 번에 여러 개의 일을 처리합니다. 닷넷 프레임워크는 당연히 멀티 스레드를 지원하고 있지만 유니티 엔진은 사용자 스크립트에 한해 별도의 설정 없이는 단일 스레드 방식으로 작동하게 됩니다. 그 이유는 멀티 스레드 사용 시 데이터 충돌을 고려해야 해서 동기화라는 과정을 거쳐야 하고 이는 꽤 복잡하기 때문입니다. 하지만 게임 제작 시 동시성의 활용은 중요하기 때문에 유니티 엔진은 별도로 코루틴이라는 비동기 문법을 제공해줍니다.

함수는 크게 2가지로 나눌 수 있습니다. 프로그램이 시작하자마자 호출되는 함수인 메인루틴 (Main-routine), 그 외의 함수인 서브루틴(sub-routine). 일반적인 서브루틴은 메인루틴 실행 중 호출 시 서브루틴을 전부 실행하고, 다시 메인루틴으로 돌아온다는 특징이 있습니다. 그러나 코루틴은 조금 특별한 서브루틴입니다. 루틴 진행 도중 멈추고 다시 메인루틴으로 돌아올 수 있습니다. 즉, 메인 루틴과는 협력적인 관계라는 의미로 Co-routine이라는 이름을 가집니다.

함수의 반환값은 C# 기본 인터페이스인 IEnumerator로, 이는 반복을 도와주는 인터페이스입니다. 함수 내부에서는 return이 아닌 yield return을 사용하는데 해당 예약어를 만나면 메인루틴으로 돌아가고(양보해주고), yield return값에 따라 조건을 만족하면 다시 코루틴으로 돌아옵니다. 일반 함수와 달리 코루틴은 중단이 가능하기에 아직 끝까지 실행이 안 됐는데도 다시 호출이 될 수도 있다는 점을 주의해야 합니다. 이 경우 이전 진행상황은 사라지고 처음부터 다시 실행됩니다.

그런데 한편, 함수의 작동방식을 생각한다면 이런 의문이 들 겁니다. 함수를 실행하면 메모리가 스택에 쌓이고 함수가 끝나면 메모리가 전부 사라졌을텐데 어떻게 코루틴은 함수의 기존 정보를 저장할 수 있었을까요? 코루틴은 메인 스택과는 별도의 공간에 자신만의 스택공간을 가지게 됩니다. 스택을 메인스택과 같은 형태로 별도의 스택이 존재한다면 Stackful Coroutine이고 힙 메모리에 코루틴의 상태를 저장한다면 Stackless Coroutine입니다. 유니티는 후자의 경우를 사용합니다.

따라서 코루틴 호출 시 힙메모리가 할당되고 가비지 대상이 된다는 것을 인지해야 합니다. 코루틴을 규칙적으로 호출하는 경우, 코루틴 내부 반복으로 바꿀 수 없는지 생각해보는 것이 좋습니다.

코루틴은 정말 다양한 곳에 활용되고 있습니다. 페이드인, 페이드아웃은 대표적인 코루틴의 예시이고 응용하면 대화시스템에도 쓰입니다. 특히 밑에 설명할 설명할 라이프사이클처럼 프레임단위 반복호출이 아니기 때문에 특정 조건 만족 시 단 1회 호출로 사용해 최적화에 많은 도움을 주기도 합니다. 또한 물리검사를 코루틴으로 횟수를 제한하는 고급최적화에도 이용됩니다.

+a) C# 기본 비동기 문법인 async와 await도 사용할 수 있습니다. 코루틴과는 사용법이 또 다르니 궁금하다면 아래를 참고해보세요. (링크에서 TAP 부분 보시면 됩니다)

<https://docs.microsoft.com/ko-kr/visualstudio/gamedev/unity/unity-scripting-upgrade>

**4. 변수 참조**

첫째로, **유니티에서 직접 참조**하는 방법입니다. 유니티에서 변수의 값을 확인하고 직접 초기화하기 위해서는 접근지정자 **public**을 사용해야 합니다. public으로 만든 변수들은 유니티 인스펙터 창에서 값을 확인 및 수정할 수 있습니다. 만약 접근지정자 private을 쓴다면 인스펙터창에 값이 표시되지 않지만 창 모드를 **디버깅 모드**로 바꾸면 값을 확인할 수 있습니다(초기화하거나 값을 바꿀 수는 없습니다.) 유니티의 이러한 특징 때문에 OOP의 캡슐화를 지키기 어렵습니다.

참고로 어트리뷰트 [Serializefield]를 사용하면 private 변수를 public처럼 유니티 인스펙터창에 값을 확인하고 수정 가능하게 만들 수 있는데 이는 보안적으로 public과 똑같기 때문에 다른 클래스에서 접근 불가능하게 하려는 장점밖에 없습니다. 유니티 개발자 중에서도 의견이 분분한 부분으로 (가독성 VS 코드적 캡슐화) 중 선택하여 사용하시면 될 것 같습니다. 다만, 다른 클래스에서 접근해야 하는 변수가 있는데 인스펙터에서 직접 초기화가 필요 없을 때는 평소의 설계처럼 private과 프로퍼티를 사용하는 걸 추천드립니다.

둘쨰로, **스크립트에서 참조**하는 방법입니다. GetComponent<T> 함수는 해당 스크립트를 가지고 있는 게임 오브젝트 내부에 부착되어 있는 클래스를 찾게 됩니다. 참고로 이런 Get류 함수들은 비용이 크기 때문에 Update() 등 프레임 단위로 자주 호출하는 것은 피해야 합니다. 그러나 어쩔 수 없이 update()에서 호출해야 할 경우가 생길 경우에는 어떻게 해야 해당 함수의 호출을 줄일 수 있을 것인가에 대해 고민해봐야 합니다. 매 프레임마다 Get류 함수를 쓰는 것은 정말 피해야 하고 대부분의 경우 설계를 바꾸거나 코루틴 또는 bool을 사용함으로써 해결 가능합니다.

이 외에 Find류 함수들도 존재하는데 씬에 존재하는 모든 게임오브젝트들을 대상으로 탐색합니다. 비용은 굉장히 크므로 실제 게임 빌드 시에는 없는 것을 권장합니다. 만약 쓰더라도 Start()나 Awak()에서 1번 쓰고 캐싱(변수 저장)을 이용하는 게 좋습니다. (ex. Camera.main은 내부적으로 Find를 쓰고, 초기화 함수에서 Camera 변수에 담아서 쓰는 게 좋습니다)

**5. 인스펙터창과 public**

자신이 만든 게 아닌 에셋 등을 사용하려다 보면 어떻게 사용할 지 감이 안 올 수도 있습니다. 그런데 유니티에서는 인스펙터창만 보고도 변수조작을 하기가 쉽습니다. 위에서 배웠듯이 직접 참조를 하기 위해 변수를 public으로 만들어서 인스펙터창에서 변수를 관리했었습니다. 반대로 말하면, 인스펙터창에 보이는 변수는 코드로 바로 접근할 수 있습니다.

**6. 유니티의 기본 클래스와 객체**

유니티의 기본 오브젝트는 UI를 제외한다면 전부 Transform 클래스가 부착되어 있는 상태이며 이는 지울 수 없습니다. Monobehaviour 클래스는 이런 게임 오브젝트에 부착하는 것이기 때문에 미리 GameObject 객체와 Transform 객체가 구현되어 있습니다. 따라서 이들은 별도의 참조나 선언없이 바로 gameObject, transform으로 접근할 수 있습니다.

**7. Transform과 상속**

모든 게임 오브젝트의 자식과 부모 관계는 Transform 클래스로 이어져 있습니다. 이 말을 실용적으로 활용할 수 있게 바꾸자면 Transform 클래스로 부모와 자식관계에 대해 접근할 수 있습니다. transform.GetChildCount()와 같이 부모와 자식과의 관계를 얻는 함수는 transform으로 접근하게 됩니다.

**8. Tag와 Layer의 차이**

Tag는 보통 충돌과 관련되어 쓰이는 반면, Layer는 레이캐스트라는 물리 엔진을 쓸 때 사용하거나 카메라가 보여주는 부분을 제한할 때 사용합니다. 또한 태그와 레이어는 얼핏 보면 Enum처럼 보이지만 태그는 String클래스, Layer는 LayerMask 라는 별도의 구조체로 구성됩니다.

**9. Transform의 회전문제와 Quaternion**

Position과 Scale은 그냥 Vector3를 사용하지만 문제는 Rotation입니다. 3D에서는 아래 링크와 같은 벡터값을 사용한 회전에 문제가 있습니다. 이 문제를 **짐벌락(gimbal lock)** 문제라고 하며 이를 해결하기 위해 내부적으로는 쿼터니언이라는 방식을 사용합니다. 따라서 유니티창에는 작업하기 편리하게 벡터값처럼 보여주고 있지만 실제로 회전과 관련된 코드를 작성할 때는 Vector3가 아닌 Quaternion 클래스를 이용해야 합니다.

<https://docs.unity3d.com/2021.2/Documentation/Manual/QuaternionAndEulerRotationsInUnity.html>

**10. 이동 시 가변 프레임 문제점**

Transform 클래스에서 이동함수를 쓰다 보면 분명 유니티창에서는 원하는대로 잘 작동됐는데 이후에 게임을 빌드해보니 원했던 결과가 아니었을 수 있습니다. 이는 바로 Update()의 문제점입니다. 게임 환경에 따라 프레임이 변하다보니 이동함수를 Update()에 구현해버려서 원하는 결과가 나오지 않는 것입니다. 그래서 코드를 FixedUpdate()에 옮겼더니 움직임이 너무 버벅여 맘에 안 들 수도 있습니다(웬만하면 가변프레임이 훨씬 짧습니다). 이럴 때 필요한 것이 바로 Time.deltaTime입니다. 이는 바로 전 프레임과 현재 프레임까지 걸린 시간을 의미하는데 가변프레임에 맞는 결과를 내놓습니다. 다음과 같은 예시를 봅시다.

/\*1초에 x축으로 100만큼 이동하고 싶은 상황\*/

transform.position.x += 100f / Time.deltaTime;

위의 코드를 Update()에 적게 되면 부드럽게 x축으로 1초에 100만큼씩 이동하는 걸 볼 수 있습니다. 이런 코드는 이해하기에도 쉬운데 그 이유가 바로 프레임 단위가 아닌 초 단위로 생각할 수 있기 때문입니다.

<https://docs.unity3d.com/2021.1/Documentation/Manual/TimeFrameManagement.html>

**11. 물리 엔진**

대부분의 게임은 전부 물리 이벤트를 사용합니다. 여러 물리 이벤트가 존재하지만 그 중 가장 많이 보게 될 물리 이벤트는 충돌 이벤트입니다. 유니티에서는 충돌 이벤트를 확인하기 위해서 **Collider**라는 컴포넌트를 이용합니다. 종류는 크게 보면 (2D or 3D), (내부가 차있는 것 or 아닌 것)으로 나뉩니다.

충돌 이벤트 감지와 관련된 것은 대표적으로 크게 2개로 나뉩니다. 하나는 Rigidbody이고 하나는 Raycast입니다.

Collider가 영역이었다면 **Rigidbody**는 강체입니다. 예를 들어 영역 2개가 겹쳐있다고 해도 사건이 발생하지는 않겠죠. 영역 2개 중 적어도 하나가 Rigidbody를 가지고 있다면 이는 충돌로 받아들이게 됩니다. 웬만하면, 동적인 물체에 부착하는 것이 좋습니다.

충돌이 발생했을 때 어떤 동작을 하고 싶을 수 있습니다. 이는 Monobehaviour 클래스에 내장된 이벤트 함수들을 이용하면 됩니다. 충돌은 처음 영역에 들어왔을 때, 영역 안에 있을 때, 영역 밖으로 나갔을 때로 구분됩니다. 각각 OnCollision류 이벤트 함수가 구비되어 있습니다.

한편, OnTrigger 이벤트 함수들도 존재하는데 이는 Rigidbody에서 isTrigger 옵션을 true로 만들어야 작동되는 함수들입니다. 이는 충돌감지는 하고 싶은데 Rigidbody의 옵션은 사용하고 싶지 않을 때 사용합니다. 즉, 강체의 개념을 사용하고 싶지 않을 때 사용합니다.

**Raycast**는 오브젝트에 부착하는 클래스가 아니고 내부적으로 사용되는 함수입니다. Raycast는 Physics.Raycast()로 사용할 수 있고 선, 또는 물체를 특정 방향으로 쏴서 특정 영역을 감지하는 방식입니다. 즉, 시각적으로 충돌을 발생시키는 것은 아니고 무언가를 ‘감지’하는데 쓰입니다.

이러한 리지드바디나 레이캐스트와 같은 물리 관련 코드는 전부 **FixedUpdate()**에 있어야 합니다.

**12. Input**

유니티는 입력을 3가지로 분류합니다. 누르는 순간, 누르는 도중, 떼었을 때. 이러한 기본적으로 입력은 Input 클래스가 별도로 존재합니다.

입력에서 중요한 점은 누르는 순간과 떼었을 때와 관련된 함수는 **꼭 Update()**에 존재해야 합니다. 보통 가변프레임이 고정프레임보다 빠르고 누르는 순간과 떼었을 때는 키입력 중 단 1번 일어나는 것이기 때문에 FixedUpdate()를 포함한 다른 별도의 이벤트함수에 존재할 경우 키입력이 제대로 안 될 가능성이 큽니다.

Input.GetKey 함수는KeyCode라는 enum 데이터를 이용하는데 KeyCode.A는 A를 의미하는 것처럼 직관적이기 때문에 사용이 편리합니다. Input.GetButton류 함수는 GetKey류 함수보다 덜 직관적이므로 GetKey를 애용합시다.

Input.GetAxis(“Horizontal”)은 수평키입력(A, D, 왼쪽화살표, 오른쪽화살표)을 받았을 때 float 값을 내놓습니다. Input.GetAxis(“Vertical”)는 수직키입력(S, W, 위아래화살표)을 받았을 때 float 값을 내놓습니다. 이들은 보통 입력장치가 조이스틱 등 입력에 정도가 있을 때 사용하게 됩니다.

Input.GetAxisRaw()는 GetAxis()와 다른 점은 반환값이 오직 -1f, 0f, 1f 중 하나입니다. 이동과 관련된 함수를 쓸 때 입력을 안 받았는지, 받았다면 방향이 어디인지만 확인할 때 편합니다. 따라서 대부분의 이동입력은 이 GetAxisRaw()를 이용할 것이고 위의 GetAxis()는 조이스틱 같은 입력에 차이가 있는 입력도구일 경우에 사용됩니다.

Input 함수의 일부는 매개변수로 string값을 주게 되는데 이는 유니티 내부에서 미리 정의된 입력값들을 사용하는 것입니다. Unity에서 Edit-ProjectSettings-InputManager에 들어가면 string값들을 어떤 것을 써야 하는지 확인할 수 있습니다.

Input.mousePosition는 현재 마우스의 좌표를 화면픽셀 기준으로 값을 내놓게 됩니다.

<https://docs.unity3d.com/kr/530/ScriptReference/Input-mousePosition.html>

한편, 마우스입력은 키입력에 비해 다양한 편입니다. 이들은 예외적으로 Input클래스가 아닌 MonoBehaviour에 있는 별도의 내장 이벤트 함수로 존재하고 이들은 꼭 게임오브젝트에 **콜라이더**가 부착되어 있어야 인식합니다. 내부적으로 마우스 입력을 Raycast로 하고 충돌이벤트를 확인하는 방식이기 때문입니다. OnMouseDrag(), OnMouseEnter() 등이 존재하며, 이는 Input 함수와는 달리 콜라이더를 기준으로 함수가 작동됩니다.

복잡하겠지만 게임 오브젝트 중에서도 UI는 특별취급 당합니다. 따라서 UI와 관련된 마우스입력은 또 다른 곳에 정의되어 있습니다. 이는 내장함수가 아닌 인터페이스 형태로 구성되어 있습니다. 예를 들어 마우스 드래그와 관련된 UI 동작이 있다면 IDragHandler를 클래스에 인터페이스해주고 해당 함수인 OnDrag()를 작성하면 됩니다. 이 외에도 다양한 인터페이스가 존재합니다.

**13. 사용자 지정 클래스**

유니티 엔진에 보이는 데이터는 전부 직렬화(serialization)라는 과정을 거친 데이터입니다. 그런데 만약 스크립트로 클래스를 만들었는데 유니티 인스펙터창에서 이를 조절하고 싶다면 해당 클래스를 직렬화해줘야 합니다. 이를 위해 클래스 구현 위에 [Serializeable]을 추가합니다.

Ⅳ. 게임엔진 이해 – 유니티 기초 지식

**1. 릴리즈 버전**

릴리즈는 LTS와 아닌 것, 베타버전 이렇게 3가지로 구분할 수 있습니다. 보통 개발을 시작할 때 가장 최근의 LTS를 쓰는 것을 권장합니다(LTS = Long Term Support). 가장 최신 버전을 쓰는 이유는 당연하게도 더 좋아진 기능들과 추가된 기능을 사용하기 위해서이고 LTS는 프로젝트를 이후에 더 상위 버전으로 마이그레이션 하더라도 높은 호환성을 보장받기 위해서입니다.

**2 2D VS 3D**

유니티2D와 3D 옵션을 선택할 수 있는데 이는 처음에 선택을 잘 하셔야 합니다. 언뜻 보면 2개의 차이가 없어 보이지만 해당 옵션에 따라 변경되는 기본 에셋들이 있습니다.

**3. 게임오브젝트의 이해**

게임 오브젝트는 그 자체로 객체 개념이라고 보시면 돼요. 또한 게임을 이루는 가장 기본적인 단위이며 게임오브젝트가 모여 씬이 되고 씬이 모여 게임이 완성된다는 점을 기억합시다.

**4. 프리펩(Prefab)**

반복되는 게임오브젝트를 한 번에 변경할 수 있다면 얼마나 좋을까요? 프리펩은 게임오브젝트 자체를 또 하나의 새로운 클래스로 만든다고 생각하시면 됩니다. 즉, 게임 오브젝트의 청사진을 만든다는 개념입니다. 프리펩은 언제든 가져다 쓸 수 있고 프리펩을 변경하면 프리펩을 쓴 오브젝트는 전부 프리펩에 따라 변경됩니다. 프리펩 변형이라는 것도 존재하는데 프리펩을 프리펩으로 만들었을 때 변경되지 않은 부분은 프리펩 원형을 따르고 변경된 부분은 또 다시 독립적으로 프리펩 변형 오브젝트의 청사진을 가지게 됩니다. 클래스의 상속 개념과 유사합니다. 프리펩은 데이터 관리에도 중요하지만 런타임 중 게임 오브젝트 생성에도 쓰이는 중요한 개념입니다.

<https://docs.unity3d.com/2021.1/Documentation/Manual/InstantiatingPrefabs.html>

**5. 애니메이션**

코드로 모든 동작을 제어하는 것은 한계가 있습니다. 따라서 애니메이션이라는 기능이 존재하고, 키 프레임창에서 C키를 누르면 부드러운 연출이 가능하도록 변화값에 커브를 줄 수 있습니다.

블렌드 트리와 애니메이션 레이어의 개념도 있지만, 사실 동아리에 있으면서 방대한 애니메이션을 경험할 일은 거의 없을 겁니다. 그러나 복잡한 애니메이션에 대한 관리에 대해 궁금하다면 별도로 찾아보시면 될 것 같아요.

**6. UI**

UI에 있어서 처음에 많이 하는 실수는 UI 요소를 한 Canvas에 넣는 것입니다. Canvas는 용도에 맞게 분할하고 쓰지 않는 캔버스는 캔버스 자체를 SetActive(false) 해주시는 게 좋습니다. 이 이유는 캔버스 내부에서 변경점이 하나라도 생기면 유니티가 캔버스 전체를 다시 그리기 때문입니다. 따라서 캔버스를 분할하는 것은 중요한 일입니다.

<https://docs.unity3d.com/kr/2018.4/Manual/UICanvas.html>

**7. 에셋스토어**

유니티가 흥할 수 있었던 이유가 바로 이 에셋스토어 때문입니다. 정말 다양한 에셋들이 나와 있고 자신이 직접 무언가를 만들려고 하는 것보다 에셋을 찾아보는 것이 훨씬 빠른 경우도 빈번합니다.

**8. 예약된 폴더**

유니티엔 미리 예약된 폴더가 존재합니다. 대표적인 예시는 Resources 폴더와 Editor 폴더입니다. Resources 폴더에 있는 에셋은 사용하기 편리하지만 데이터를 쓰든, 안 쓰든 무조건 메모리에 올라간다는 단점이 있습니다. 따라서 싱글톤 패턴을 적용한 게임 오브젝트를 저장하는 경우가 아닌 매터리얼이나 스프라이트 등의 에셋을 저장하는 경우에는 신중히 판단해야 될 것입니다. 한편, 다른 에셋 저장 방법인 에셋번들도 존재합니다. 모바일 게임에서는 업데이트의 편리함과 용량을 줄이기 위해 많이 사용하는 방법이지만 그 과정이 까다롭습니다.

<https://docs.unity3d.com/kr/2021.2/Manual/SpecialFolders.html>

<https://docs.unity3d.com/kr/2018.4/Manual/AssetBundlesIntro.html>

**9. 외부 파일과 파싱(Parsing)**

대화 중심의 게임은 특히 많은 텍스트 데이터를 필요로 합니다. 이를 유니티 엔진 내부에서만 관리하기는 굉장히 힘들기 때문에 보통 엑셀 등으로 외부파일을 만든 후 그 내부 데이터를 파싱해서 실제 데이터에 사용하게 됩니다.

**10. 스크립터블 오브젝트(Scriptable object)**

유니티 엔진에 내장된 데이터 관리법 중 하나이며, 일반 유니티 데이터보다 좋은 점은 바이너리 코드로 관리된다는 점입니다. IO 커리큘럼에서 배웠듯이 바이너리 코드는 접근도 더 빠르고 용량도 더 적어 효율적이라는 장점을 가지고 있습니다. 정형화되어 있는 많은 종류의 데이터를 다루거나, 위에서 언급한 파싱을 해야 할 때 사용하곤 합니다. 또는 이진코드의 안정성 때문에 후술할 싱글톤 패턴 오브젝트를 대신하기도 합니다.

**11. 게임 데이터 저장**

PlayerPrefs 클래스는 유니티 내장 클래스로, 사용하기 편하다는 장점이 있지만 자료형이 제한적이고 보안이 취약하다는 단점이 있습니다. 따라서 보통 볼륨이나 그래픽 세팅 등 게임에 큰 지장을 주지 않는 단순한 요소들을 저장합니다.

BinaryFormatter는 데이터를 이진파일로 저장하는 것을 돕는 클래스입니다. 유니티 내장 클래스라 사용하기 편리하기도 하고 안전하며 가장 범용적입니다. 플레이에 지장을 줄 수 있는 변형되면 안 되는 핵심 데이터를 저장합니다.

그 외 외부파일(XML, JSON) 등을 이용하는 방식도 있습니다. 이 경우 암호화를 통해 중요한 데이터는 보호하는 걸 추천합니다.

Ⅴ. 프로젝트와 협업

**1. 협업과 일정 관리**

프로젝트 관리는git으로 합니다. 따라서 협업 전 github 사용법을 익혀두면 좋을 것 같습니다. 주의할 점은 github에서 새 프로젝트 설정 시 .gitignore 파일 타입을 유니티로 설정해줘야 합니다. 그렇지 않으면 불필요하게 큰 파일도 같이 공유하게 되기 때문입니다.

프로젝트에서 기획만큼이나 중요한 게 바로 일정관리입니다. 이를 위해 **트렐로**나 **노션** 등을 쓰게 됩니다. 특히 팀원들과의 소통이 굉장히 중요합니다. 예를 들어 PM이 기획서에 제대로 명시를 안 한 부분이 있다면 그 즉시 물어보는 게 이후에 일이 커지지 않습니다. 소통이 많으면 많을수록 오히려 더 빠르게 일처리가 가능합니다.

**2. 프로토타입 만들기**

PM이 기획서를 만들어왔을 때 에셋스토어를 쓰는 게 아니라면 당연히 아트 리소스는 존재하지 않겠죠. 정말 백지의 상태에서 프로그래머는 팀원들에게 무언가를 보여줘야 합니다. 보통 프로토타입은 단순한 객체(3D라면 큐브, 2D라면 단순 스프라이트)를 이용해서 핵심 시스템을 구현해서 보여줍니다.

**3. 디자인 패턴**

디자인 패턴은 설계를 돕기 위해 나온 개념입니다. 설계를 어떻게 해야 더 효율적이거나 확장성있거나 안전한가를 논합니다. 디자인패턴에 정말 많은 종류가 있지만 유니티에서 가장 많이 쓰이는 것은 **싱글톤 패턴**이며 최적화 기법으로는 **오브젝트 풀링**이 많이 쓰이게 됩니다.

-싱글톤 패턴(Singleton Pattern)

싱글톤 패턴은 프로그램에서 단 1개 존재해야 되는 객체에 대한 클래스를 정의할 때 사용합니다. 게임에서 많이 쓰이는 이유는 사라지지 않는 데이터 관리를 위해서입니다. 유니티는 하나의 씬이 끝나고 다른 씬으로 넘어갈 때 해당 씬에 있는 모든 오브젝트를 정리하게 됩니다. 즉, 모든 데이터가 씬이 이동될 때 사라집니다. 하지만 게임에서는 스테이지를 깼다고 해서 현재 가지고 있는 돈이 사라지진 않을 겁니다. 이런 데이터를 관리할 때 싱글톤 패턴을 사용하고 클래스 이름은 관용적으로 GameManager라고 합니다. 실제 구현은 아래를 참고하시면 좋을 것 같아요.

<https://www.youtube.com/watch?v=-wzULWMvFu0>

-오브젝트 풀링(Object pooling)

오브젝트 풀링은 최적화 기법 중 하나입니다. 생성과 삭제가 빈번한 게임인 슈팅게임에서 가장 많이 사용됩니다. 생성과 삭제가 많다는 의미는 가비지가 많이 쌓인다는 의미입니다. 가비지 컬렉터가 작동할 때는 다른 일을 하지 않고 오직 메모리 수집만을 하기 때문에 프레임 드롭이 일어났었죠. 이는 게임 플레이어 입장에선 굉장히 불편한 일입니다. 따라서 미리 객체를 생성해두고 만약 객체가 필요하다면 Pool에서 꺼내쓰고, 객체가 파괴된다면 삭제하지 않고 다시 Pool에 돌려줍니다. 그러나 이는 항상 차지하는 메모리 사용공간이라는 단점이 있기 때문에 프로파일링을 잘 해서 최적의 오브젝트 풀을 생성해야 합니다. 실제 구현은 아래를 참고해보세요.

<https://www.youtube.com/watch?v=xiojw0lHzro&t=734s>

Ⅵ. 추가 내용

1. **데이터 관리를 어트리뷰트로 더 편리하게**

유니티에서는 클래스와 단순 변수에 어트리뷰트를 붙여줌으로써 관리를 더 편리하게 할 수 있습니다.

[Header()]: 인스펙터를 정리할 수 있도록 구간을 나눌 수 있게 됩니다.

[Range()]: 정수 또는 소수에 사용 가능. 슬라이드바 형태로 값을 조정할 수 있게 됩니다.

[TextArea()]: 스트링에 사용 가능. 문자열 편집창이 보기 쉽게 만들어집니다.

[RequireComponent(typeof(클래스명))]: 해당 클래스를 사용하기 위해서는 해당 클래스가 꼭 같이 부착되어야 한다는 어트리뷰트입니다. 만약 이 클래스를 게임 오브젝트에 부착하게 되면 기술한 클래스가 같이 부착되게 됩니다. 그리고 같이 부착된 클래스는 에디터에서 지울 수 없습니다. 해당 어트리뷰트에서 리플렉션을 하는데 이것은 인게임에서 실행되는 것이 아니라 에디터에서 실행되는 것이므로 성능문제는 안심해도 됩니다.

**2. 유니티 빌드 심화**

위에서 언급했던 IL2CPP는 C# 컴파일러가 바이너리 코드로 만든 CIL을 .cpp파일로 만든 후 다시 바이너리 파일로 만든 다음 마치 C++과 같이 링킹의 과정을 거쳐서 하나의 실행파일을 만들게 됩니다. 이런 일을 하는 가장 큰 이유는 보안 때문이고 부차적인 이유는 게임 용량의 감소입니다. 용량은 게임에 따라 다르겠지만 체감 상 20~30% 정도 감소하는 것 같습니다. 그러나 고전적인 컴파일러의 특성 상 호환이 안 되는 플랫폼도 있으므로 주의해야 하며 빌드 시간이 말도 안 되게 길어지니 꼭 최종 빌드 시에만 사용하는 것이 좋습니다.

<https://docs.unity3d.com/2021.2/Documentation/Manual/IL2CPP.html>

**3. 세간의 잘못된 정보 ~ foreach와 딕셔너리 박싱**

유니티는 2017버전 이전까지는 호환성 문제 때문에 아주 예전의 닷넷 프레임워크를 사용했었습니다. 그래서 foreach를 쓰거나 제네릭 컬렉션이더라도 딕셔너리에서 key로 enum이나 struct를 사용하면 박싱이 일어나서 성능에 큰 문제를 일으켰습니다. 그리고 이 문제가 현재도 지속된다고 널리 퍼뜨리는 사람이 여전히 있습니다.

하지만 현재는 박싱은 당연히 일어나지 않으며 유니티의 프레임워크는 최신 닷넷으로 업데이트가 지속되고 있습니다. 해당 문서 작성일 기준 가장 최신 유니티 버전은 가장 최신 닷넷을 사용하고 있습니다. 다소 오해의 소지가 있는 게 유니티의 기본 빌드는 .Net Standard 2.0 이기 때문입니다(가장 최신 닷넷은 .Net 4.8). 하지만 이는 최신 닷넷을 반영하는 빌드입니다. 단지 .Net 4.x 빌드와 다른 점은 적용되는 API의 수가 다를 뿐이며 웬만하면 이 빌드 설정을 건드릴 일은 없습니다. 그러니 안심하고 foreach와 딕셔너리 key로 enum을 사용하는 건 물론이고 가장 최신 C#의 문법을 사용해도 됩니다.

<https://docs.unity3d.com/2021.2/Documentation/Manual/dotnetProfileSupport.html>

<https://dotnet.microsoft.com/platform/dotnet-standard>

**4. 최적화 (단순코드)**

언급할 최적화는 단순코드 최적화로, 대부분의 경우는 알고리즘이나 설계를 바꾸는 것이 더 효율적입니다. 또한 효과가 미미하여 크게 신경쓰지 않아도 될 부분도 많습니다. 실제로 최적화는 프로파일링을 통해 문제가 되는 부분만 캐치해서 하는 것이 보편적입니다. 그러나 굳이 궁금한 분들을 위해 준비했습니다.

- 자주 쓰는 메모리는 캐싱(Caching)을 합니다.

- 특히 배열을 리턴하는 API의 빈번한 엑세스는 캐싱을 이용합니다.

- 1번만 해야 한다면 반드시 1번만 하도록 설계합니다.

- 제네릭 컬렉션과 배열의 경우, 재사용을 고려합니다. 예를 들어 컬렉션의 경우 Clear() 함수를 쓴 후 재사용, 배열은 전부 쓰더라도 삭제하지 않고 빈 배열을 남겨두는 등

- 클로저 사용과 델리게이트를 이용해 함수를 매개변수로 받는 것은 매 프레임마다 진행되는 곳에서는 지양합니다.

- 코루틴은 실행 시 메모리를 사용하므로 반복적인 코루틴이 존재한다면 내부에서 반복문을 돌릴 수 없는지 고려합니다.

- 문자열은 클래스로, 가비지에 가장 주의해야하는 객체입니다.

- 문자열을 Enum으로 변경할 수 없을지 고려합니다.

- 문자열 비교는 ==이 아닌 string.Compare()를 이용합니다.

- 태그 비교는 ==이 아닌 CompareTag()를 이용합니다.

- Debug.Log()는 최종 빌드 시 반드시 삭제합니다.

- 빈 문자열은 “”가 아닌 String.Empty를 사용합니다.

- 문자열의 +는 string.Concat과 동일하며 +를 할 때마다 가비지를 생성합니다. 하지만 속도는 빠릅니다. 5개 이상의 문자열을 합칠 때는 StringBuilder 클래스를 이용하는 것이 좋습니다. 속도는 느리지만 가비지 생성을 하지 않습니다.

- 애니메이터나 머터리얼, 쉐이더 이용 시 해시코드를 캐싱해서 매개변수로 사용합니다.

- 비할당 물리 API가 있다면 그것을 사용합니다. (ex. RaycastAll => RaycastNonAlloc)

- 클래스 null 비교는 프레임 단위에선 지양합니다.

- 정수, 부동소수점, 구조체와 클래스 순서로 연산이 느리고 덧셈, 뺄셈, 곱셈, 나눗셈 순서로 연산이 느립니다. (ex. vec / c => vec \* (1 / c))

- Find류 함수는 최대한 피하고 필요하다면 1번만 호출하고 캐싱합니다.

- Camera.main은 내부적으로 Find류 함수를 씁니다. 꼭 1번만 호출해 캐싱합니다.

- 다차원 배열 필요 시 가변 배열을 사용합니다. (type[x][y] => type[x, y])

- 벡터에서 normalized보다는 magnitude를 사용합니다.

- 벡터에서 단순 거리 비교는 sqrMagnitude를 사용합니다.

- 굳이 실행하지 않아도 되는 환경의 업데이트 함수는 바로 종료하도록 bool로 관리합니다.

- 빈 콜백함수(start, update) 등은 삭제합니다.

**5. 최적화 (리소스)**

- 이미지는 2의 멱수로 설정합니다. (32, 64, 128, 256, …)

- 2의 멱수가 비효율적인 이미지는 다른 이미지와 합쳐 2의 멱수인 아틀라스를 만듭니다.

- 알파값을(불투명도를) 쓰지 않습니다.

- 이미지 포멧은 최적화와 아무 관련이 없습니다. (유니티는 어떤 파일이라도 가공작업을 합니다.)

- 좌우음향이 필요없다면 소리는 mono로 바꿉니다.

- 효과음은 무압축인 wav, 배경음이나 인물대사 크기가 큰 것은 mp3나 ogg를 이용합니다. (wav는 무압축이라 읽는 속도가 빠르지만 용량이 큽니다.)

- 모바일 3D의 경우 로우 폴리 에셋을 사용합니다.

**6. 최적화 (유니티 엔진)**

- 캔버스 분할을 합니다.

- 쓰지 않는 캔버스는 캔버스 통째로 비활성화합니다.

- 애니메이션 사용 시 scale 값 변경은 지양합니다.

- UI에는 애니메이션 사용을 지양합니다.

- 레이아웃그룹을 지양합시다.

- 화면 전체를 가리는 UI 시 모든 카메라를 끕니다.

- 이미지 리소스가 분할인 경우, 슬라이스를 사용합니다.

- 반복되는 이미지 사용 시 9슬라이싱을 사용합니다.

- 이미지의 Max Size는 꼭 해당 이미지에 맞게 변경합니다.

- 터치 불가능한 UI는 Raycast를 체크 해제합니다.

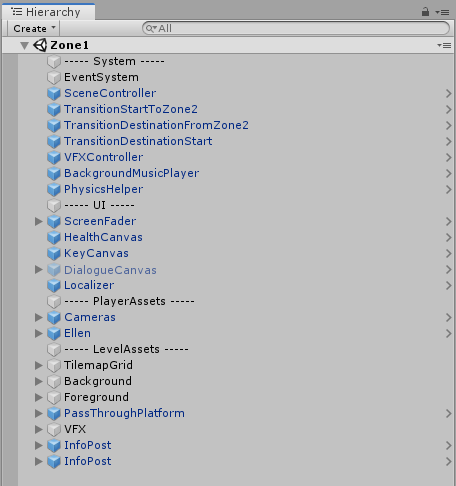
- 의미없이 게임오브젝트의 상속관계는 만들지 않습니다. (ex. 오브젝트 정리용)

- 3D가 아니라면 static 옵션을 건드리지 않습니다.

- 3D라면 상황에 맞는 static 옵션을 사용합니다.

이 외에도 유니티 엔진 자체 최적화엔 정말 다양한 방법이 존재합니다.

8. **유니티 게임오브젝트 관리 팁**



위처럼 비활성화 오브젝트로 게임오브젝트를 구분하면 씬 관리가 더 쉽습니다.

[문제]

방대한 양의 문서 읽느라 고생하셨습니다. 자세한 내용은 독학해야겠지만 이 문서가 방향성을 제시해준다고 느끼면 좋겠네요.

시작일은 2022.00.00() 입니다. 마감기한은 **2022.00.00() 23:59**입니다. 특별한 사정없이 마감 기한까지 제출을 못하면 경고를 부여받으므로 주의해주세요.

이번 과제는 커리큘럼 조건에 맞춰 미니게임을 만드는 것입니다.