**유니티 5장**

**5장에서는 총 발사 로직을 배울 예정이다.** 크게 2가지 방식으로 구분된다 :

**Projectile 방식** :

총알이 물리적으로 발사되는 방식. 직접 날아가 타격을 입힘.

총알이 시각적으로 표현, but 게임 속도 저하 원인이 될 수도

해결을 위해 Low Poly 모델이나 평면 메시 모델에 텍스처를 입혀 사용

총알 역시 동적으로 생성하기보다는 Object Pool 개념을 도입해 총알을 미리 생성해두고

번갈아 가며 발사되게 구현하는 편이 속도 향상에 유리

**RayCasting 방식** :

히트스캔 방식. 보이지 않는 레이저 빔을 발사해 적 또는 물체를 검출하는 방식.

총알 발사 루틴 뿐 아니라 미로를 탐색하는 로봇의 센서 기능이나 추적 기능을 구현할 수 있음

**Rigidbody 컴포넌트 = 강체**

유니티에서 게임오브젝트를 이동할 때는 보통 Transform의 Position 속성을 이용하거나 물리 엔진을 이용. 총알은 물리 엔진의 힘을 이용한다. 그러기 위해서는 Rigidbody 컴포넌트가 필요하다. 게임오브젝트에 Rigidbody컴포넌트를 추가하면 해당 게임오브젝트는 물리 엔진의 영향을 받는다. 코드를 작성 안해도 FixedUpdate 함수의 주기로 물리적인 연산을 함.

**Mass** : 질량. 상대적인 의미의 질량으로 1kg, 1g 등의 무게 단위가 아니다. 즉, A 물체의 Mass가 1이고 B 물체는 10이라 한다면 B가 A보다 10배 무겁다고 해석. 편의상 1KG로 가정

**Drag** : 이동할 때 적용되는 마찰계수(저항)

**Angular Drag** : 회전할 때 적용되는 마찰계수(저항)

**Use Gravity** : 중력 적용 여부

**Is Kinematic** : 체크하면 물리 시뮬레이션을 이용해 이동하지 않고 Transform 컴포넌트를 이용해 이동한다. 즉, 이 옵션을 체크하면 물리 엔진의 영향 안 받음

**Interpolate** : 물리력을 이용한 움직임이 끊어지는 현상이 발생할 때 보간해준다.

Interpolate : 이전 프레임의 Transform에 맞춰 움직임을 부드럽게 처리

Extrapolate : 다음 프레임의 Transform 변화를 추정해 움직임을 부드럽게 처리

**Collision(Detection)** : 충돌 감지 옵션으로 아주 빠른 물체는 물리 엔진이 충돌 검출을 놓칠 수 있다. 따라서 좀 더 세밀한 충돌을 검출하기 위한 옵션값이다.

빠르게 이동하는 물체의 충돌을 감지하려면 3가지 옵션을 사용함

\* Continuous

\* Continuous Dynamic

\* Continuous Speculative

**Freeze Position** : X, Y, Z 축 중에서 해당 축으로의 이동을 막는다.

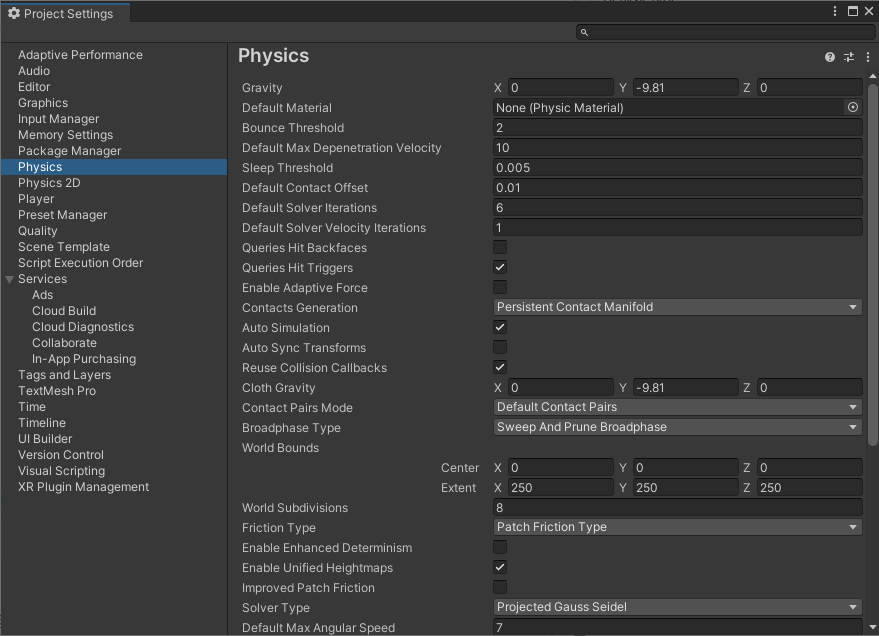
**Freeze Rotation** : x, y, z 축 중에서 해당 축을 기준으로 회전을 막는다.

강체? 외력을 가해도 크기가 형태가 변하지 않는 이상적인 물체를 말함.

**물리 엔진 속성 설정 – Physics Manager**

유니티에 탑재된 물리엔진의 설정은 Physics Manager에서 확인할 수 있다.

메뉴에서 Edit -> Project Settings 에 Physics 섹션을 선택한다.

****

**\* Gravity**

Rigidbody의 Use Gravity 속성에 체크하면 Gravity에 설정된 값으로 중력이 작용한다.

기본값은 Y축으로 -9.81이다. 즉, 중력가속도 -9.8m/s2 로 적용됨

**\* Default Material**

두 물체가 충돌했을 때 물리적인 반작용에 대한 속성을 전역적으로 설정한다.

None으로 설정하면 각 Rigidbody에서 개별적으로 물리적인 반작용을 설정할 수 있다.

**\* Bounce Threshold**

충돌한 두 물체의 상대 속도가 이 값보다 작으면 충돌이 발생하지 않는다.

또한 떨림 현상(Jitter)을 감소시키는 역할을 한다.

\* Layer Collision Matrix

Build in Layer 또는 사용자 정의 레이어(Layer) 간의 충돌 감지 여부를 선택적으로 활성화하거나 비활성화할 수 있다. 즉, Rigidbody를 포함한 게임오브젝트가 특정 레이어에서만 물리 엔진의 영향을 받게 할 수 있다.

**Collider 컴포넌트\*\***

유니티는 충돌 감지를 위해 다양한 콜라이더 컴포넌트를 제공한다. Collider는 충돌을 감지하는 일종의 센서라고 생각하면 된다.

**\* Box Collider**

가장 다양한 용도로 사용되는 콜라이더. Center와 Size 속성으로 박스 형태를 조절.

Edit Collider 토글 버튼을 클릭하면 씬 뷰에서 직접 크기를 조절할 수 있다.

**\* Sphere Collider**

가장 속도 처리가 빠른 콜라이더. Radius 속성으로 구체의 반지름을 조절. 정밀한 충돌 감지를 해야 하는 경우를 제외하곤 대부분 Sphere Collider를 활용할 것을 권장함.

**\* Capsule Collider**

주로 인체 또는 나무, 가로등과 같은 모델의 충돌체로 사용한다. Height 속성은 캡슐의 높이를 조절한다. Direction 속성은 Height 속성값을 변경했을 때 길어치는 축을 설정할 수 있다.

**\* Mesh Collider**

CPU 부하가 가장 높은 Collider다. 추가한 모델의 폴리곤 수가 아주 많다면 굉장히 세밀한 Collider가 형성되기 때문에 로우 폴리곤에서만 사용해야 한다. 따라서 이 충돌체는 세밀한 충돌 감지에 사용한다. 유니티에서는 Mesh Collider 간의 충돌 감지가 안 되도록 기본값이 설정돼 있다. 이는 속도 저하를 방지하기 위함이며 충돌 감지를 해야 할 때는 Mesh Collider 속성 중 Convex 옵션을 체크해야만 충돌이 감지된다.

인터넷 상의 자료나 문서를 보면 Mesh Collider는 절대로 사용하지 말라는 내용을 볼 수 있으나, 미로처럼 복잡한 게임 스테이지에서 플레이어가 스테이지를 벗어나지 못하게 하려고 여러 개의 Collider를 배치한다면 작업 시간도 오래 걸리고 작업량도 많아서 개발자가 누락하는 실수를 범할 수 있다. 이럴 Mesh Collider를 활용할 수 있다. 게임 스테이지 전체를 로우폴리 메시로 제작하고, Mesh Collider를 적용한 후 Mesh Renderer는 제거해서 게임 스테이지에 오버랩해 사용하는 방식이다. 물론 성능 모니터링을 통해 구동 디바이스의 성능이 충분히 수용 가능한지 먼저 검토해야 한다.

**\* Wheel Collider**

차량의 바퀴에 사용할 목적으로 제공되는 Collider다. 바퀴의 서스펜션, 바닥과의 마찰 계수와 미끄러지는 저항을 세밀하게 설정할 수 있다.

**\* Terrain Collider**

유니티에 내장된 Terrain Engine을 이용해 생성한 지형에 적용되는 Collider다. 지형의 복잡도에 따라 Collider의 부하가 높아진다. 로우폴리 메시를 이용해 Mesh Collider로 대체하는 것도 좋은 방법이다.

**충돌 감지 조건**

Bullet이 발사되어 일정거리를 날아가서 벽이나 적에게 맞았을 때 충돌을 감지하려면 다음의 두 가지 조건을 만족해야 한다 :

1. 충돌을 일으키는 양쪽 게임오브젝트 모두 Collider 컴포넌트가 있어야 함

2. 두 게임오브젝트 중 움직이는 쪽에는 반드시 Rigidbody 컴포넌트가 있어야 함

**충돌 이벤트**

Collider 컴포넌트를 포함한 게임오브젝트 간의 충돌이 발생하면 OnCollision~ 이벤트 함수 또는 OnTrigger~ 이벤트 함수가 발생한다. 이벤트 함수라고 부르는 이유는 언제 충돌이 발생할지 모르는 상태에서 충돌이 감지되는 시점에 시스템에서 호출해주기 때문이다. 또 다른 용어로 콜백 함수(Call Back function)라고 부르기도 한다. 콜백은 충돌이 감지되면 시스템(Back)에서 호출(Call)한다는 것을 의미한다.

|  |  |  |
| --- | --- | --- |
| 함수명 | 호출 조건 | is Trigger 옵션 |
| Void OnCollisionEnter | 두 물체 간의 충돌이 일어나기 시작했을 때 | 언체크 |
| Void OnCollisionStay | 두 물체 간의 충돌이 지속할 때 | 언체크 |
| Void OnCollisionExit | 두 물체가 서로 떨어졌을 때 | 언체크 |
| Void OnTriggerEnter | 두 물체 간의 충돌이 일어나기 시작했을 때 | 체크 |
| Void OnTriggerStay | 두 물체 간의 충돌이 지속될 때 | 체크 |
| Void OnTriggerExit | 두 물체가 서로 떨어졌을 때 | 체크 |

모든 Collider 컴포넌트에는 is trigger 속성이 있다. 이 속성을 체크하면 충돌 감지는 되지만 물리적인 충돌은 일어나지 않는다. 즉 물체가 서로 부딪혀 정지하거나 튕기는 물리적 현상을 일으키지 않는다.

따라서 Bullet의 Sphere Collider에서 is trigger에 체크하면 물체를 관통하는 효과가 발생한다. 이때 발생하는 충돌 이벤트는 OnTrigger~ 계열의 함수다. 반대로 is trigger 속성에 체크하지 않으면 OnColission~ 계열의 함수가 발생한다.

**Tag 활용**

총알이 Wall에 충돌했을 때 Wall의 관점에서 생각해보면 Collider 컴포넌트를 가진 어떤 게임오브젝트가 날아와서 충돌이 일어났는지 알 필요가 있다. 물론 충돌한 게임오브젝트의 이름으로 판단할 수도 있지만 결코 좋은 방법은 아니다. 따라서 게임오브젝트에 태그를 지정하고 충돌 시 해당 태그값을 식별해 태그값에 따라 적절한 로직을 처리해야 한다.

**OnCollisionEnter 콜백 함수**

OnCollisionEnter 함수는 Collision 타입의 인자를 넘겨준다. Collision은 클래스이며 충돌한 게임오브젝트의 정보, 충돌 위치 및 충돌 속도 등 다양한 정보가 담겨있다. 이러한 정보가 필요하지 않을 때는 인자를 생략할 수 있다.

Void OnCollisionEnter(Collision coll) {} // 충돌이 시작할 때 발생하는 이벤트

Coll 파라미터는 Collision 타입으로 충돌 시 다음과 같은 정보를 담고 있다.

|  |  |
| --- | --- |
| **속성** | **설명** |
| Collider | 충돌한 게임오브젝트의 Collider 컴포넌트(읽기 전용) |
| Contacts | 물체 간의 충돌 지점으로 물리 엔진에 의해 생성 |
| gameObject | 충돌한 게임오브젝트 |
| Impulse | 충돌 시 발생한 충격량 |
| relativeVelocity | 충돌한 두 객체의 상대적인 선 속도(읽기 전용) |
| Rigidbody | 충돌한 게임오브젝트의 Rigidbody 컴포넌트(읽기 전용) |
| Transform | 충돌한 게임오브젝트의 Transform 컴포넌트(읽기 전용) |

**CompareTag 함수**

위 코드에서 사용한 coll.collider.tag == “문자열” 또는

“coll.gameObject.tag == “문자열”과 같이 게임오브젝트의 문자 속성을 가져오는 코드는 실행 시 해당 문자열의 복사본을 생성한다. 이 복사본은 특정 메모리에 할당되면 가비지 컬렉션의 대상이 된다.

가비지 컬렉션은.. 알죠?

따라서 가비지 컬렉션이 처리해야 할 대상이 많아질수록 해제하는 시점에 잠시 멈추는 랙이 발생한다. 이를 위해 CompareTag 함수를 적극 권장함. 비교할 태그명을 tag 인자로 넘겨주면 해당 게임오브젝트의 태그와 비교해 같으면 true 값을 반환함.

**총알 발사 로직**

이제 총알 발사 로직을 구현한다(222p). 개발을 하다보면 게임오브젝트 하위로 차일드화해서 빈 게임오브젝트를 생성해야 할 경우가 빈번하다.

다음 2가지 방법 중 1개를 꼭 숙지하자.

1. 하이러키 뷰에서 부모가 될 게임오브젝트 선택 -> + 눌러 -> Create Empty Child 선택

2. 부모가 될 게임오브젝트 오른클릭 -> 팝업되는 Create Empty Child 생성

**총알의 발사 위치 설정**

모델의 하위에 빈 오브젝트를 생성. FirePos 게임오브젝트는 총알을 발사시키는 위치니까 좀 조정해줘야 함. 참고로 유니티 run 모드 중에서 맞춰놓은 포지션은 초기화되므로 transform 오른클릭 -> copy component 하고, 에딧모드로 돌아와서 오른클릭 -> paste -> component value 해주면 됨.

**기즈모의 활용**

FirePos와 같이 Transform 컴포넌트만 가진 게임오브젝트는 씬 뷰에서 직접 마우스로 클릭하거나 영역을 드래그해서 선택할 수 없다. 따라서 FirePos 게임오브젝트의 위치를 변경하려면 하이러키 뷰에서 FirePos를 선택한 후 Transform 축을 클릭해 옮겨야 하는 불편함이 있다.이처럼 메시가 없는 빈 게임 오브젝트를 어떻게 편하게 컨트롤 하느냐? 바로 사용자 정의 기즈모다.

MyGizmos라는 C# 스크립트 활용.(225p) void OnDrawGizmos()



여기 네모 상자 누르고 기즈모처럼 사용할 수 있다. 레이블은 오브젝트 이름 나오고 밑에 그림들은 아이콘.

**스크립트 작성 방식(228p)**

스크립트를 작성하는 방식은 개발자의 성향에 따라 다를 수 있다. 해당 게임오브젝트가 수행해야 할 로직을 하나의 스크립트에 몰아서 작성하는 개발자도 있고, 특정 기능별로 스크립트를 분리해서 작성하는 개발자도 있다. 두 방식 모두 장단점이 있으나 스크립트를 너무 세분화하는 방식은 나중에 관리하기가 어려워지고, 한 스크립트에 작성하는 방식 또한 스크립트의 소스가 길어져 관리 측면에서 비효율적일 수 있다. 초보자에게는 어려울 수 있으나 구현하려는 로직을 적절히 분리해 스크립트를 작성해야 한다.

생성과 동시에 앞으로 날아가는 총알과 총알이 생성될 좌표인 FirePos가 준비됐다. 이제 마우스 왼쪽 버튼을 클릭할 때마다 총알을 발사해보자. 총알 발사 로직을 기존의 PlayerCtrl 스크립트에 추가해도 관계없지만, 별도의 스크립트로 작성해 구현해보자. FireCtrl 스크립트를 생성해 다음과 같이 작성하고 Player 게임오브젝트에 추가해보자.

**INPUT 클래스**

Input 클래스에서 제공하는 마우스 클릭과 관련된 이벤트는 다음과 같다.

|  |  |
| --- | --- |
| GetMouseButton(int button) | 마우스 버튼을 클릭하고 있을 때 계속 발생 |
| GetMouseButtonDown(int button) | 마우스 버튼을 클릭했을 때 한번 발생 |
| GetMouseButtonUp(int button) | 마우스 버튼을 놓았을 때 한번 발생 |

**총알 발사 궤적 효과 만들기 – Trail Renderer**

궤적을 만들면 타격감 향상 및 보기에 좋다. 책 참고. 어렵진 않음

\* Min Vertex Distance : 생성하는 버텍스 간의 최솟값을 설정하는 것. 값이 작을수록 조밀한 폴리곤을 생성함. 포물선처럼 곡선으로 이동한다면 값을 작게 수정해 부드러운 궤적으로 표현될 수 있게, 직선이면 값을 크게해서 폴리곤의 개수를 줄이고.

**파티클 활용하기**

벽에 총알이 충돌했을 때 불꽃이 튀는 효과를 만들자. 게임에서 화염, 연기, 타격 등의 효과에는 주로 파티클(Particle)을 사용한다. 파티클은 표현하고자 하는 텍스처가 적용된 메시(Mesh)를 입자(Emitter)처럼 생성하고, 그 입자를 이동 및 회전시켜 효과를 표현한다. 그러나 입자가 많아질수록 성능에 큰 부하를 주기 때문에 적절히 사용해야 한다.

에셋 스토어에는 “Unity Particle”이라는 꽁짜 에셋이 있음.

RemoveBullet 스크립트에 다음을 추가한다.

Public GameObject sparkEffect; // 스파크 파티클 프리팹을 연결할 변수

// 스파크 파티클을 동적으로 생성

Instantiate(sparkEffect, coll.transform.position, Quaternion.identity);

충돌 콜백 함수인 OnCollisionEnter에 추가한 Instantiate 함수에서 스파크 파티클을 생성한다. 위치는 총알이 충돌한 시점의 총알 위치로 이동한다. 각도는 Quaternion.identity로 설정해 회전 없이 생성한다.

쿼터니언(Quarternion) 구조체에 관해서는 뒤에서 자세히 소개한대요.

지금은 “유니티에서 사용하는 각도의 단위는 쿼터니언이다” 라고 말할 수 있다.

**충돌 지점과 법선 벡터**

앞서 Instantiate 함수의 3번째 인자를 Quaternion.identity로 설정했다. 이것의 의미는 spark effect 프리팹의 원래 각도 그대로 생성하겠다는 뜻으로, 4개의 벽에 총을 골고루 스파크가 튀는 방향이 항상 일정하다. 따라서 충돌한 지점에서 법선 벡터(Normal Vector)를 구해 해당 방향으로 스파크가 튀게 변경해보자. 법선 벡터는 두 물체가 충돌한 평면 또는 구면의 접점에서 수직 방향으로 바라보는 벡터를 말한다. 그니까 사선에서 벽으로 총을 쏴도 이펙트가 벽과 90도로, 직각으로 생성된다. 243p 참고.

GetContactPoint 함수의 반환 타입은 ContactPoint 구조체 타입으로 다음과 같이 정의됨 :

|  |  |
| --- | --- |
| 속성 | 설명 |
| Normal | 충돌 지점의 법선 |
| otherCollider | 충돌 지점의 다른 Collider |
| Point | 충돌 지점의 위치 |
| Separation | 충돌한 두 Collider의 거리 |
| thisCollider | 충돌 지점의 첫 번째 collider |

**쿼터니언(Quaternion)**

쿼터니언 타입은 유니티에서 사용되는 각도의 단위다. 수학 용어로는 복소수 4차원 벡터

(four-dimensional complex number)라고 한다. 말 그대로 각도를 4개의 원소로 표현한 것으로 인스펙터 뷰를 Debug 모드로 변경하고 Transform 컴포넌트를 보면 4개의 원소(x,y,z,w)로 구성된 것을 볼 수 있음

오일러 각은 수학자 오일러가 고안한 것으로 3차원 공간의 절대 좌표를 기준으로 물체가 얼마나 회전했는지 측정하는 방식이다. 오일러 각을 이용한 회전방식은 x,y,z 축 차례대로 회전시키는데, 회전하는 동안 x,y,z 축 중 2개의 축이 겹쳐지면 어느 축으로도 회전하지 않고 잠기는 현상이 발생한다. 이 현상을 짐벌 락(Gimbal Lock)이라고 한다.

이러한 문제점을 해결하기 위해 쿼터니언이라는 대안이 제시됐다. 쿼터니언은 4차원 복소수로서, 오일러 각이 각축에 대해 차례로 회전시켰다면 쿼터니언은 세 개의 축을 동시에 회전시켜 짐벌락 현상을 방지한다. 따라서 짐벌락을 방지하기 위해 유니티는 모든 객체의 회전을을 쿼터니언을 이용해 처리한다.

**폭발 효과 및 폭발력 생성**

Barrel 모델을 씬에 배치, 총알 발사해서 폭파시키자.

상자 모양 아이콘이 원본, 그냥 육면체가 프리팹이다. 그냥 봐도 구별이 가능해야 함.



**Scale Factor**

외부에서 임포트한 3d 모델은 Transform의 scale 속성을 수정하지 말고, FBX Import Setting의 Scale Factor 속성을 수정한다. 프로젝트 뷰의 Barrel 3D 모델을 선택하면 인스펙터 뷰에 3D 모델의 여러 정보가 표시됨. Scale Factor은 퍼센티지 개념임(0.4 = 40%)

**BarrelCtrl 스크립트 설명**

드럼통에 총알이 3방 명중했을 때 폭발하는 파티클을 생성하고 해당 드럼통을 위로 솟구치게 한다.

하이러키 뷰에서 빈 게임오브젝트를 생성하고, 복사한 바렐들을 거기에 몽땅 넣자. 하이러키 뷰 관리가 편해질 것이다.

**텍스처 변경하기 – Mesh Renderer**

여러 개 배치한 드럼통은 모두 같은 텍스처로 되어있는데, 시작과 동시에 랜덤하게 변경해보자.

텍스처의 적용은 Mesh Renderer 컴포넌트에 연결된 머티리얼에서 지정한다. 여기서 사용한 Barrel 모델의 Mesh Renderer 컴포넌트는 Barrel 하위에 있는 Barrel에 추가돼 있다. 즉, 부모 오브젝트인 Barrel은 빈 오브젝트다.

따라서 하위에 있는 Mesh Renderer를 연결한 변수를 선언하고 미리 연결한 다음 스크립트에서 텍스처만 변경할 수도 있지만, 스크립트에서 직접 연결하는 방식으로 구현해보자.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

클래스 선언부에 선언한 renderer 변수는 유니티의 Component.renderer로 정의된 멤버 변수로서 new 키워드를 사용하지 않으면 CS0108 경고 문구를 표시함.

New 키워드는 상속받은 Base 멤버 변수가 아니라 새롭게 선언한 변수라는 의미.

**폭발력 적용하기 – AddExplosionForce**

드럼통 로직은 구현했다. 실제로 폭발하면서 주변의 물체까지 폭발력을 전달해 간접적으로 데미지를 주는 로직을 구현한다.

1. 폭발하는 드럼통 주변의 물체가 어떤 것인지 알아야 한다. 지금까지 배웠던 범위에서 생각해본다면 드럼통에 Sphere collider를 추가해 충돌된 물체가 무엇인지 알 수 있지만, 폭발하는 순간 주변의 물체를 추출하는 방식으로 구현해보자. 또한, 드럼통만 데미지를 적용한다.

2. 레이어(Layer)를 추가한다. 유니티에서의 레이어 == Group. 레이어는 다양한 로직이나 기능에 활용할 수 있다. 예를 들어 스크립트 로직이 특정 레이어에만 반응하게 하거나 조명이 특정 레이어에만 비치게 하는 식으로. Tag나 레이어는 대문자로 표기하길 권장.

3. 배럴이 현재 씬에 여러 개 있는데 프리팹 원본을 수정했기 때문에 모든 복사본 배럴의 레이어는 BARREL로 구분된다. 하위에 있는 게임오브젝트까지 레이어를 지정할 것인지 묻는 팝업창이 나오면 판단 후 선택한다. 현재의 경우 적용할 필요 없다.

폭발할 드럼통 주위에 어떤 드럼통이 있는지 판단할 때엔 Physics.OverlapSphere 함수를 사용한다. 반경 몇 m 이내에 근접해있는 드럼통을 가져올 수 있다.

Physics.OverlapSphere 함수의 반환 값은 조건에 맞고 해당 범위에 들어온 모든 Collider 컴포넌트가 배열에 담겨 넘어온다. 따라서 foreach 구문을 이용해 배열의 처음부터 마지막까지 순회하면서 폭발력을 적용한다.

폭발력을 생성하는 Rigidbody, AddExplosionForce 함수의 인자는 다음과 같다 :

            rb.AddExplosionForce(횡 폭발력, 폭발 원점, 폭발 반경, 종 폭발력);

**OverlapSphereNonAlloc**

Physcis.OverlapSphere 함수는 실행 시 Sphere 범위에 검출될 개수가 명확하지 않을 때만 사용해야 한다. 메모리 가비지가 발생하기 때문. 따라서 Sphere 범위에 검출될 개수가 명확할 때는 가비지가 발생하지 않는 Physics.OverlapSphereNonAlloc 함수를 사용하길 권장.

            Physics.OverlapSphereNonAlloc(pos, radius, colls, 1<<3);

**오디오**

오디오는 매우 중요함. 대부분의 오디오 파일(AIFF, WAV, MP3, OGG 등)을 지원한다.

리버브(Reverb), 에코(Echo), 필터링(Filtering)과 같은 음원효과도 낼 수 있음

**AudioListener, AudioSource 컴포넌트**

총알 발사 시 효과음 추가하자.

AudioListener : 소리를 듣는 역할. 씬에서 반드시 한 개만 존재해야 함

AudioSource : 소리를 발생시키는 역할. 여러 개 있어도 됨

\* 한 씬에 카메라가 여러 대 있을 경우, 1개 카메라를 정해서 그 오디오 리스너만 사용하자. 성능 저하를 경험할 수 있음

오디오 파일 임포트 -> 인스펙터 창에서 각종 설정을 확인할 수 있음.

오디오 파일은 2개의 채널로 이루어져 있다. 즉 스테레오 음원으로 음향 효과를 극대화한 게임이 아니라면 모노 음원으로 변환하는 게 좋다. 특히 모바일 게임에서 스테레오 사운드는 성능 저하의 원인이 될지도? Force to Mono를 체크하면 강제로 모노 사운드가 됨

**오디오 임포트 옵션 – Compression Format**

압축 포맷은 오디오 파일의 용도에 따라 다르게 설정해야 한다. 유니티에서 지원하는 오디오 압축 포맷으로는 PCM, ADPCM, Vorbis/MP3가 있다.

|  |  |
| --- | --- |
| 오디오 압축 포맷 | 설명 |
| PCM | 품질이 높은 대신 파일의 크기는 상대적으로 크기 때문에 짧은 오디오에 적합하다(비압축 포맷). |
| ADPCM | 압축비가 PCM 대비 3.5배이기에 파일의 크기가 작지만, CPU 자원은 좀 더 사용한다. 노이즈가 포함돼 있어서 총소리, 발걸음 소리와 같이 잡음이 일부 포함된 오디오에 적합하다. |
| Vorbis / MP3 | 중간 길이 정도의 효과음 또는 배경음악에 적합하다. PCM 대비 품질은 떨어지며 압축률(Quality)을 조절할 수 있다. 보통 70% 정도 설정해서 쓰는 듯. |

배경음과 환경음은 대부분 큰 파일이기 때문에 실행할 때 미리 압축을 풀어서 메모리에 보관하면 가용 메모리가 줄어든다. 따라서 큰 파일의 오디오는 다음과 같은 임포트 설정을 권장한다. 특히 Compressed In Memory를 사용할 경우 압축해 메모리에 올리는 것으로 품질을 떨어뜨리면 파일 크기를 줄일 수 있다. 따라서 귀로 들어보고 음질의 손상이 없는 범위 내에서 적절하게 Quality 속성값을 설정하기를 권장함.

|  |  |  |
| --- | --- | --- |
| **Load Type** | **Compression Format** | **Quality** |
| Streaming | Vorbis |  |
| Compressed In Memory | Vorbis | 70% |

(배경음악 또는 환경음의 임포트 권장 설정법)

또한 오디오의 발생 빈도나 오디오 파일의 크기에 따라 다음 표와 같은 임포트 설정을 권장함

|  |  |  |  |
| --- | --- | --- | --- |
| **발생 빈도** | **파일의 크기** | **Load Type** | **Compression Format** |
| 빈번히 발생 | 작은 크기 | Decompress On Load | PCM 또는 ADPCM |
| 빈번히 발생 | 중간 크기 | Compressed In Memory | ADPCM |
| 가끔 발생 | 작은 크기 | Compressed In Memory | ADPCM |
| 가끔 발생 | 중간 크기 | Compressed In Memory | Vorbis |

Preload Audio Data 속성은 씬이 로딩될 때 씬에서 사용하는 모든 오디오 클립이 미리 로딩된다. 기본값은 체크돼 있으며 만약 언체크돼 있으면 처음 오디오를 재생할 때 로딩되기 때문에 약간의 레이턴시가 발생할 수 있다.

**총소리 구현!!**

Sound Source 속성들은 다음과 같다 :

|  |  |  |
| --- | --- | --- |
| **속성** | **설명** | **설정값** |
| Audio Clip | 발생시킬 음원 파일 | P\_m4\_1.wav |
| Play On Awake | 해당 컴포넌트가 활성화될 때 자동 재생 여부 | 체크 |
| Min Distance | 볼륨 100% 값으로 음원이 들리는 영역 범위 | 5 |
| Max Distance | 음원이 들리는 최대 영역 범위 | 10 |

Min, Max Distance 범위를 확인하고 싶으면 프리팹 Bullet 더블클릭해서 봐라.

메인 카메라에 있는 Audio Listener가 AudioSource의 Min Distance 범위에 있을 때는 100%의 볼륨 크기로 들리며, Min Distance부터 Max Distance 사이에 있을 때는 Volume Rolloff 옵션에 따라 볼륨이 서서히 감소한다. 그리고 AudioListener의 위치가 Max Distance를 넘어서면 소리가 50%의 볼륨 크기로 들린다.

Volume Rolloff 옵션은 Audio Source가 멀어질수록 볼륨이 감소하는 세 가지 프리셋(Preset) 옵션을 제공한다. 또한 하단의 커브드 에디터(Curved Editor)를 이용해 감소 효과의 세부적인 조정이 가능하다.

자.. 총알에 총소리를 적용시켰는데, 총알 오브젝트가 빨리 없어지면 소리도 같이 삭제되는 문제가 발생한다. 그래서 캐릭터에게 Sound Source 컴포넌트를 적용할 것이다.

// 반드시 필요한 컴포넌트를 명시해 해당 컴포넌트가 삭제되는 것을 방지하는 어트리뷰트

[RequireComponent(typeof(AudioSource))]

어떤 스크립트에서 반드시 함께 있어야 하는 컴포넌트를 자동으로 추가하고, 실수로 의존성이 있는 컴포넌트를 삭제하는 것을 방지하기 위해 RequireComponent 어트리뷰트를 사용한다.

또한, FireCtrl에 설정한 RequireComponent 어트리뷰트 때문에 이미 추가된 AudioSource 컴포넌트를 삭제하려 한다면 “스크립트에서 참조하고 있는 컴포넌트는 삭제할 수 없다!”라는 메시지가 나온다! 안전빵인 거시다.

        // 총소리 발생

    audio.PlayOneShot(fireSfx, 1.0f);

// AudioSource.PlayOneShot(오디오 클립, 볼륨);

…

총소리를 발생시키는 로직은 Fire 함수에서 처리하며 AudioSource 컴포넌트의 PlayOneShot 메소드를 사용한다.

이렇게 하면 총알이 없어져도 총소리는 끊기지 않는다.

**총구 화염 효과 – Muzzle Flash**

자, 이제 총구 화염 효과를 만들자.

1. 총 끝(FirePos)에 Quad 차일드 오브젝트 만들기

2. 콜라이더 삭제(충돌 필요없으니까), Mesh Renderer에서 그림자랑 조명 옵션 끄기

3. 총구 화염 텍스처를 붙여넣기. 이쯤되면 이해되겠지? 클릭할 때마다 보이게 하는거다.

4. 총구 화염 이미지를 Quad에 붙여넣는다.

5. 셰이더 속성을 Mobile -> Particles -> Additive로 변경. 타일링 속성 xy 각 0.5로

즉 Offset 속성을 (0.5, 0.5)로 하면 오른쪽 위에 있는 이미지가 표현된다. 289p 참고

6. 자.. 이제 스크립트 타임. 290p 참고

    // Muzzle Flash의 MeshRenderer 컴포넌트

    private MeshRenderer muzzleFlash;

  // FirePos 하위에 있는 MuzzleFlash의 Material 컴포넌트를 추출!

        muzzleFlash = firePos.GetComponentInChildren<MeshRenderer>();

        // 처음 시작할때 비활성화

        muzzleFlash.enabled = false;

**자… 드디어 코루틴 함수에 대해 배운다…**

유니티는 게임을 실행하면 모든 스크립트에 일종의 메시지 루프(Message Loop)가 동작한다. 메시지 루프란 유니티의 다양한 이벤트 함수가 정해진 순서대로 실행되는 순환 구조를 말한다. 일반적인 함수를 호출하면 해당 함수 안의 로직을 다 수행해야만 실행이 끝난다.

만약 함수 안에서 수행하는 로직이 10초 정도의 시간이 걸린다고 가정하면 10초 동안 메시지 루프의 다른 로직을 실행할 수 없다는 것을 의미한다. 즉, 게임이 10초 동안 멈추는 현상이 발생한다. 다른 용어로는 이를 블로킹(Blocking)이라고 한다. 그리고 메시지 루프의 다른 함수가 정상적으로 실행되면서 시간이 오래 걸리는 함수를 병렬로 호출하는 것을 멀티스레드(Multi thread)라고 한다.

유니티에서는 멀티 스레드와 같이 비동기(Asynchronous)로 처리해야 하는 로직을 구현하기 위해 멀티스레드와 유사한 코루틴(Coroutine)을 제공한다.

코루틴을 협력(Co-) + 루틴(routine)의 합성어로 생각하면 “협력 루틴” 으로 생각할 수 있음.

즉 메시지 루프와 코루틴이 서로 번갈아가면서 로직을 수행한다고 볼 수 있다!

코루틴 함수는 열거형 함수 Enum과 비슷한 듯.

Ienumerator 타입으로 선언해야하고, 함수 내에 하나 이상의 yield 키워드를 사용해야 한다.

IEnumerator 함수명(인자1, 인자2 ...) {

    [로직1] // 로직 1을 실행

    yield return null; // 코루틴 함수를 대기(정지)시키고 메시지 루프로 제어권 양보

    [로직2] // 로직 2를 실행

    yield return null; // 마찬가지

[로직 3] // 로직3 실행 후 종료

}

코루틴 함수는 일반 함수를 호출하듯 함수명으로 호출할 수 없다. 다음과 같이 StartCoroutine() 함수를 이용해 호출한다. 코루틴 함수를 호출할 때 전달하는 인자는 함수의 원형(포인터)를 사용한다.

호출할 코루틴 함수명을 문자열로 전달할 수도 있지만 그럴 경우 가비지 컬렉션이 발생하며 문자열로 호출한 코루틴은 개별적으로 정지시킬 수 없다는 문제가 있다. 따라서 문자열로 전달하는 방식을 웬만해선 쓰지 말자.

스타크래프트의 wait() 과 비슷한 것 같다…… 이것도 터보트리거 비슷한 게 있을지도????

**MuzzleFlash의 텍스처 오프셋 변경**

이제 Muzzle에 연결된 텍스처를 불규칙하게 변경해봅시다. 또한 불규칙적인 각도로 화전시키고 크기도 조절하면 좀 더 실감날 듯.

텍스처의 오프셋 값은 SetTextureOffset 함수와 mainTextureOffset 속성으로 설정할 수 있음

여기 좀 어렵다.

    IEnumerator ShowMuzzleFlash() {

        // 오프셋 좌푯값을 랜덤 함수로 생성

        Vector2 offset = new Vector2(Random.Range(0, 2), Random.Range(0, 2)) \* 0.5f;

        // 텍스처의 오프셋 값 설정

        muzzleFlash.material.mainTextureOffset = offset;

자. 텍스처의 오프셋 값을 수정하기 위해 mainTextureOffset 속성을 사용했다. 텍스처의 오프셋은 Vector2 타입으로 (x, y) 값을 갖는다. 그면 4가지 값을 갖게 되겠죠. 참고로 0~2가 아니고 0~1임

그래서 0.5를 곱하면?

0-0, 0.5-0, 0-0.5, 0.5-0.5 값이 나오죠? 그러면 4가지 오프셋 중 랜덤한 게 나오겠죠?

        // MuzzleFlash의 회전 변경

        float angle = Random.Range(0, 360);

        muzzleFlash.transform.localRotation = Quaternion.Euler(0, 0, angle);

자 muzzleFlash는 FirePos의 아들이잖슴? 그래서 좌표, 회전각도, 스케일을 수정하려면 무조건 localPosition / ~Rotation / ~Scale속성을 써야함. MuzzleFlash를 회전시키기 위한 localRotation 속성은 Quaternion 타입이니까네 오일러 각을 쿼터니언으로 변환할라면

Quaternion.Euler(x,y,z) 함수를 쓴다. 먼저 0도부터 360도 각 중 난수로 생성한 값을 이용함.

        // MuzzleFlash의 크기 조절

        float scale = Random.Range(1.0f, 2.0f);

        muzzleFlash.transform.localScale = Vector3.one \* scale;

스케일 역시? localScale이겠죠? 왜냐? 자식이니까~ 여기에 사용된 Vector3.one은 (1,1,1)을 나타냄.

Vector3.one \* 1.5f는 Vector3(1.5, 1.5, 1.5)랑 똑같다; 왜 이케쓰지; one땜에 헷갈릴뻔

따라서 Vector3.one에 곱하는 스칼라(Scala) 값을 난수로 생성한다.

스칼라? 스칼라는 방향을 갖지 않고 크기만 갖는 개념의 수임……………ㅅㅂ?

        // MuzzleFlash 활성화

        muzzleFlash.enabled = true;

        // 0.2초 동안 대기(정지)하는 동안 메시지 루프로 제어권 양보

        yield return new WaitForSeconds(0.05f);

        // MuzzleFlash 비활성화

        muzzleFlash.enabled = false;

    }

**코루틴의 응용 – 임계치**

예제 게임을 시작할 때마다 에디트 모드에서 주인공 캐릭터가 바라보는 방향이 불규칙하게 변경된다. 이런 현상은 시작할 때 마우스의 좌우 이동값이 불규칙하게 넘어와서 발생하는 현상이다. 따라서 테스트를 쉽게 하기 위해 Start 함수를 코루틴 함수로 변경시켜 해결해보자. PlayerCtrl 스크립트 변경. 303p 참고

여기서 5장이 마무리된다…..