



목차

#01 준비하기	01
● 1. 프로그램 설치	
+ 개발 도구	
+ Visual Studio 설치	
+ Unity 설치	02
● 2. 프로젝트 관리	03
+ 프로젝트 만들기	
+ 레이아웃 설정	04
#02 구성과 조작	05
● 1. 윈도우 종류와 역할	
+ ① Hierarchy	
+ ② Scene	
+ ③ Game	
+ ④ Inspector	
+ ⑤ Project	
+ ⑥ Console	
+ 그 외...	06
● 2. Scene 조작	
+ 도구	
- 1. Hand Tool (단축키: Q)	
- 2. Rect Tool (단축키: T)	
+ 시점 조작	
- 확대/축소	
- 이동	
● 3. 게임 오브젝트	07
+ 구성	
+ 태그	
- 태그 관리	
+ 레이어	08
- 레이어 관리	
+ 컴포넌트	
+ 오브젝트 만들기	
● 4. 파일	09
+ 파일 만들기	
- 폴더	
- C# 스크립트	
- Scene	
+ 파일 추가	
+ 파일 관리	10
- 스프라이트 설정	
- 스프라이트 에디터	

#03 게임 만들기	11
● 1. 게임 오브젝트 종류	
+ 빈 오브젝트	
+ 카메라	
+ 스프라이트	
+ UI	
● 2. 게임 오브젝트 배치	12
+ 부모와 자식	
+ 프리팹	
● 3. 컴포넌트 종류	13
+ Transform	
+ Camera	
+ Sprite Renderer	
+ Rigidbody 2D	14
+ Collider 2D	
- 콜라이더와 트리거	
● 4. 스크립트 제작	15
+ 변수	
+ 생명주기	16
- Start()	
- Update()	
- 충돌과 트리거	
+ 입력받기	17
+ 컴포넌트 가져오기	18
- Transform 조작	
+ Vector	19
● 5. 플레이어 만들기	
+ 이동 구현	
- 바라보기	22
● 6. 맵 만들기	23
+ 배치	
+ 콜라이더	24
● 7. 다른 오브젝트 만들기	25
+ 상자	
+ 동전과 치즈	
- Scene 관리	26
+ 열쇠와 문	27
+ 대포	28
- 오브젝트 생성	29
- 코루틴	
+ 맵 꾸미기	31
- 카메라 따라가기	
● 8. UI	32
+ UI 오브젝트	
- 캔버스	
- 텍스트	33
- 이미지	
- 버튼	34
+ 엔딩 화면	35
+ 시작 화면	37
● 9. 마무리	38
+ 빌드	

#01 준비하기

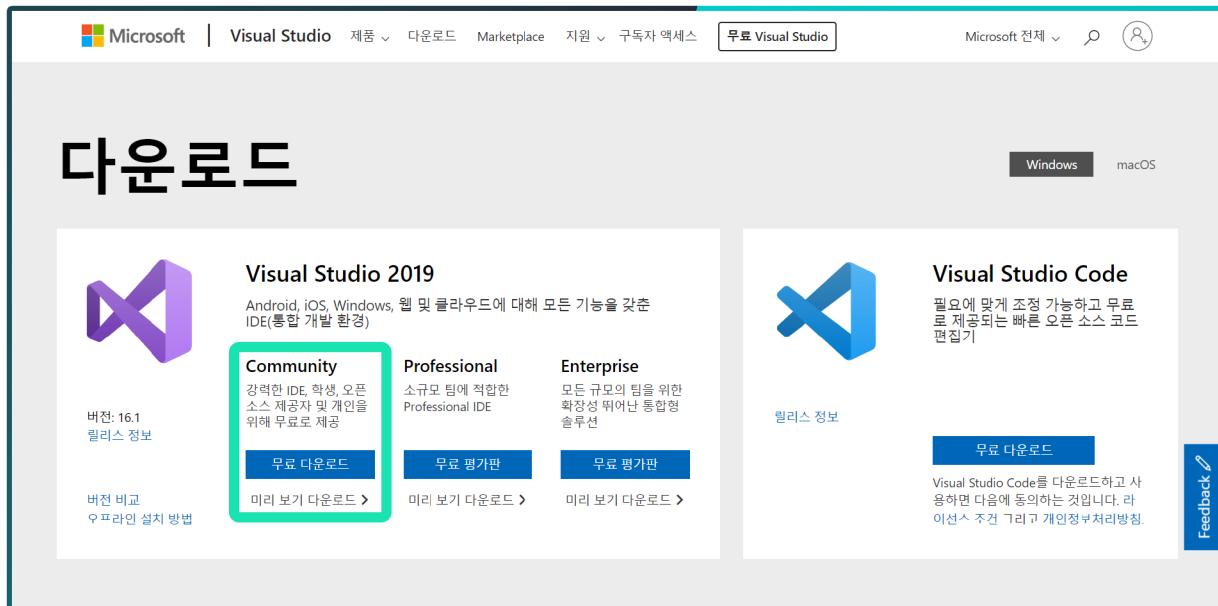
1. 프로그램 설치

+ 개발 도구

프로그램을 개발할 때 필요한 도구들을 개발 도구라고 합니다. 게임을 개발할 때 필요한 도구들로는 게임 엔진과 코드 에디터 등이 있습니다.

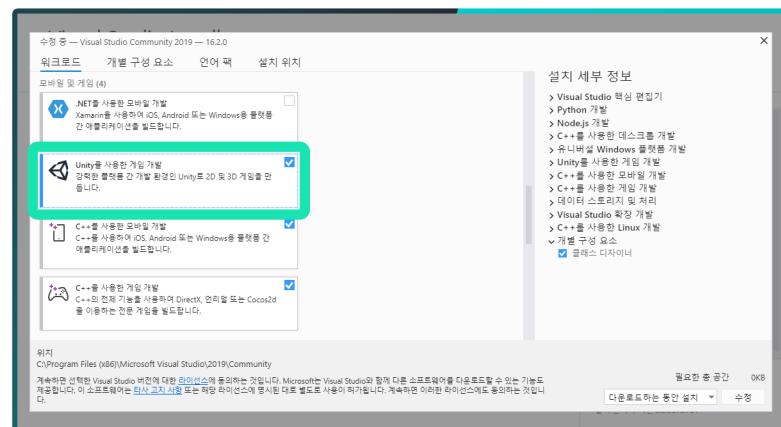
게임 엔진은 Unity나 Unreal 등 게임에 필요한 기능들을 제공하는 도구입니다. 코드 에디터는 코드 작성과 편집을 도와주고 보기 좋게 정리해주는 도구입니다. 우리는 Unity 엔진과 Visual Studio를 사용합니다.

+ Visual Studio 설치



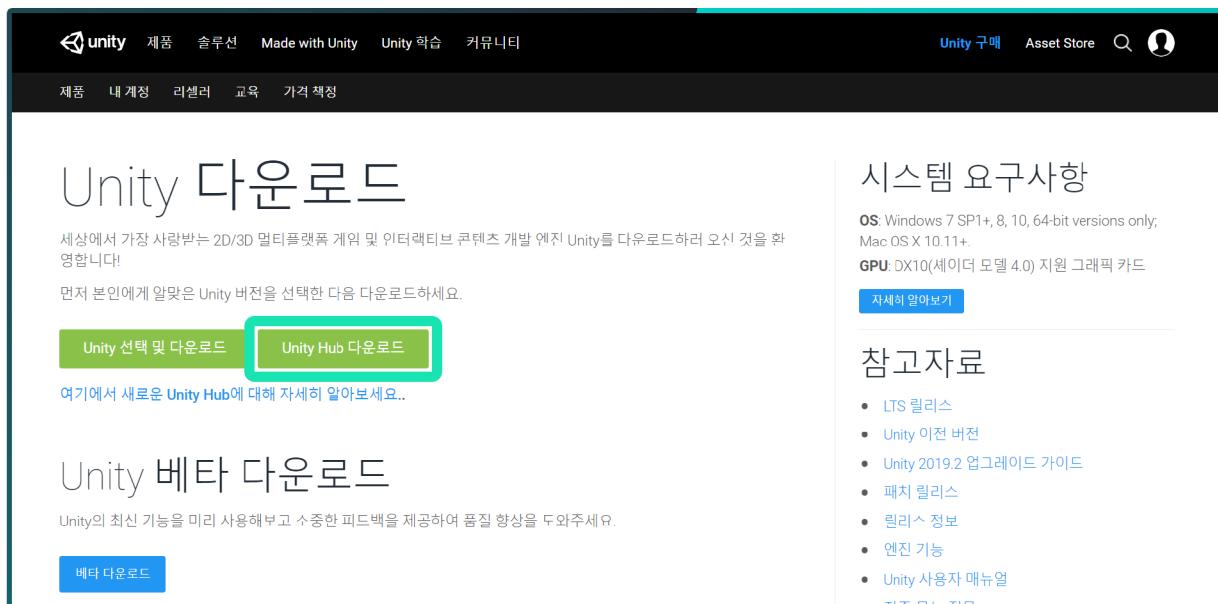
Visual Studio 설치 사이트(visualstudio.microsoft.com/ko/downloads/)

1. 'Visual Studio 설치'를 검색하거나 위 주소를 통해 설치 사이트로 이동합니다.
2. Community 버전을 다운로드한 후 다운로드한 파일을 클릭해 실행합니다.
3. 'Unity를 사용한 게임 개발'을 선택하고 설치합니다.



+ Unity 설치

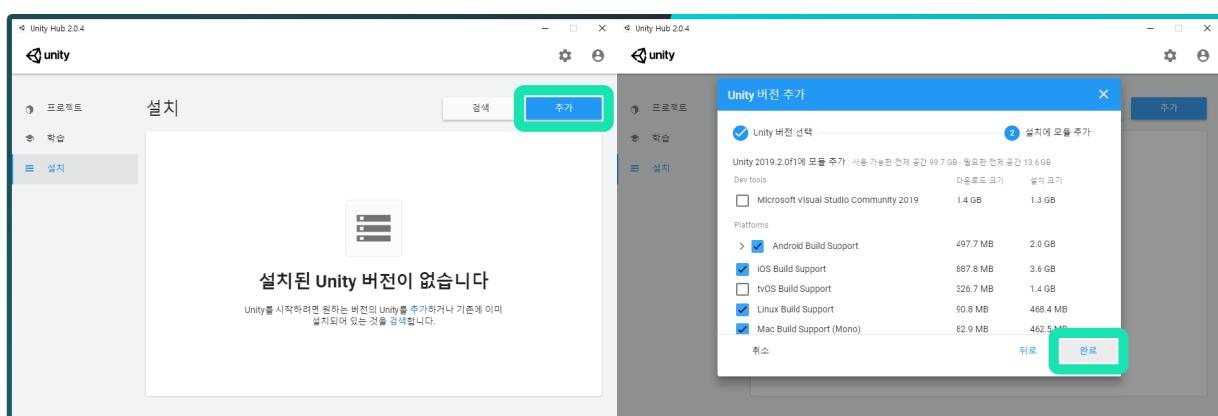
Unity는 Hub를 이용해 프로젝트와 버전들을 관리합니다. 원하는 버전의 엔진이 있다면 허브에서 추가로 설치해야 됩니다.



The screenshot shows the Unity Hub download page. At the top, there's a navigation bar with links for 제품 (Products), 슬루션 (Solutions), Made with Unity, Unity 학습 (Unity Learning), 커뮤니티 (Community), Unity 구매 (Buy Unity), Asset Store, and a search icon. Below the navigation is a secondary menu with links for 제품 (Products), 내 계정 (My Account), 리셀러 (Reseller), 교육 (Education), and 가격 책정 (Pricing). The main content area has a large title 'Unity 다운로드' (Unity Download) and a sub-section 'Unity 베타 다운로드' (Unity Beta Download). A green button labeled 'Unity Hub 다운로드' is highlighted with a red box. To its left is another button 'Unity 선택 및 다운로드' (Select and Download). Below these buttons is a link '여기에서 새로운 Unity Hub에 대해 자세히 알아보세요.' (Learn more about the new Unity Hub here). On the right side, there's a section titled '시스템 요구사항' (System Requirements) with details for OS (Windows 7 SP1+, 8, 10, 64-bit versions only, Mac OS X 10.11+), GPU (DX10(제이더 모델 4.0) 지원 그래픽 카드), and a '자세히 알아보기' (Learn more) button. Below this is a '참고자료' (Reference) section with links to LTS 릴리스, Unity 이전 버전, Unity 2019.2 업그레이드 가이드, 패치 릴리스, 릴리스 정보, 엔진 기능, Unity 사용자 매뉴얼, and 저작권 및 저작권.

└ Unity 설치 사이트(unity3d.com/kr/get-unity/download)

1. 'Unity 다운로드'를 검색하거나 위 주소를 통해 설치 사이트로 이동합니다.
2. Unity Hub를 다운로드한 후 다운로드한 파일을 클릭해 실행합니다.
3. '설치'에서 '추가'를 눌러 원하는 버전을 선택한 후 원하는 플랫폼을 선택해 설치합니다. (Visual Studio는 앞에서 설치했으므로 설치하지 않고 넘어갑니다)

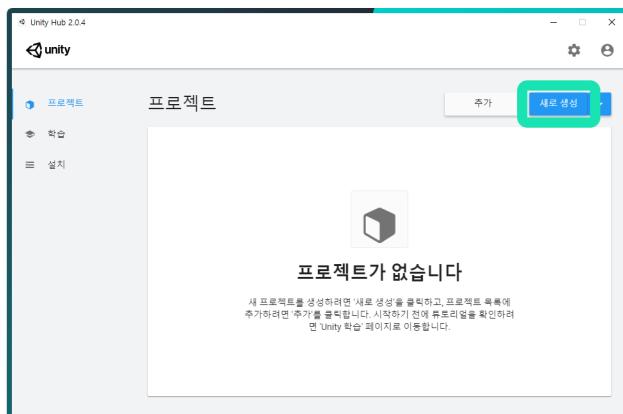


! 이 책은 Unity 2019.2.0f1 버전과 Visual Studio 2019를 사용했습니다. 다른 버전과 차이가 있을 수 있습니다.

2. 프로젝트 관리

+ 프로젝트 만들기

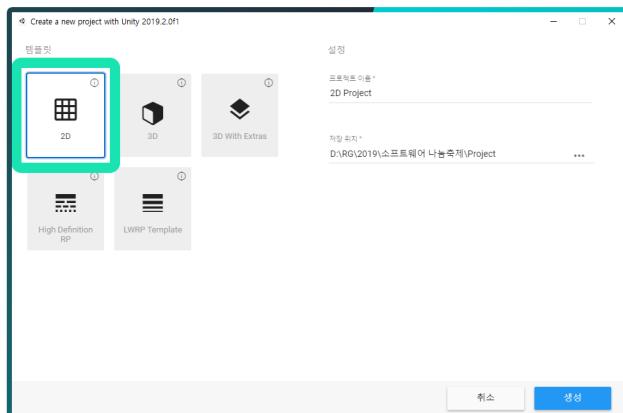
유니티 프로젝트는 크게 3D와 2D로 나뉩니다. 우리는 2D 게임을 제작할 것이기 때문에 2D 프로젝트로 제작하겠습니다.



새로 생성 버튼을 눌러 프로젝트를 만들 수 있습니다.

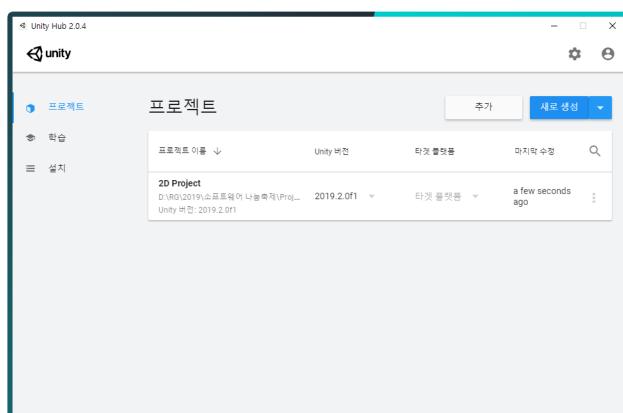
다른 버전의 프로젝트를 만들고 싶다면 오른쪽의 삼각형 버튼을 눌러 버전을 선택하세요.

이미 있는 프로젝트를 불러오고 싶다면 왼쪽의 추가 버튼을 눌러주세요.



먼저 좌측에서 원하는 템플릿을 고릅니다. 우리는 2D 템플릿을 사용합니다.

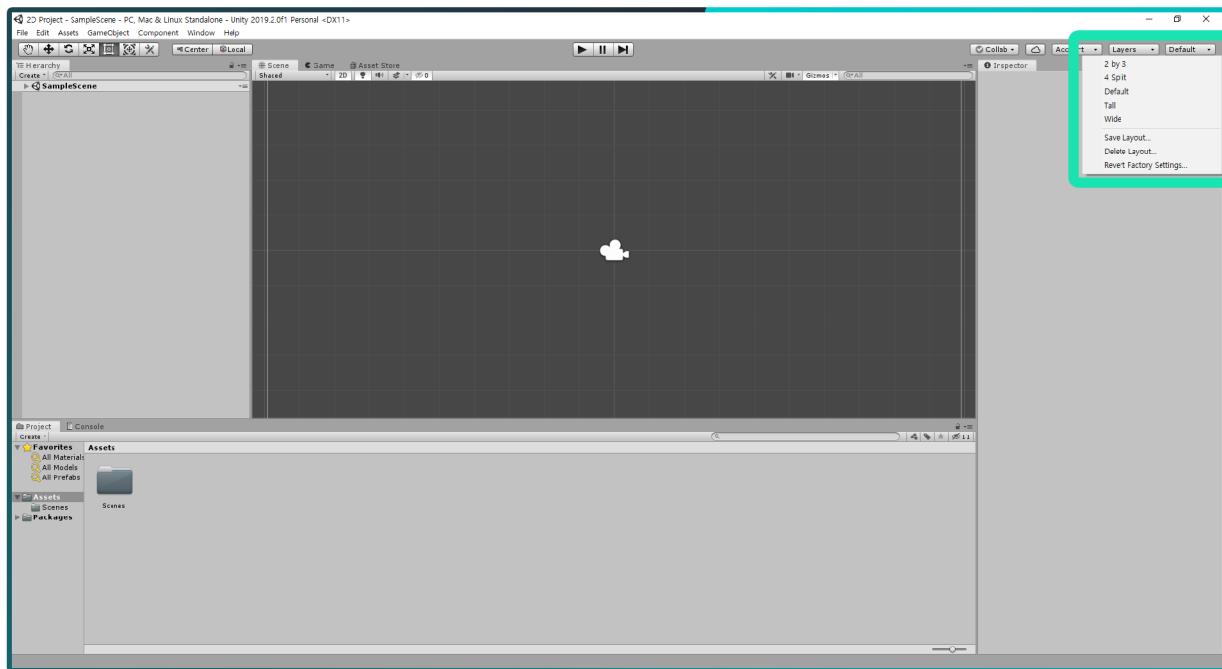
프로젝트 이름과 저장 경로를 확인하고 생성 버튼을 눌러 프로젝트를 만듭니다.



다시 허브에 들어가면 지금까지 만든 프로젝트들을 볼 수 있습니다.

버전을 바꾸면 켜지는 버전이 바뀌지만 버그가 있을 수 있으니 백업 후 진행해주세요.

+ 레이아웃 설정



Unity 에디터에서는 여러 윈도우들을 오가며 게임을 제작하게 됩니다. 윈도우가 많은 만큼 자신의 프로젝트에 맞는 레이아웃을 만드는 것이 좋습니다.

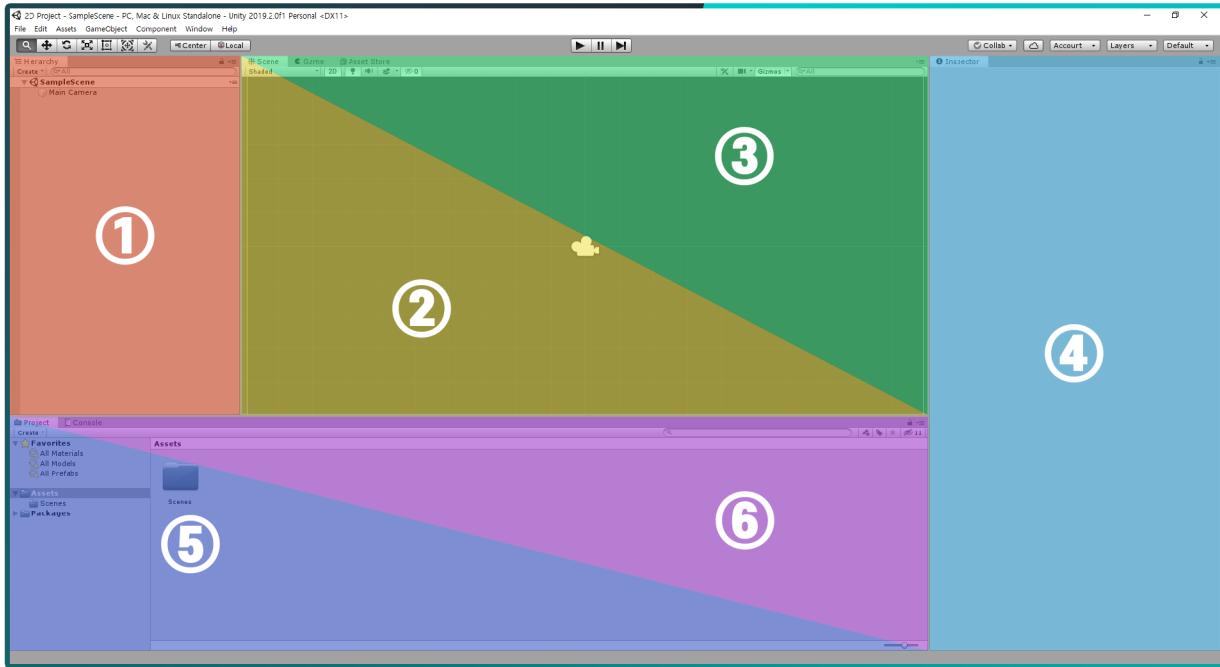
에디터를 처음 킨다면 기본 레이아웃으로 설정되어 있습니다. 윈도우를 드래그해 옮겨 편한 레이아웃을 만들 수 있습니다. 만약 레이아웃 배치가 힘들다면 우측 상단에 버튼을 클릭하거나 'Window > Layouts'에서 미리 만들어진 레이아웃 중 하나를 골라보세요.

추가로 필요한 윈도우가 있다면 'Window'에서 원하는 윈도우를 골라 만드세요. 필요한 윈도우와 역할을 알고 싶다면 [#02-1 챕터](#)를 참고해주세요.

! 'Save Layout' 버튼을 누르면 지금 레이아웃 배치를 저장할 수 있습니다. 자신만의 배치를 저장해 편리하게 사용하세요.

#02 구성과 조작

1. 윈도우 종류와 역할



+ ① Hierarchy

Scene과 Scene 안의 게임 오브젝트들을 계층적으로 보여줍니다. 게임 오브젝트를 클릭하면 ④ Inspector에서 그 오브젝트의 정보를 볼 수 있습니다.

+ ② Scene

Scene의 모습을 자유 시점에서 보여줍니다. 게임 오브젝트들의 위치를 쉽게 수정해 배치할 수 있습니다.

+ ③ Game

실제 게임에 보이는 모습, 카메라가 보고 있는 모습을 보여줍니다. 상단의 재생 버튼을 누르면 전환되고 게임 테스트를 할 수 있습니다.

+ ④ Inspector

프로젝트나 설정, 게임 오브젝트 등의 정보를 보여줍니다. 현재 선택한 항목의 정보를 보여주며 정확한 수치로 정보를 수정할 수 있습니다.

+ ⑤ Project

프로젝트에 필요한 리소스 등의 파일들이 표시해줍니다. 탐색기와 비슷하게 동작하며 많은 스크립트와 리소스들을 폴더로 관리할 수 있습니다.

+ ⑥ Console

디버그 메시지나 오류, 경고 메시지를 보여줍니다. 테스트할 때 사용됩니다.

+ 그 외...

지금 소개한 윈도우 말고도 애니메이션 효과를 담당하는 Animator와 Animation, 많은 사람들이 만들어둔 부가 기능들을 받아쓸 수 있는 Asset Store, Unity에서 유용한 Asset을 빠르게 받을 수 있는 Package Manager 등 다양하게 있습니다. 필요할 때 'Window'에서 찾아 사용하세요.

● 2. Scene 조작

+ 도구



에디터 좌측 상단에는 오브젝트나 시점을 조작하는 여러 도구들이 나열되어있습니다. 상황에 맞는 도구를 골라 사용하면 됩니다. 도구를 단축키로 고를 수도 있는데 왼쪽부터 Q, W, E, R, T, Y 키로 바로 전환할 수 있습니다. 2D 프로젝트에서는 주로 1번과 5번 도구만 사용하므로 이 도구들만 다루겠습니다.

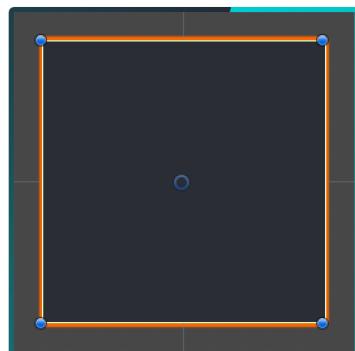
- 1. Hand Tool (단축키: Q)

Scene의 시점을 옮길 수 있는 도구입니다. 커서가 손 모양으로 바뀌며 드래그하면 Scene을 자유롭게 둘러볼 수 있습니다. 오브젝트를 클릭해도 선택되지 않습니다.

- 5. Rect Tool (단축키: T)

오브젝트를 선택할 수 있고 선택한 오브젝트를 드래그해 위치를 옮기거나 크기, 각도 등을 조절할 수 있는 도구입니다.

꼭짓점의 원이나 모서리를 클릭하고 드래그하면 크기를 조절할 수 있고, 꼭짓점과 가까운 지점에서 커서에 회전 표시가 뜨면 각도를 조절할 수 있습니다. 또, 가운데 원의 위치를 옮겨 오브젝트의 기준점을 바꿀 수 있습니다.



+ 시점 조작

2D 시점은 3D 시점과 달리 회전이 없습니다. 오브젝트를 다른 각도에서 보고 싶다면 3D 시점으로 변경해주세요.

- 확대/축소

마우스 휠을 사용해 확대와 축소가 가능합니다. 현재 마우스 커서를 기준으로 확대/축소가 됩니다. Alt 키를 누른 상태로 휠을 굴리면 Scene 화면 가운데를 기준으로 확대/축소가 됩니다.

- 이동

우클릭으로 드래그하면 사용중인 도구와 상관없이 오브젝트를 선택하지 않고 Scene을 둘러볼 수 있습니다.

3. 게임 오브젝트

+ 구성

'Main Camera' 오브젝트를 선택하면 Inspector 창에 오른쪽과 같은 정보가 표시됩니다.

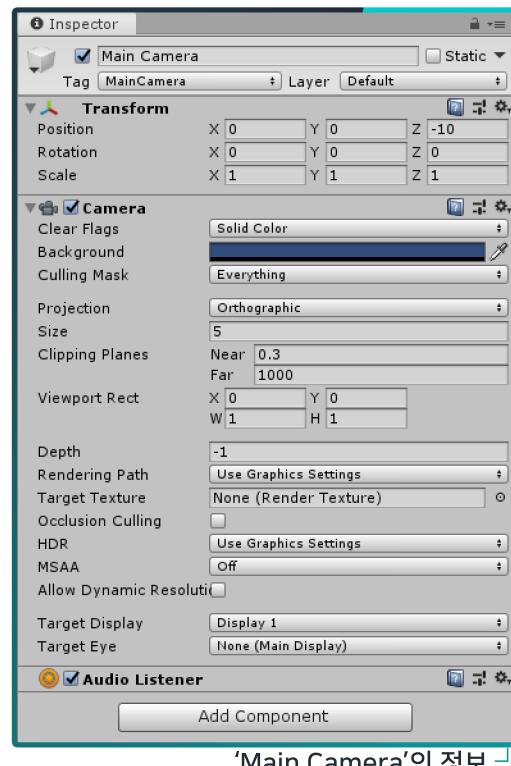
가장 위에는 오브젝트의 이름이 표시됩니다. 이름은 오브젝트를 구분하는데 사용됩니다. Hierarchy 창에서 표시되는 이름이기도 합니다.

이름 왼쪽의 체크 박스는 오브젝트의 활성 여부를 표시합니다. 오브젝트가 비활성화되면 Scene에서 보이지 않게 됩니다.

이름 밑에는 태그와 레이어가 표시됩니다. 태그와 레이어는 오브젝트를 구분하는데 사용됩니다.

태그와 레이어 밑에는 컴포넌트들이 표시됩니다. 모든 오브젝트들은 컴포넌트의 조합으로 구성됩니다. 미리 만들어진 많은 컴포넌트들을 조합하면 다양한 오브젝트를 만들 수 있습니다.

컴포넌트 중 'Transform'은 모든 오브젝트에 필수로 들어가는 컴포넌트입니다.



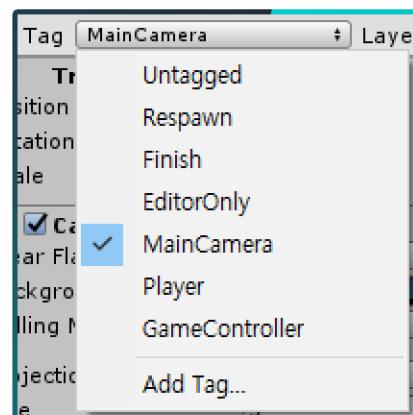
'Main Camera'의 정보

+ 태그

태그는 오브젝트에 붙여줄 수 있는 참고 단어입니다. 예를 들어 플레이어가 직접 조종하는 오브젝트에 'Player' 태그를 붙여 구분합니다.

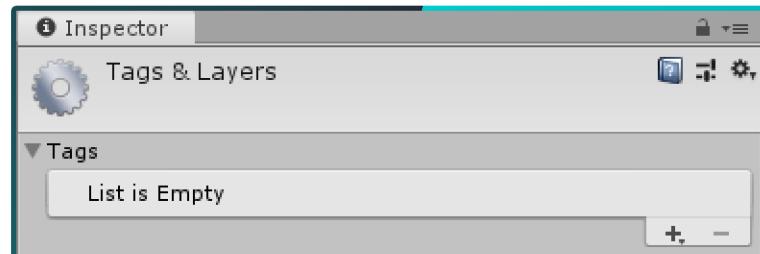
태그가 붙어있는 오브젝트는 스크립트에서 더 쉽게 찾을 수 있습니다.

태그를 클릭하면 태그 목록이 나타나면서 태그를 선택할 수 있게 됩니다. 새로운 태그를 추가하고 싶다면 목록 하단의 'Add Tag...'를 클릭해 'Tags & Layers'에 들어가 추가하면 됩니다.



- 태그 관리

'Tags'를 열어 태그 리스트를 확인합니다. 리스트 우측 하단의 '+' 버튼을 클릭해 새 태그를 추가합니다. 지우고 싶은 태그를 클릭한 후 '-' 버튼을 누르면 지울 수 있습니다.

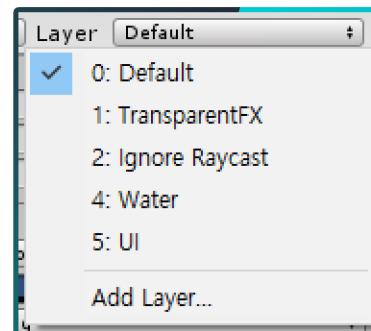


+ 레이어

레이어는 오브젝트를 분류할 수 있는 층입니다. 예를 들어 UI 오브젝트는 'UI' 레이어에 넣어 구분합니다.

레이어는 일부 레이어만 카메라에 표시하거나 빛이 비치는 레이어를 구분하는 등 선택적인 분류에 효과적입니다.

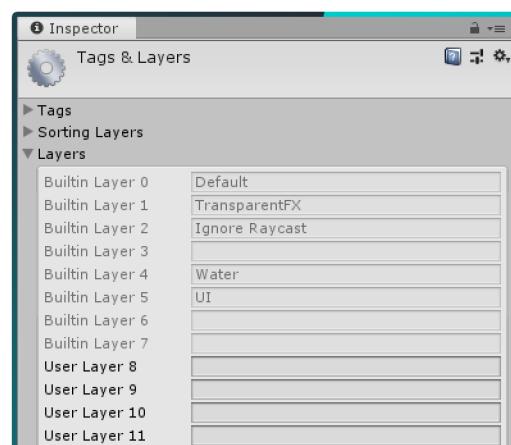
레이어를 클릭하면 레이어 목록이 나타나면서 레이어를 선택할 수 있게 됩니다. 새로운 레이어를 추가하고 싶다면 목록 하단의 'Add Layer...'를 클릭해 'Tags & Layers'에 들어가 추가하면 됩니다.



- 레이어 관리

'Layers'를 열어 레이어 리스트를 확인합니다. 미리 만들어진 빈 공간에 원하는 레이어 이름을 적으면 레이어가 추가됩니다. 미리 만들어진 레이어를 제외하고 최대 25개를 만들 수 있습니다.

레이어를 지우고 싶다면 이름을 지우면 됩니다.



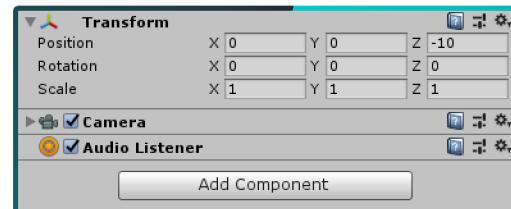
+ 컴포넌트

컴포넌트는 오브젝트에게 기능을 부여합니다.

스크립트도 컴포넌트의 형식으로 들어갑니다.

미리 준비된 오브젝트들은 각자 기능이 구현된 컴포넌트들을 가지고 있습니다. 빈 오브젝트에 직접 컴포넌트를 조합해 새로운 오브젝트를 만들 수도 있습니다.

새로운 컴포넌트를 추가하고 싶다면 컴포넌트 아래의 'Add Component' 버튼을 클릭하거나 'Component'에서 원하는 컴포넌트를 선택해주세요. 컴포넌트의 종류와 사용법은 #03-01 챕터를 참고해주세요.



'Main Camera'의 컴포넌트

+ 오브젝트 만들기

게임을 만들기 위해선 다양한 오브젝트들이 필요합니다. 직접 오브젝트에 컴포넌트를 추가하는 건 너무 오래 걸리므로 보통 미리 준비된 오브젝트를 만들어 사용합니다.

Hierarchy 창의 빈 공간을 우 클릭하거나 'GameObject'에서 원하는 오브젝트를 선택하면 새로운 오브젝트가 만들어집니다. 이제 오브젝트의 컴포넌트를 수정하면서 게임을 만들어가면 됩니다.

오브젝트의 종류와 사용법은 #03-03 챕터를 참고해주세요.

4. 파일

+ 파일 만들기

Project 창의 빈 공간을 우 클릭하면 'Create' 안에 만들 수 있는 파일들이 표시됩니다. 우리는 폴더와 C# 스크립트, Scene만 만들어 사용합니다.

- 폴더

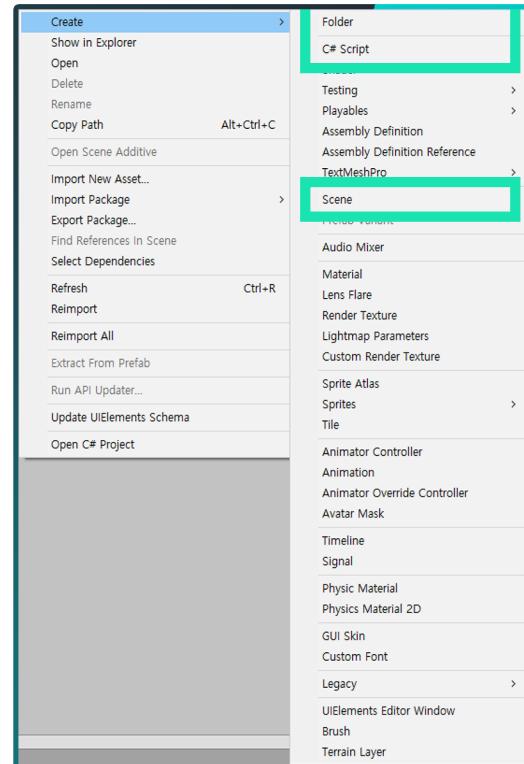
흔히 사용하는 탐색기의 폴더와 같습니다.
프로젝트를 정리하기 위해 잘 분류해야 합니다.

- C# 스크립트

코드를 적는 스크립트 파일입니다. 더블클릭하면 Visual Studio가 열리고 수정할 수 있습니다.
스크립트의 이름은 중복될 수 없고 대문자로 시작해야 됩니다.

- Scene

카메라만 들어있는 빈 Scene을 만듭니다.
더블클릭하거나 Hierarchy 창에 드래그하면 그 Scene을 열 수 있습니다.

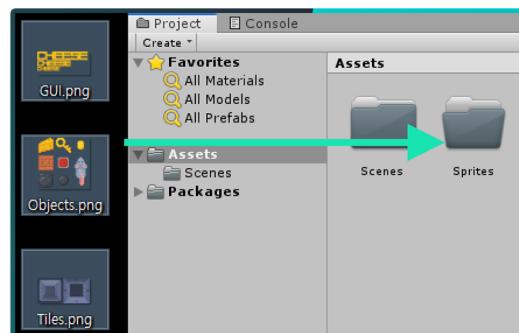


Project 창에서 우 클릭 'Create'

+ 파일 추가

기존의 파일은 드래그로 쉽게 추가할 수 있습니다. 직접 Assets 폴더에 Sprites 폴더를 만들고 그 폴더 안에 이미지들을 넣어보겠습니다.

Hierarchy 창의 오브젝트를 Project에 드래그하면 오브젝트가 파일 형식으로 저장됩니다. 파일로 저장된 오브젝트는 프리팹이라고 합니다. 프리팹에 대한 자세한 설명은 #03-02 챕터를 참고해주세요.



! 프로젝트에 사용되는 이미지는

github.com/Sunrin-RG/SSF-2019/tree/master/Sprites에서 다운로드 할 수 있습니다.

+ 파일 관리

파일을 클릭하면 Inspector에 정보가 표시됩니다. 폴더와 Scene은 정보가 없지만 스크립트는 미리 보기와 끄고 스프라이트는 많은 설정들이 있습니다. 스프라이트 설정을 통해 더 사용하기 편하게 만들겠습니다.

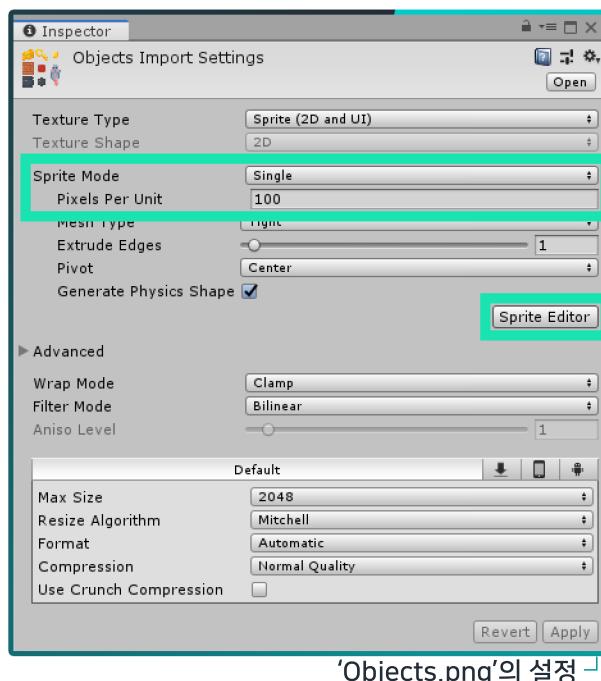
- 스프라이트 설정

스프라이트 설정에는 많은 설정이 있지만 직접 사용할 몇 가지만 알아보겠습니다.

'Sprite Mode'는 스프라이트의 형식을 정해줍니다. 'Single'은 단일 이미지이고 'Multiple'은 한 이미지에 여러 스프라이트가 들어있는 형식입니다. 우리는 여러 오브젝트 스프라이트가 같이 있으므로 'Multiple'로 합니다.

바로 밑에 있는 'Pixel Per Unit'은 좌표 1이 차지할 픽셀의 수입니다. 기본 설정 100은 좌표 1당 100칸의 픽셀이 들어가게 됩니다. 이 스프라이트는 160 픽셀을 한 칸으로 제작되었으니 160으로 합니다.

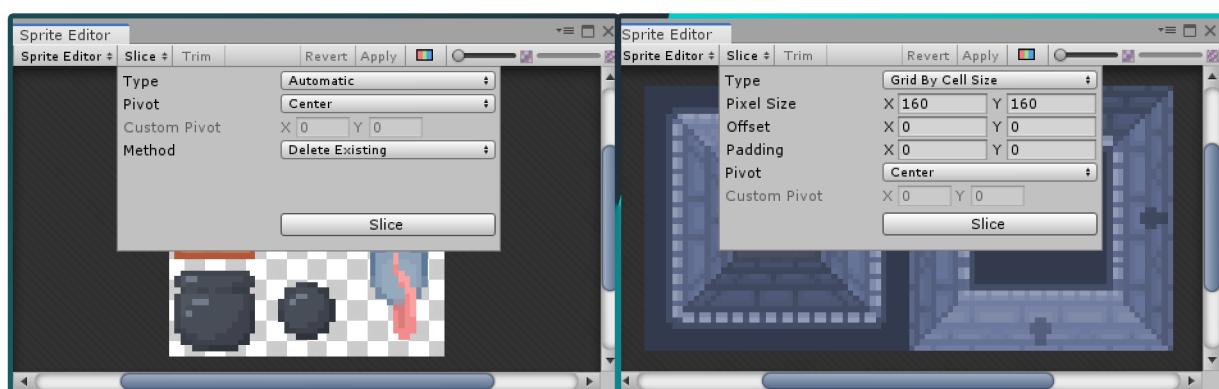
이제 'Sprite Editor'로 한 이미지를 여러 스프라이트로 나눠주면 따로 사용할 수 있습니다.



! 도트 디자인에 경우 'Filter Mode'를 'Point'로 설정하고 'Compression'을 'None'으로 설정하면 스프라이트를 더 선명하게 할 수 있습니다.

- 스프라이트 에디터

스프라이트 에디터에서 'Slice'를 누르면 여러 스프라이트로 자를 수 있습니다. 각 스프라이트가 'Objects.png'처럼 서로 거리가 있다면 'Automatic'으로 쉽게 자를 수 있습니다. 'Tiles.png'처럼 빈 공간 없이 전부 붙어있다면 'Grid By Cell Size'로 설정한 뒤 한 칸의 크기를 정하고 잘라주면 됩니다. 'Tiles.png'는 한 칸을 160픽셀로 제작했으므로 160 × 160 크기로 잘라주면 됩니다.



#03 게임 만들기

1. 게임 오브젝트 종류

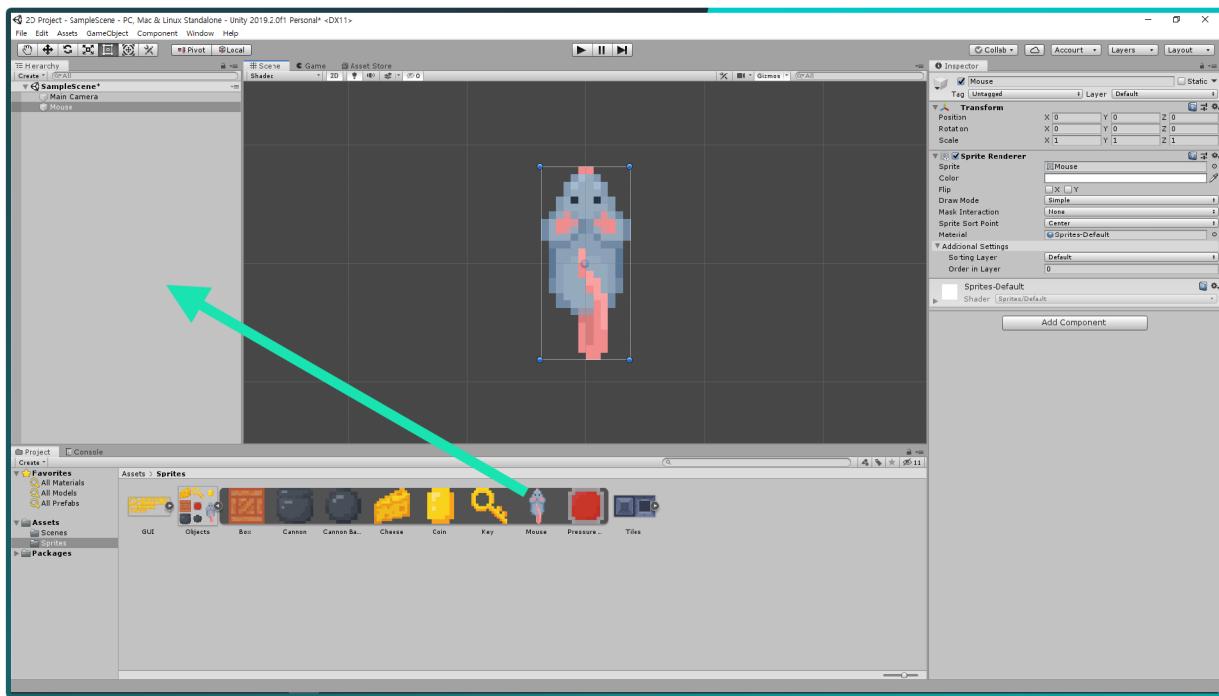
+ 빈 오브젝트

Transform을 제외한 어떤 컴포넌트도 없는 오브젝트입니다. 새로운 오브젝트를 만들거나 무안가를 표시(메모)하는 용도로 사용됩니다.

+ 카메라

화면을 볼 수 있게 해주는 오브젝트입니다. 'Camera' 컴포넌트를 가지고 있으며 Scene을 만들면 Main Camera로 하나가 자동으로 생성됩니다.

+ 스프라이트



스프라이트는 2D 이미지(스프라이트)를 보여주는 오브젝트로 'Sprite Renderer' 컴포넌트를 가지고 있습니다. Project 창의 스프라이트를 Hierarchy 창이나 Scene 창에 드래그해도 만들 수 있습니다. 기본적으로 겉 모습을 가져야되는 오브젝트들은 스프라이트에서 시작합니다.

+ UI

User Interface의 약자로 사용자가 조작하기 편하도록 만든 버튼이나 텍스트 등을 말합니다. Unity에는 많은 UI 오브젝트들이 있으며 텍스트, 버튼, 이미지 등 역할도 다양합니다. UI에 대한 자세한 정보는 #03-10 챕터를 참고해주세요.

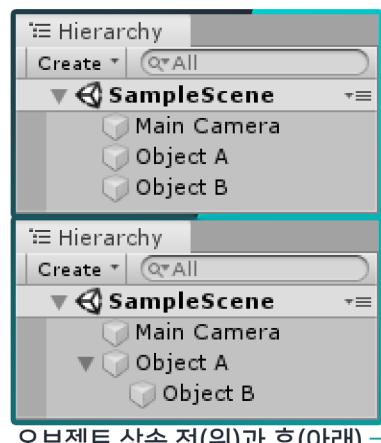
● 2. 게임 오브젝트 배치

+ 부모와 자식

오브젝트들은 배치할 때 서로 독립적으로 배치됩니다. 하지만 서로 상대적인 위치(혹은 각도나 크기)를 가지면 편해지는 상황이 있습니다. 이때 우리는 두 오브젝트를 부모와 자식 관계로 만들어 자식이 부모에 대해 상대적인 성질을 가지게 합니다. 그리고 이걸 상속이라고 부릅니다.

아무 관계없는 두 오브젝트 중 하나를 옮기거나 회전시켜도 다른 오브젝트는 변하지 않습니다. 하지만 오브젝트 하나를 드래그해 폴더에 파일을 넣듯 다른 오브젝트에 넣으면 부모 자식 관계가 됩니다. 여기서 위에 있는 오브젝트를 부모, 아래 있는 오브젝트를 자식으로 합니다.

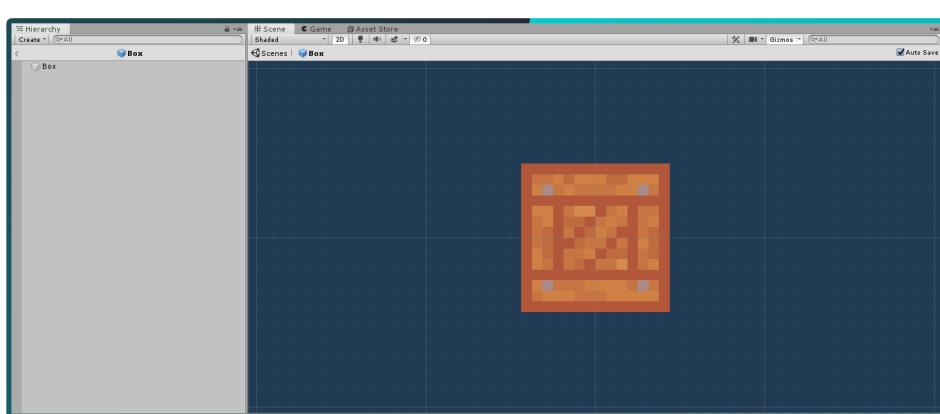
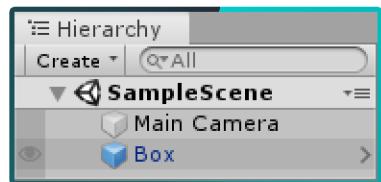
자식 오브젝트는 부모 오브젝트가 오른쪽으로 이동하면 따라가게 됩니다. 회전도 마찬가지입니다. 하지만 자식 오브젝트의 'Transform' 값을 보면 변하지 않는데, 부모가 있는 오브젝트는 부모 오브젝트의 'Transform' 값을 기준으로 상대적인 값이 표시되기 때문입니다. 자식 오브젝트의 X 좌표를 1로 바꾸면 부모 오브젝트의 중심을 기준으로 X 좌표로 1만큼 움직입니다.



! 글로만 보면 이해하기 힘들 수 있습니다. 직접 오브젝트를 상속시켜보고 부모를 움직이고 자식의 'Trnasform' 값을 바꿔보세요. 상대적인 값을 금방 이해할 수 있을 겁니다.

+ 프리팹

프리팹을 Scene에 가져오면 다른 오브젝트와 달리 파란색으로 표시됩니다. 프리팹은 미리 저장된 파일을 불러오기 때문에 저장된 파일의 오브젝트를 바꾸면 같은 프리팹으로 생성된 모든 오브젝트가 변경되게 됩니다. 저장된 파일을 수정하고 싶다면 해당 프리팹 파일을 더블클릭하거나 Hierarchy 창 오브젝트 오른쪽의 꺽새를 클릭해 프리팹 설정 화면에서 변경하세요.

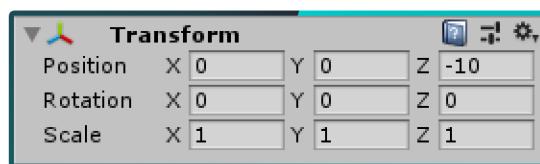


프리팹 설정 화면: 배경이 파란색으로 바뀝니다

3. 컴포넌트 종류

+ Transform

모든 오브젝트에 필수로 들어가는 컴포넌트로 오브젝트의 위치와 회전(각도), 크기를 저장합니다. 값을 수정해 정확한 수치를 정해줄 수 있습니다.



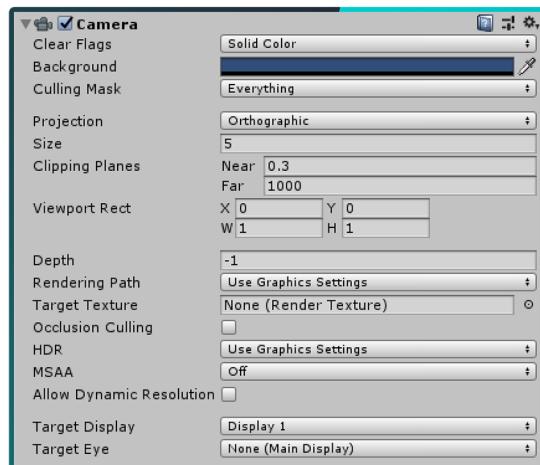
+ Camera

카메라 오브젝트에 쓰이는 컴포넌트로 실제 게임의 시점을 조작할 수 있게 해줍니다.

'Background'의 색을 변경하면 빈 공간의 배경 색이 변경됩니다.

'Culling Mask'는 이 카메라가 보여줄 레이어를 선택합니다. 선택 해제된 레이어는 카메라에 보이지 않게 됩니다.

'Size'를 변경하면 카메라가 보여주는 공간의 크기를 조절할 수 있습니다. 값이 클수록 더 넓은 공간을 보여주게 됩니다.



+ Sprite Renderer

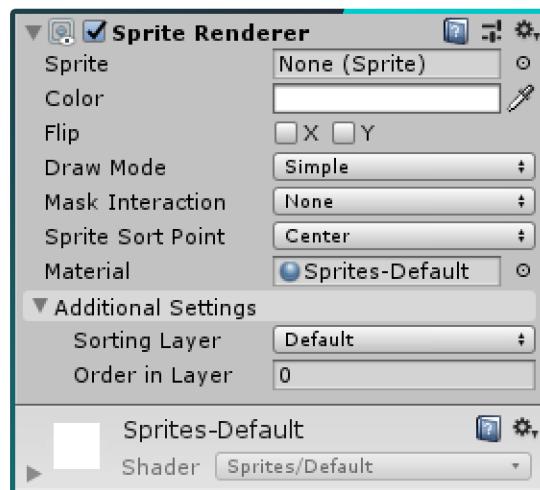
2D 스프라이트를 렌더링 할 수 있게 해주는 컴포넌트입니다.

'Sprite'에 원하는 스프라이트를 드래그해 넣거나 오른쪽의 원을 클릭해 프로젝트에 있는 스프라이트 중 하나를 골라 변경할 수 있습니다.

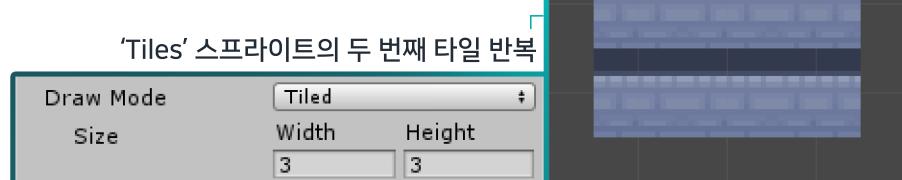
'Color'의 색을 바꾸면 스프라이트에 색이 덮어집니다. 흑백의 스프라이트에 색을 줄 수도 있습니다.

'Flip'은 해당 좌표 축을 기준으로 스프라이트를 뒤집어줍니다.

'Additional Settings'에 'Order in Layer'의 값을 변경하면 스프라이트의 순서가 바뀝니다. 숫자가 더 클수록 위에 나타납니다.



'Draw Mode'를 'Tiled'로 바꾸면 한 스프라이트를 반복시켜 넓은 면을 채울 수 있습니다. 'Size'에 원하는 면적을 적으면 됩니다.



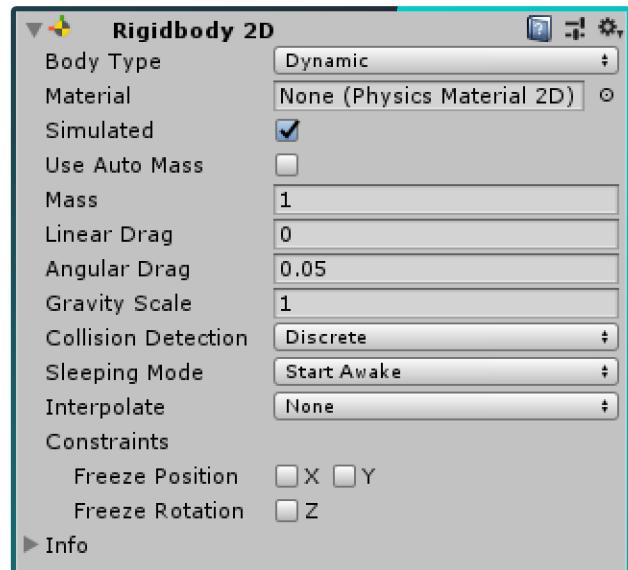
+ Rigidbody 2D

2D에서 물리효과를 적용시켜주는 컴포넌트입니다.

'Mass'는 오브젝트의 무게입니다. 수치를 조절해 가볍게, 혹은 무겁게 만들 수 있습니다.

'Gravity Scale'은 이 오브젝트에 작용하는 중력의 크기입니다. 우리는 탑뷰 게임을 제작하기 때문에 주로 0으로 설정합니다.

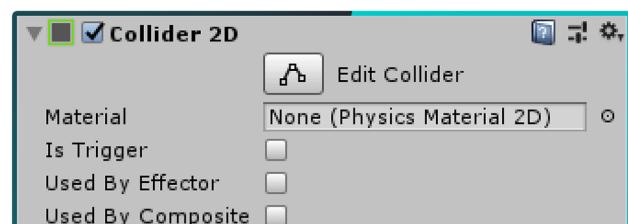
'Constraints'는 여러 Freeze 설정을 가지고 있습니다. 이 설정으로 얼린 위치나 각도는 물리적인 현상으로만 변하지 않고 스크립트 등의 영향은 그대로 받습니다. 우리는 물리 현상으로 인한 회전은 필요 없기 때문에 주로 'Rotation'을 알려둡니다.



+ Collider 2D

콜라이더는 오브젝트에게 충돌 판정을 줍니다. 만약 콜라이더가 있는 오브젝트가 게임 도중에 움직인다면 Rigidbody 컴포넌트를 넣어주세요.

'Rdit Collider' 버튼을 누르면 마우스로 콜라이더의 크기나 모양을 수정할 수 있습니다. 조작법은 콜라이더 종류마다 다릅니다.



- 콜라이더와 트리거

콜라이더는 자신과 부딪히거나 겹치는 등 접촉한 다른 오브젝트의 콜라이더를 검사합니다. 평소의 콜라이더는 충돌할 수 있기 때문에 접촉한 오브젝트의 콜라이더를 검사합니다.

하지만 'Is Trigger'를 사용하면 더 이상 물리적으로 충돌하지 않고 자신과 겹치는 다른 오브젝트의 콜라이더를 검사합니다.

콜라이더는 검사하면서 이 오브젝트의 이벤트 함수를 호출합니다. 이벤트 함수에 대한 자세한 정보는 #03-04 생명주기 챕터를 참고해주세요.

콜라이더는 Box, Circle, Capsule 등 다양한 모양이 있습니다. 한 오브젝트에 여러 콜라이더를 넣을 수도 있으니 모양에 맞춰 적절한 콜라이더를 사용하면 됩니다.

! 탑뷰 게임에서는 중력이 필요 없지만 한 번 Rigidbody와 콜라이더를 여러 오브젝트에 적용시키고 게임을 테스트해보세요. 물리 엔진을 이해하기 좋습니다.

4. 스크립트 제작

스크립트는 C# 스크립트를 작성합니다. 기초 문법은 C언어나 C++과 비슷하니 C언어 계열로 기초 문법을 익힌 후 시작하는 걸 추천합니다.

+ 변수

Unity는 많은 객체들이 있고 여러 컴포넌트나 게임 오브젝트를 위해 기본 자료형과는 다른 자료형을 사용하게 됩니다. 우리는 게임 오브젝트나 프리팹을 저장하는 'GameObject', 물리 효과를 담당하는 컴포넌트 Rigidbody를 저장하는 'Rigidbody(2D)', 그리고 세 좌표 축(X, Y, Z)의 좌표 값을 저장하는 'Vector3'를 사용합니다.

```
1 using UnityEngine;
2
3 public class Variable : MonoBehaviour {
4     float number = 3.141f;
5 }
```

Unity 스크립트는 파일 이름과 같은 클래스 안에 작성합니다. 만약 파일 이름과 클래스 이름이 다르면 오류가 생기니 조심해주세요.

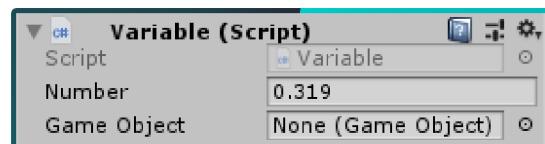
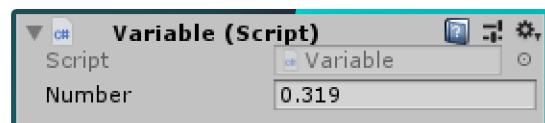
위 코드처럼 평범하게 변수를 선언해 사용할 수 있습니다. 하지만 값을 수정하기 위해서는 매번 코드를 수정해야 된다는 불편함이 있습니다. 그래서 이 스크립트가 적용된 오브젝트의 Inspector 창에서 쉽게 값을 수정하기 위해 변수 선언 앞에 'public'을 붙여 접근 권한을 낮춰줍니다.

```
1 using UnityEngine;
2
3 public class Variable : MonoBehaviour {
4     public float number = 3.141f;
5 }
```

이제 Inspector 창에서 값을 즉석으로 수정하며 게임을 테스트할 수 있습니다.

게임 오브젝트도 같은 방법으로 변수를 만들면 오브젝트가 들어갈 수 있는 공간이 생깁니다.
Hierarchy 창에서 드래그하거나 Project 창의 프리팹을 드래그해서 넣을 수 있습니다.

다른 변수들도 Inspector 창에 노출시킬 수 있습니다.



+ 생명주기

게임 오브젝트가 생성되고 삭제되기까지 특정 순간에 호출되는 함수들을 이벤트 함수라고 합니다. 그리고 이 함수들의 주기를 오브젝트의 생명주기라고 합니다.

```
1 using UnityEngine;
2
3 public class LifeCycle : MonoBehaviour {
4     void Start() { }
5     void Update() { }
6     void OnCollisionEnter(Collision collision) { }
7 }
```

이벤트 함수가 실행되는 타이밍은 로그를 출력해주는 함수인 'Debug.Log()'로 확인하면 좋습니다.

- Start()

Start는 오브젝트가 처음 시작될 때 호출됩니다. 기존 Scene에 이미 있는 오브젝트라면 그 Scene이 시작될 때, 만약 뒤늦게 생성된 오브젝트라면 생성될 때 한번 호출되게 됩니다.

- Update()

Update는 오브젝트가 있다면 매 프레임마다 호출됩니다. Unity는 60프레임 제한이 있어 1초에 최대 60번 호출될 수 있습니다. 기기마다 성능 차이로 호출 횟수가 달라지곤 합니다.

- 충돌과 트리거

콜라이더와 관련된 이벤트 함수는 충돌과 트리거에 각각 3개씩, 총 6개입니다. 이 스크립트가 적용된 오브젝트의 모든 콜라이더를 대상으로 호출됩니다. 우리는 2D 프로젝트를 다루기 때문에 2D 이벤트 함수만 사용합니다.

충돌과 트리거에 관한 이벤트 함수는 앞에서 알아본 이벤트 함수들과 달리 자동으로 넘어오는 값이 있습니다. 그리고 그 값을 이벤트 함수의 인자로 받게 됩니다. 충돌은 'Collision(2D)'을 가져와 충돌에 관한 정보(충돌 지점, 충돌 속도 등)를 모두 얻을 수 있습니다. 트리거는 'Collider(2D)'를 가져와 감지된 콜라이더만 가져오게 됩니다.

* OnCollision[Enter/Stay/Exit]2D(Collision2D)

'Enter'는 다른 콜라이더와 충돌이 일어난 순간에 한 번, 'Stay'는 계속 닿아있는 상태라면 반복적으로, 'Exit'는 닿아있던 다른 콜라이더가 떨어지는 순간에 한 번 호출됩니다.

* OnTrigger[Enter/Stay/Exit]2D(Collider2D)

충돌 이벤트 함수와 비슷하지만 충돌이 아닌 트리거 콜라이더 안에 들어온 다른 콜라이더를 감지해 호출됩니다.

! 콜라이더 관련 이벤트 함수들은 Rigidbody가 있는 콜라이더만 감지합니다.

! 충돌과 트리거 관련 이벤트 함수들은 직접 두 오브젝트에 콜라이더를 적용시키고 실험해보면 이해하기 편합니다. 한 번 직접 확인해 보는 걸 추천합니다.

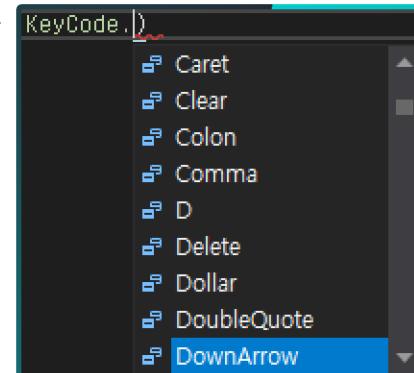
+ 입력받기

PC의 입력 도구는 키보드와 마우스가 대표적입니다. Unity에서는 키보드와 마우스 말고도 많은 입력 도구를 지원해 조절할 수 있습니다.

```
1  using UnityEngine;
2
3  public class KeyInput : MonoBehaviour {
4      void Update() {
5          bool inputW = Input.GetKey(KeyCode.W);
6
7          if (inputW == true) {
8              Debug.Log("Pressed W");
9          } else {
10              Debug.Log("Not Pressed W");
11          }
12      }
13 }
```

'Input.GetKeyDown(KeyCode)' 함수는 인자의 키 코드에 해당하는 키가 눌렸는지 bool 형식으로 알려주는 함수입니다. KeyCode에는 키보드나 조이스틱 컨트롤러, 마우스 등 지원하는 입력 도구들의 키들이 enum으로 정의되어 있습니다.

KeyCode에 정의된 키들을 보고 싶다면 자동완성을 이용하거나
docs.unity3d.com/kr/530/ScriptReference/KeyCode.html
사이트를 참고하세요.



! GetKey 함수는 눌리고 있다면 계속 true를 반환하지만 GetKeyDown과 GetKeyUp은 각각 해당 키가 눌린 순간과 땐 순간에만 true를 반환합니다.

+ 컴포넌트 가져오기

게임 도중에 스크립트로 컴포넌트를 조작할 수 있습니다. 콜라이더를 비활성화하거나 Rigidbody의 날아가는 방향을 수정하는 등 오브젝트의 상태를 조절할 수 있습니다.

```
1 using UnityEngine;
2
3 public class GetComponent : MonoBehaviour {
4     CapsuleCollider2D collider;
5
6     void Start() {
7         collider = GetComponent<CapsuleCollider2D>();
8         collider.enabled = false;
9     }
10 }
```

컴포넌트를 수정하기 위해서는 먼저 원하는 컴포넌트를 저장할 변수를 만들어야 됩니다. 모든 컴포넌트는 사용자가 만든 스크립트와 같은 형식이기 때문에 컴포넌트의 이름, 스크립트라면 클래스의 이름을 적으면 변수 자료형으로 사용할 수 있습니다.

이제 컴포넌트를 찾아서 변수에 넣어주면 됩니다. ‘GetComponent’ 함수는 이 스크립트가 적용된 오브젝트의 컴포넌트를 찾아줍니다. ‘GetComponent<>()’에서 화살괄호 안에 가져오고 싶은 컴포넌트를 적으면 됩니다. 보통 오브젝트가 처음 시작할 때 미리 컴포넌트들을 받아둬야 하므로 Start 이벤트 함수에 작성합니다.

컴포넌트를 가져왔다면 원하는 순간에 원하는 방법으로 조작하면 됩니다. 모든 컴포넌트는 ‘enabled’라는 상태를 가지고 있고 이 값이 false라면 컴포넌트는 비활성화됩니다. 그 외에도 콜라이더는 모양마다 다른 크기 조절 값, Rigidbody는 날아가고 있는 힘이나 적용되고 있는 중력의 크기, Transform은 위치, 방향, 각도를 조작할 수 있습니다.

- Transform 조작

하지만 Transform 컴포넌트는 모든 오브젝트에 들어있어 가져오지 않아도 사용할 수 있습니다. 스크립트에서 transform은 이 오브젝트의 Transform 컴포넌트를 가리킵니다.

```
:
:
11     void Update() {
12         transform.position = new Vector3(3.141f, 2.718f, 0);
13     }
14 }
```

위치 값은 세 좌표 축을 사용하는 Vector3 형식이므로 Vector3를 원하는 값으로 만들어 적용시켰습니다. 위 코드처럼 Transform 컴포넌트는 따로 가져오지 않아도 사용할 수 있습니다.

+ Vector

Unity에는 좌표 축의 값을 저장하는 Vector들이 있습니다. Vector2부터 Vector4까지 있지만 주로 Vector2와 Vector3만 사용합니다.

두 Vector를 더하면 각 좌표 값에 더해집니다. 값이 (1, 2, 3)인 Vector3와 (3, 4, 5)인 Vector3를 더하면 값이 (4, 6, 8)인 Vector3가 됩니다. 그 외 사칙연산도 동일하게 동작합니다.

값의 수가 더 적은 Vector2를 Vector3에 더할 수는 있지만 반대는 안됩니다. 반드시 형 변환을 해주세요.

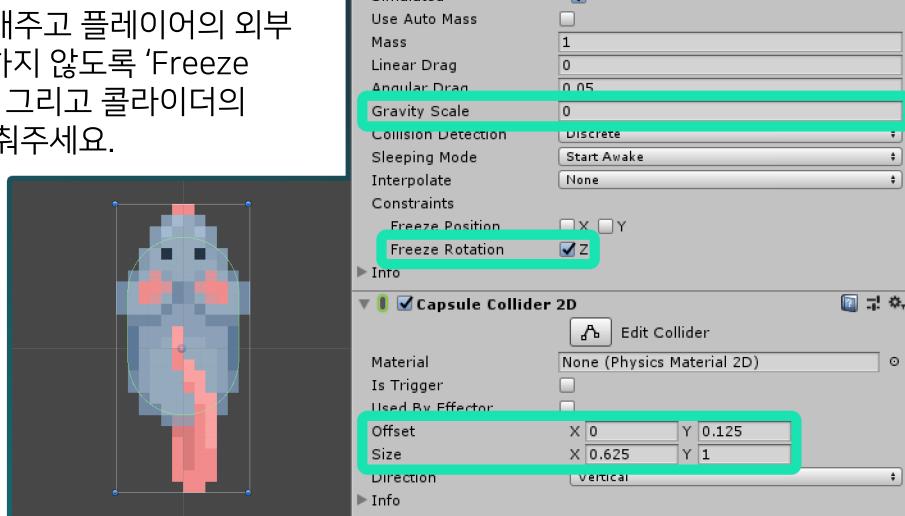
5. 플레이어 만들기

+ 이동 구현

먼저 움직일 오브젝트를 만들어줘야 됩니다. 우리는 쥐 스프라이트를 이용해서 만들겠습니다.

쥐 스프라이트 둔 후 Rigidbody와 캡슐 모양의 콜라이더를 넣었습니다.

탑뷰 게임이니 중력을 없애주고 플레이어의 외부 요인으로 플레이어가 회전하지 않도록 'Freeze Rotation'을 체크해주세요. 그리고 콜라이더의 크기와 위치를 적절하게 맞춰주세요.



이제 플레이어가 움직일 수 있도록 스크립트를 만듭니다. 입력을 받고 Rigidbody 컴포넌트를 가져와 위치를 수정하겠습니다.

- 움직이기

```
1 using UnityEngine;
2
3 public class PlayerMove : MonoBehaviour {
4     Rigidbody2D rigidbody;
5
6     void Start() {
7         rigidbody = GetComponent<Rigidbody2D>();
8     }
9 }
```

먼저 Rigidbody를 가져와 준비해줍니다. 이제 Update에 W, A, S, D 키 입력을 확인하는 조건문을 만들고 Rigidbody의 Position 값에 각 방향을 더해줍니다. 위와 아래 방향은 Y좌표를, 좌우는 X좌표를 조절합니다.

```
:  
10 void Update() {
11     if (Input.GetKey(KeyCode.W))
12         rigidbody.position += new Vector2(0, 1);
13     else if (Input.GetKey(KeyCode.S))
14         rigidbody.position += new Vector2(0, -1);
15     if (Input.GetKey(KeyCode.A))
16         rigidbody.position += new Vector2(-1, 0);
17     else if (Input.GetKey(KeyCode.D))
18         rigidbody.position += new Vector2(1, 0);
19 }
20 }
```

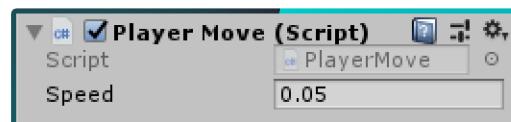
하지만 지금 이대로 적용시키고 실행한다면 굉장히 빠르게 이동합니다. 속도를 조절하기 위해 'speed'라는 Float형 변수를 Public으로 만들어 Inspector에서 값을 조절하며 속도를 정하겠습니다.

! 위치를 옮길 때 Transform이 아닌 Rigidbody를 사용하면 물리적인 효과를 유지한 채 위치를 옮길 수 있습니다.

```

4     :
5     public float speed;
6     :
7
8     void Update() {
9         if (Input.GetKey(KeyCode.W))
10            rigidbody.position += new Vector2(0, speed);
11        else if (Input.GetKey(KeyCode.S))
12            rigidbody.position += new Vector2(0, -speed);
13        if (Input.GetKey(KeyCode.A))
14            rigidbody.position += new Vector2(-speed, 0);
15        else if (Input.GetKey(KeyCode.D))
16            rigidbody.position += new Vector2(speed, 0);
17    }
18
19
20
21 }
```

이제 Inspector에서 속도를 조절해 사용할 수 있습니다. 하지만 Update는 기기마다 실행 횟수가 달라질 수 있어 이동속도가 다를 수도 있습니다. 기기별로 차이가 생기지 않도록 개선해보겠습니다.



```

4     :
5     public float speed;
6     :
7
8     void Update() {
9         float moveDistance = speed * Time.deltaTime;
10
11        if (Input.GetKey(KeyCode.W))
12            rigidbody.position += new Vector2(0, moveDistance);
13        else if (Input.GetKey(KeyCode.S))
14            rigidbody.position += new Vector2(0, -moveDistance);
15        if (Input.GetKey(KeyCode.A))
16            rigidbody.position += new Vector2(-moveDistance, 0);
17        else if (Input.GetKey(KeyCode.D))
18            rigidbody.position += new Vector2(moveDistance, 0);
19    }
20
21
22
23 }
```

Time.deltaTime을 곱하면 Update 호출 횟수 차이를 없앨 수 있습니다. 대신 값이 작아져 속도를 다시 조절해줘야 됩니다.

속도를 조절해줬다면 쥐가 적당한 속도로 움직이게 됩니다. 하지만 항상 위를 보고 움직여 어색합니다. 이번에는 바라보는 방향을 조절해주겠습니다.

- 바라보기

쥐를 돌리기 위해서는 Z 각도를 조절하면 됩니다. Z 각도가 커질수록 반 시계 방향으로 돌아갑니다. 왼쪽은 90도, 아래는 180도, 오른쪽은 270도로 설정합니다.

오브젝트의 각도를 수정할 때 Transform의 Rotation 값을 수정하게 됩니다. 하지만 Rotation에 들어가는 값은 우리가 사용하는 육십분법과 달리 호도법을 사용합니다. 그래서 호도법을 사용하는 'transform.rotation'이 아닌 육십분법을 사용하는 'transform.eulerAngles'를 수정하겠습니다.

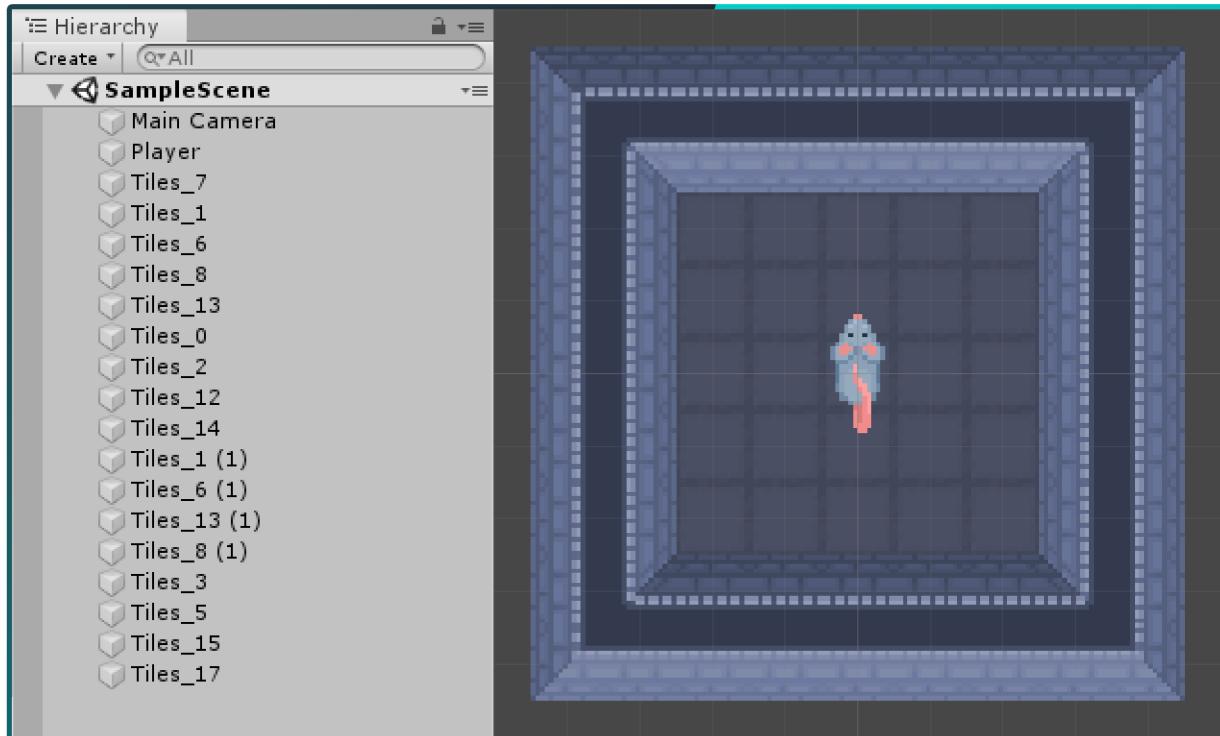
```
:  
14    :  
15        if (Input.GetKey(KeyCode.W)) {  
16            rigidbody.position += new Vector2(0, moveDistance);  
17            transform.eulerAngles = new Vector3(0, 0, 0);  
18        }  
19        else if (Input.GetKey(KeyCode.S)) {  
20            rigidbody.position += new Vector2(0, -moveDistance);  
21            transform.eulerAngles = new Vector3(0, 0, 180);  
22        }  
23        if (Input.GetKey(KeyCode.A)) {  
24            rigidbody.position += new Vector2(-moveDistance, 0);  
25            transform.eulerAngles = new Vector3(0, 0, 90);  
26        }  
27        else if (Input.GetKey(KeyCode.D)) {  
28            rigidbody.position += new Vector2(moveDistance, 0);  
29            transform.eulerAngles = new Vector3(0, 0, 270);  
30        }  
31    :  
32    :
```

이제 향하는 방향을 바라봐 좀 더 자연스럽게 이동합니다. 다음은 쥐가 돌아다니게 될 맵을 만들어보겠습니다.

! 다 만든 플레이어 오브젝트의 이름을 'Player'로 바꾸고 태그에 'Player' 태그를 달면 더 구분하기 쉬워집니다.

6. 맵 만들기

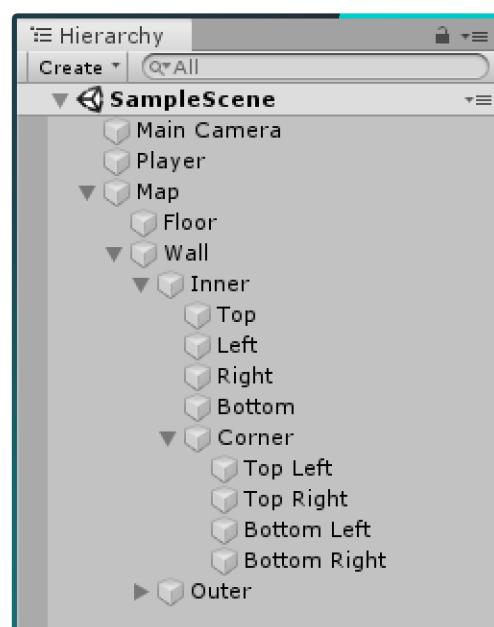
+ 배치



간단하게 사각형 모양으로 벽을 둘렀습니다. ‘Sprite Mode’를 ‘Tiled’로 바꿔 반복되는 부분을 빠르게 채웠습니다. 하지만 Hierarchy 창이 정리가 되지 않아 보기 힘듭니다. 빈 오브젝트를 폴더처럼 사용하고 이름을 위치와 역할에 맞게 바꿔보겠습니다.

그림에서는 바닥과 벽, 벽의 방향과 위치로 구분해 정리했습니다. 각자 원하는 형식으로 정리해 보기 편하게 만들면 됩니다. 상속된 오브젝트는 접어서 줄일 수 있어 작게 숨길 수 있습니다.

! 스프라이트는 ‘Order in Layer’ 설정으로 정렬됩니다. 플레이어를 위로 설정하세요.



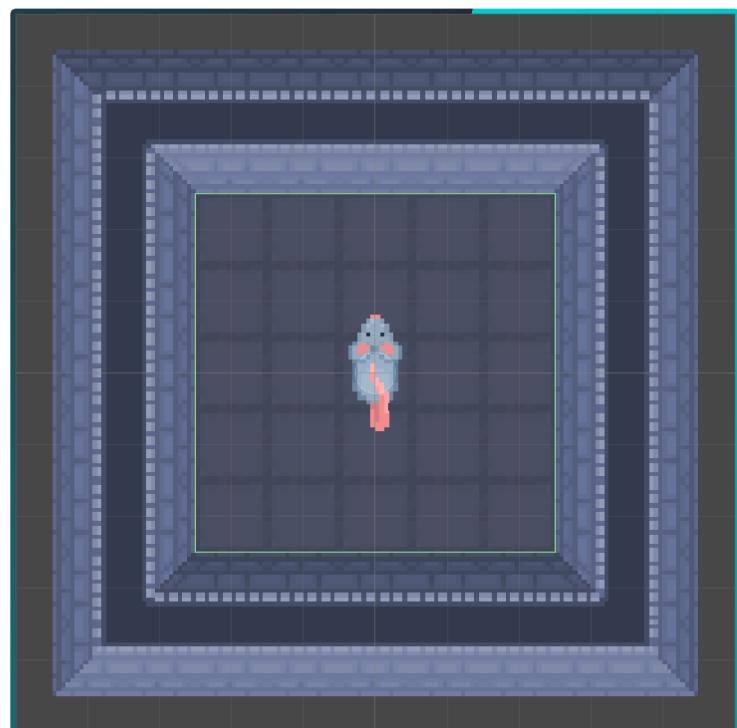
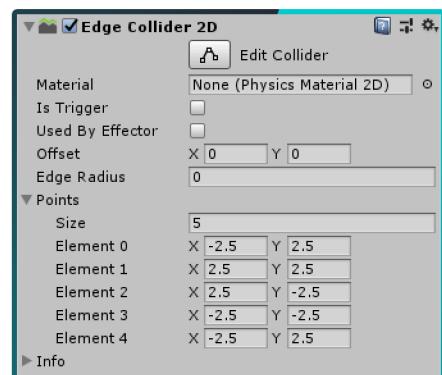
+ 콜라이더

맵과 같이 콜라이더에 채워진 부분이 따로 없고 벽만 있는 경우 'Edge' 콜라이더가 유용합니다. 점들의 위치를 적어주면 각 점을 이어 선을 만들고 그 선은 다른 콜라이더와 같이 충돌할 수 있습니다.

Edge 콜라이더는 채워지는 부분이 없어 울타리 같은 역할을 하기 때문에 상자와 같이 채워져있는 물체에는 적합하지 않지만 어느 부분도 채워지지 않는 벽이나 울타리에 좋습니다.

벽에는 충돌 시 콜라이더 이벤트 함수를 호출할 수 있도록 Rigidbody를 넣어주세요. 그리고 위치와 회전을 알려주세요.

외부를 두른 벽은 충돌 판정이 없지만 필요 없기 때문에 생략할 수 있습니다. 만약 Edge 콜라이더를 사용하기 힘들다면 여러 Box 콜라이더를 사용해도 좋습니다.



벽 콜라이더를 가지고 있는 오브젝트에게 'Wall' 태그를 만들어 붙여주면 더 편하게 구분할 수 있습니다.

7. 다른 오브젝트 만들기

+ 상자

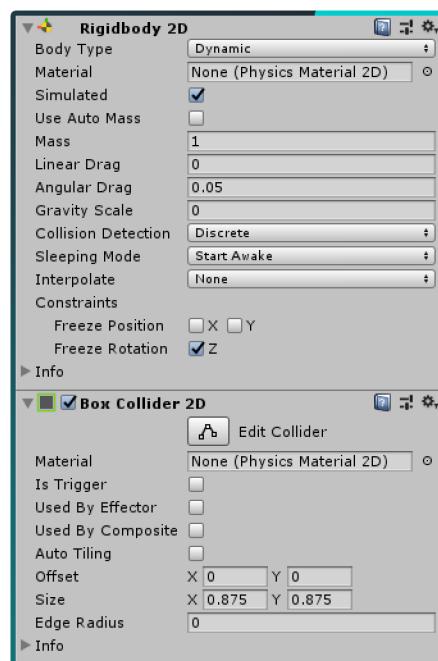
Rigidbody와 콜라이더만 있으면 되는 간단한 밀 수 있는 상자를 만들어봅시다. 'Objects.png'의 Box 스프라이트를 사용하겠습니다.

Rigidbody를 넣어주고 역시 중력 크기를 0으로 바꾸고 회전을 잠가주세요. Box 콜라이더를 넣으면 크기가 스프라이트 크기로 자동 조절됩니다.

태그로 'Box' 태그를 만들어 달아주세요. 더 편하게 사용할 수 있습니다.

테스트로 밀어본다면 플레이어가 조금 느려지는 걸 볼 수 있습니다. 만약 너무 느리거나 빠르다면 Mass(무게)를 수정해 조절하세요.

이제 편하게 상자를 사용하기 위해 Hierarchy의 상자를 Project로 드래그해 프리팹으로 저장하세요. 다시 사용할 때 얼마든지 꺼내 사용하면 됩니다.



+ 동전과 치즈

동전은 점수를 주는 오브젝트, 치즈는 게임 클리어 목표가 되는 오브젝트로 만들겠습니다. 먼저 점수를 만들고 동전을 만들겠습니다.

```
1  using UnityEngine;
2
3  public class Score : MonoBehaviour {
4      static public int score = 0;
5  }
```

변수를 선언할 때 Public처럼 'static'을 붙인다면 스크립트를 오브젝트에 넣지 않고도 값을 조절할 수 있습니다. 이제 어디에서든 'Score.score'로 점수를 조정할 수 있습니다.

동전 스프라이트를 오브젝트로 만든 후 적절한 콜라이더를 넣으세요. 크기를 조절한 후 'Is Trigger'를 체크해 충돌되지 않도록 합니다. 그리고 'Coin' 스크립트를 만들어 플레이어가 닿을 경우 점수를 올리고 동전을 없애는 코드를 작성해줍니다.

'Desctroy' 함수는 인자로 전달된 오브젝트를 없애줍니다. 스크립트에서 자신이 적용된 오브젝트는 'gameObject'로 가져올 수 있습니다. 즉 자신을 없애기 위해선 'Destroy(gameObject)'와 같이 사용하면 됩니다.

```

1  using UnityEngine;
2
3  public class Coin : MonoBehaviour {
4      void OnTriggerEnter2D(Collider2D collision) {
5          if (collision.tag == "Player") {
6              Score.score += 100;
7              Destroy(gameObject);
8          }
9      }
10 }

```

이제 동전 오브젝트에 스크립트를 넣고 테스트한후 프리팹으로 만들어 사용하세요.

치즈도 동전처럼 적당한 콜라이더를 넣고 크기를 조절하세요. 치즈는 닿으면 엔딩 화면으로 넘어가 게임이 종료되게 만들겠습니다.

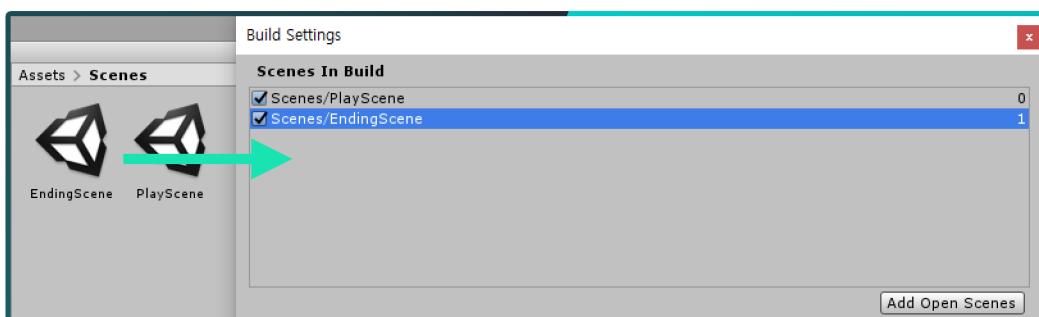
- Scene 관리

Scene은 오브젝트들이 들어있는 공간입니다. 실제 게임에 타이틀 화면, 게임 플레이 화면, 엔딩 화면 등 다양한 화면이 있듯 Scene도 다양하게 만들어 서로 전환할 수 있습니다.

우선 지금까지 게임을 만든 Scene을 플레이 화면으로 만들겠습니다. Project에서 해당 Scene의 이름을 바꾸면 Hierarchy에서도 이름이 바뀝니다.



그리고 게임을 완료하면 전환될 엔딩 화면도 만듭니다. 만든 Scene은 'File > Build Settings...'의 'Scenes In Build'에 드래그해 추가해야 제대로 동작합니다.



이제 치즈의 스크립트를 만듭니다. 전체적인 모양은 동전과 같지만 기능만 다르게 만들겠습니다.

```

1  using UnityEngine;
2  using UnityEngine.SceneManagement;
3
4  public class Cheese : MonoBehaviour {
5      void OnTriggerEnter2D(Collider2D collision) {
6          if (collision.tag == "Player")
7              SceneManager.LoadScene("EndingScene");
8      }
9  }

```

동전과 달리 사라질 필요 없이 Scene만 전환해줬습니다.

+ 열쇠와 문

문은 진행을 방해하는 요소로 열쇠를 얻어야 문이 열려 진행할 수 있습니다. 열쇠에 닿으면 문이 사라져 앞으로 나아갈 수 있도록 만들겠습니다.

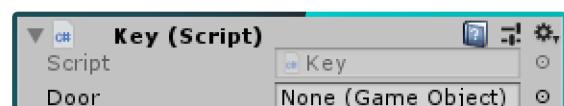
우선 열쇠도 동전이나 치즈같이 트리거 콜라이더를 가진 오브젝트로 만들어주세요. 그럼 스크립트를 만들어 적용시키겠습니다.

```

1  using UnityEngine;
2
3  public class Key : MonoBehaviour {
4      public GameObject door;
5
6      void OnTriggerEnter2D(Collider2D collision) {
7          if (collision.tag == "Player") {
8              Destroy(door);
9              Destroy(gameObject);
10         }
11     }
12 }

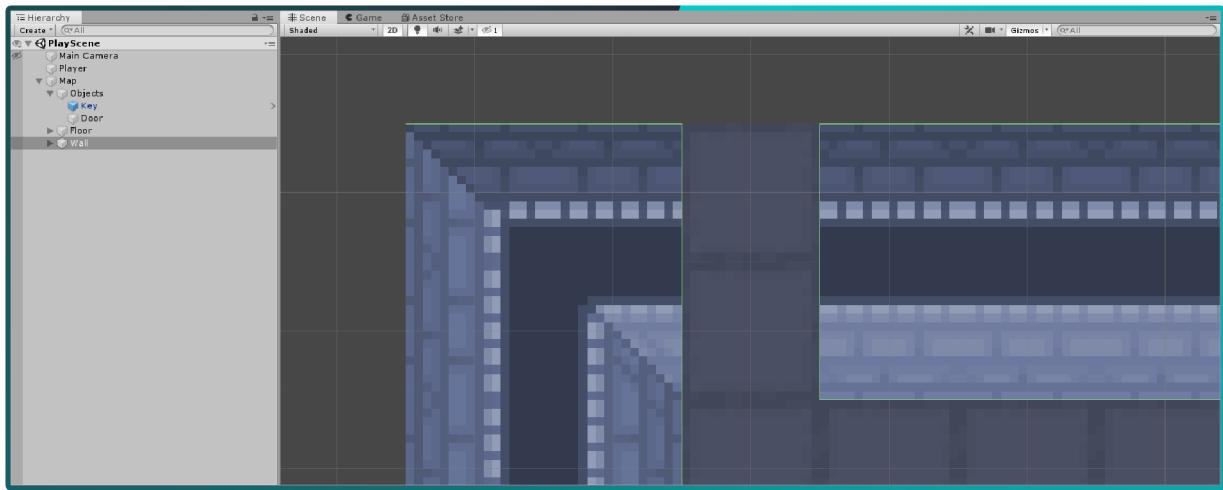
```

게임 오브젝트로 열릴 문을 Inspector에서 받아 없애도록 했습니다. 그럼 열릴 문을 만들어보겠습니다.



문은 빈 오브젝트에 문 스프라이트들을 넣어 한 번에 관리할 수 있도록 만듭니다. 그리고 문이 사라지면 빈 공간이 생기니 미리 바닥을 깔아두겠습니다.

! 바닥 스프라이트가 문 스프라이트보다 위로 올라올 수도 있습니다. 'Order in Layer' 값을 조절하세요.



문이 들어갈 공간에는 벽 콜라이더가 있으면 안 되니 조절해줘야 됩니다.

이제 문 스프라이트를 넣고 벽이랑 똑같이 Rigidbody와 콜라이더를 넣고 설정하세요. 완성된 문을 열쇠의 Key 스크립트의 Door에 넣고 테스트해보세요.

지금처럼 상하로 통하는 문과 좌우로 통하는 문을 만들어 프리팹으로 사용하면 편하게 사용할 수 있습니다.

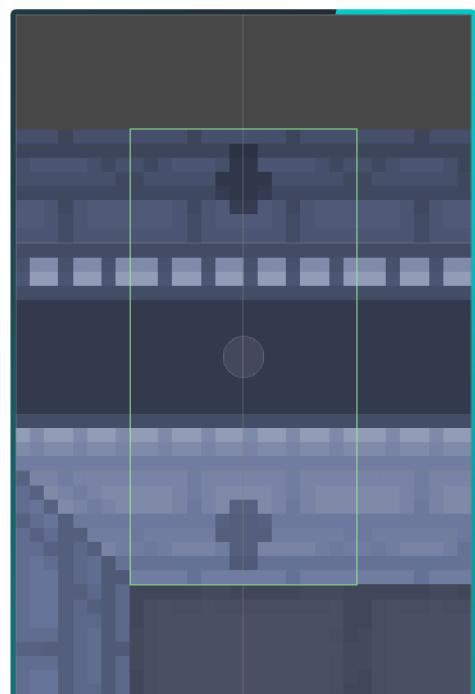
문도 벽과 비슷하게 동작하므로 태그를 'Wall'로 설정하겠습니다.

+ 대포

대포는 플레이어가 맞으면 게임을 다시 시작하기 만드는 오브젝트입니다. 일정 간격으로 대포를 쏘고 벽이나 상자로 막을 수 있게 만들겠습니다.

먼저 대포가 발사할 대포알을 준비하겠습니다. 다른 오브젝트처럼 트리거 콜라이더를 넣어 지나치는 콜라이더를 확인하겠습니다. 그리고 대포가 복사해 생성할 수 있도록 프리팹으로 저장하세요.

그리고 대포알을 발사할 대포를 만듭니다. 대포 스프라이트에 콜라이더를 넣은 후 조절합니다. 그럼 지금부터 대포의 기능을 위한 스크립트를 만들겠습니다.



! 대포알이 대포보다 위에 표시되지 않도록 설정해주세요. 그럼 더 자연스러운 모습으로 발사할 수 있습니다.

- 오브젝트 생성

'Instantiate' 함수는 오브젝트를 게임 도중에 생성할 수 있습니다. 다양한 사용법이 있지만 우린 3개의 인자를 쓰는 방법을 사용하겠습니다. 인자로 (생성할 오브젝트, 생성할 위치, 회전(각도))를 입력하면 오브젝트가 해당 위치에 회전 값을 가지고 생성됩니다.

```
1  using UnityEngine;
2
3  public class Cannon : MonoBehaviour {
4      public GameObject bullet;
5
6      void Update() {
7          Instantiate(bullet, transform.position, transform.rotation);
8      }
9  }
```

이제 대포 Inspector에 있는 Bullet 항목에 방금 프리팹으로 만들어둔 대포알을 넣으면 굉장히 많은 양의 대포알이 계속 생성됩니다. 그럼 일정 간격으로 생성되도록 만들겠습니다.

- 코루틴

코루틴은 시작하면 평범한 함수처럼 동작합니다. 하지만 중간에 'yield return'을 만나면 평범한 함수의 return처럼 호출한 자리로 돌아가 본래 명령들을 다시 실행하게 됩니다. 하지만 코루틴은 일정 조건이 만족되면 다시 'yield return' 자리로 돌아와 자신의 남은 명령들을 수행합니다. 조건에는 많은 종류가 있지만 우리는 입력한 초만큼 기다리는 'WaitForSeconds'를 사용하겠습니다.

```
1  using System.Collections;
2  using UnityEngine;
3
4  public class Cannon : MonoBehaviour {
5      public GameObject bullet;
6      bool canShoot = true;
7
8      void Update() {
9          if (canShoot == true)
10              StartCoroutine("SpawnBullet");
11      }
12
13      IEnumerator SpawnBullet() {
14          Instantiate(bullet, transform.position, transform.rotation);
15          canShoot = false;
16          yield return new WaitForSeconds(1.0f);
17          canShoot = true;
18      }
19  }
```

우선 스크립트 상단에 'using System.Collections'를 추가해야 됩니다. 코루틴은 함수를 반드시 'IEnumerator'로 생성해야 되는데 이 자료형이 저 위치에 정의되어 있습니다. 그리고 발사 여부를 조절할 'canShoot' 변수를 만들었습니다. 참일 때 발사할 수 있고 거짓이면 발사할 수 없게 만들었습니다.

코루틴은 다른 함수처럼 호출하지 않고 'StartCoroutine'이라는 함수로 호출합니다. 함수 안에 원하는 코루틴 함수의 이름을 적으면 실행할 수 있습니다.

코루틴 함수는 명령을 실행하다 중간에 조건이 걸리기 때문에 먼저 대포알을 생성하고 발사가 되지 않도록 'canShoot'을 거짓으로 만듭니다. 그러면 바로 아래 yield return의 조건인 1초가 지나기 전까진 Update에서 코루틴을 시작하지 않게 됩니다. 그리고 1초가 지나면 다시 canShoot을 참으로 만들어 발사되게 합니다.

지금은 발사 속도가 1초로 고정이지만 플레이어의 속도처럼 Public 변수로 발사 간격을 조절할 수 있게 만들어보겠습니다.

```
1   :
2   :
3   public float delay;
4   :
5   yield return new WaitForSeconds(delay);
6   :
```

이제 대포가 일정 간격으로 대포알을 생성하지만 대포알이 날아가지 않아 효과가 없습니다. 그럼 대포알의 기능을 위한 스크립트를 만들겠습니다. 먼저 날아가게 합시다.

대포는 기본적으로 위를 바라보기 때문에 대포알의 Y좌표를 계속해서 늘려주면 됩니다. 하지만 만약 대포가 회전되어 있다면 그 방향으로 날아가야 되기 때문에 단순히 Transform의 Y 좌표만 늘리면 안됩니다. 이때 'transform.up'을 사용하면 오브젝트가 바라보는 방향을 기준으로 위 방향을 가리킬 수 있습니다. 우리는 이 값에 대포알의 속도를 곱해서 사용하겠습니다.

```
1  using UnityEngine;
2
3  public class CannonBall : MonoBehaviour {
4      public float speed;
5
6      void Update() {
7          transform.position += transform.up * speed * Time.deltaTime;
8      }
9  }
```

이제 대포알이 설정한 속도만큼 날아갑니다. 하지만 아직도 충돌이 일어나도 반응하지 않습니다. 플레이어가 맞으면 게임을 다시 시작하고 벽이나 상자 등 'Wall'이나 'Box' 태그가 붙은 오브젝트가 맞으면 대포가 사라지게 만들겠습니다.

```

1  using UnityEngine;
2  using UnityEngine.SceneManagement;
3  :
4  void OnTriggerEnter2D(Collider2D collision) {
5      if (collision.tag == "Player") {
6          Score.score = 0;
7          SceneManager.LoadScene("PlayScene");
8      }
9      else if (collision.tag == "Wall" || collision.tag == "Box")
10         Destroy(gameObject);
11     }
12 }
```

플레이어에 닿으면 점수를 초기화하고 Play Scene을 다시 불러와 초기화하고 벽이나 상자에 닿으면 자신을 없애도록 했습니다.

+ 맵 꾸미기

이제 간단한 오브젝트들을 제작했으니 맵을 꾸며봅시다. 대포의 발사 속도를 빠르게 해 반드시 상자로 막아야만 지나갈 수 있게 설계하거나 넓은 미로를 만들어 열쇠와 문으로 어지럽게 할 수도 있습니다.

하지만 지금은 카메라가 고정이라 플레이어가 카메라 밖으로 나가면 보이지 않습니다. 카메라가 플레이어 중앙에 고정되도록 간단한 스크립트를 만들어주겠습니다.

- 카메라 따라가기

```

1  using UnityEngine;
2
3  public class CameraMove : MonoBehaviour {
4      GameObject player;
5
6      void Start() {
7          player = GameObject.FindGameObjectWithTag("Player");
8      }
9
10     void Update() {
11         Vector3 playerPosition = player.transform.position;
12         playerPosition.z = -10;
13         transform.position = playerPosition;
14     }
15 }
```

'GameObject.FindGameObjectWithTag' 함수는 태그 이름으로 오브젝트를 찾아줍니다. 그리고 카메라는 Z 좌표가 0이면 아무것도 비출 수 없기 때문에 위치를 따로 저장해 Z 좌표만 바꾼 후 적용시킵니다.

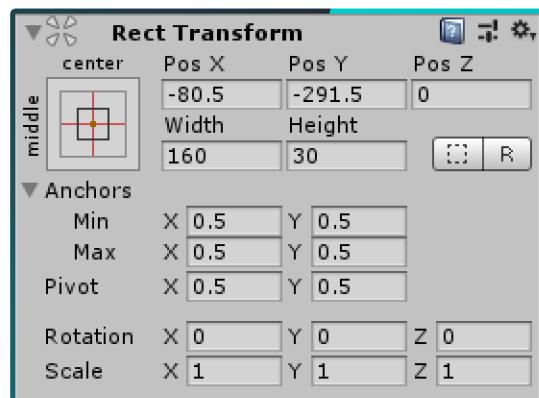
8. UI

+ UI 오브젝트

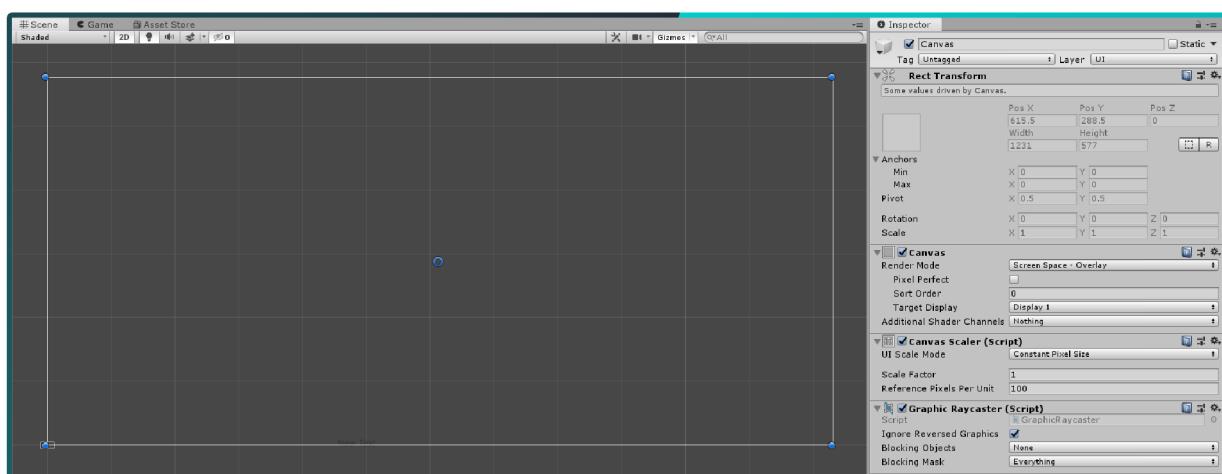
UI 오브젝트는 일반 오브젝트와 달리 평범한 Transform이 아닌 Rect Transform을 사용합니다. Rect Transform은 부모를 기준으로 배치됩니다.

Anchor의 위치를 세로와 가로 방향으로 각각 양 끝과 중간을 기준으로 바꿀 수 있고 그 기준을 중심으로 상대적인 좌표를 사용합니다.

크기는 Pivot을 중심으로 퍼지듯 커집니다.



- 캔버스

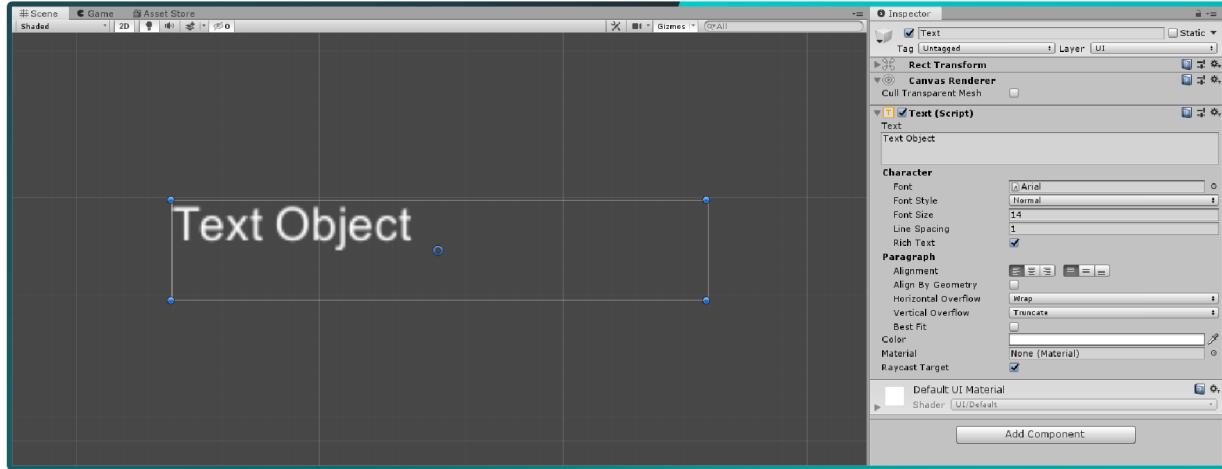


캔버스는 UI 오브젝트를 자동으로 생성되며 크기가 항상 화면의 크기로 고정됩니다. 모든 UI 오브젝트는 캔버스의 자식으로 생성됩니다.

캔버스에는 UI 오브젝트들의 크기를 담당하는 'Canvas Scaler'라는 컴포넌트가 있습니다. 여기서 'UI Scale Mode' 설정은 UI의 크기를 정하는 단위를 설정하는데 기본 설정인 'Constant Pixel Size'는 기기마다 다른 해상도를 고려하기 힘듭니다. 그래서 보통 'Scale With Screen Size'로 바꿔 사용합니다.

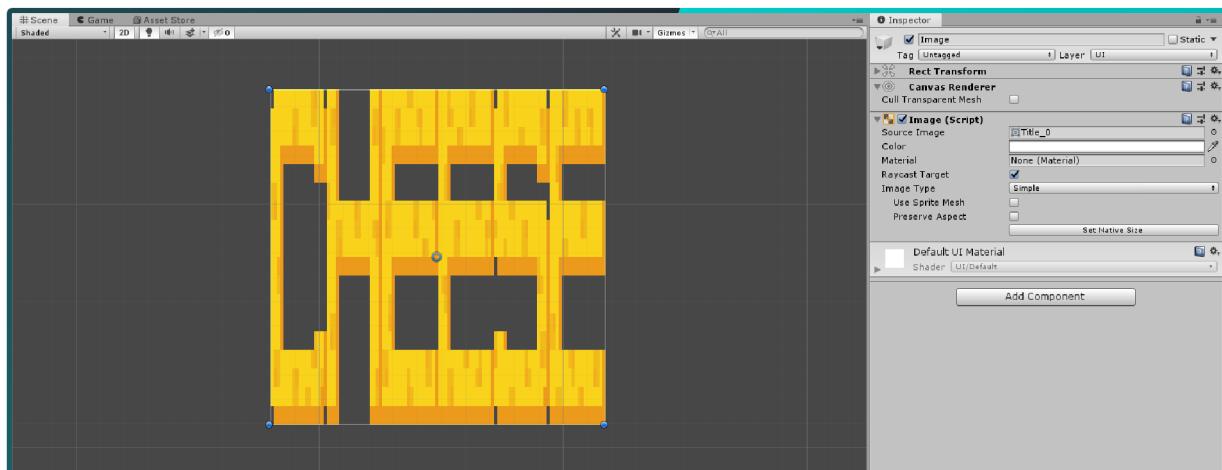
! 캔버스는 카메라가 보여주는 공간을 아득히 벗어났지만 정상적으로 게임 화면에 표시됩니다. 캔버스를 게임 화면이라 생각하고 작업하세요.

- 텍스트



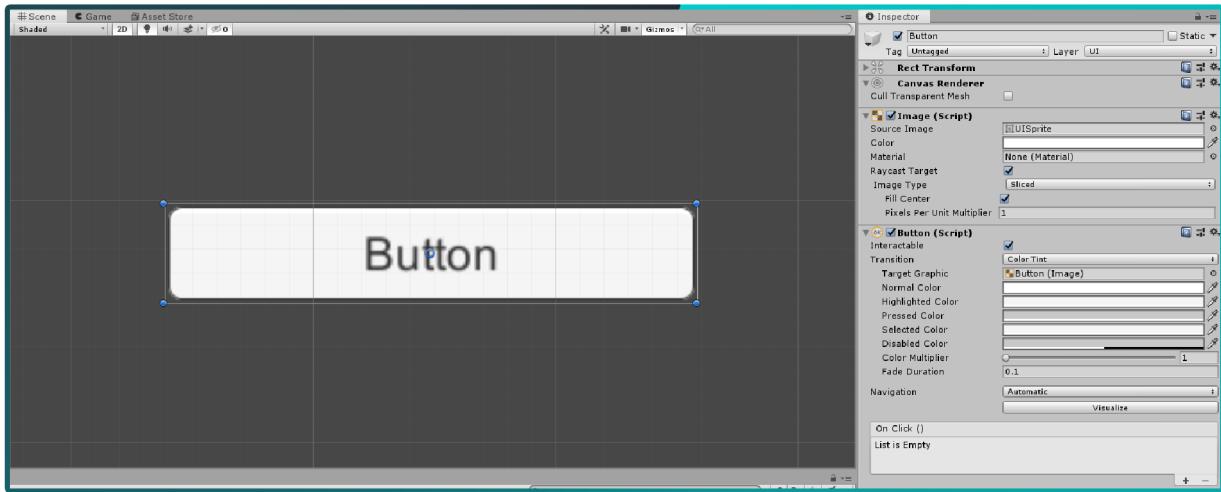
텍스트는 화면에 글자를 보여줍니다. Text 컴포넌트로 글자와 크기, 정렬, 색 등을 설정할 수 있습니다.

- 이미지



이미지는 화면에 그림을 보여줍니다. Image 컴포넌트의 Source Image에 원하는 스프라이트를 넣어 이미지를 변경할 수 있고 컴포넌트 크기에 맞춰 이미지가 늘어나지 않도록 하려면 'Preserve Aspect' 설정을 체크해주세요.

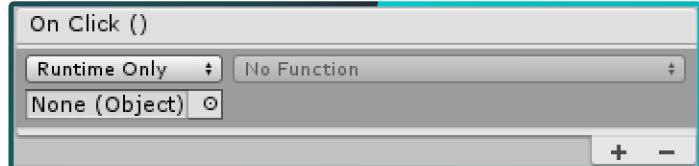
- 버튼



버튼은 Image와 Button 컴포넌트를 가지고 있고 자식으로 텍스트가 있는 오브젝트입니다. 이미지는 버튼의 배경이고 텍스트는 버튼의 글씨입니다.

Button 컴포넌트에는 버튼에 마우스를 올렸을 때, 버튼을 눌렀을 때 등의 상황에서 버튼을 강조해 보여주는 'Transition'이라는 설정이 있습니다. 기본 설정인 'Color Tint'는 배경 이미지에 색을 변경하는 설정입니다.

버튼을 눌렀을 때 기능은 Button 컴포넌트 아래에 있는 'On Click()' 이벤트에서 정합니다. 태그 설정처럼 우측 하단의 '+' 버튼을 눌러 추가할 수 있고 원하는 오브젝트(스크립트나 게임 오브젝트)를 넣어 그 오브젝트에 있는 Public 함수를 하나 선택하게 됩니다.



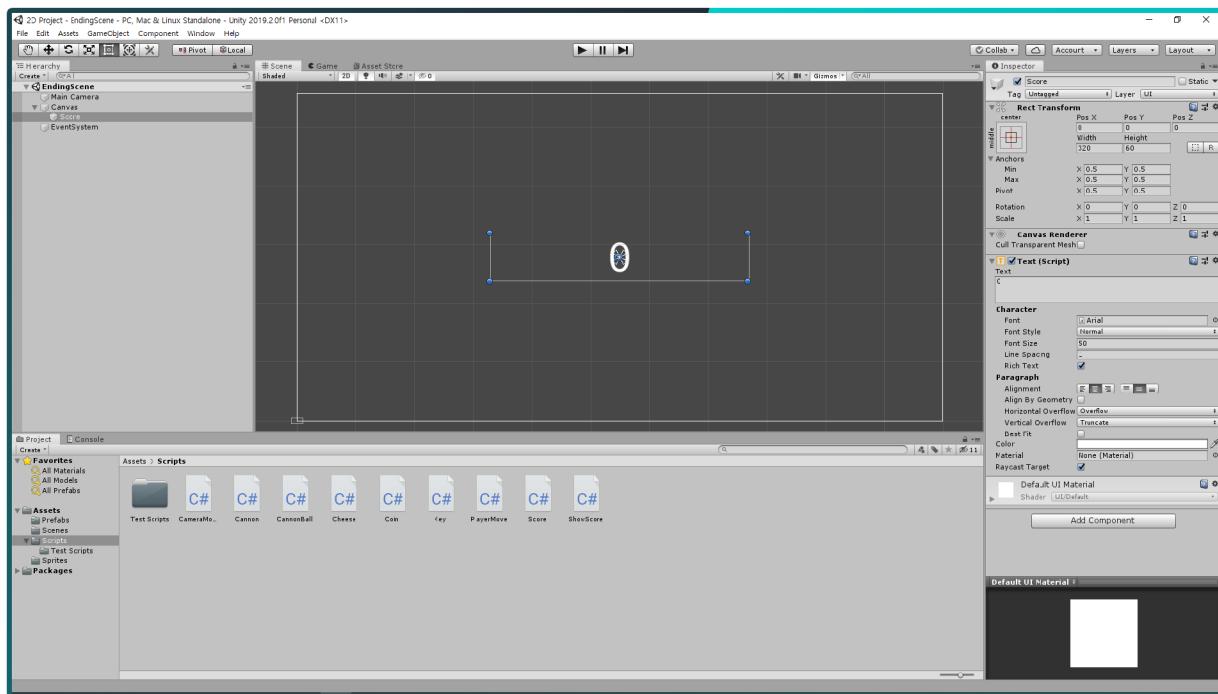
```
1 using UnityEngine;
2
3 public class ButtonTest : MonoBehaviour {
4     public void Test() {
5         Debug.Log("Button Click");
6     }
7 }
```

'Test' 함수처럼 Public으로 만들면 오브젝트에 이 스크립트를 넣어 실행시킬 수 있습니다.

! 캔버스에 만든 빈 오브젝트는 UI 오브젝트처럼 Rect Transform을 가집니다. 여러 UI 컴포넌트들을 조합해 자신만의 오브젝트를 만들어보세요.

+ 엔딩 화면

이번에는 이전에 만들어둔 엔딩 화면을 UI를 배치해 꾸며보겠습니다. 먼저 게임에서 얻은 점수를 표시해주겠습니다.

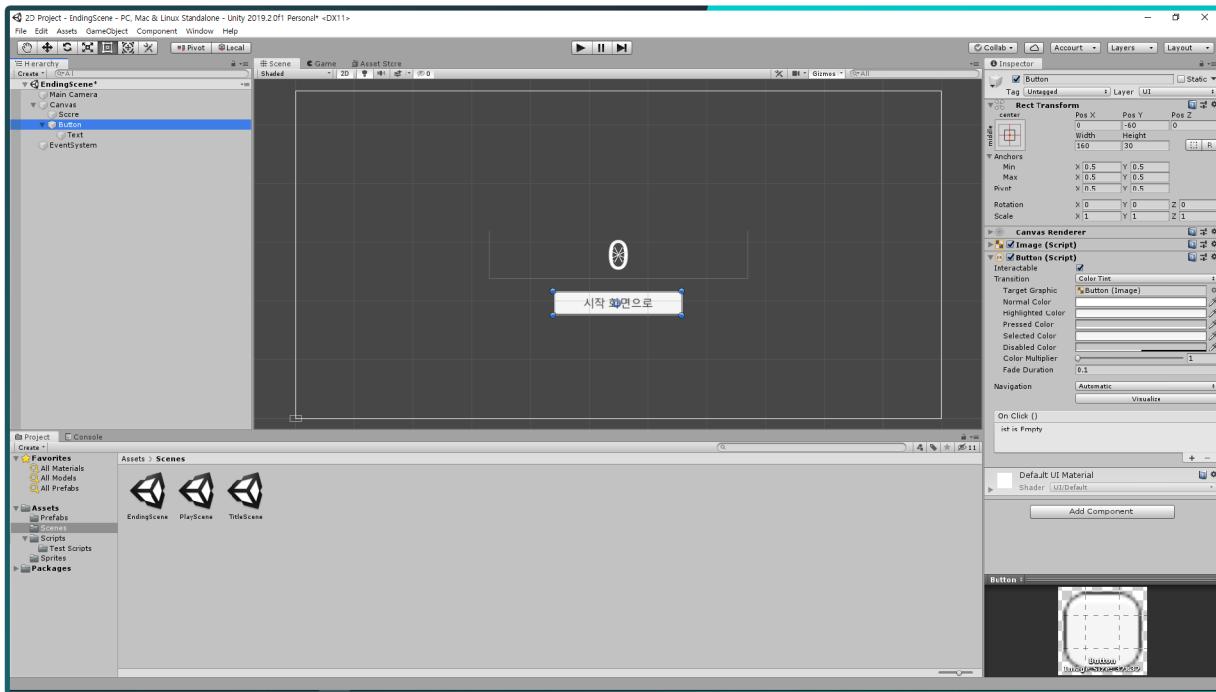


중앙에 임시로 0을 적은 텍스트를 배치했습니다. 점수가 너무 커지면 오브젝트의 너비보다 길어질 수 있어서 Paragraph의 'Horizontal Overflow' 설정을 Overflow로 고쳤습니다. 이제 점수를 가져와 표시해주는 스크립트를 만들겠습니다.

```
1  using UnityEngine;
2  using UnityEngine.UI;
3
4  public class ShowScore : MonoBehaviour {
5      void Start() {
6          Text text = GetComponent<Text>();
7          text.text = Score.score + " 점";
8      }
9 }
```

UI 컴포넌트를 사용하기 위해선 스크립트 상단에 'using UnityEngine.UI'를 추가해야 됩니다. 그리고 Text 컴포넌트의 텍스트는 'text'에 들어있습니다. 이 text에 점수를 뒤에 '점' 문자열을 붙여 넣었습니다. 정수는 문자열과 더하면 자동으로 문자열로 합쳐집니다.

이제 엔딩 화면이 뜨면 점수가 표시됩니다. 엔딩 화면이 있다면 시작 화면도 있어야 됩니다. 일단 엔딩 화면에서 시작 화면으로 넘어가는 버튼을 만들어주겠습니다.

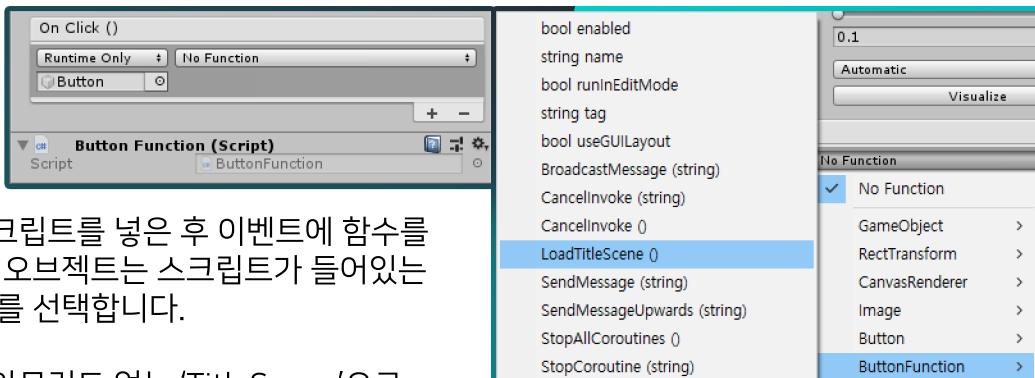


먼저 텍스트를 '시작 화면으로'로 바꿨습니다. 버튼을 눌렀을 때 넘어갈 시작 화면도 미리 만들어둡니다. 저는 'TitleScene'으로 만들겠습니다.

버튼에는 스크립트의 Public 함수만 이벤트에 넣을 수 있습니다. 그럼 버튼의 기능을 간단한 스크립트로 만들겠습니다.

```

1  using UnityEngine;
2  using UnityEngine.SceneManagement;
3
4  public class ButtonFunction : MonoBehaviour {
5      public void LoadTitleScene() {
6          SceneManager.LoadScene("TitleScene");
7      }
8 }
```



이제 버튼에 스크립트를 넣은 후 이벤트에 함수를 추가합니다. 이때 오브젝트는 스크립트가 들어있는 자신을 넣고 함수를 선택합니다.

버튼을 누르면 아무것도 없는 'TitleScene'으로 넘어갑니다. 지금부터 시작 화면을 꾸며봅시다.

+ 시작 화면

시작 화면에는 이미지 오브젝트를 이용해 게임 제목을 보여주고 버튼 두 개를 사용해 각각 게임 시작과 게임 종료로 만들겠습니다.

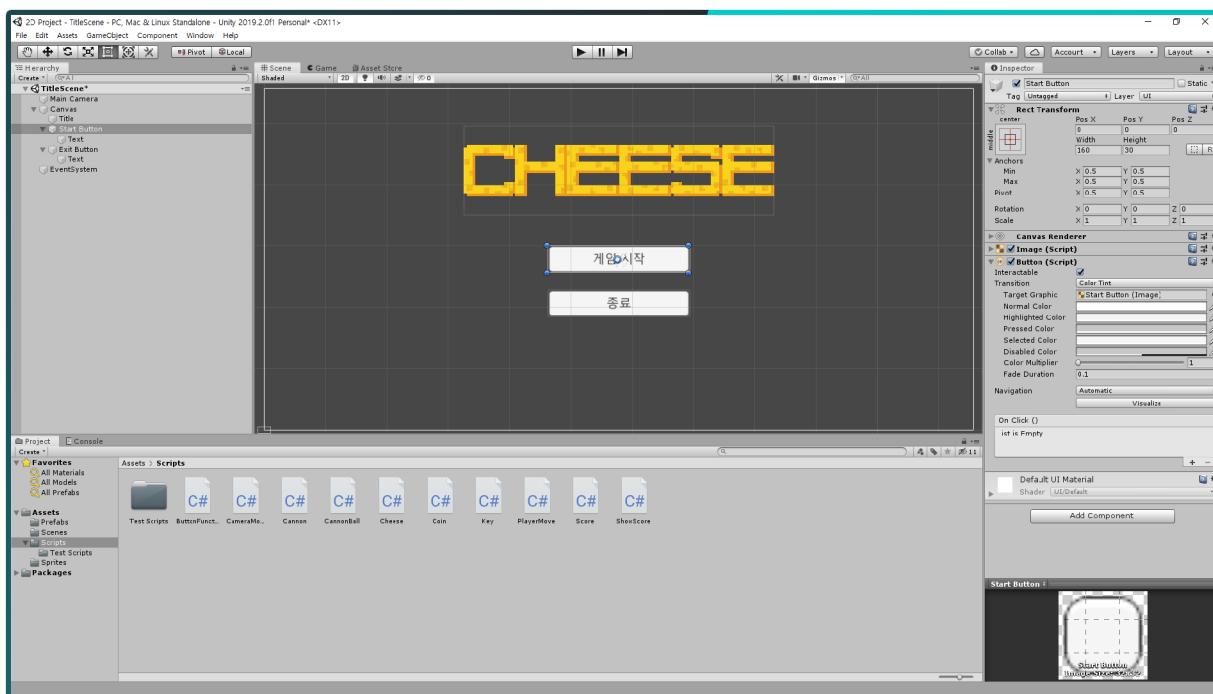


Image 컴포넌트에 제목 이미지를 넣고 비율이 변하지 않도록 설정해줬습니다. 위치는 중앙에서 조금 위로 가도록 했습니다.

시작 버튼과 종료 버튼은 텍스트를 바꿔준 후 종료 버튼을 조금 아래로 내려줬습니다. 이제 게임 시작 버튼의 기능과 종료 버튼의 기능을 방금 만든 'ButtonFunction' 스크립트에 추가해 적용시키겠습니다.

```
9     public void StartGame() {
10        Score.score = 0;
11        SceneManager.LoadScene("PlayScene");
12    }
13
14    public void ExitGame() {
15        Application.Quit();
16    }
17 }
```

게임을 시작할 때는 점수를 초기화해주고 'PlayScene'을 불러와 시작합니다. 게임을 끌 때는 'Application.Quit' 함수를 사용하면 됩니다. 그럼 엔딩 화면의 '시작 화면으로' 버튼처럼 스크립트를 적용시킨 후 오브젝트에 자신을 넣어 함수를 골라주세요.

게임 테스트에서는 게임 종료가 동작하지 않습니다. 바로 다음 단계인 빌드를 진행한 후 확인해주세요.

9. 마무리

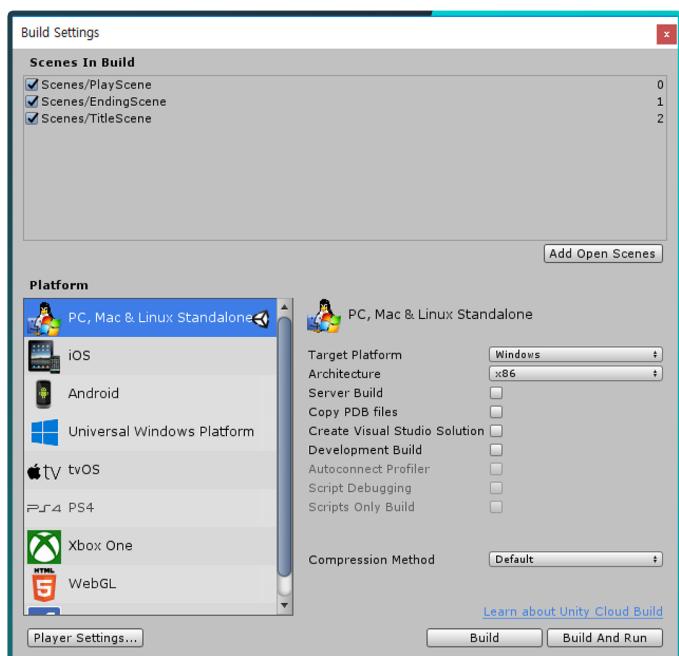
+ 빌드

'File > Build Settings...'로 가면 빌드 설정 화면이 나옵니다.

먼저 Scene의 순서를 정해주겠습니다.
가장 위에 올라와 있는 Scene이 게임을 키면
먼저 나오는 Scene입니다. 우리는
'TitleScene'을 제일 위로 옮리겠습니다.

이제 우측 하단에 'Build' 버튼을 누르면
빌드 할 폴더를 정하게 됩니다. 자신이 원하는
폴더를 지정하면 빌드가 진행됩니다. 빌드는
시간이 걸리니 기다려주세요.

빌드가 완료되면 빌드 된 폴더가 열립니다.
프로젝트 이름으로 된 exe 파일을 실행하면
Unity 로고와 함께 게임이 실행됩니다.



게임을 테스트해보고 즐겨보세요. 그리고 다른 프로젝트도 고민하고 또 진행해보세요. 앞으로 더 재밌는 게임을 만들어봅시다. 수고하셨습니다.

Website
sunrin-rg.github.io/
Facebook
facebook.com/sunrintrg/



