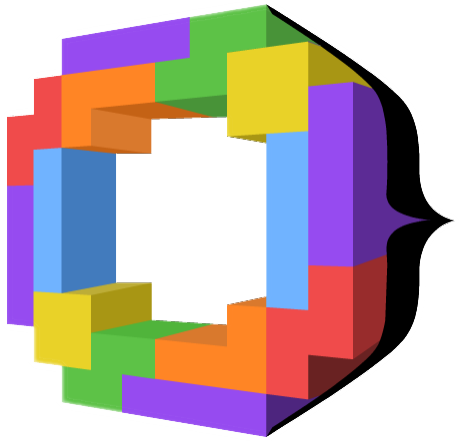
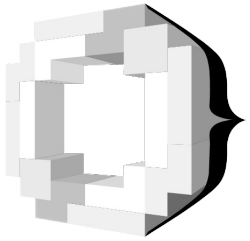


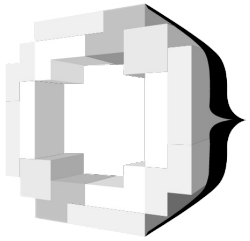
# 2023 유니티 중급반 2주차

오파츠 12기 여정인, 유태환



# Git 설치 및 Github 가입

- Git 설치
  https://goddaehee.tistory.com/216
- Github 가입
  https://goddaehee.tistory.com/218
- Github 학생 계정 인증 (하루정도 걸려요)
  https://goddaehee.tistory.com/219
- GUI로 깃 다루기 (하나만 설치해도 돼요! 수업은 일단 Github Desktop 사용하겠습니다.)
  - Github Desktop 설치
    https://desktop.github.com/
  - Git Kraken 설치 (학생 계정 인증 후 프로 버전 사용 가능)
    https://www.gitkraken.com/



# Git 기본 명령어

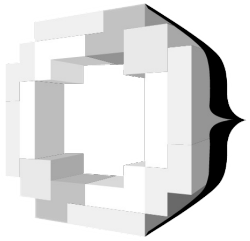
- 사용자 설정

```
git config --global user.name “이름”
git config --global user.email “이메일”
```

- 레포지토리 초기화 (.git 폴더 생성)

```
git init
```

.git 폴더는 숨김 파일로 처리되어 보이지 않을 수 있어요.
이 폴더 안에 버전관리를 위한 파일들이 들어있습니다.

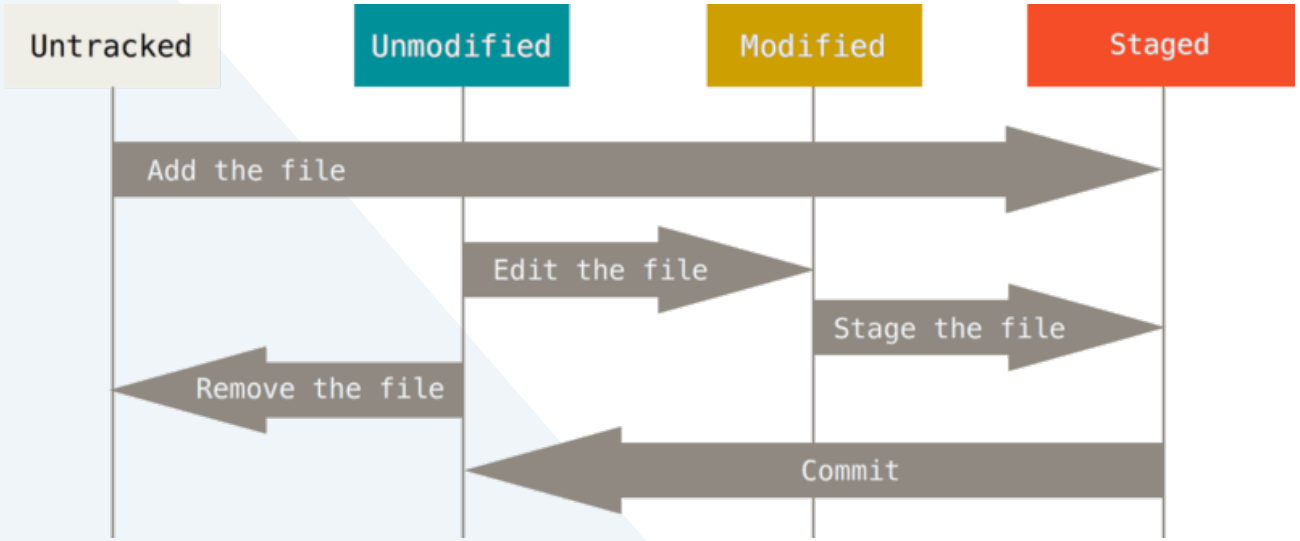


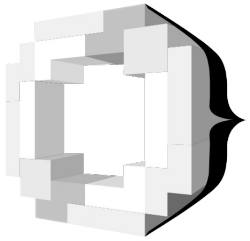
# Git 기본 명령어

- **Staging (git 커밋 전 준비)**

```
git add 파일명
git add . (전체 파일 스테이징)
```

Commit(커밋)은 수정사항을 깃 레포지토리에 저장한다는 뜻입니다.
파일을 커밋하기 전에 준비하는 것을 스테이징이라고 표현합니다.





# Git 기본 명령어

- **Commit**

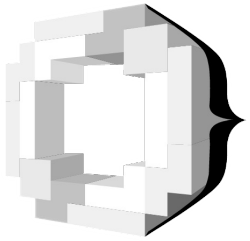
```
git commit -m “커밋 메시지”
```

커밋 메시지에는 어떤 수정이 이루어졌는지 간단하게 작성합니다.
보통 프로젝트 관리의 용이함과 가독성을 위해 규칙을 정하여 정해진 형식으로 작성합니다.

- **Remote 저장소 연결**

```
git remote add origin 저장소 주소
```

리모트 저장소는 쉽게 말해 서버를 의미합니다.
origin은 리모트 저장소 이름입니다. 기본은 origin이고 리모트 저장소를 여러 개 연결한다면 다른 이름을 사용해도 됩니다.



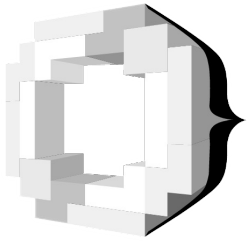
# Git 기본 명령어

- **Push**

```
git push -u origin main
```

푸시는 리모트 저장소에 레포지토리를 업로드하는 것 입니다.
origin은 리모트 저장소, main은 브랜치 이름이며, -u 옵션을 주면 이후에 push할 때 이름을 명시해주지 않아도 됩니다.

리모트 저장소에 내가 push하기 이전에 누군가가 push를 했다면 나는 push를 하지 못합니다.
이럴 때는 먼저 저장소의 변경 사항을 내 저장소에 반영을 해준 다음 push를 해주어야 합니다.



# Git 기본 명령어

- **Pull**

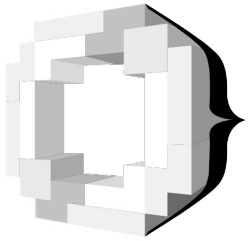
```
git pull
```

풀은 반대로 리모트에서 로컬로 가져오는 것입니다.
깃이 자동으로 변경된 사항을 작업중인 폴더에 합쳐 버리기 때문에 (Merge) 변경사항이 있다면 커밋을 먼저 해줘야 합니다.

- **Fetch**

```
git fetch
```

자동으로 합쳐지는게 싫다면 수동으로 확인하고 직접 합쳐줄 수 있습니다. 그럴 때는 pull 대신 fetch를 합니다.



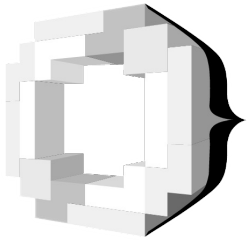
# Git으로 협업할 때

- 내 변경 사항이 있다면

1. add (스테이징)
2. commit
3. pull
4. push

- 내 변경 사항이 없다면 (자주 해주자!)

1. pull (자동으로 합쳐지고 커밋됨)
2. add
3. commit
4. push



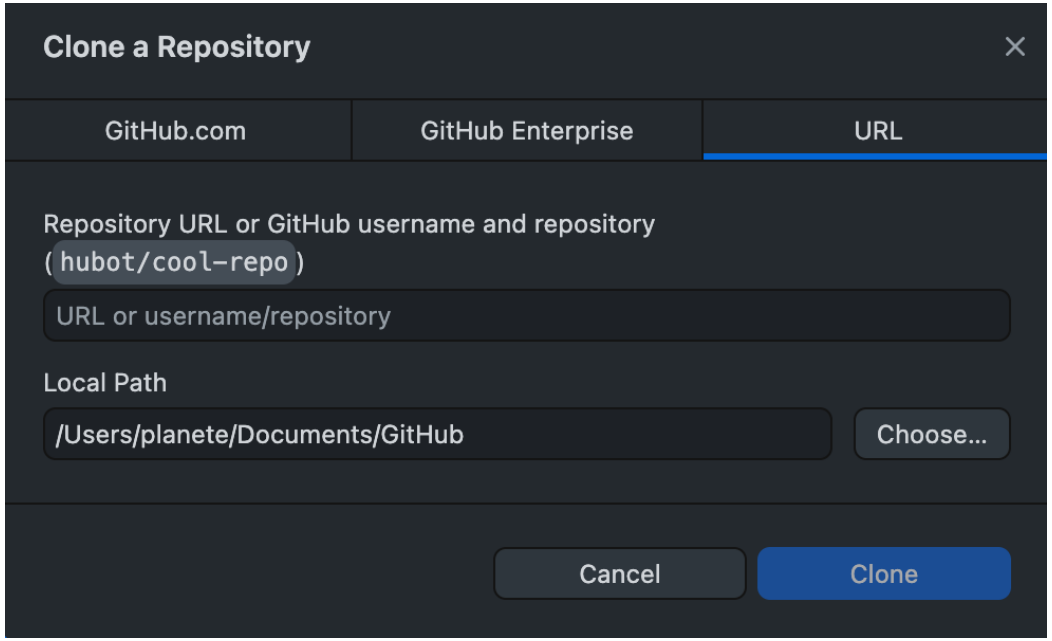
# 프로젝트 내려받기

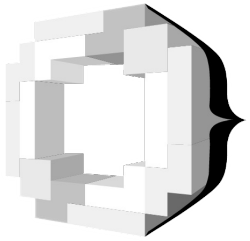
- 리모트 저장소를 내 노트북에 내려받고 싶어요!

git clone 저장소 주소

git clone https://github.com/OOPARTS-2023-Unity/Week-1.git

- 수업에 사용할 프로젝트 깃허브 레포입니다.
  원하시는 장소에 터미널을 열고 클론해주세요.
- 깃허브 데스크탑으로 클론 하셔도 됩니다.

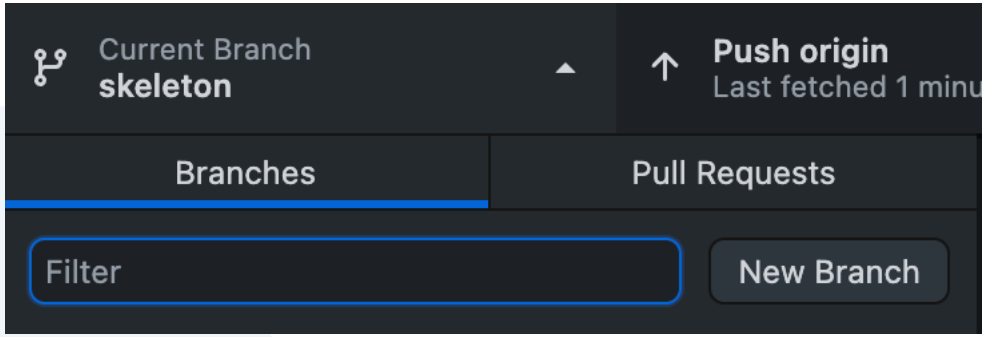


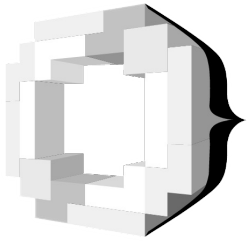


# 브랜치 생성해보기

- **새 브랜치 생성**
  **브랜치는 다른 브랜치와 독립적으로 작업할 수 있는 공간입니다.**

- **깃허브 데스크탑에서 Current Branch 선택 후 New Branch를 눌러주세요**



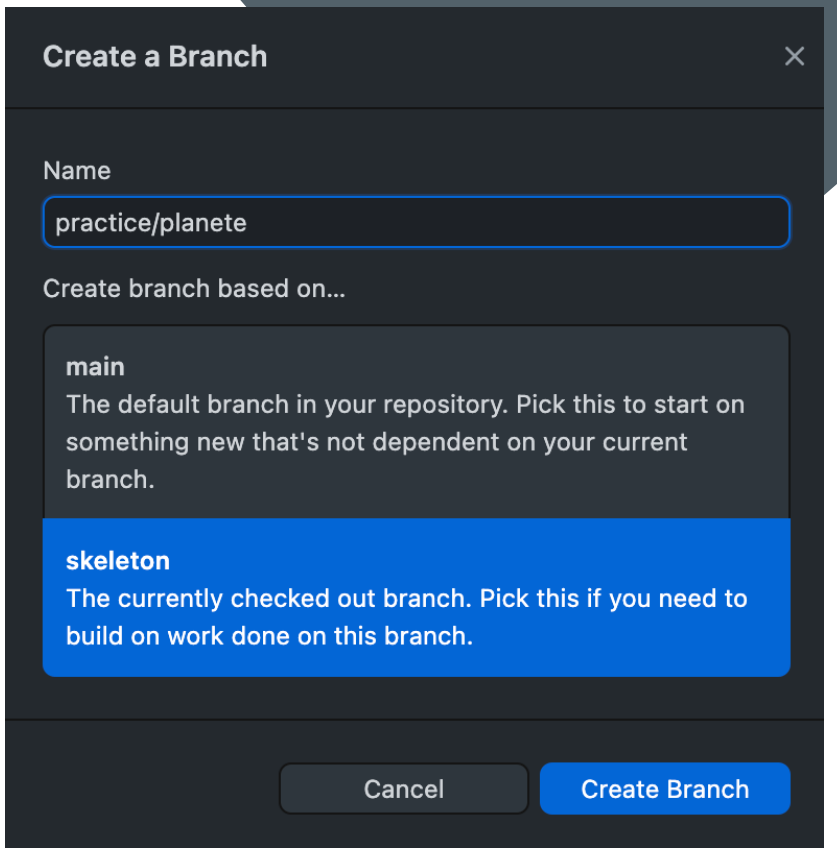


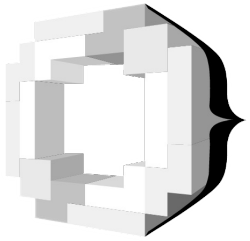
# 브랜치 생성해보기

- 브랜치 이름은
  practice/닉네임
  으로 하겠습니다.

- 기존 브랜치 중 어떤 브랜치에서 새롭게 시작할지 결정할 수 있습니다.
  skeleton으로 선택하고 브랜치를 만들겠습니다.

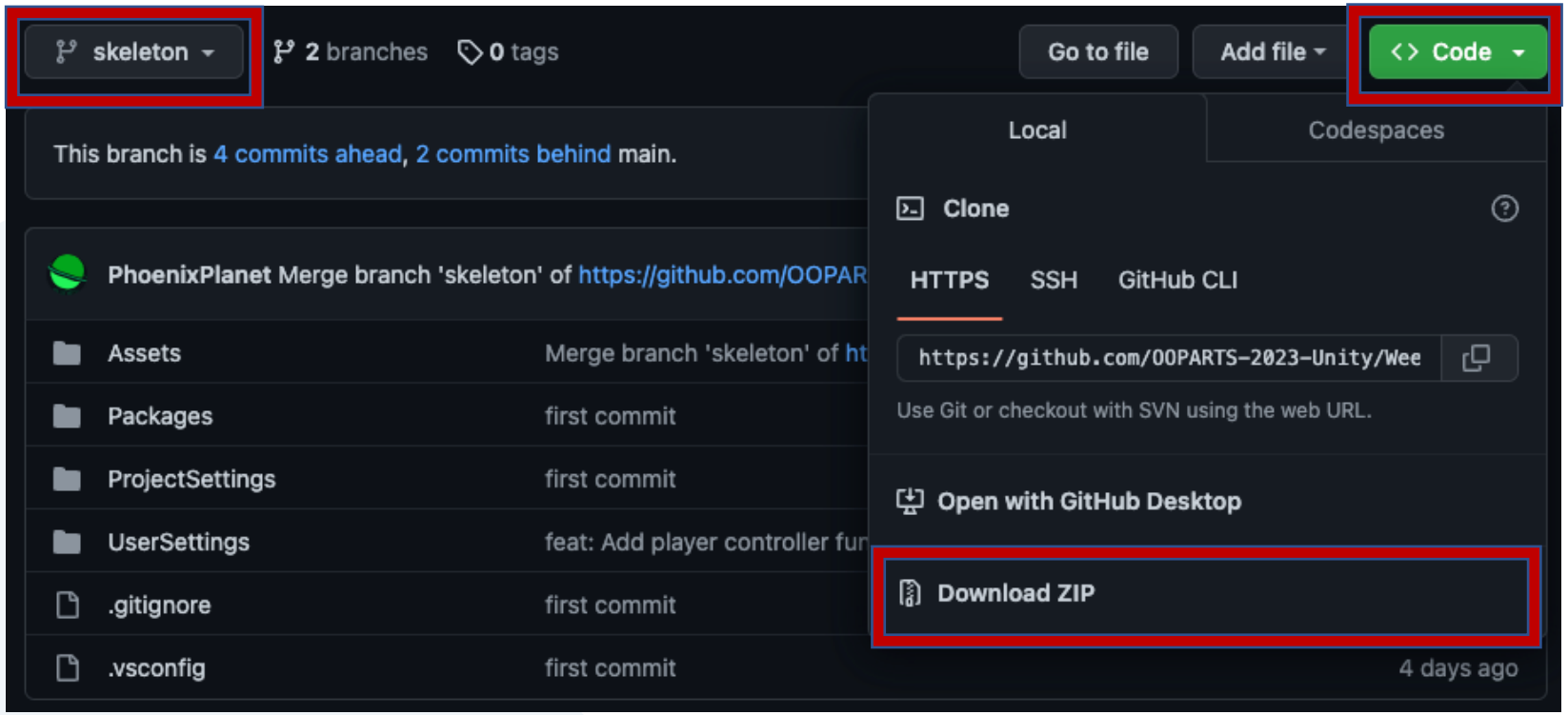
- 아마도 아직 Push를 할 수는 없을거에요.
  해당 레포지토리의 콜라보레이터로 추가되거나,
  (github 가입 후 원하시는 분들은 아이디/이메일 알려주시면 추가해드리겠습니다)
  아니면 '포크'를 통해 자신의 레포지토리로 복제 해야합니다.

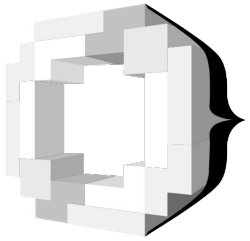




# 깃이 없다면

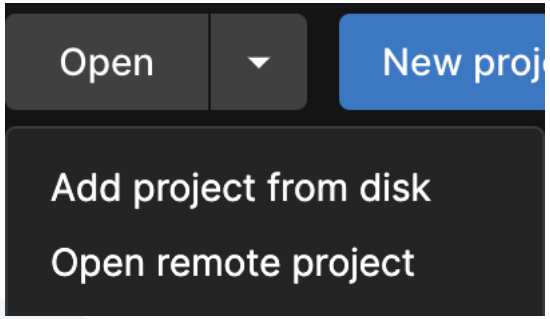
- https://github.com/OOPARTS-2023-Unity/Week-1/tree/skeleton
  여기에서 skeleton으로 브랜치 변경해주신 후에 Code -> Download ZIP으로 다운로드가 가능합니다.



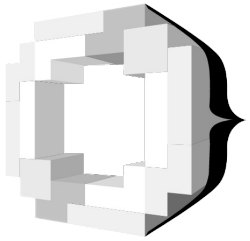


# 프로젝트 열기

- 클론 받은 폴더를 유니티 허브에서 열어줍니다.

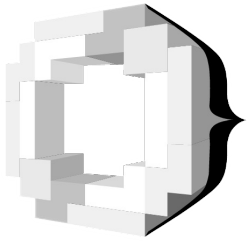



- 유니티에서는 실행 도중에 생성되고 삭제되는 임시 파일들이 많아요
  꼬임을 방지하기 위해서는 프로젝트를 닫고 스테이징 및 커밋해주는 것이 안전합니다



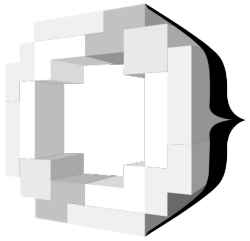
# 목차

- 타일맵으로 맵 그려보기
- 콜라이더로 충돌 감지하기
- 캐릭터에 물리 적용하고 이동하기



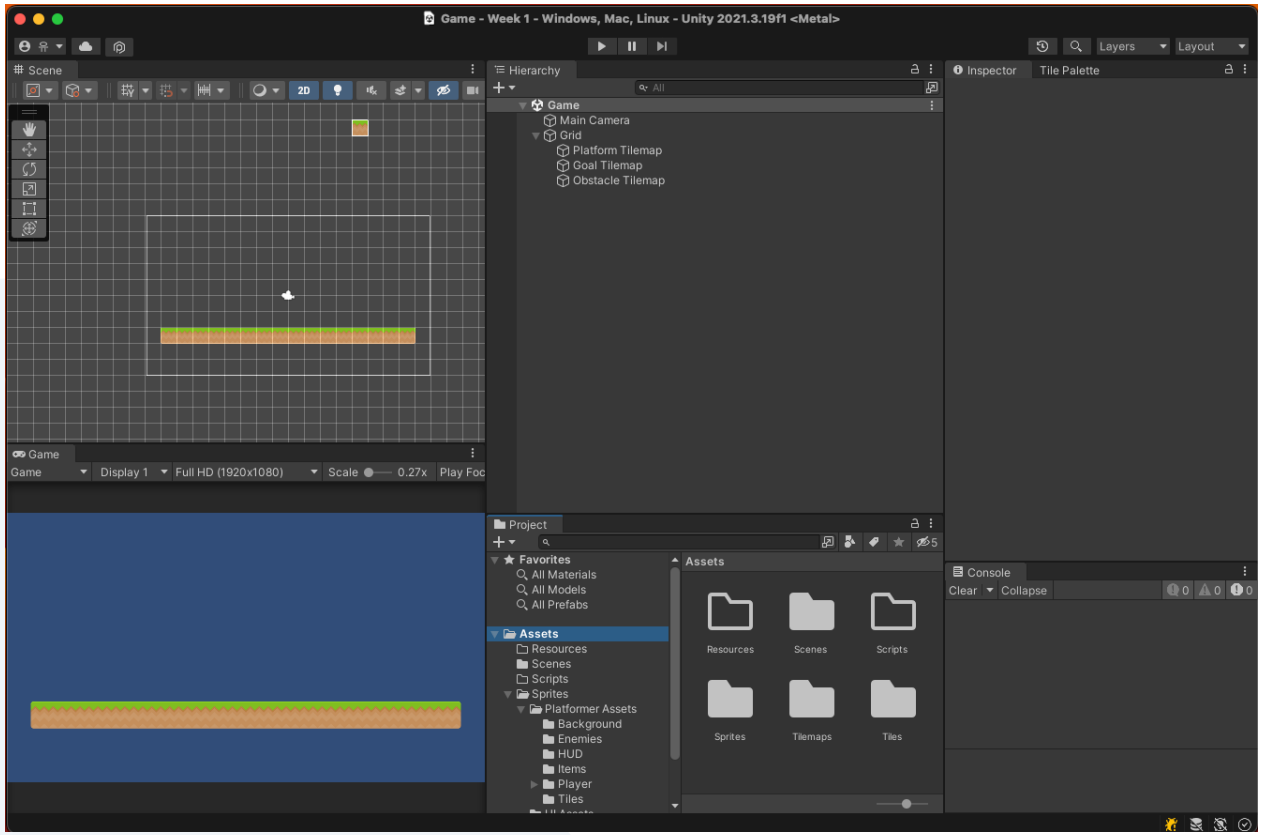
# 시작하기 전에

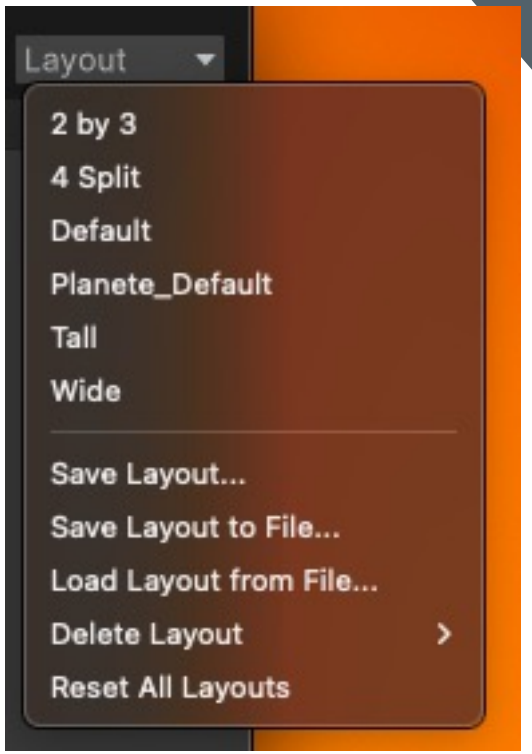
- 이미 아래 설명한 대부분은 구현되어 있어요.
- 다만 PPT 내용을 따라가면서 각 부분이 어떻게 설정되어 있는지 살펴봐요

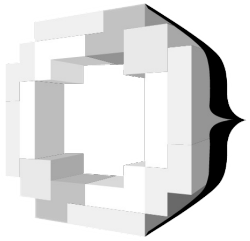


# 유니티 에디터 레이아웃

- 제 화면은 여러분과 다를거에요

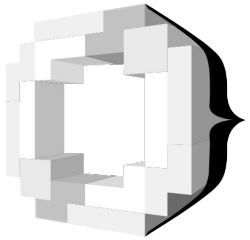




# 게임 씬

- Scenes 폴더에 Game이라는 이름의 씬을 생성해 두었어요.
  - 씬은 유니티에서 한 장면을 의미합니다.
  - 보통 하나의 씬이 하나의 레벨을 포함해요.
  - 씬은 Hierarchy에서 수정 가능해요.
  - 플레이 상태에서 씬을 수정하면 반영되지 않아요!



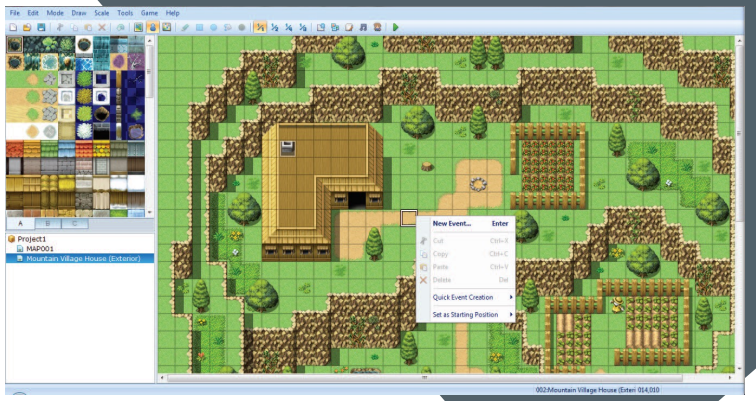
# 타일맵이란?

정사각형의 그림(타일)을 배치하여 맵을 제작하는 기법.

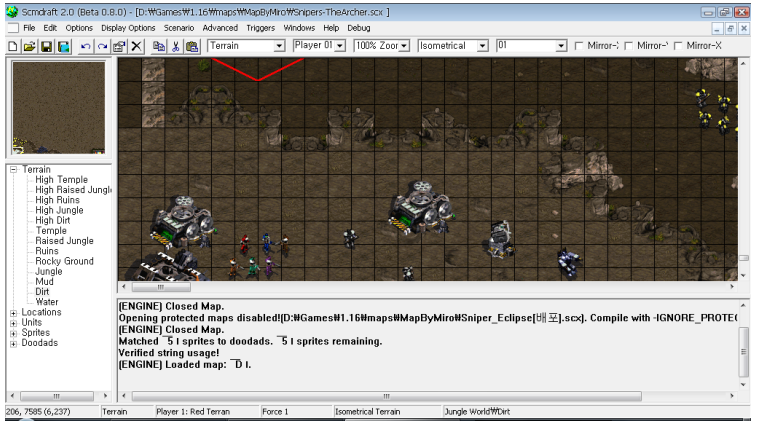
타일 몇 개만으로 다양한 맵을 만들 수 있어요.

쉽고 빠르게 맵을 완성할 수 있어요.

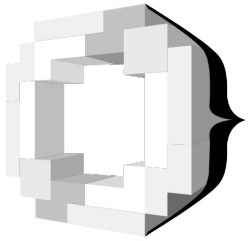
캐릭터 상호작용도 간단하게 구현할 수 있어요.



RPG Maker

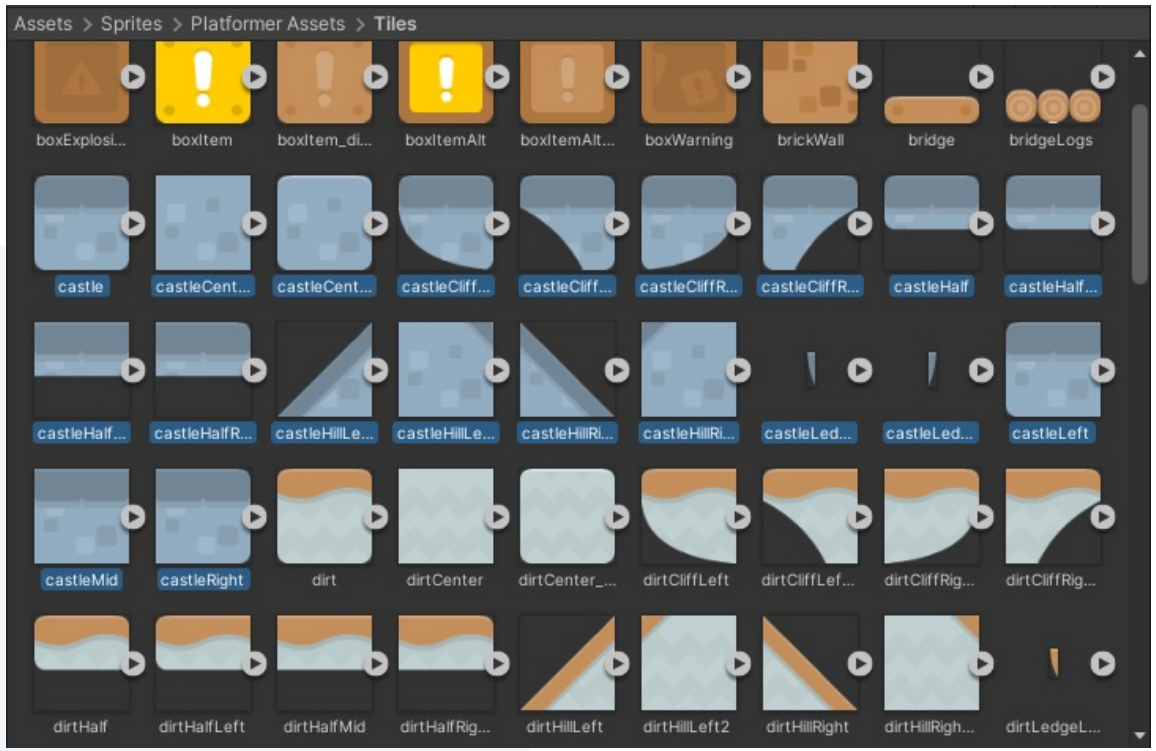


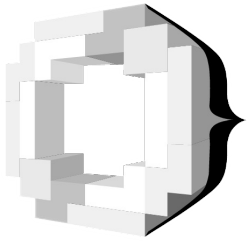
스타크래프트 맵 에디터



# 에셋 설정하기

- Sprites/Platformer Assets/Tiles 폴더





# 인스펙터

Inspector에서 세팅을 해줍니다.

- Texture Type : Sprite
- Pixels Per Unit: 70

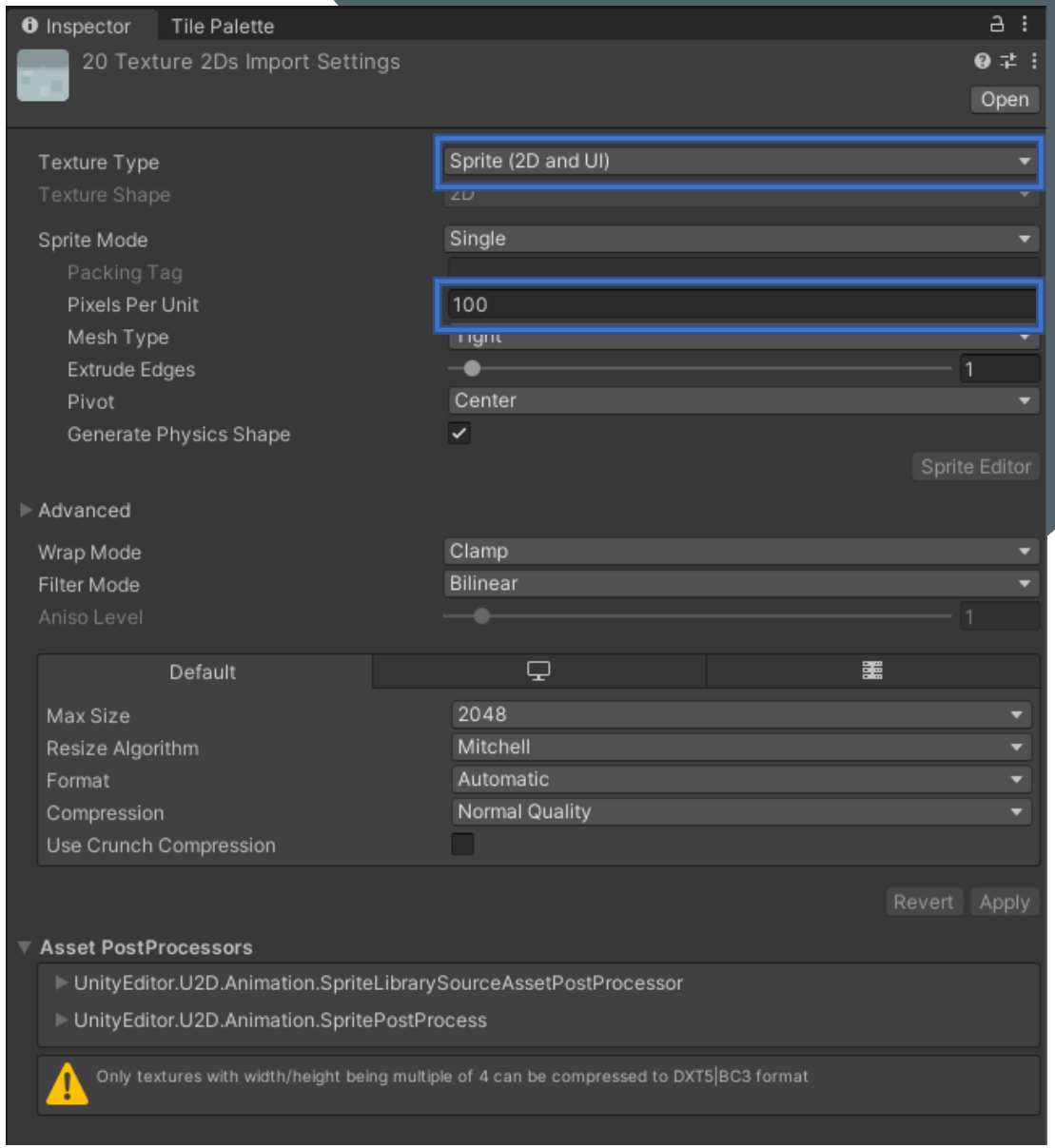
픽셀이 깨져보이거나 뿌옇다면

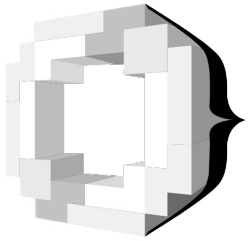
- Filter Mode: Point (no filter)

  이 설정을 적용하면 안티 앨리어싱이 적용되지 않습니다.

- Compression: None

  이 설정을 적용하면 텍스쳐가 압축되지 않습니다.



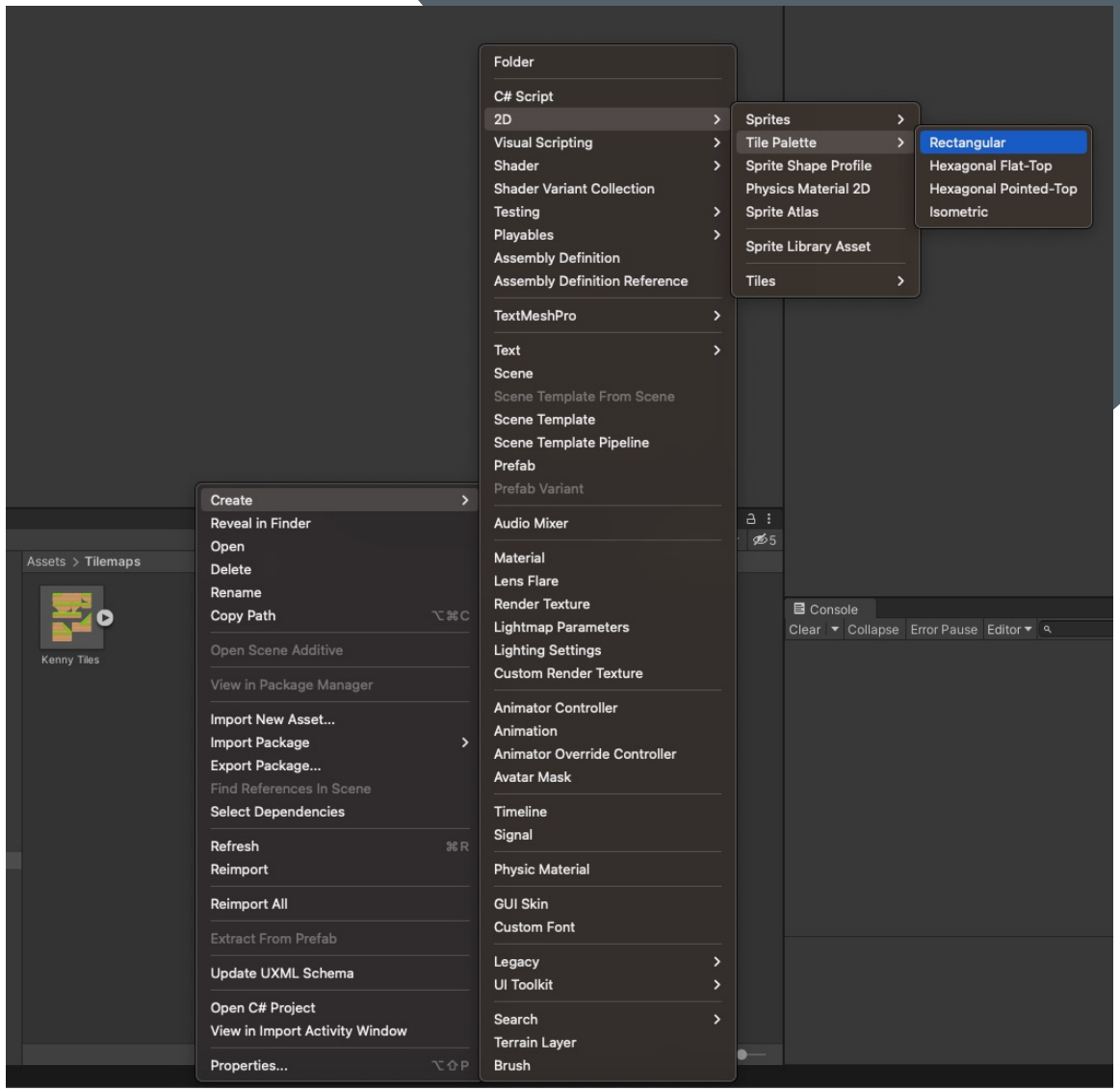


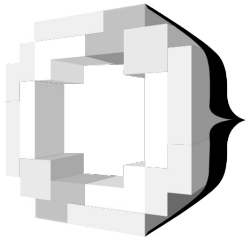
# 타일 팔레트

Tilemaps 폴더에서 우클릭 후

Create -> 2D -> Tile Palette -> Rectanguar
을 선택해줍니다.

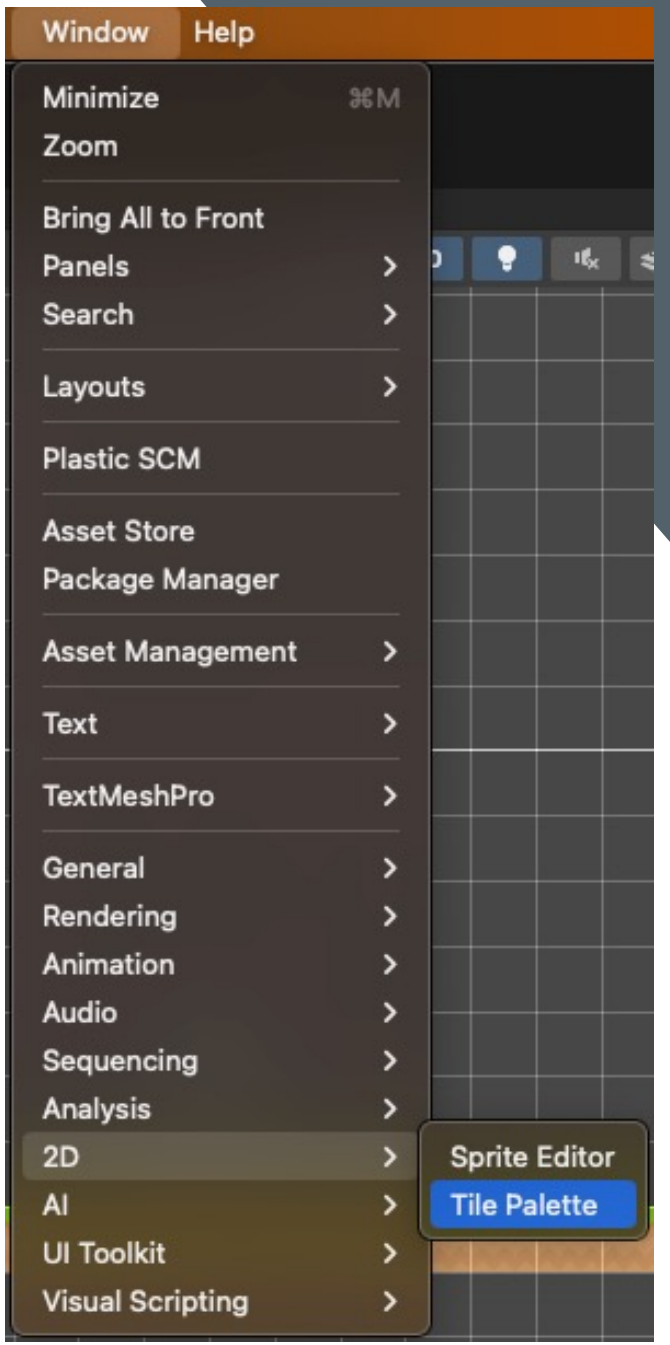
이름은 Castle Tile Palette 라고 하겠습니다.
여러분 원하시는대로 지어도 돼요!

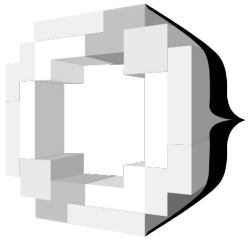




# 타일 팔레트

Window 메뉴에서 2D -> Tile Palette를 선택해줍니다.



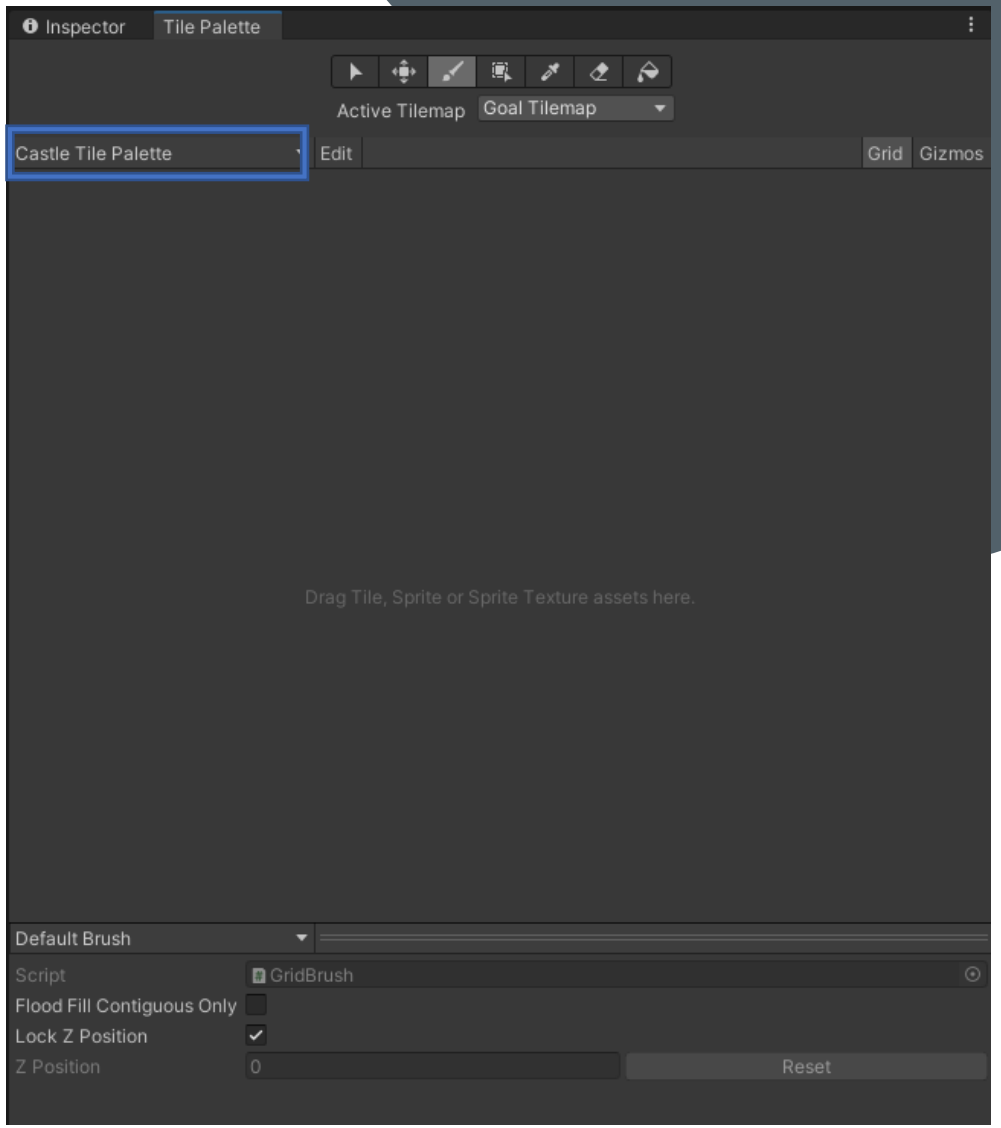


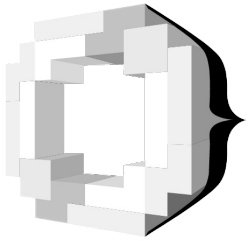
# 타일 팔레트

이런 창이 뜰거에요!

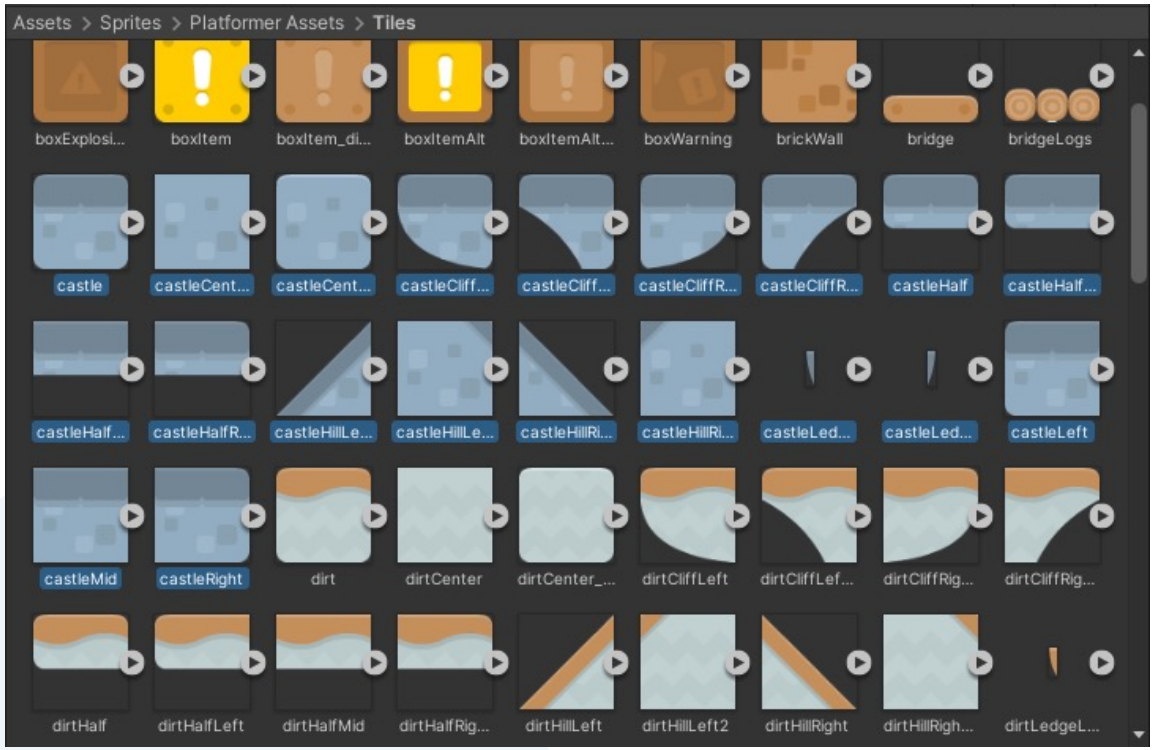
만약 이미 타일들이 보인다면 그건 제가 만들어둔 타일 팔레트에요.

표시한 곳을 눌러 새로 만든 타일 팔레트를 선택해봅시다.



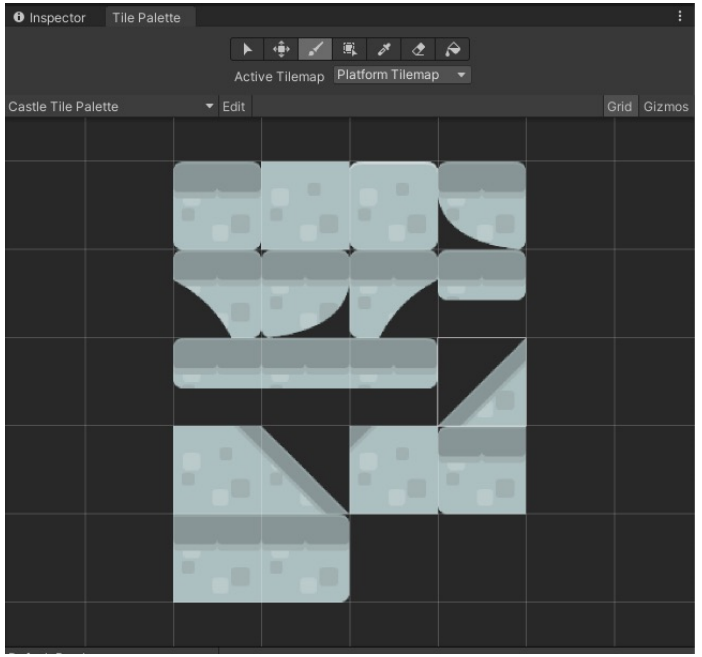


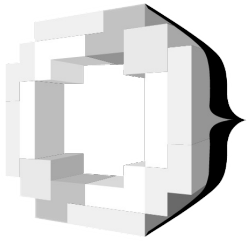
# 타일 팔레트



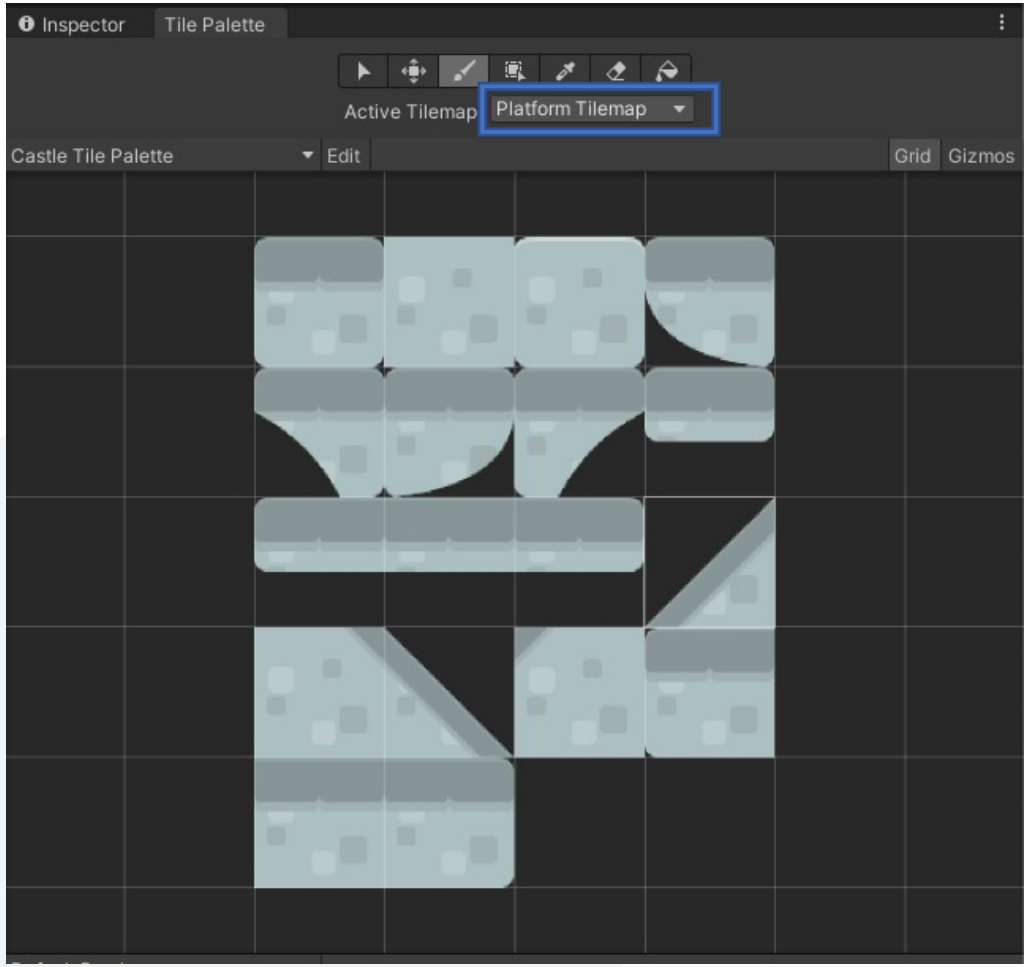

타일 팔레트 창에 드래그 앤 드롭으로 넣어줍니다.

파일을 저장하라고 창이 뜰텐데, Assets/Tiles 폴더에 새롭게 castle 폴더를 생성해주고 해당 폴더에 저장해주도록 합시다. 아래처럼 뜨면 제대로 된 거에요!





# 타일 팔레트

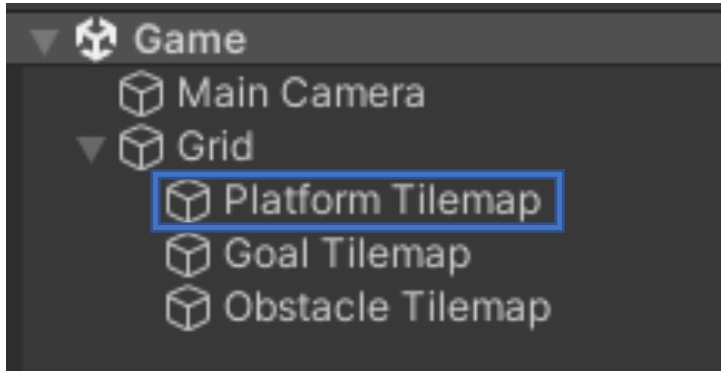


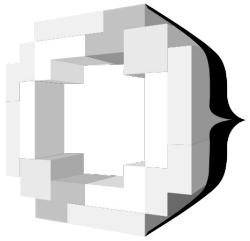

이제 씬에 맵을 그려봅시다!

Active Tilemap은 하이어라키의 어떤 타일맵에 그릴 지 선택하는 거에요.

저희는 플랫폼을 Platform Tilemap에 그릴거에요.

붓, 스포이드 등등 다양한 도구로 그려봅시다.





# 캐릭터 생성

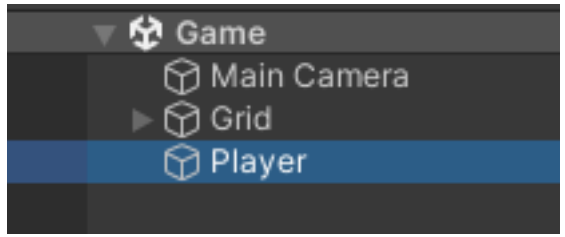
하이어라키에서 우클릭 후 Create Empty로
빈 게임 오브젝트를 생성해줍니다.

이름은 Player로 할까요?

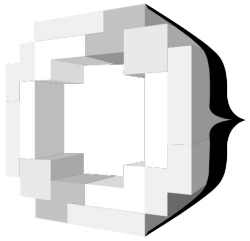
폴더에서 스프라이트들을 선택해서 빈 오브젝트로 드래그 앤드롭 해주면 자동으로 애니메이션 컨트롤러와 애니메이션 파일이 생성됩니다.

Animations 폴더에 저장해줍시다.

생성된 애니메이터 이름은 Player,
애니메이션 이름은 Player Walk로 해주겠습니다.





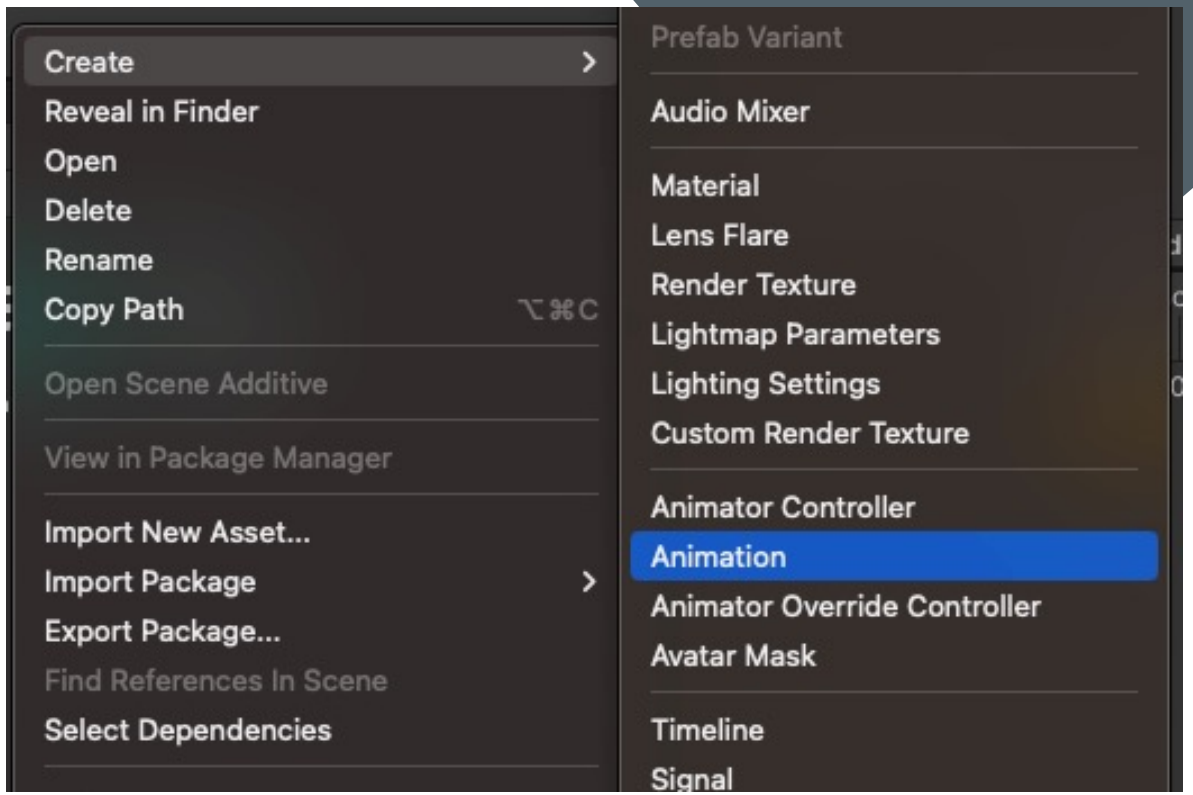


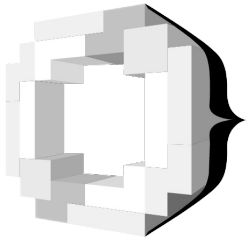
# 애니메이션

Animations 폴더에서 우클릭하여

Create -> Animation을 통해

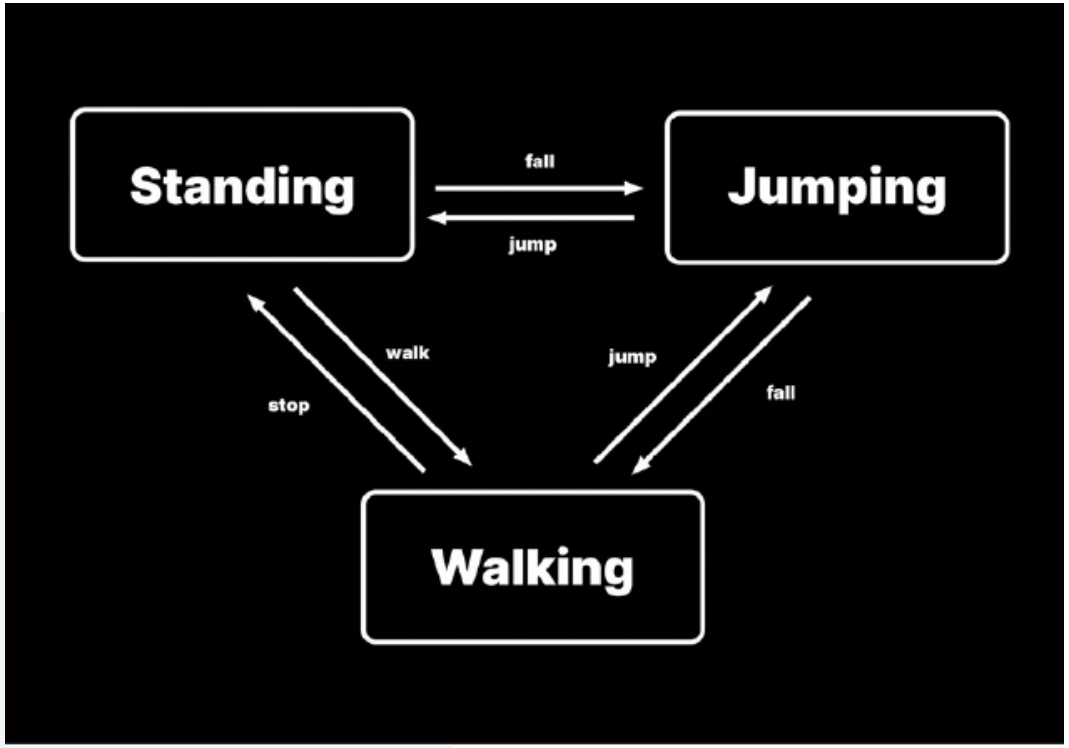
Player Idle, Player Jump
애니메이션을 생성해주겠습니다.



# 스테이트 머신 (스테이트 패턴)

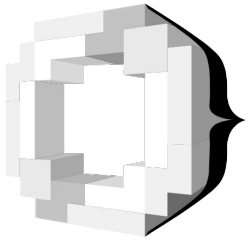
- 들어가기 전에 잠깐 스테이트 머신에 대해서 알아봅시다




오토마타를 배우셨다면 익숙한 그림일거에요

조건 여러개가 같이 엮여있다면?

```
if ( 윗쪽 버튼 )
{
    점프;
} else if ( 오른쪽 버튼 )
{
    오른쪽으로 걷기;
} else if ( 왼쪽 버튼 )
{
    왼쪽으로 걷기;
}
```



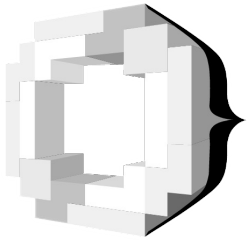
# 스테이트 머신 (스테이트 패턴)

- **방법 1 - enum 사용하기**

```
public enum PlayerControllerState
{
    Idle,
    Walk,
    Jump
}
```

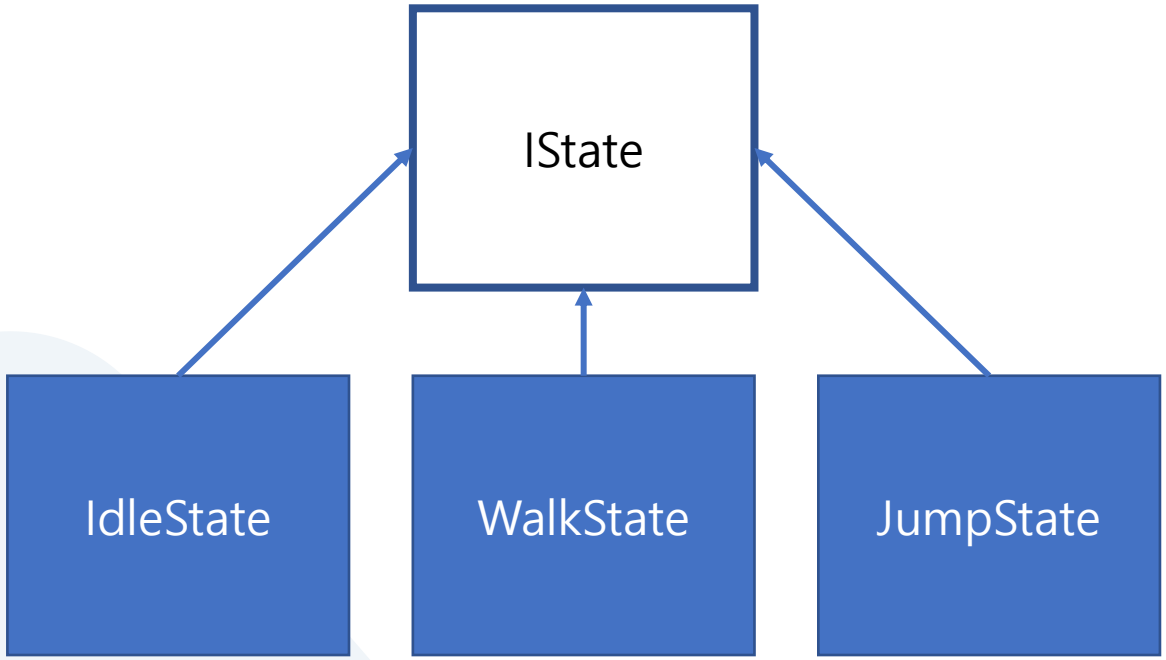
```
private void Update()
{
    GetInput();

    switch (state)
    {
        case PlayerControllerState.Idle:
            Idle();
            break;
        case PlayerControllerState.Walk:
            Walk();
            break;
        case PlayerControllerState.Jump:
            Jump();
            break;
    }
}
```

**조금 나아졌지만 그래도 여전히 조건문 분기를 사용합니다.**

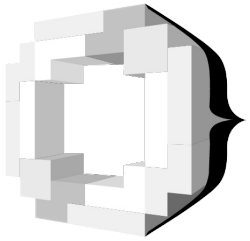


# 스테이트 머신 (스테이트 패턴)

- **방법 2 - 스테이트마다 클래스 선언해주기**




**클래스의 다형성을 활용해서 스테이트를 관리해봅시다.**



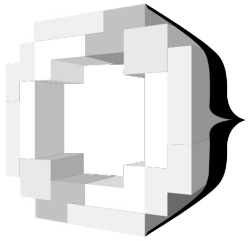
# 스테이트 머신 (스테이트 패턴)

- **방법 2 - 스테이트마다 클래스 선언해주기**

```
public interface IState
{

    public void Enter()
    {
        // code that runs when we first enter the state
    }

    public void Update()
    {
        // per-frame logic, include condition to transition to a new
state
    }

    public void Exit()
    {
        // code that runs when we exit the state
    }
}
```

**클래스의 다형성을 활용해서 스테이트를 관리해봅시다.**

```
public class IdleState : IState
{
    private PlayerController player;

    public IdleState(PlayerController player)
    {
        this.player = player;
    }

    public void Enter()
    {
        // code that runs when we first enter the state
    }

    public void Update()
    {
        // Here we add logic to detect if the conditions exist to
        // transition to another state
        …
    }

    public void Exit()
    {
        // code that runs when we exit the state
    }
}
```



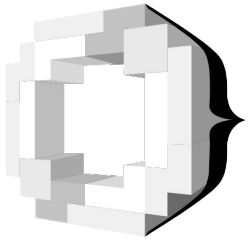
# 스테이트 머신 (스테이트 패턴)

- **방법 2 - 스테이트마다 클래스 선언해주기**

```
public void TransitionTo(IState nextState)
{
    CurrentState.Exit();
    CurrentState = nextState;
    nextState.Enter();
}

public void Update()
{
    if (CurrentState != null)
    {
        CurrentState.Update();
    }
}
```

**CurrentState를 바꿔주는 것 만으로 스테이트를 변경할 수 있습니다.**

**추후에 스테이트를 추가하더라도 다른 스테이트에 영향이 없을 뿐더러 메인 로직을 크게 수정할 필요가 없어요!**



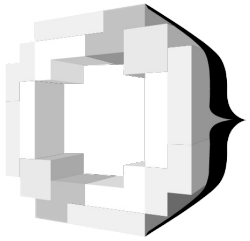
# 애니메이션

- 유니티 애니메이션은 스테이트 머신,

  그러니까 유한상태 머신으로 관리되고 있어요.

- 따라서 스테이트 패턴과 같이 사용하기에 좋아요!

- 일단... 일반적인 방법으로 애니메이션을 구현해봅시다



# 애니메이션

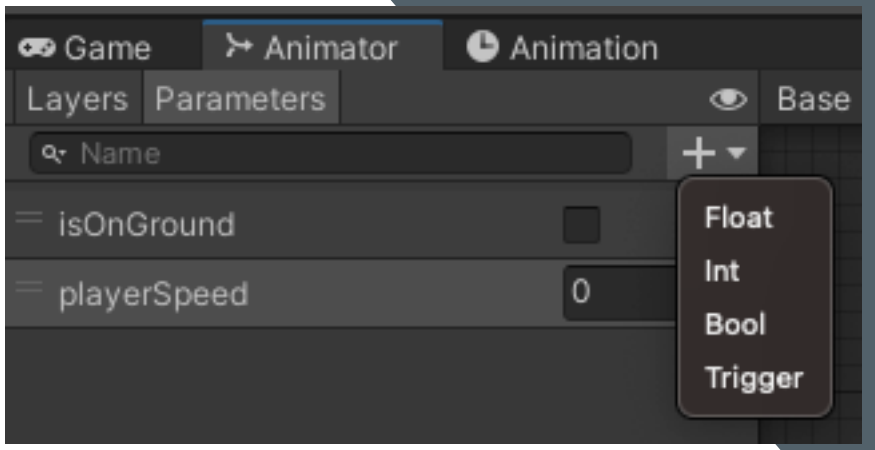
Player 애니메이터를 더블클릭하면 오른쪽 같은 창이 뜰거에요.

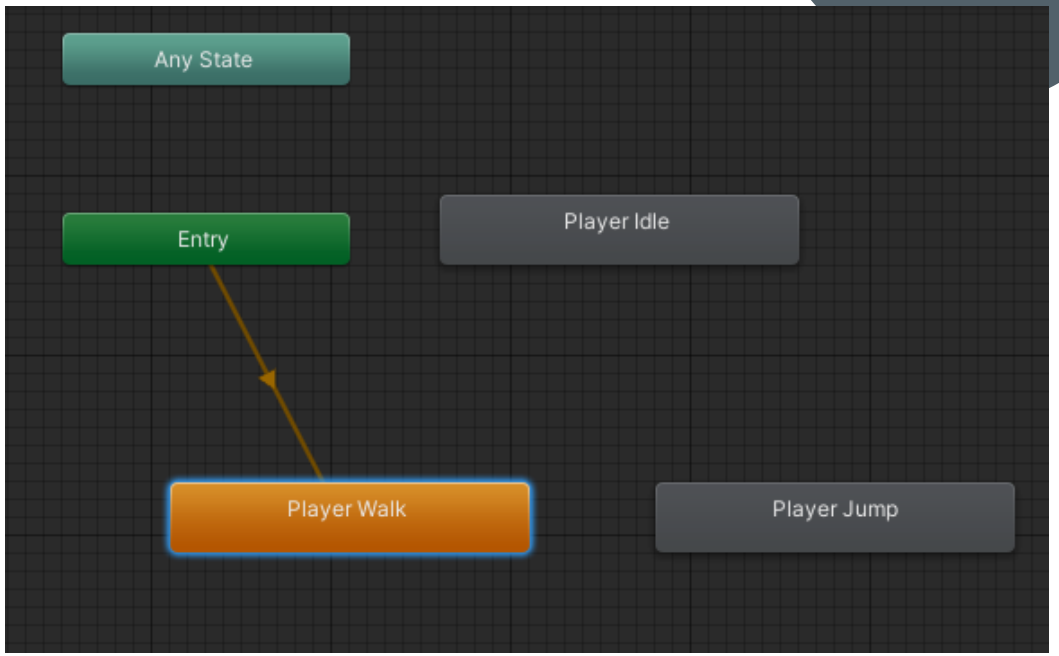
Parameters 탭을 선택해주고, + 버튼을 눌러

- Bool Type의 isOnGround
- Float Type의 playerSpeed

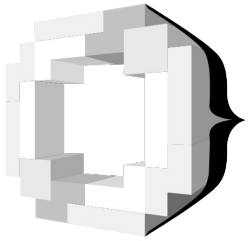
두 개의 파라미터를 생성해주겠습니다.

그리고 스테이트 머신에 Player Idle과 Player Jump 애니메이션을 드래그 앤 드롭으로 추가해주겠습니다.



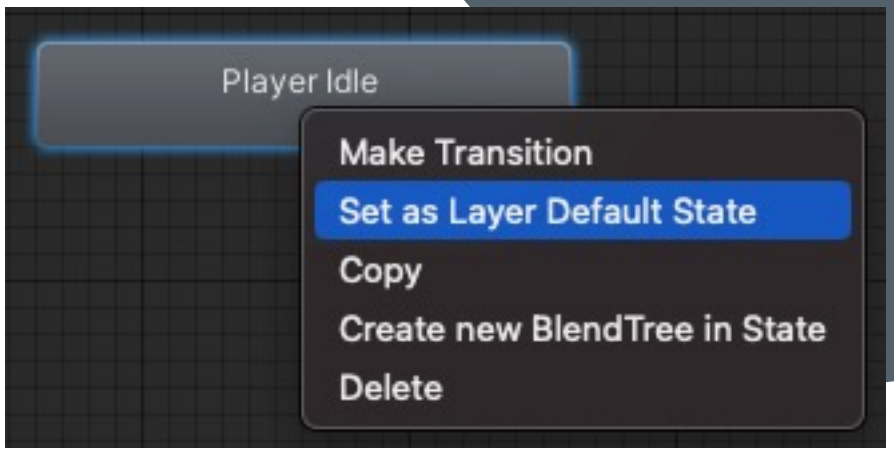

스테이트 머신! (유니티에선 애니메이션을 유한상태머신으로 관리해요)

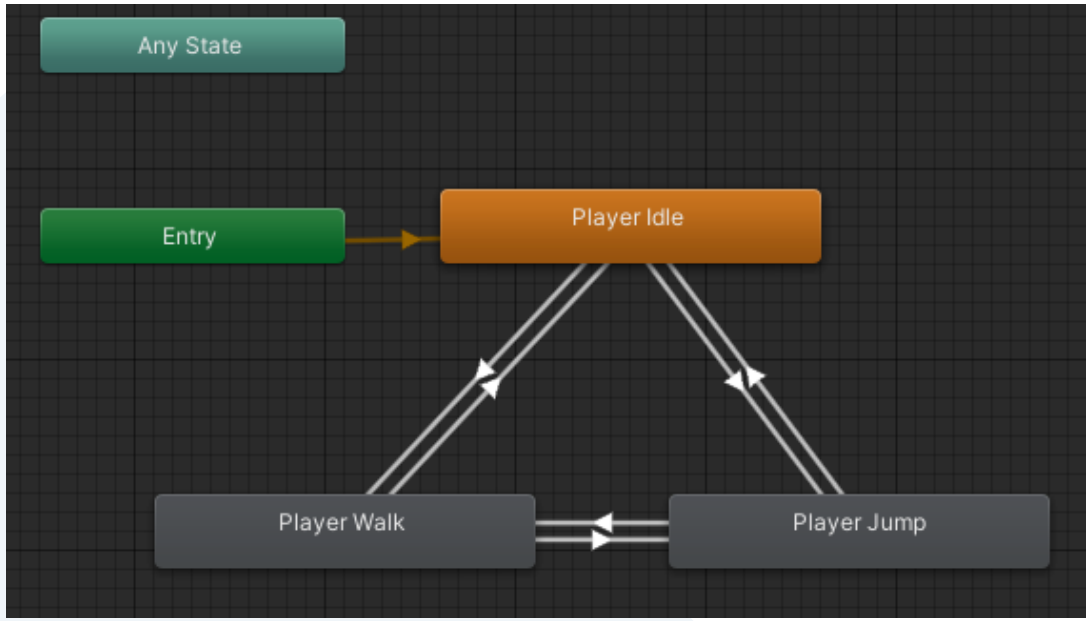


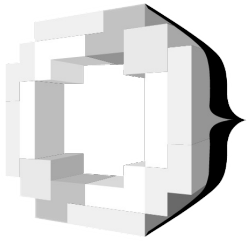
# 애니메이션

Player Idle을 우클릭하여 디폴트 스테이트로 만들어줍니다.

그리고 각 스테이트에서 Make Transition을 눌러 아래와 같이 연결해줍니다.



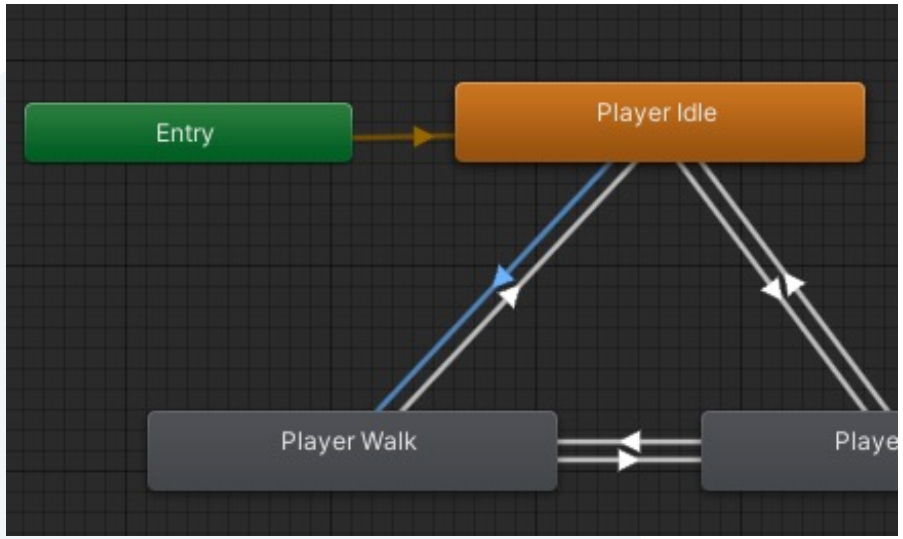



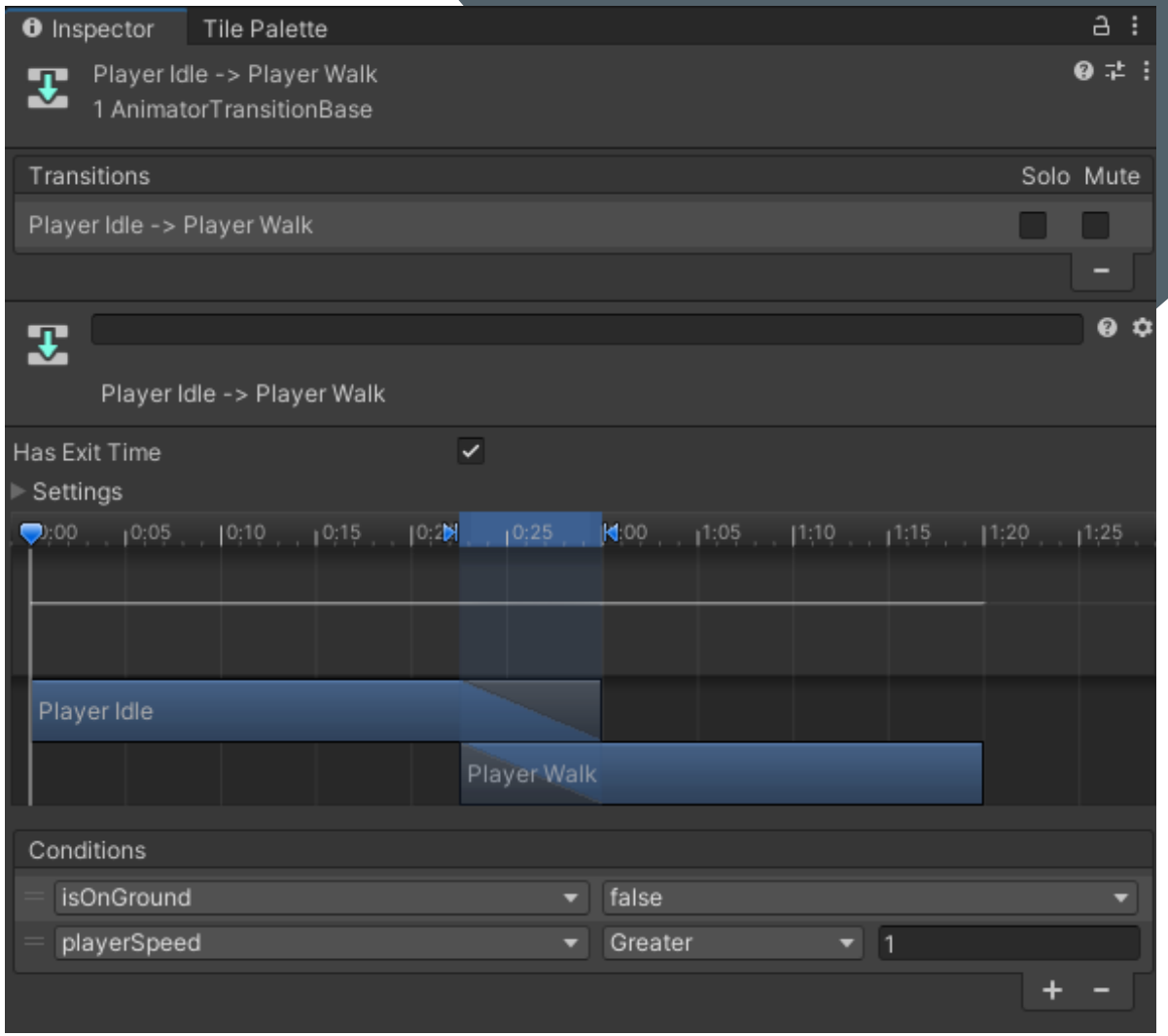


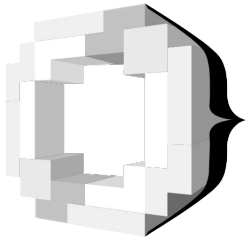
# 애니메이션

트랜지션을 클릭하면 인스펙터 창이 오른쪽처럼 뜰거에요.

Conditions에 + 버튼을 눌러 조건을 설정해줍시다.
Has Exit Time은 체크를 해제해주세요.



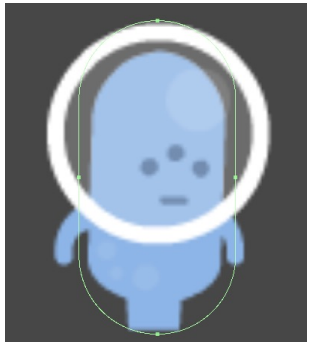


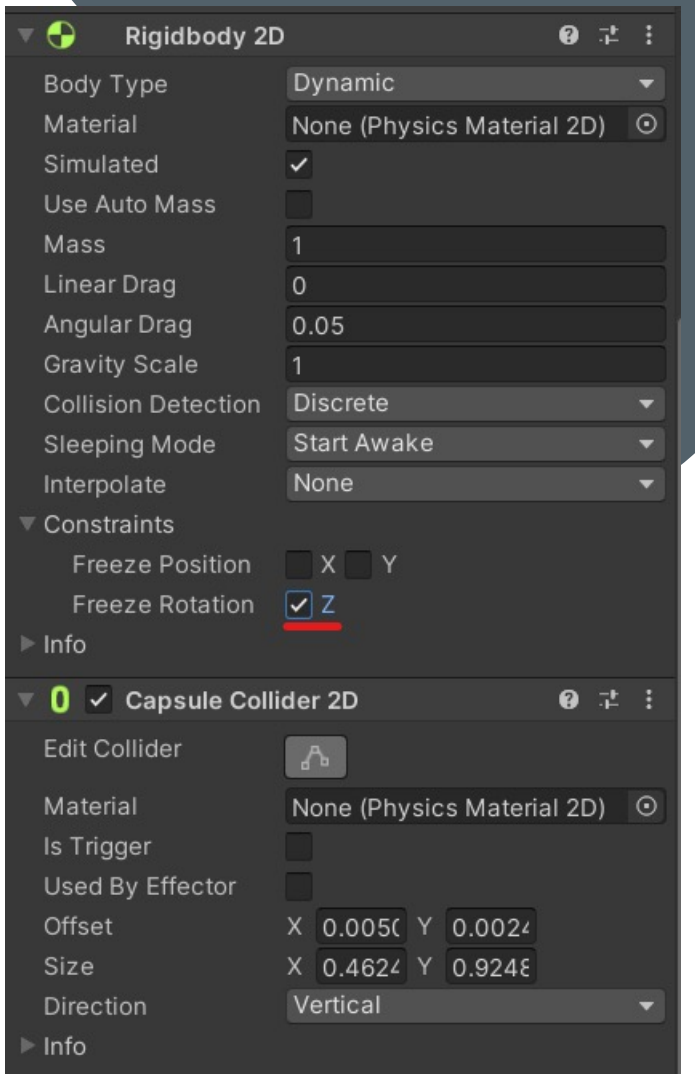
# 캐릭터 생성

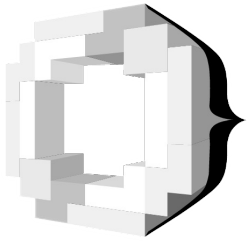
Player이 물리법칙을 적용 받을 수 있도록 Rigidbody 2D 컴포넌트를 추가합니다. 그리고, Capsule Collider 2D도 추가해서 공허 속으로 떨어지는 걸 막습니다.

Sprite Renderer에 _walk01을 넣고 Collider 크기를 맞춥니다.

캐릭터가 넘어지면 불쌍하니까 Rigidbody에서 Z축 회전을 막아줍시다.







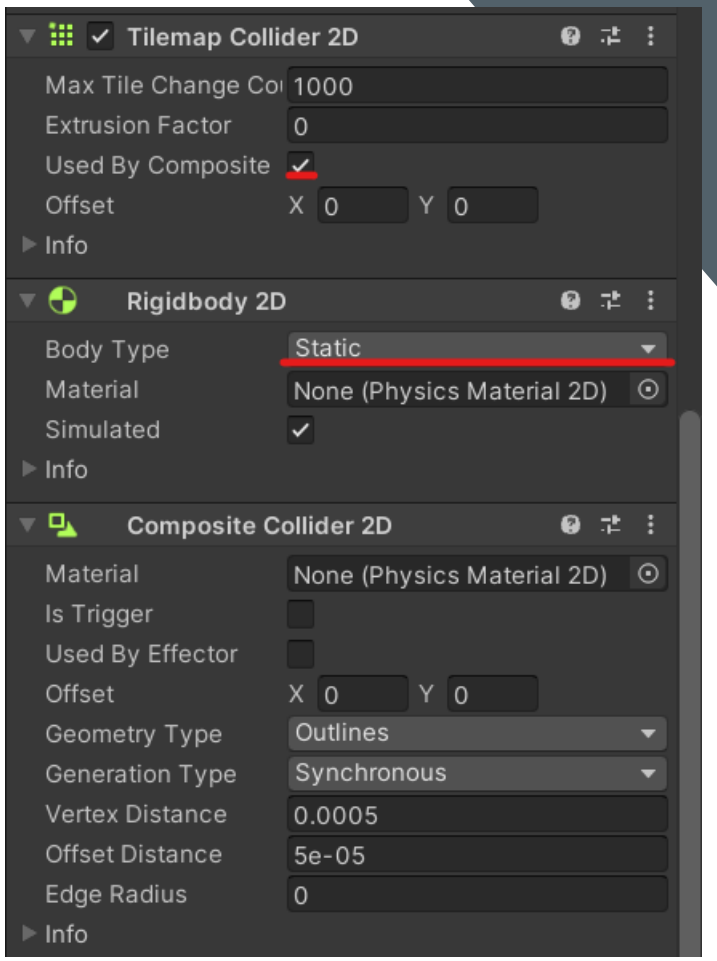
# 타일맵 Collider

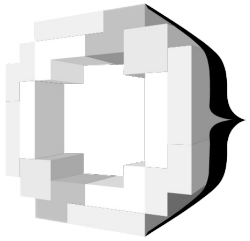
이런! 타일맵에 Collider이 빠졌군요.

Tilemap Collider 2D와 Composite Collider 2D를 추가합니다. (Rigidbody 2D는 자동으로 따라와요.)

Tilemap Collider 2D에 Used By Composite을 체크해서 Composite Collider 2D에 연결해줍니다.

맵 자체가 떨어지면 아포칼립스 장르가 되버립니다.
Rigidbody 2D의 Body Type을 Static으로 바꿔 맵을 고정해줍니다.



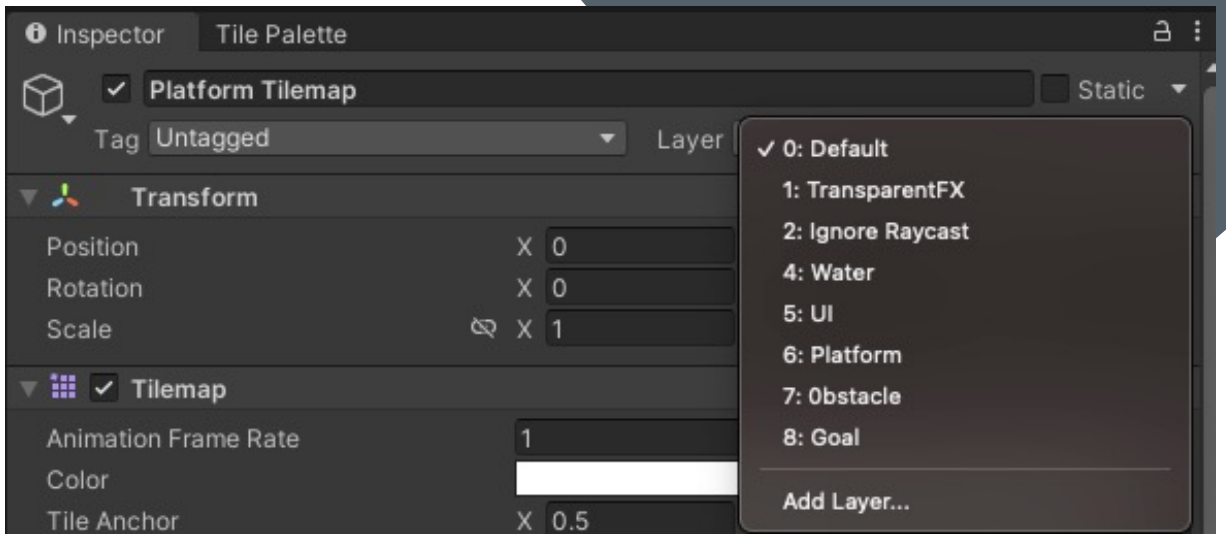


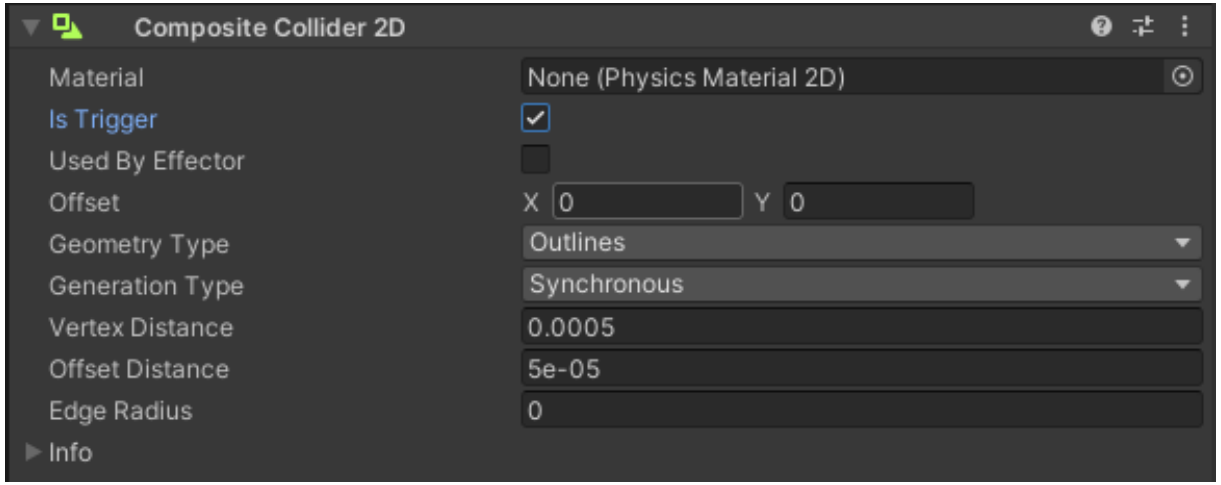
# 타일맵 Collider

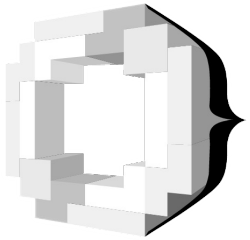
각 타일맵에 레이어를 설정해줍니다.

Add Layer로 새 레이어들을 추가해주고, 각각 알맞는 레이어를 추가해주도록 하겠습니다.

Goal과 Obstacle 타일맵에는 IsTrigger에 체크표시를 해주도록 하겠습니다



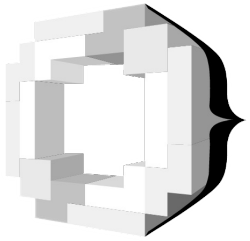





# 타일맵 Collider

이렇게 설정해 둔 레이어와 콜라이더는
후에 이벤트 처리에 활용됩니다.

```
// 타일맵에 캐릭터가 닿을 때 이벤트를 처리합니다.
void OnCollisionEnter2D(Collision2D other)
{
    if (other.gameObject.layer == LayerMask.NameToLayer("Platform"))
    {

    }
}

// 타일맵에 캐릭터가 떨어질 때 이벤트를 처리합니다.
void OnCollisionExit2D(Collision2D other)
{
    if (other.gameObject.layer == LayerMask.NameToLayer("Platform"))
    {

    }
}

// 타일맵의 지형 외에 다른 오브젝트와 부딪혔을 때 이벤트를 처리합니다.
void OnTriggerEnter2D(Collider2D trigger)
{
    if (trigger.gameObject.layer == LayerMask.NameToLayer("Goal"))
    {

    }
    if (trigger.gameObject.layer == LayerMask.NameToLayer("Obstacle"))
    {

    }
}
```



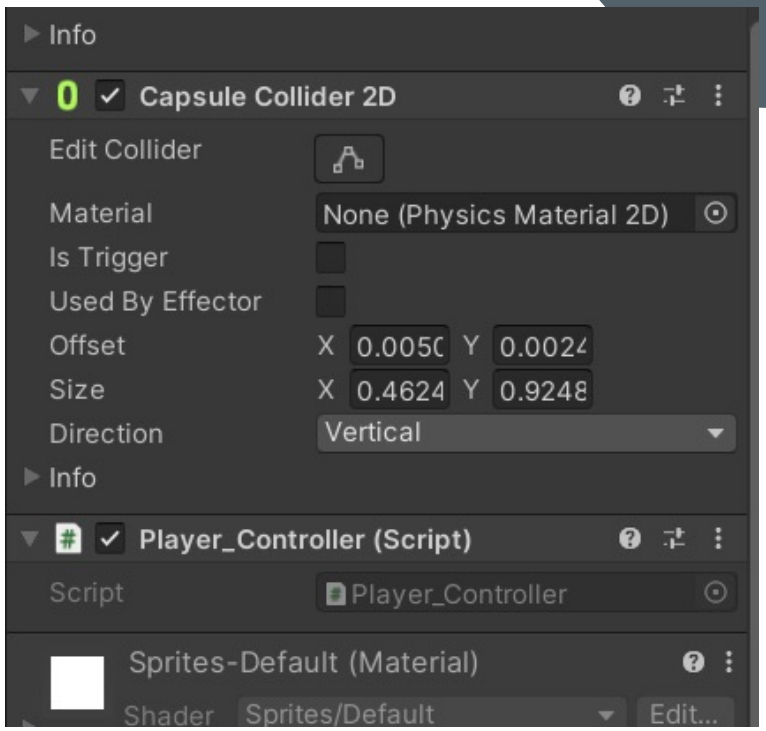
# 캐릭터 이동

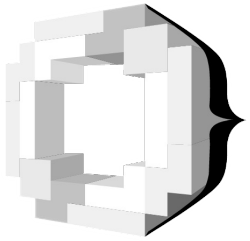
플레이를 해볼까요? 맵도 가만히 있고, 캐릭터도 맵 위에 넘어지지 않고 서있습니다.

이제 캐릭터를 움직여봅시다.

Script 폴더에 C# script를 생성합니다.
이름은 playerController로 정합시다.

그리고 아까 생성한 캐릭터에 컴포넌트로 달아줍시다.



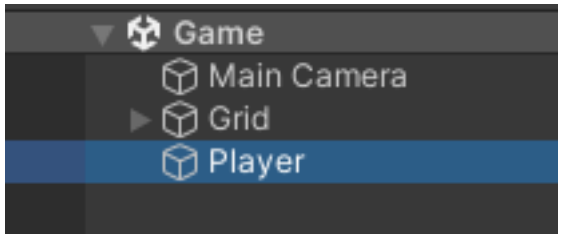


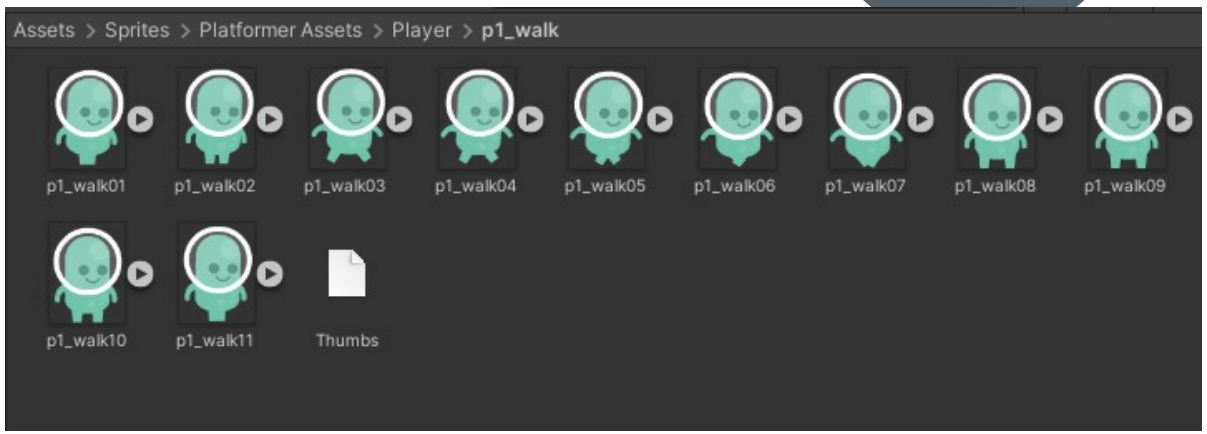
# 캐릭터 이동

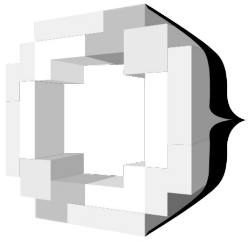
캐릭터는 우선 방향키로 움직입니다.

좌, 우로 이동하며 위쪽 방향키로 점프를 합니다.

점프는 총 두 번 뛸 수 있습니다. (더블 점프까지)





# 캐릭터 이동 관련 함수

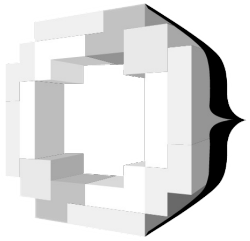
Input.GetKey(KeyCode) 키를 누르고 있을때 true를 반환합니다.

Input.GetKeyDown(KeyCode) 키를 누른 순간 true를 반환합니다.

void OnCollisionEnter2D(Collision2D other) 해당 오브젝트의 collider이 다른 오브젝트의 collider(other)과 충돌되기 시작할 때 호출되는 함수입니다.

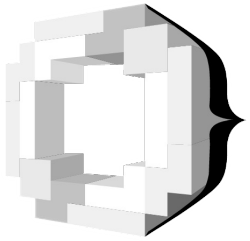
Void OnCollisionExit2D(Collision2D other) 해당 오브젝트의 collider이 다른 오브젝트의 collider(other)과의 충돌이 끝날 때 호출되는 함수입니다.

Void OnTriggerEnter2D(Collider2D trigger) 해당 오브젝트가 물리적 충돌이 아닌 가상의 영역에 닿을 때 호출되는 함수입니다.



# 코드 살펴보기

```
// Tooltip은 에디터에서 플로팅 도움말을 띄워줍니다.
[Tooltip("Jump power")]
public float JUMP_POWER;

[Tooltip("Maximum count of jump")]
public int MAX_JUMP_COUNT;

[Tooltip("Accelation of x-axis")]
public float ACCELATION;

[Tooltip("Maximum speed of x-axis")]
public float MAX_SPEED;

[Tooltip("Initial Position of the Character")]
public Vector3 INIT_POSITION;
```



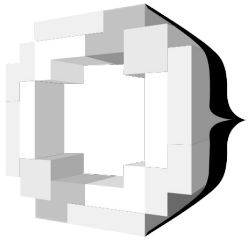
# 코드 살펴보기

```
private int currentJumpCount;
```

```
private enum KeyInput
{
    UP,
    LEFT,
    RIGHT
}

private bool[] keyInputCheck;
```

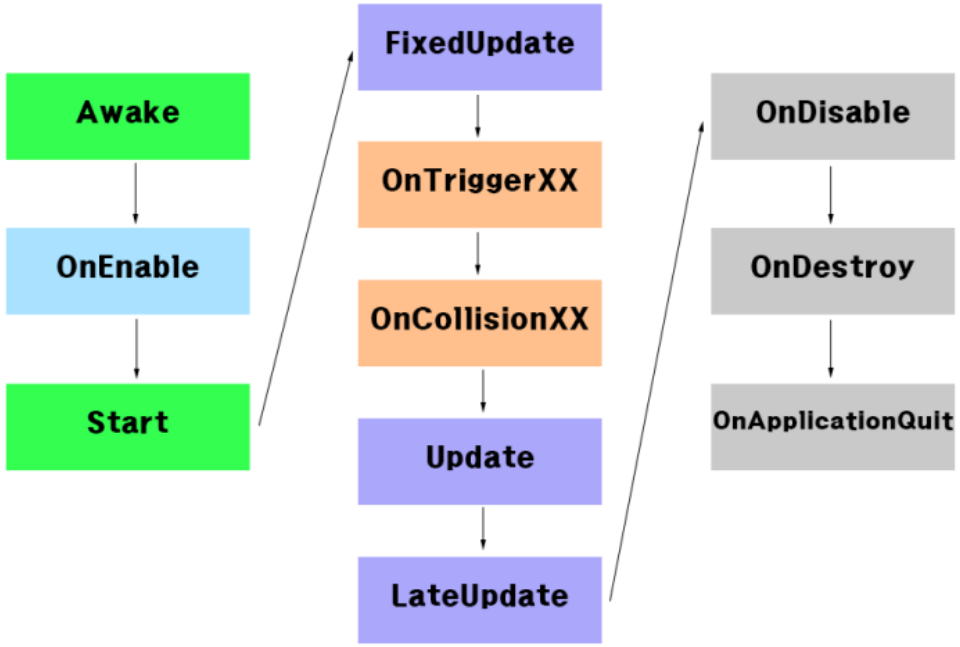
```
Animator animator;
Rigidbody2D rgbody;
SpriteRenderer spriteRenderer;
```

```
private void Init()
{
    animator = GetComponent<Animator>();
    rgbody = GetComponent<Rigidbody2D>();
    spriteRenderer = GetComponent<SpriteRenderer>();

    rgbody.position = INIT_POSITION;
    rgbody.velocity = Vector3.zero;

    currentJumpCount = 0;

    keyInputCheck = new bool[Enum.GetNames(typeof(KeyInput)).Length];

    for (int i = 0; i < keyInputCheck.Length; i++)
    {
        keyInputCheck[i] = false;
    }
}
```

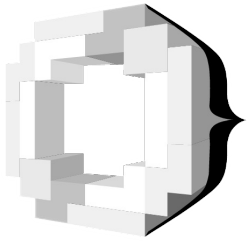


# 코드 살펴보기

## 오브젝트 생명주기

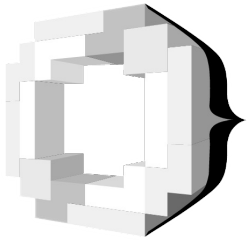



```
// 생명주기 함수들
// Awake는 항상 Start 이전에 한 번 호출됩니다. (언제 호출될지는 모름)
void Awake()
{
    Init();
}

// Start is called before the first frame update
// Awake 이후에 단 한 번 호출됩니다.
void Start()
{

}

// 오브젝트가 활성화될 때마다 호출됩니다.
// pub-sub 패턴에서 구독할 때 자주 사용됩니다.
void OnEnable()
{

}

// 오브젝트가 비활성화될 때마다 호출됩니다.
// pub-sub 패턴에서 구독 해지할 때 자주 사용됩니다.
void OnDisable()
{

}

// Update is called once per frame
// 매 프레임 호출됩니다. 주로 input 이벤트 처리와 그래픽 작업을 합니다.
void Update()
{
    GetPlayerKeyInput();

    animator.SetFloat("playerSpeed", Math.Abs(rgbody.velocity.x));
}

// 고정 프레임 시간마다 호출됩니다. 물리 연산은 여기서 해야합니다.
void FixedUpdate()
{
    PlayerJump();
    PlayerMove();
}
```



# 코드 살펴보기

```
    // 타일맵에 캐릭터가 닿을 때 이벤트를 처리합니다.
    void OnCollisionEnter2D(Collision2D other)
    {
        if (other.gameObject.layer == LayerMask.NameToLayer("Platform"))
        {
            animator.SetBool("isOnGround", true);
            currentJumpCount = 0;
        }
    }

    // 타일맵에 캐릭터가 떨어질 때 이벤트를 처리합니다.
    void OnCollisionExit2D(Collision2D other)
    {
        if (other.gameObject.layer == LayerMask.NameToLayer("Platform"))
        {
            animator.SetBool("isOnGround", false);
        }
    }

    // 타일맵의 지형 외에 다른 오브젝트와 부딪혔을 때 이벤트를 처리합니다.
    void OnTriggerEnter2D(Collider2D trigger)
    {
        if (trigger.gameObject.layer == LayerMask.NameToLayer("Goal"))
        {
            Debug.Log("Goal!");
        }
        if (trigger.gameObject.layer == LayerMask.NameToLayer("Obstacle"))
        {
            Debug.Log("Obstacle!");
        }
    }
}
```

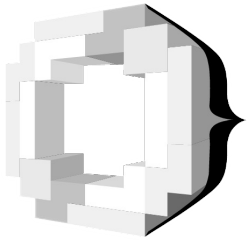


# 직접 구현해보기

**직접 마음대로 구현해봐요! 완성본 예시는**
**https://github.com/OOPARTS-2023-Unity/Week-1/blob/practice/planete/Assets/Scripts/PlayerController.cs**

```
private void GetPlayerKeyInput()
{
    // 사용자의 입력을 받아서 처리합니다.
    if (Input.GetKeyDown(KeyCode.UpArrow))
    {
        keyInputCheck[(int)KeyInput.UP] = true;
    }

    if (Input.GetKey(KeyCode.LeftArrow))
    {
        // 입력 처리
    }
    else
    {
        // 입력 처리
    }

    if (Input.GetKey(KeyCode.RightArrow))
    {
        // 입력 처리
    }
    else
    {
        // 입력 처리
    }
}
```

```
// 저장된 입력값을 확인해서 플레이어를 움직입니다.
private void PlayerJump()
{
    if (keyInputCheck[(int)KeyInput.UP])
    {
        keyInputCheck[(int)KeyInput.UP] = false;
        // 점프 구현
    }
}

private void PlayerMove()
{
    if (keyInputCheck[(int)KeyInput.LEFT])
    {
        // sprite renderer flip x -> true
        // sprite move to left
    }

    if (keyInputCheck[(int)KeyInput.RIGHT])
    {
        // sprite renderer flip x -> false
        // sprite move to right
    }
}
```

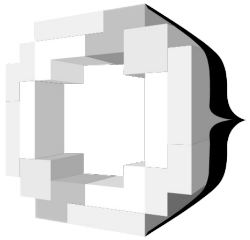


# 코드 개선?

**플레이어 이동을 아예 스테이트 패턴으로 구현!**

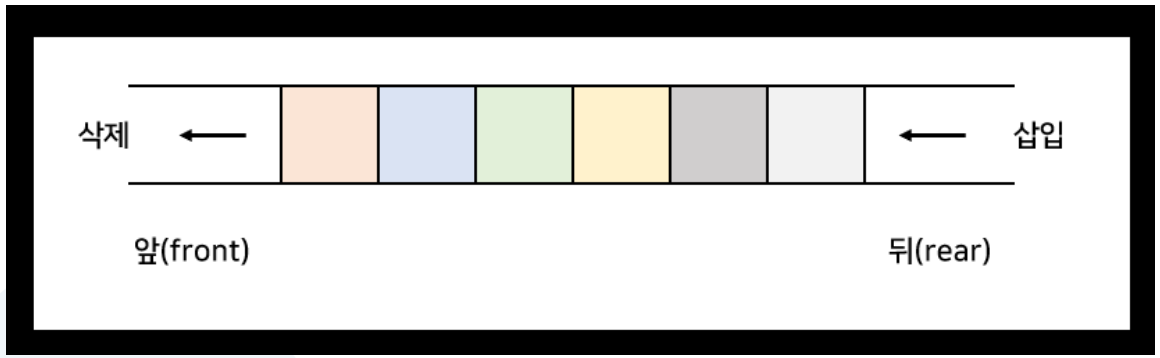
```
public interface IState
{

    public void Enter()
    {
        // code that runs when we first enter the state
    }

    public void Update()
    {
        // per-frame logic, include condition to transition to a new
state
    }

    public void Exit()
    {
        // code that runs when we exit the state
    }
}
```

```
public class IdleState : IState
{
    private PlayerController player;

    public IdleState(PlayerController player)
    {
        this.player = player;
    }

    public void Enter()
    {
        // code that runs when we first enter the state
    }

    public void Update()
    {
        // Here we add logic to detect if the conditions exist to
        // transition to another state
        …
    }

    public void Exit()
    {
        // code that runs when we exit the state
    }
}
```



# 코드 개선?

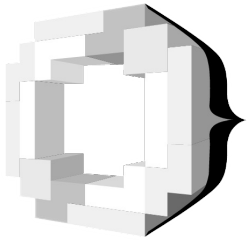
이벤트 처리를 큐로 받을 수도 있다!




```
private Queue<IPlayerEvent> playerInputs = new Queue<IPlayerEvent>();
```

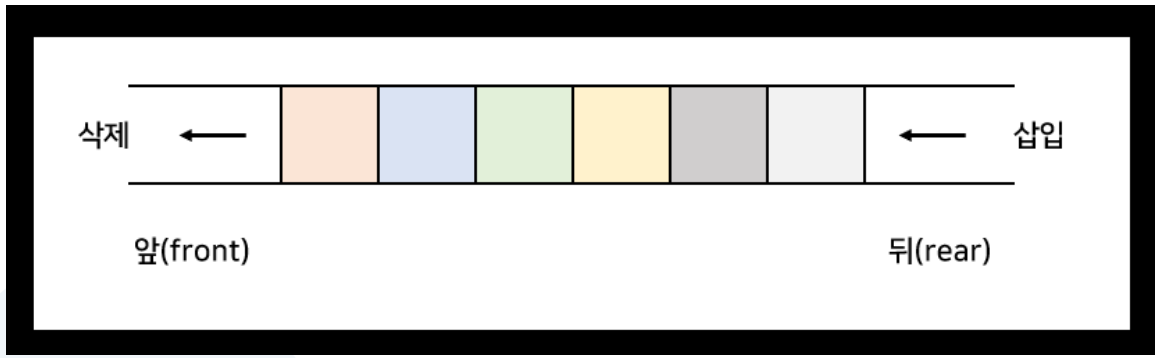
```
void FixedUpdate()
{
    while (playerInputs.Count > 0)
    {
        playerInputs.Dequeue().Action();
    }
}
```

```
private interface IPlayerEvent
{
    public void Action();
}

private class PlayerJump : IPlayerEvent
{
    private float _jumpPower;
    private Rigidbody2D _rigidbody;

    public PlayerJump(float jumpPower, Rigidbody2D rigidbody)
    {
        _jumpPower = jumpPower;
        _rigidbody = rigidbody;
    }

    public void Action()
    {
        _rigidbody.AddForce(new Vector2(0, 1) * _jumpPower);
    }
}

private class PlayerWin : IPlayerEvent
{
    public void Action()
    {
        // 승리 시 해야할 것
    }
}

private class PlayerLose : IPlayerEvent
{
    public void Action()
    {
        // 패배 시 해야할 것들
    }
}
```



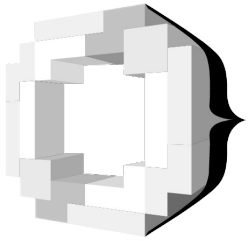
# 코드 개선?

**이벤트 처리를 큐로 받을 수도 있다!**



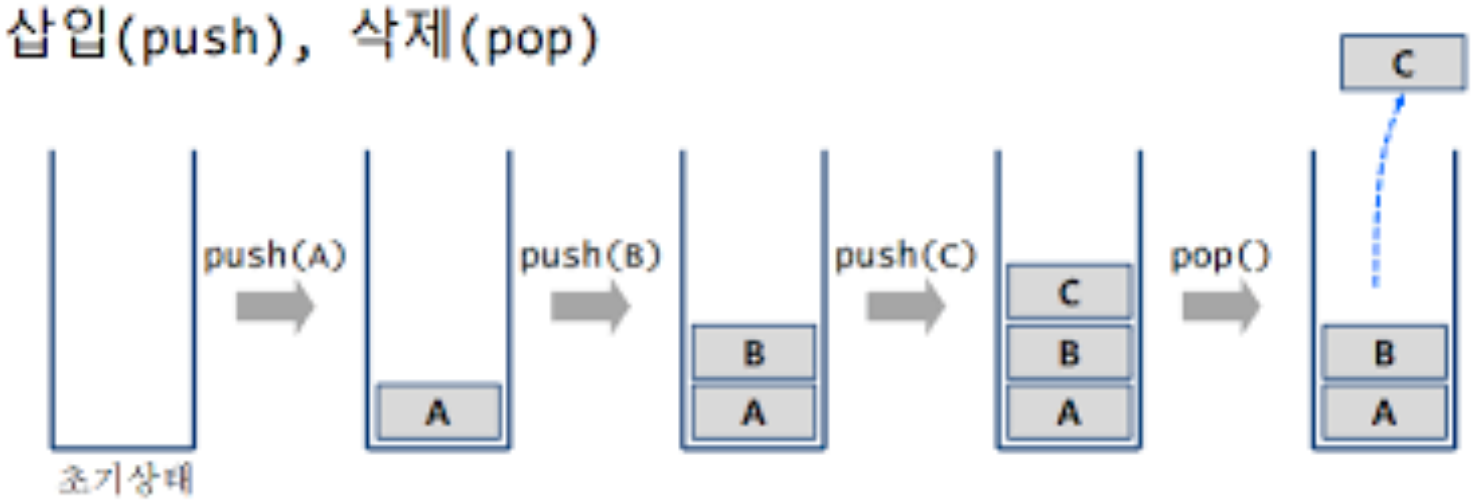

```
if (Input.GetKeyDown(KeyCode.UpArrow))
{
    if (jumpCount < maxJumpCount)
    {
        //rgbody.AddForce(new Vector2(0, 1) * jumpPower);
        playerInputs.Enqueue(new PlayerJump(jumpPower, rgbody));
        jumpCount++;
    }
}
```

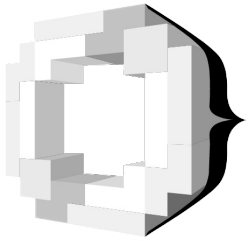
```
void FixedUpdate()
{
    while (playerInputs.Count > 0)
    {
        playerInputs.Dequeue().Action();
    }
}
```



# 코드 개선?

**스택을 써도 되지 않을까?**
- **선입선출로 이벤트를 처리하는 것이 개념적으로 맞다.**





# 해보면 좋아요

- 코드를 살펴보면서 직접 코드를 수정해봐요
- 깃의 다양한 기능들(특히 브랜치!)을 알아봐요
- 유니티 기능들은 자세히 안 다뤘어요
  - 유니티 공식 매뉴얼을 살펴봐요
  - https://docs.unity3d.com/kr/2021.3/Manual/UnityManual.html