유니티
게임 제작 입문

액션 게임 만들기

길벗

액션게임 만들기

게임 기획

제작하기

레벨 디자인

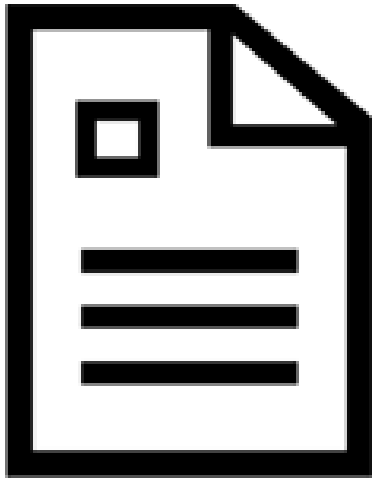
* 액션게임을 만들기까지

1 기획

2 프로그래밍

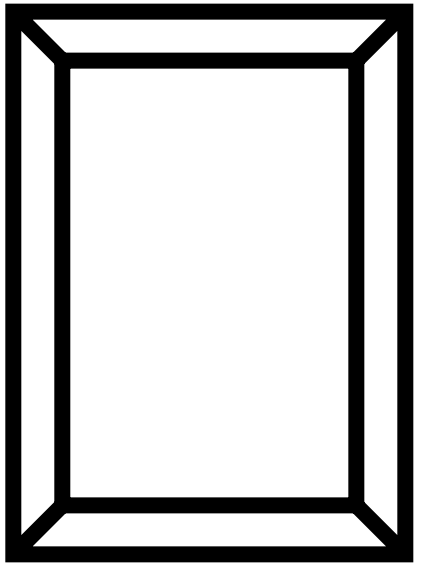
3 레벨디자인

1. 액션 게임 기획의 과정



기획서

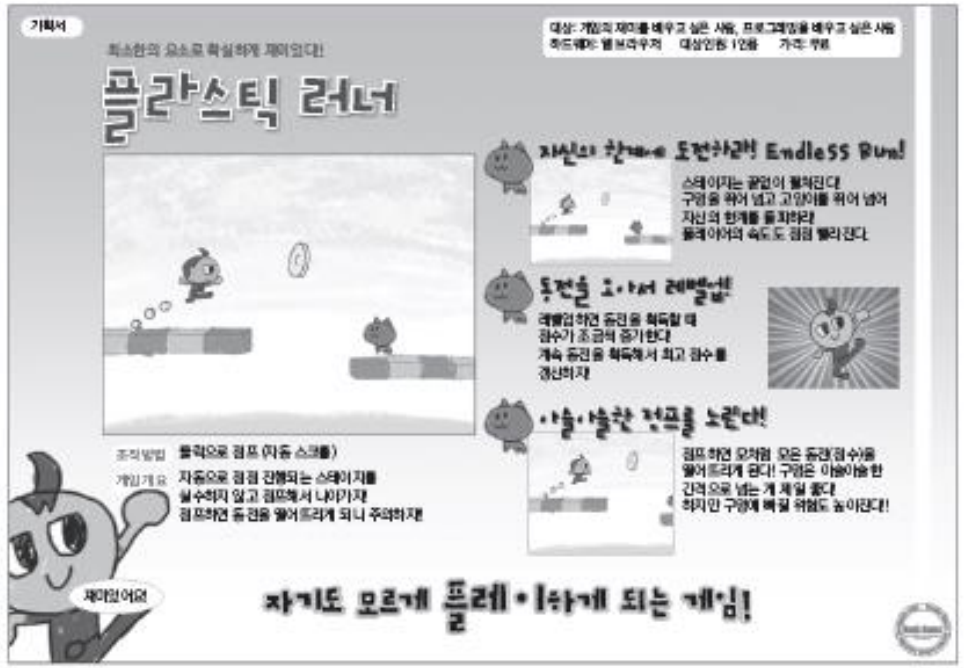
플레이어



게임의 틀

1-1. 기획서 작성

그림 3-27(c) 예쁘게 다시 작성한 기획서



어떤 게임을 만들고 싶은 지,

어떤 놀이가 필요한 지,

어떤 형태로 진행할 것인지,

이 게임이 재미있을 지,

목표

↓

기획서

1-2. 플레이어 선정

Why?

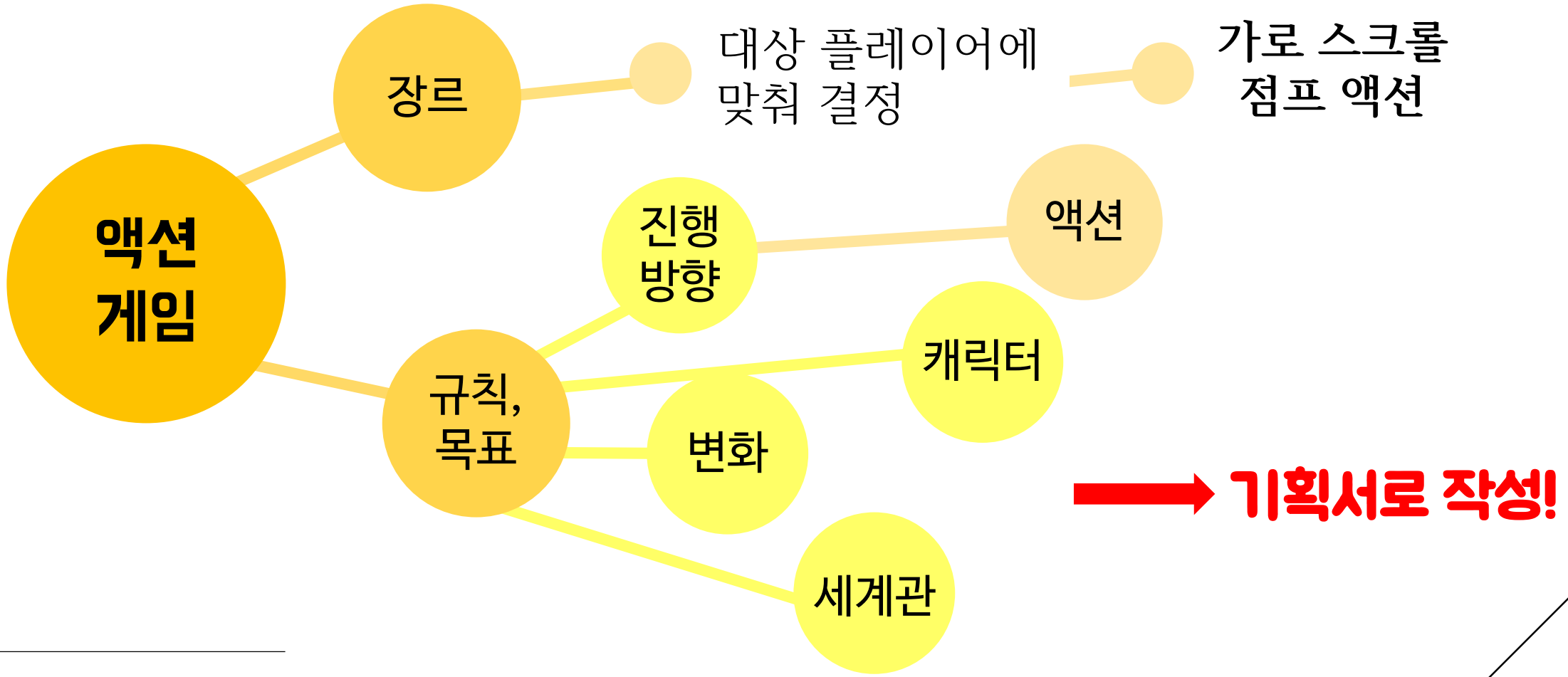
대상에 따라 아이디어 방향이 달라짐

남녀, 나이대, 생활환경, 직업 등 다양한 변수 고려

많은 시간을 투자해야 함!

Chap 5~7 대상 : 기획과 프로그래밍을 공부하고 싶은 사람

1-3. 큰 틀 짜기


액션 게임
장르
대상 플레이어에 맞춰 결정
가로 스크롤 점프 액션
진행 방향
액션
규칙, 목표
캐릭터
변화
세계관
기획서로 작성!

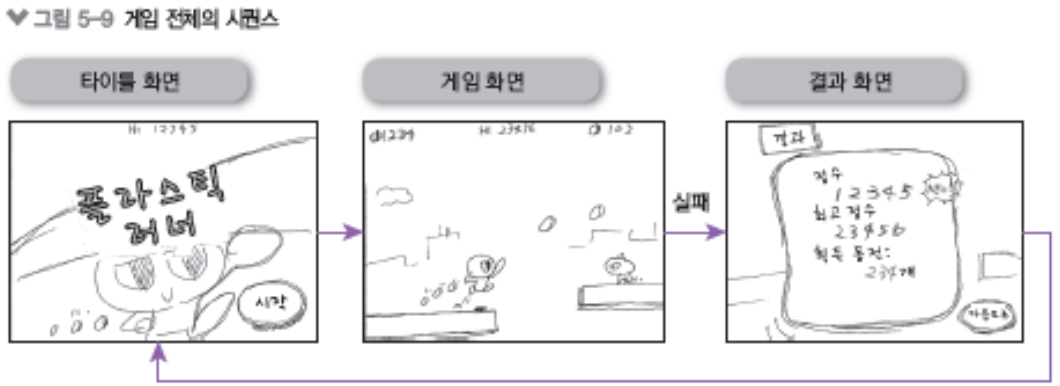
2. 사양서

기획서는 전체적인 틀을 잡아주는 역할을 한다면

사양서는 프로그래머가 무엇을 만들어야 하는 지 파악할 수 있는 기초작업.

사양서

시퀀스



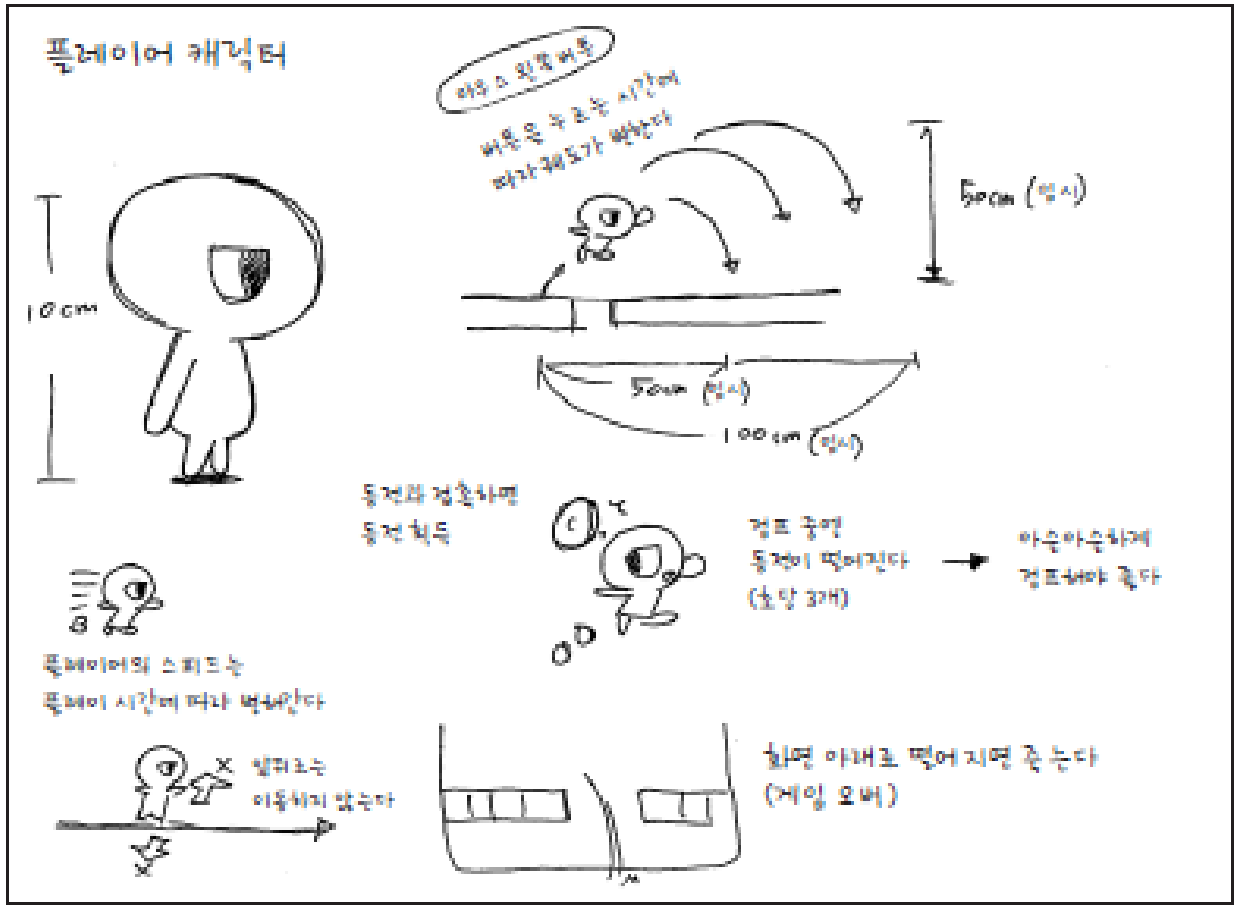
게임 속 요소

플레이어 캐릭터

그 외 ex) 방해캐릭터, 아이템 등

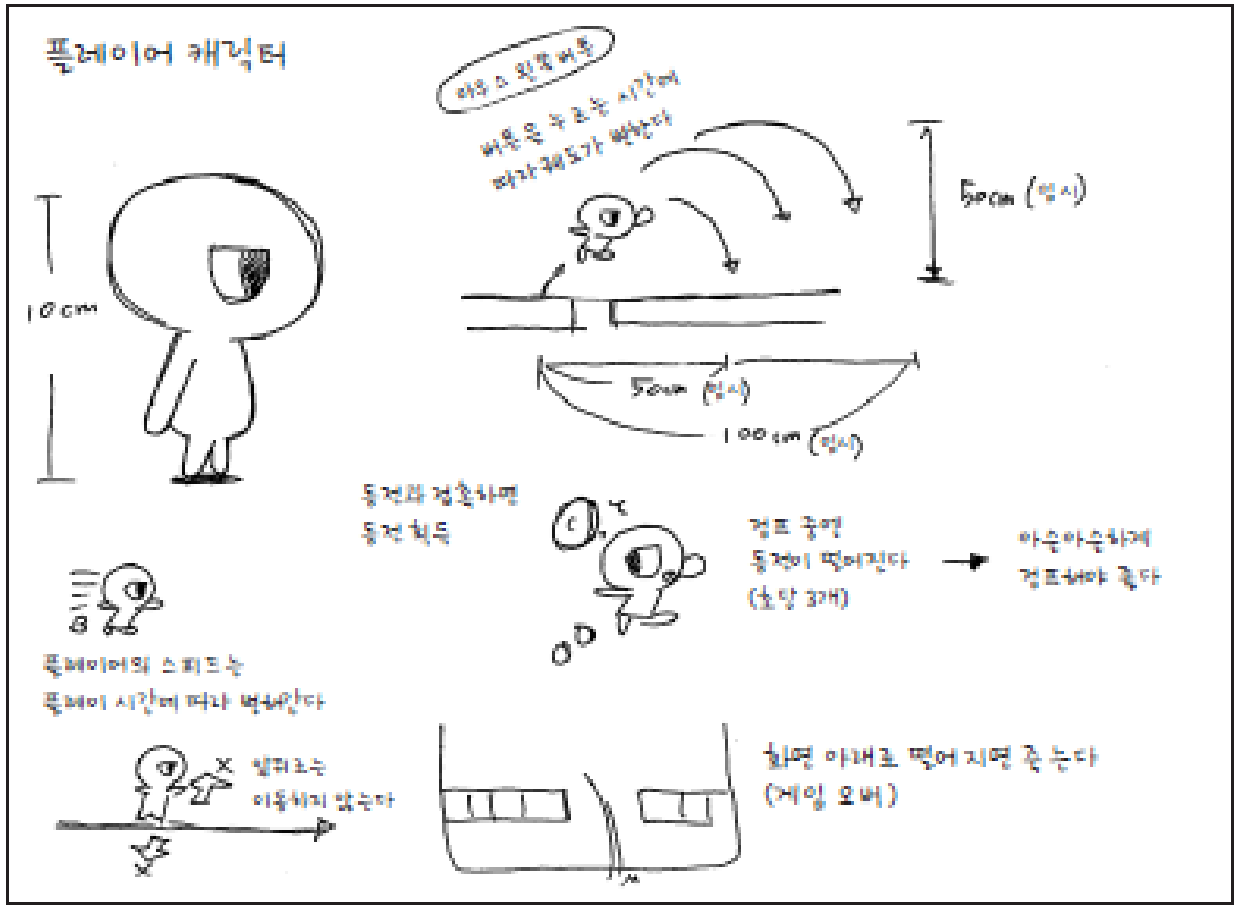
2. 사양서

* 플레이어 캐릭터 사양서



2. 사양서

* 플레이어 캐릭터 사양서



* 액션게임을 만들기까지

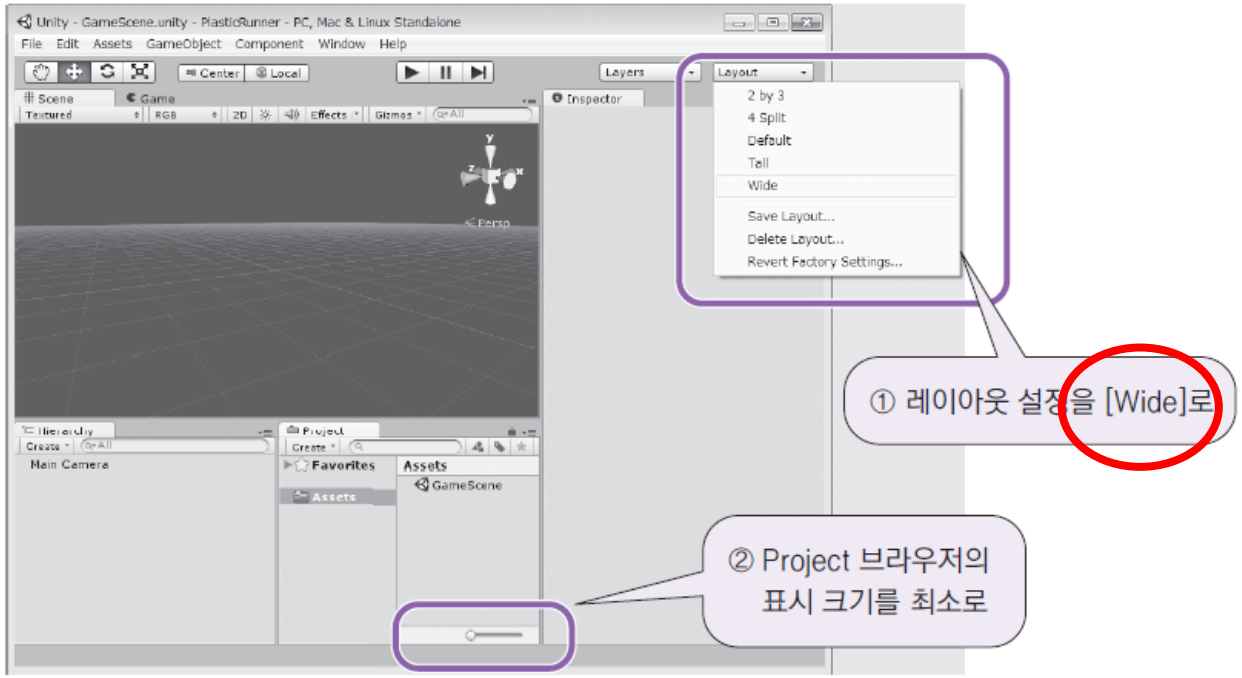
1 기획

2 프로그래밍

3 레벨디자인

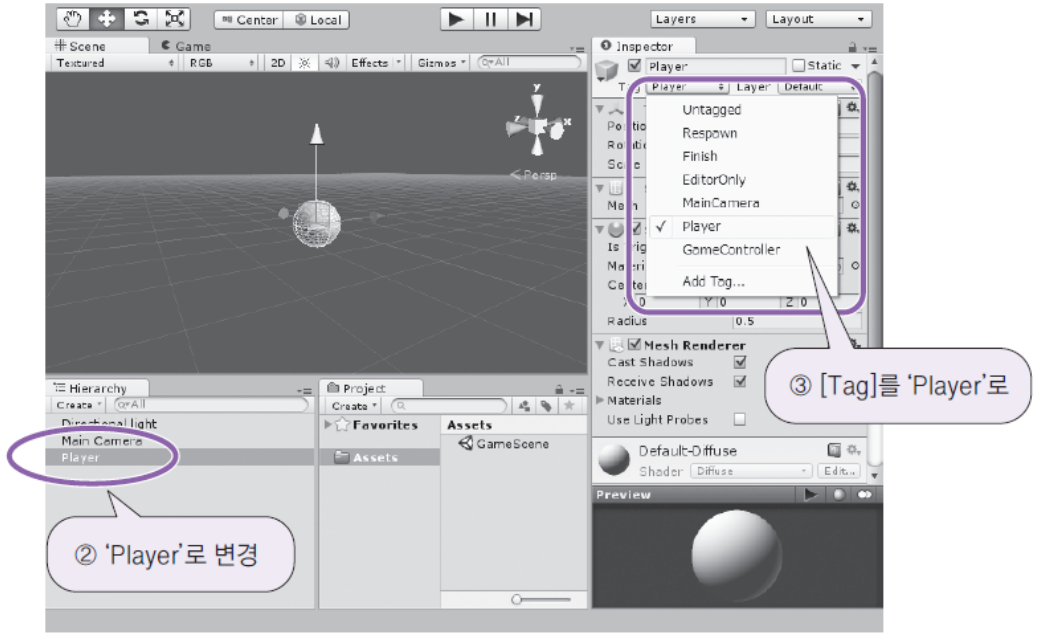
* 기초 준비

1. 새 프로젝트 만들기
2. 레이아웃 설정 맞추기



* 기초 준비

3. Player와 Floor 만들기

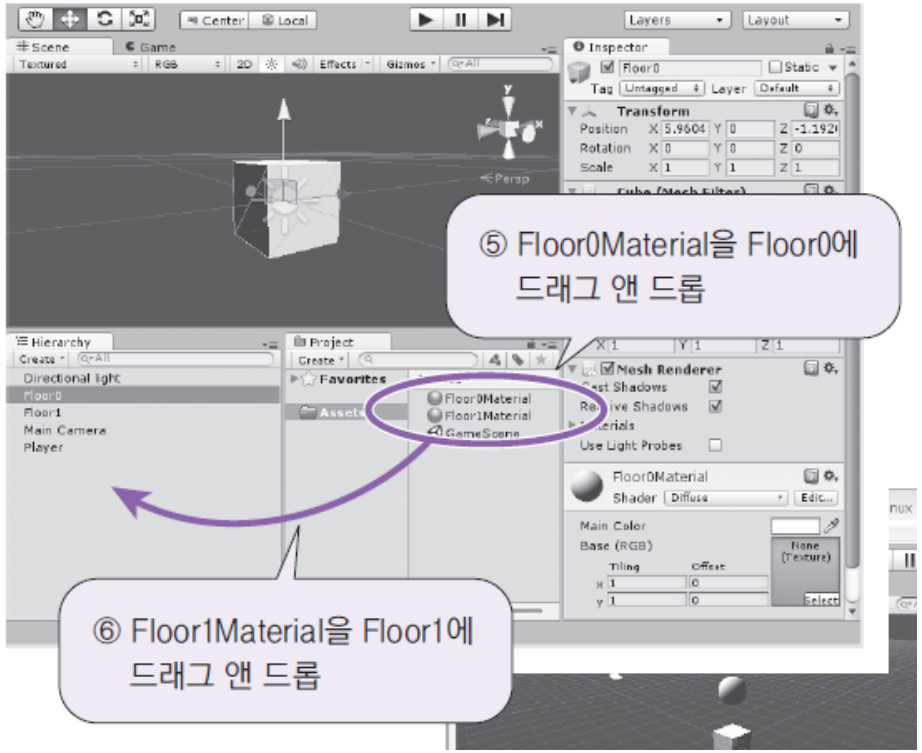


- Player

① [GameObject] → [Create Other] → [Sphere]
② Hierarchy에 생긴 Sphere를 'Player'로 변경
③ Player를 선택하고 Inspector에 있는
[Tag]를 Player로 설정한다.
④ Player의 [Transform/Position]을
(X:0, Y:2, Z:0)으로 설정한다.
⑤ Player의 [Transform/Rotation]을
(X:0, Y:90, Z:0)으로 설정한다.

* 기초 준비

3. Player와 Floor 만들기

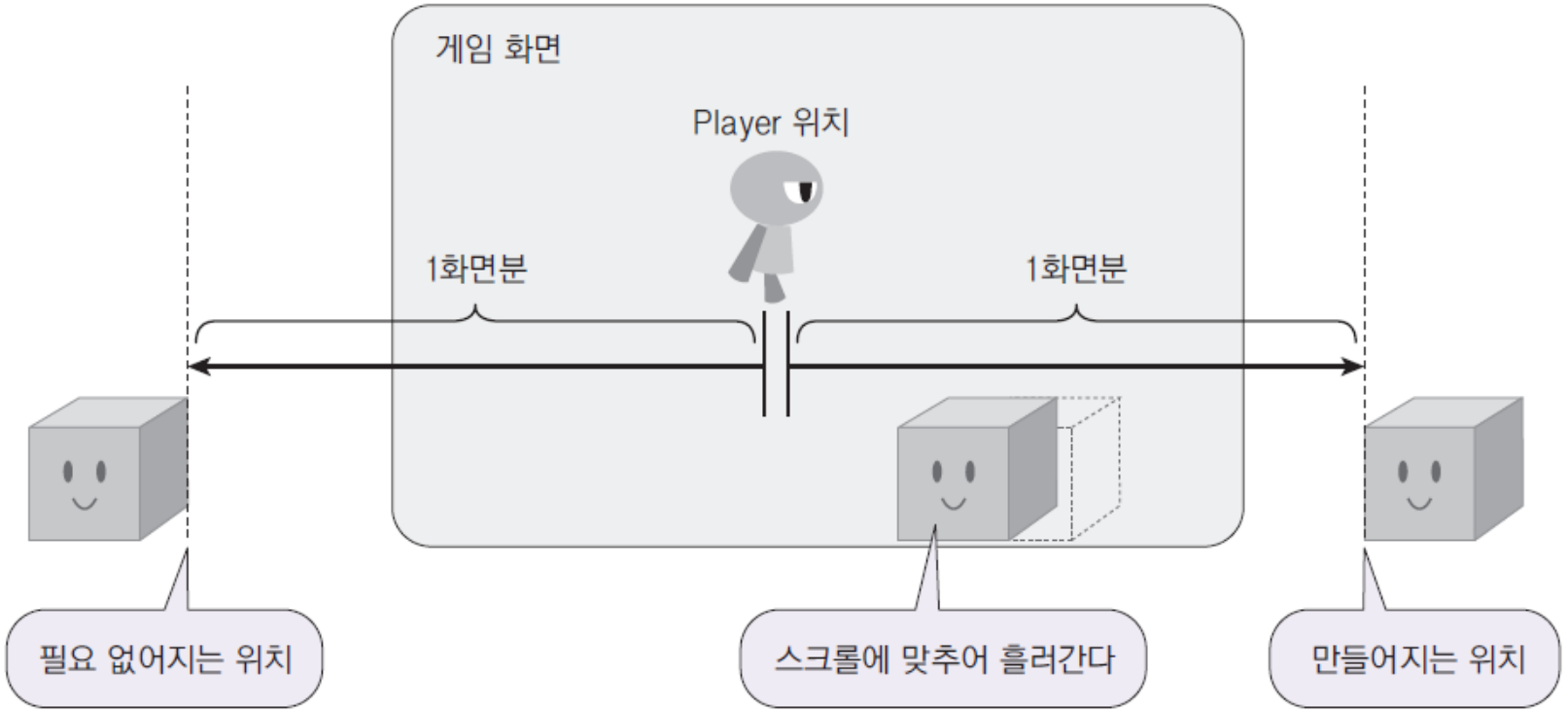


- Floor

① [GameObject] → [Create Other] → [Cube]
Cube 2개를 만들어 각각 'Floor0', 'Floor1'로 이름 지정
② [Assets] → [Create] → [Material]도 2개 만들어준다.
③ Floor0에 Floor0Material을 드래그 앤 드롭한다. (프리팹)
④ Floor1에 Floor1Material을 드래그 앤 드롭한다. (프리팹)
⑤ Floor1Material을 선택하여 Inspector를 표시한 후 색깔을 빨간색으로 설정한다.

- Block

① [GameObject] → [Create Empty]
② Hierarchy의 GameObject를 'GameRoot'로 변경

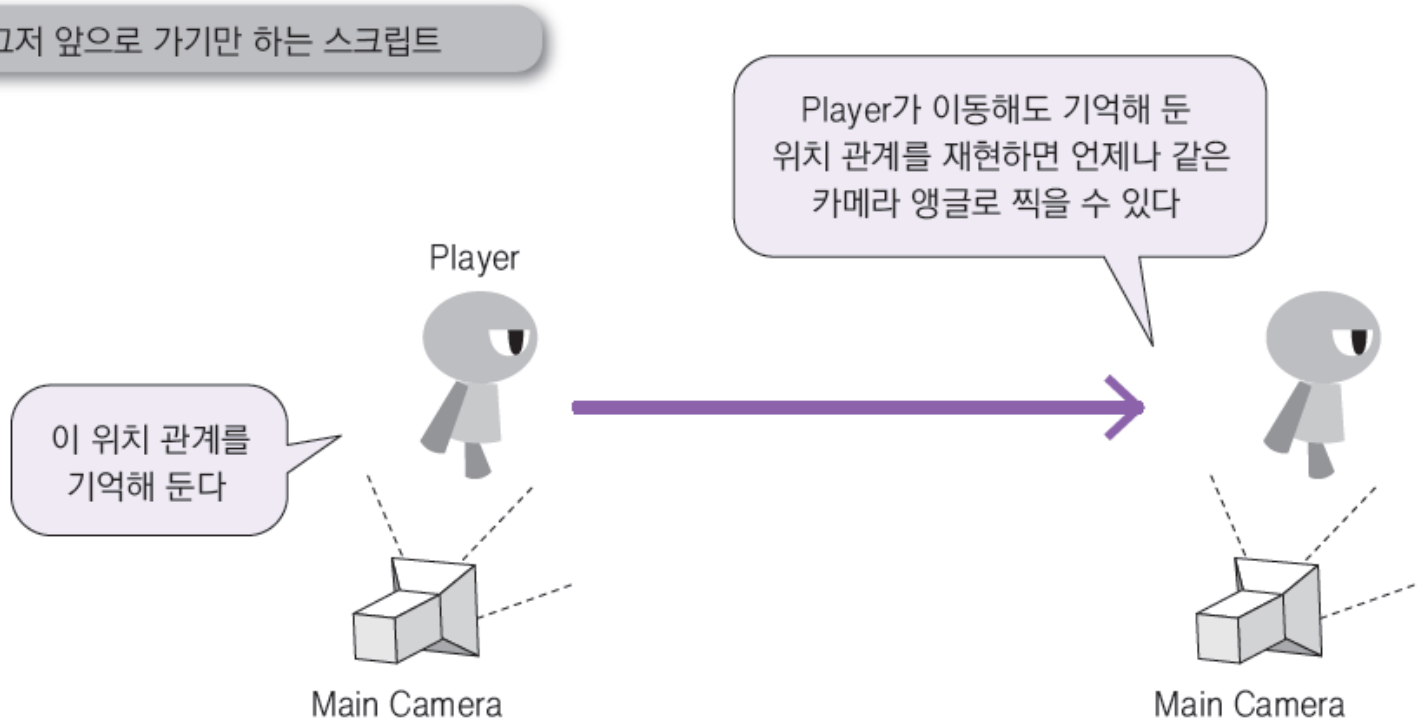


화면의 발판이 되는 블록을 만들고, 블록이 왼쪽 끝으로 넘어가면

지워지는 부분까지 구현해보도록 하자.

* 플레이어를 따라가는 카메라

앞으로 계속 나아가려면 플레이어가 있어야 하고, 그 플레이어를 쫓아가는 카메라가 필요!

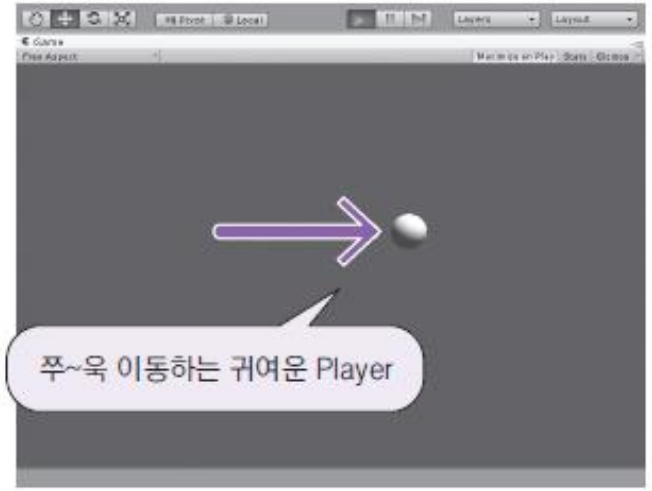


* 플레이어를 따라가는 카메라

- 오른쪽으로 진행

PlayerControl.cs

```
void Update() {
    this.transform.Translate(new Vector3(0.0f, 0.0f, 3.0f * Time.deltaTime));
}
```



* 플레이어를 따라가는 카메라

- 카메라

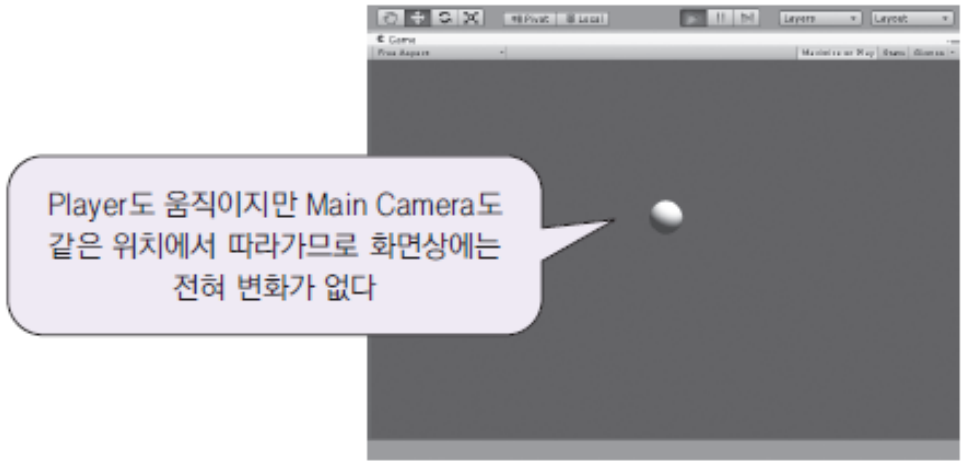
CameraControl.cs

```
private GameObject player = null;
private Vector3 position_offset = Vector3.zero;
void Start()
{
// 멤버 변수 player에 Player 오브젝트를 할당.
this.player = GameObject.FindGameObjectWithTag("Player");
// 카메라 위치(this.transform.position)와.
// 플레이어 위치(this.player.transform.position)의 차이를 보관.
this.position_offset =
this.transform.position - this.player.transform.position;
}
```

```
void LateUpdate()
{
// 카메라 현재 위치를 new_position에 할당.
```

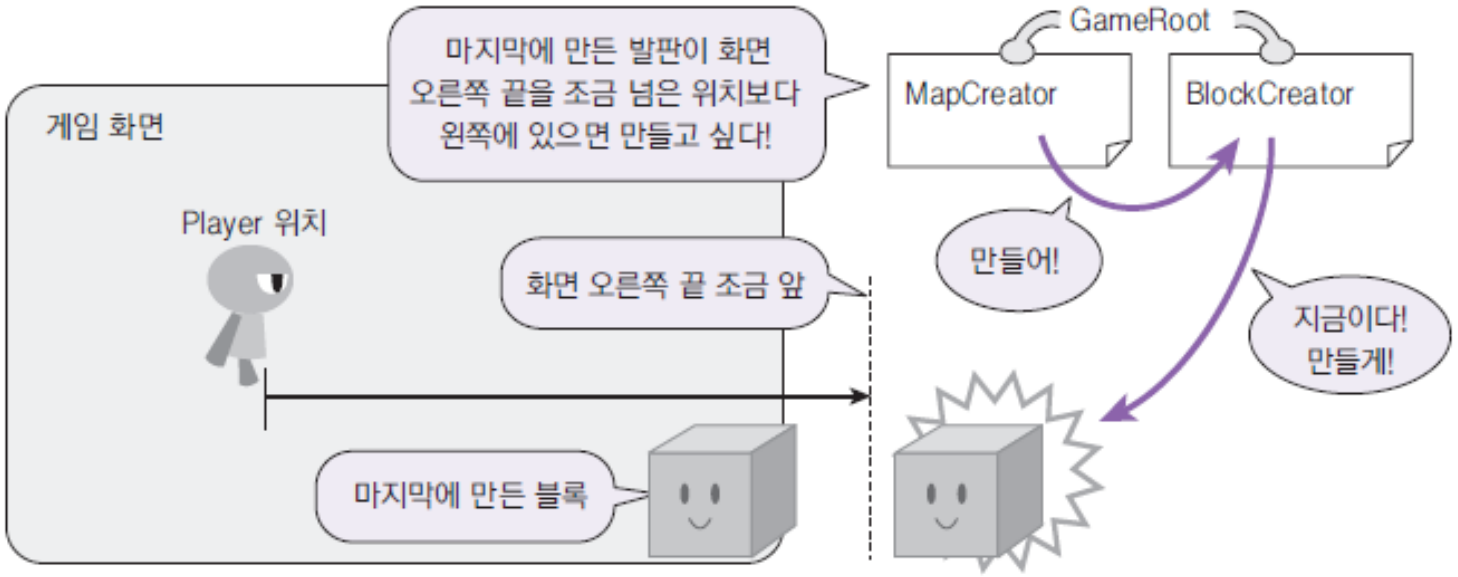
그림 6-10 Player도 움직이지만 카메라도 같이 움직인다

개입.



* 블록 만들기

전부를 미리 만드는 것보다 진행하면서 만들어내는 것이 더 효율적!



2개의 스크립트를 구현해서 블록을 제때 제때 만들어보도록 하자.

* 블록 만들기

BlockCreator.cs

```
public GameObject[] blockPrefabs; // 블록을 저장할 배열.
private int block_count = 0; // 생성한 블록의 개수.
void Start() {
}
void Update() {
}
public void createBlock(Vector3 block_position)
{
// 만들어야 할 블록의 종류(흰색인가 빨간색인가)를 구한다.
int next_block_type = this.block_count % this.blockPrefabs.Length;
// 블록을 생성하고 go에 보관한다.
GameObject go = GameObject.Instantiate(this.blockPrefabs[next_block_type]) as GameObject;
go.transform.position = block_position; // 블록의 위치를 이동.
this.block_count++; // 블록의 개수를 증가.
}
```

% : 나머지를 구하는 **연산자.**

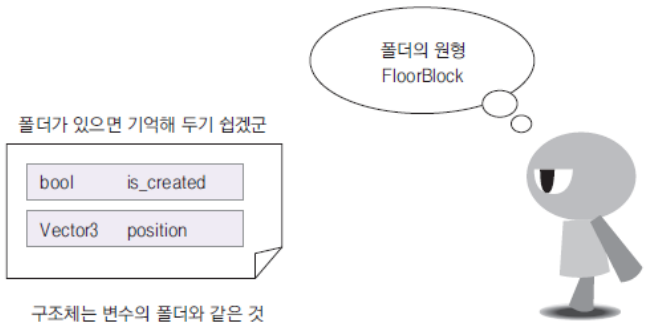
* 맵(스테이지) 만들기

MapCreator.cs

```
public static float BLOCK_WIDTH = 1.0f; // 블록의 폭.
public static float BLOCK_HEIGHT = 0.2f; // 블록의 높이.
public static int BLOCK_NUM_IN_SCREEN = 24; // 화면 내에 들어가는 블록의 개수.
// 블록에 관한 정보를 모아서 관리하는 구조체.
```

구조체 여러 개의 정보를 하나로 묶을 때 사용

```
private struct FloorBlock {
public bool is_created; // 블록이 만들어졌는가.
public Vector3 position; // 블록의 위치.
};
```




```
private FloorBlock last_block; // 마지막에 생성한 블록.
private PlayerControl player = null; // 씬상의 Player를 보관.
private BlockCreator block_creator; // BlockCreator를 보관.
```

* 맵(스테이지) 만들기

MapCreator.cs

```
void Start() {
this.player = GameObject.FindGameObjectWithTag("Player").
GetComponent<PlayerControl>();
this.last_block.is_created = false;
this.block_creator =
this.gameObject.GetComponent<BlockCreator>();
}
```

```
private void create_floor_block()
{
Vector3 block_position; // 이제부터 만들 블록의 위치.
if(! this.last_block.is_created) { // last_block이 생성되지 않은 경우.
// 블록의 위치를 일단 Player와 같게 한다.
block_position = this.player.transform.position;
// 그러고 나서 블록의 X 위치를 화면 절반만큼 왼쪽으로 이동.
block_position.x -=
BLOCK_WIDTH * ((float)BLOCK_NUM_IN_SCREEN / 2.0f);
// 블록의 Y위치는 0으로.
block_position.y = 0.0f;
} else { // last_block이 생성된 경우.
// 이번에 만들 블록의 위치를 직전에 만든 블록과 같게.
block_position = this.last_block.position;
```

* 맵(스테이지) 만들기

MapCreator.cs

```
void Update() {
        // 플레이어의 X위치를 가져온다.
        float block_generate_x = this.player.transform.position.x;
        // 그리고 대략 반 화면만큼 오른쪽으로 이동.
        // 이 위치가 블록을 생성하는 문턱 값이 된다.
        block_generate_x +=
        BLOCK_WIDTH * ((float)BLOCK_NUM_IN_SCREEN + 1) / 2.0f;
        // 마지막에 만든 블록의 위치가 문턱 값보다 작으면.
        while(this.last_block.position.x < block_generate_x) {
        // 블록을 만든다.
        this.create_floor_block();
        }
}
```

Player의 위치에서 화면 절반의 앞(오른쪽)에
문턱 값을 설정하고, 마지막에 만든 블록이
그 문턱 값보다 왼쪽에 있을 때는
그 문턱 값을 넘을 때까지 블록을 계속 만든다.

* 불필요한 블록 지우기

사용한 블록을 계속 남겨두면 프로그램 처리도 무거워지고, 게임이 멈출 수도 있기 때문에 지워주어야 한다.

BlockControl.cs

```
public MapCreator map_creator = null; // MapCreator를 보관하는 변수.
void Start()
{
        // MapCreator를 가져와서 멤버 변수 map_creator에 보관.
        map_creator = GameObject.Find("GameRoot")
.GetComponent<MapCreator>();
}
void Update()
{
        if(this.map_creator.isDelete(this.gameObject)) {
        // 카메라에게 나 안보이냐고 물어보고 안 보인다고 대답하면
        GameObject.Destroy(this.gameObject); // 자기 자신을 삭제.
        }
}
```

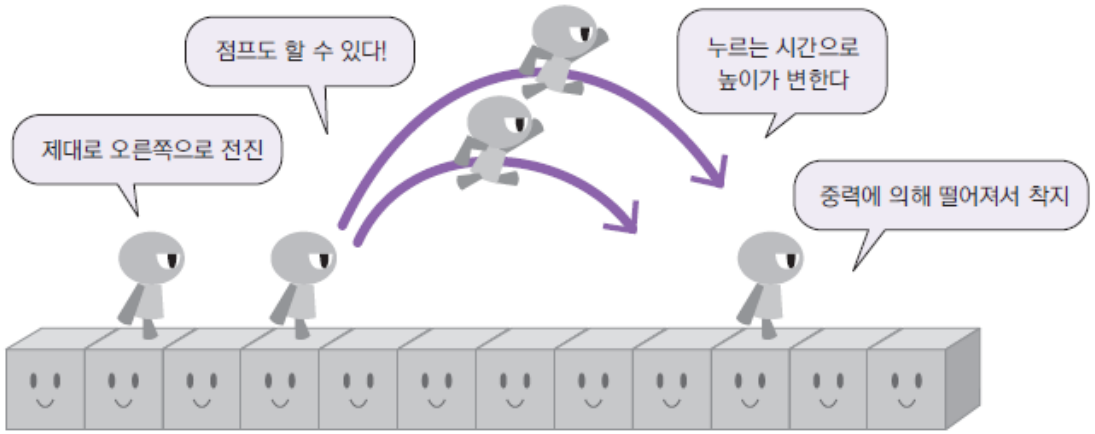
```
public bool isDelete(GameObject block_object)
{
        bool ret = false; // 반환값.
        // Player로부터 반 화면만큼 왼쪽에 위치.
        // 이 위치가 사라지느냐 마느냐를 결정하는 문턱 값이 됨.
        float left_limit = this.player.transform.position.x -
        BLOCK_WIDTH * ((float)BLOCK_NUM_IN_SCREEN / 2.0f);
        // 블록의 위치가 문턱 값보다 작으면(왼쪽).
        if(block_object.transform.position.x < left_limit) {
                ret = true; // 반환값을 true(사라져도 좋다)로.
        }
        return(ret); // 판정 결과를 돌려줌.
}
```

* 점프 - 리지드바디로 물리효과를 먼저 줄 것

PlayerControl.cs

```
// '점프'에 필요한 전역변수 선언 먼저.
public static float ACCELERATION = 10.0f; // 가속도.
public static float SPEED_MIN = 4.0f; // 속도의 최솟값.
public static float SPEED_MAX = 8.0f; // 속도의 최댓값.
public static float JUMP_HEIGHT_MAX = 3.0f; // 점프 높이.
public static float JUMP_KEY_RELEASE_REDUCE = 0.5f;
// 점프 후의 감속도.
public enum STEP { // Player의 각종 상태를 나타내는 자료형.
NONE = -1, // 상태정보 없음.
RUN = 0, // 달린다.
JUMP, // 점프.
MISS, // 실패.
NUM, // 상태가 몇 종류 있는지 보여준다(=3).
};
```

```
public STEP step = STEP.NONE; // Player의 현재 상태.
public STEP next_step = STEP.NONE; // Player의 다음 상태.
public float step_timer = 0.0f; // 경과 시간.
private bool is_landed = false; // 착지했는가.
private bool is_colided = false; // 뭔가와 충돌했는가.
private bool is_key_released = false; // 버튼이 떨어졌는가.
```

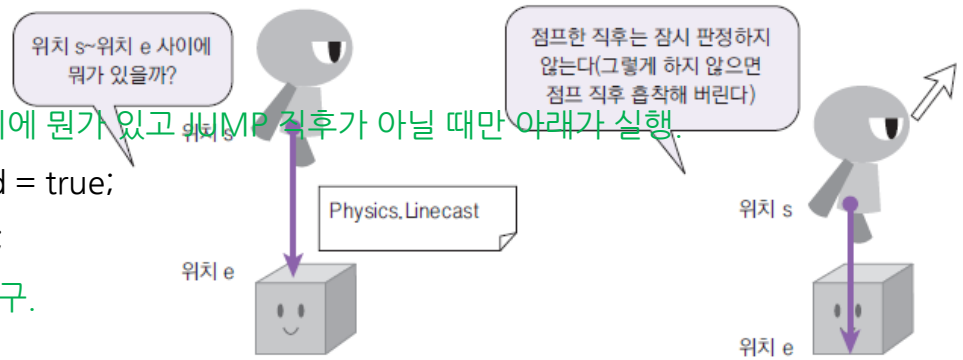


* 점프 - 리지드바디로 물리효과를 먼저 줄 것

PlayerControl.cs

```
void Start() {
this.next_step = STEP.RUN;
}

private void check_landed() //착지했는지 조사
{
this.is_landed = false; // 일단 false로 설정.
do {
Vector3 s = this.transform.position; // Player의 현재 위치.
Vector3 e = s + Vector3.down * 1.0f; // s부터 아래로 1.0f로 이동한 위치.
RaycastHit hit;
if(! Physics.Linecast(s, e, out hit)) { // s부터 e 사이에 아무것도 없을 때.
break; // 아무것도 하지 않고 do~while 루프를 빠져나감(탈출구로).
}
// s부터 e 사이에 뭔가 있을 때 아래의 처리가 실행.
if(this.step = = STEP.JUMP) { // 현재, 점프 상태라면.
// 경과 시간이 3.0f 미만이라면.
if(this.step_timer < Time.deltaTime * 3.0f) {
break; // 아무것도 하지 않고 do~while 루프를 빠져나감(탈출구로).
}
}
// s부터 e 사이에 뭔가 있고 JUMP 직후가 아닐 때만 아래가 실행.
this.is_landed = true;
} while(false);
// 루프의 탈출구.
}
```



* 점프 - 리지드바디로 물리효과를 먼저 줄 것

PlayerControl.cs

```
void Update() {
Vector3 velocity = this.rigidbody.velocity; // 속도를 설정.
this.check_landed(); // 착지 상태인지 체크.
this.step_timer += Time.deltaTime; // 경과 시간을 진행한다.
// 다음 상태가 정해져 있지 않으면 상태의 변화를 조사한다.
if(this.next_step = = STEP.NONE) {
switch(this.step) { // Player의 현재 상태로 분기.
case STEP.RUN: // 달리는 중일 때.
if(! this.is_landed) { // 달리는 중이고 착지하지 않은 경우 아무것도 하지 않는
다. } else {
If(Input.GetMouseButtonDown(0)) {
// 달리는 중이고 착지했고 왼쪽 버튼이 눌렸다면.
// 다음 상태를 점프로 변경.
                this.next_step = STEP.JUMP; }}
break;
case STEP.JUMP: // 점프 중일 때.
if(this.is_landed) {
// 점프 중이고 착지했다면 다음 상태를 주행 중으로 변경.
                this.next_step = STEP.RUN; }
break; }
}
// '다음 정보'가 '상태 정보 없음'이 아닌 동안(상태가 변할 때만).
while(this.next_step != STEP.NONE) {
this.step = this.next_step; // '현재 상태'를 '다음 상태'로 갱신.
this.next_step = STEP.NONE; // '다음 상태'를 '상태 없음'으로 변경.

// 계속
```

* 점프 - 리지드바디로 물리효과를 먼저 줄 것

PlayerControl.cs

```
switch(this.step) { // 갱신된 '현재 상태'가.
case STEP.JUMP: // '점프'일 때.
// 점프할 높이로 점프 속도를 계산(마법의 주문임).
velocity.y = Mathf.Sqrt(
2.0f * 9.8f * PlayerControl.JUMP_HEIGHT_MAX);
// '버튼이 떨어졌음을 나타내는 플래그'를 클리어한다.
this.is_key_released = false;
break; }
this.step_timer = 0.0f; // 상태가 변했으므로 경과 시간을 제로로 리셋.
}
```

```
// 상태별로 매 프레임 갱신 처리.
switch(this.step) {
case STEP.RUN: // 달리는 중일 때.
// 속도를 높인다.
velocity.x += PlayerControl.ACCELERATION * Time.deltaTime;
// 속도가 최고 속도 제한을 넘으면.
if(Mathf.Abs(velocity.x) > PlayerControl.SPEED_MAX) {
// 최고 속도 제한 이하로 유지한다.
            velocity.x *= PlayerControl.SPEED_MAX /
            Mathf.Abs(this.rigidbody.velocity.x);
}
break;
```

* 점프 - 리지드바디로 물리효과를 먼저 줄 것

PlayerControl.cs

```
case STEP.JUMP: // 점프 중일 때.
do {
// '버튼이 떨어진 순간'이 아니면.
if(! Input.GetMouseButtonUp(0)) {
                break; // 아무것도 하지 않고 루프를 빠져나간다.
}
// 이미 감속된 상태면(두 번이상 감속하지 않도록).
if(this.is_key_released) {
                break;}
// 상하방향 속도가 0 이하면(하강 중이라면).
if(velocity.y <= 0.0f) {
                break;}
// 버튼이 떨어져 있고 상승 중이라면 감속 시작.
// 점프의 상승은 여기서 끝.
velocity.y *= JUMP_KEY_RELEASE_REDUCE;
this.is_key_released = true;
} while(false);
break;
}
// Rigidbody의 속도를 위에서 구한 속도로 갱신.
// (이 행은 상태에 관계없이 매번 실행된다).
this.rigidbody.velocity = velocity;
}
```

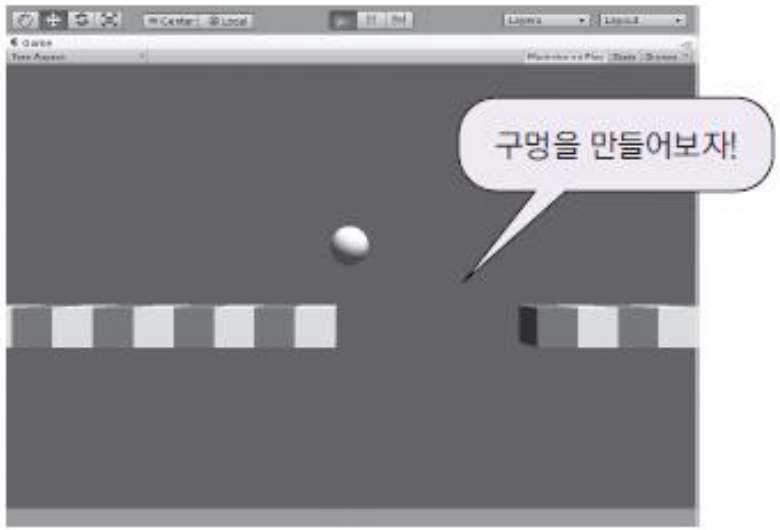
* 바닥에 구멍내기

LevelControl.cs

```
// 만들어야 할 블록에 관한 정보를 모은 구조체.
public struct CreationInfo {
public Block.TYPE block_type; // 블록의 종류.
public int max_count; // 블록의 최대 개수.
public int height; // 블록을 배치할 높이.
public int current_count; // 작성한 블록의 개수.
};

public CreationInfo previous_block; // 이전에 어떤 블록을 만들었는가.
public CreationInfo current_block; // 지금 어떤 블록을 만들어야 하는가.
public CreationInfo next_block; // 다음에 어떤 블록을 만들어야 하는가.
public int block_count = 0; // 생성한 블록의 총 수.
public int level = 0; // 난이도.
```

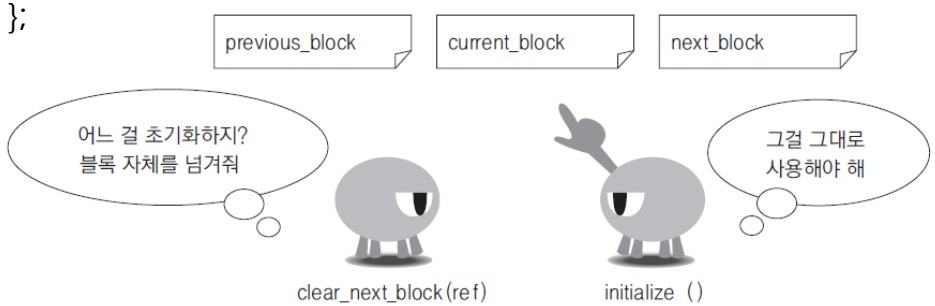
그림 6-25 구멍을 만들어보자



* 바닥에 구멍내기

MapCreator.cs

```
// Block 클래스 추가
public class Block {
// 블록의 종류를 나타내는 열거체.
public enum TYPE {
        NONE = -1, // 없음.
        FLOOR = 0, // 마루.
        HOLE, // 구멍.
        NUM, // 블록이 몇 종류인지 나타낸다(=2).
};
};
```



LevelControl.cs

```
private void clear_next_block(ref CreationInfo block)
{ //프로필 노트에 실제로 기록하는 처리를 한다.
        // 전달받은 블록(block)을 초기화.
        block.block_type = Block.TYPE.FLOOR;
        block.max_count = 15;
        block.height = 0;
        block.current_count = 0;
}
public void initialize()
{
        this.block_count = 0; // 블록의 총 수를 초기화.
        // 이전, 현재, 다음 블록을 각각.
        // clear_next_block()에 넘겨서 초기화한다.
        this.clear_next_block(ref this.previous_block);
        this.clear_next_block(ref this.current_block);
        this.clear_next_block(ref this.next_block);
}
```

* 바닥에 구멍내기

LevelControl.cs

```
private void update_level(ref CreationInfo current, CreationInfo previous)
{
switch(previous.block_type) {
case Block.TYPE.FLOOR: // 이번 블록이 바닥일 경우.
    current.block_type = Block.TYPE.HOLE; // 다음 번은 구멍을 만든다.
    current.max_count = 5; // 구멍은 5개 만든다.
    current.height = previous.height; // 높이를 이전과 같게 한다.
    break;
case Block.TYPE.HOLE: // 이번 블록이 구멍일 경우.
    current.block_type = Block.TYPE.FLOOR; // 다음은 바닥 만든다.
    current.max_count = 10; // 바닥은 10개 만든다.
    break; }
}
```

대소문자 구분 주의

```
public void update(){
// 이번에 만든 블록 개수를 증가.
this.current_block.current_count++;
// 이번에 만든 블록 개수가 max_count 이상이면.
if(this.current_block.current_count >= this.current_block.max_count) {
    this.previous_block = this.current_block;
    this.current_block = this.next_block;
    // 다음에 만들 블록의 내용을 초기화.
    this.clear_next_block(ref this.next_block);
    // 다음에 만들 블록을 설정.
    this.update_level(ref this.next_block, this.current_block); }
this.block_count++; // 블록의 총 수를 증가.
}
```

* 바닥에 구멍내기

MapCreator.cs

```
public static float BLOCK_WIDTH = 1.0f;
public static float BLOCK_HEIGHT = 0.2f;
public static int BLOCK_NUM_IN_SCREEN = 24;
private LevelControl level_control = null;

void Start() {
    this.player = GameObject.FindGameObjectWithTag(
    "Player").GetComponent<PlayerControl>();
    this.last_block.is_created = false;
    this.block_creator =
    this.gameObject.GetComponent<BlockCreator>();
    this.level_control = new LevelControl();
    this.level_control.initialize();
}
```

LevelControl과 MapCreator를 연계시킴

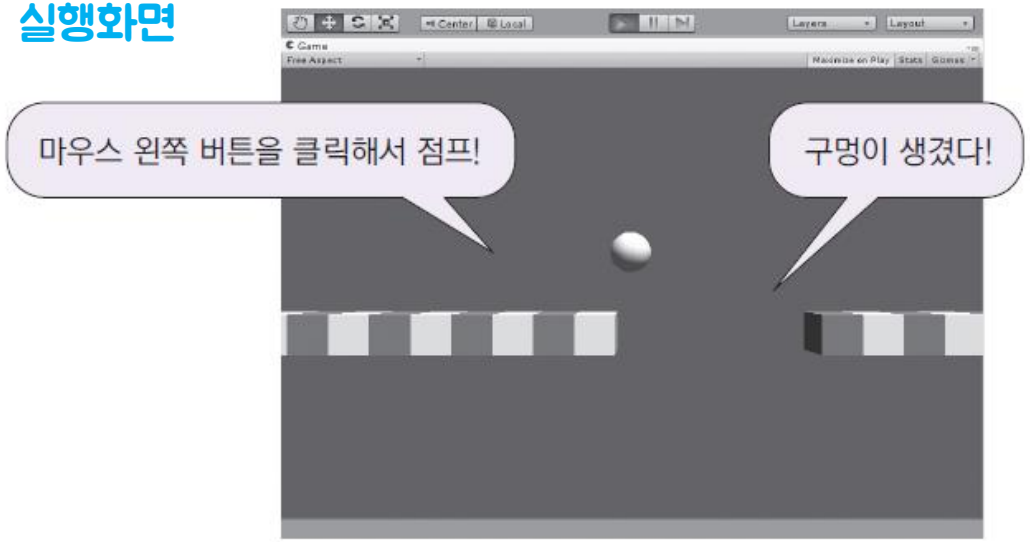
기존 코드에 추가할 것

* 바닥에 구멍내기

MapCreator.cs

```
private void create_floor_block(){
    Vector3 block_position;
    if(! this.last_block.is_created) {
        block_position = this.player.transform.position;
```

실행화면




```
        // level_control에 저장된 current_block(지금 만들 블록 정보)의.
        // height(높이)를 씬 상의 좌표로 변환.
        block_position.y = level_control.current_block.height * BLOCK_HEIGHT;
        // 지금 만들 블록에 관한 정보를 변수 current에 넣는다.
        LevelControl.CreationInfo current = this.level_control.current_block;
        // 지금 만들 블록이 바닥이면 (지금 만들 블록이 구멍이라면)
        if(current.block_type == Block.TYPE.FLOOR) {
        // block_position의 위치에 블록을 실제로 생성.
        this.block_creator.createBlock(block_position);
        }
        this.last_block.position = block_position;
        this.last_block.is_created = true;
    }
```

* 구멍에 떨어졌을 때

구멍에 떨어졌을 때,
게임이 바로 끝나지 않는다.

계속 밑으로 떨어지는 화면 반복.

= 문턱 값보다 더 낮은 장소에 Player 위치 존재.

PlayerControl.cs

```
public static float NARAKU_HEIGHT = -5.0f;

void Update() {
    Vector3 velocity = this.rigidbody.velocity;
    this.check_landed();
switch(this.step) {
    case STEP.RUN:
    case STEP.JUMP:
    // 현재 위치가 한계치보다 아래면.
        if(this.transform.position.y < NARAKU_HEIGHT) {
            this.next_step = STEP.MISS; // '실패' 상태로 한다.
        }
    break;}

...
```

* 구멍에 떨어졌을 때

구멍에 떨어졌을 때,
게임이 바로 끝나지 않는다.

계속 밑으로 떨어지는 화면 반복.

= 문턱 값보다 더 낮은 장소에 Player 위치 존재.

PlayerControl.cs

```
switch(this.step) {
…
case STEP.JUMP: // 점프 중일 때.
do {
    …
    velocity.y *= JUMP_KEY_RELEASE_REDUCE;
    this.is_key_released = true;
} while(false);
break;
case STEP.MISS:
    // 가속도(ACCELERATION)를 빼서 Player의 속도를 느리게 해 간다.
    velocity.x -= PlayerControl.ACCELERATION * Time.deltaTime;
    if(velocity.x < 0.0f) { // Player의 속도가 마이너스면.
    velocity.x = 0.0f; // 0으로 한다.
    }
    break; }
```

* 액션게임을 만들기까지

1 기획

2 프로그래밍

3 레벨디자인

그림 7-1 레벨 디자인에 사용할 수 있는 요소

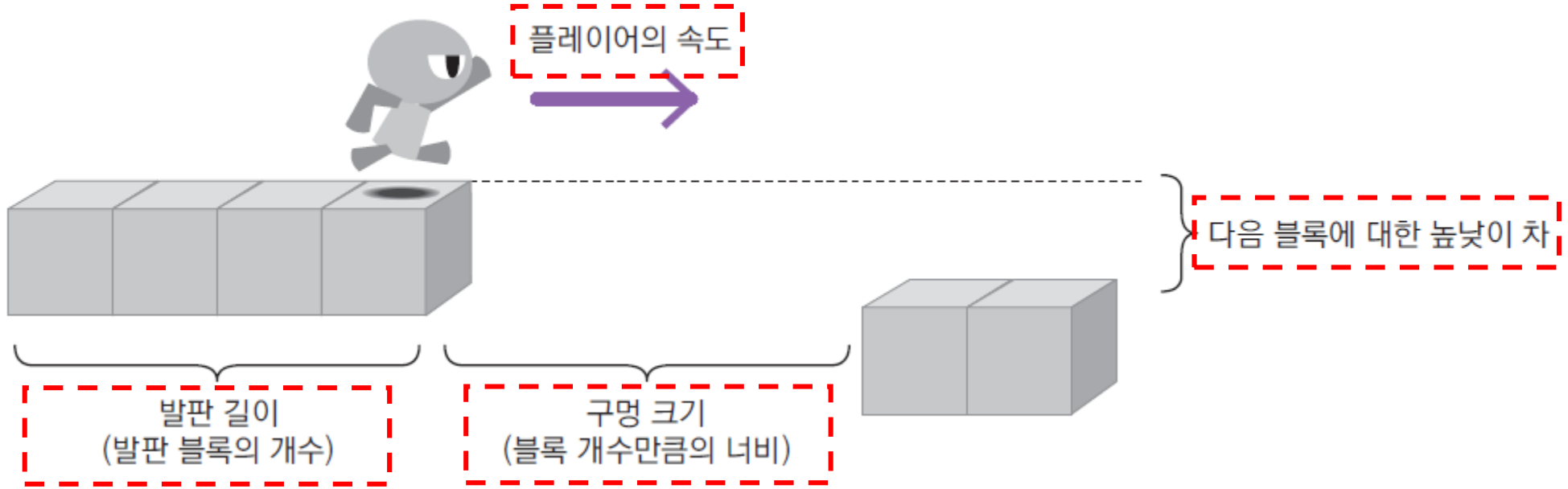


그림 7-1에서 나온 요소들을 고려하여 레벨 디자인을 해보자.

* 각 요소별 고려사항

요소	고려사항 (예시)
플레이어 속도	0~15초: 속도 7.0 15~30초: 속도 9.0 (갑자기 빨라진다)
발판 길이	0~15초: 9~10블록 15~30초: 8~9블록
구멍 크기	0~15초: 1~2블록 15~30초: 3~5블록
발판 높이	0~15초: 0블록 15~30초: -4~+4블록 (※갑자기 높이 차이가 생긴다)

level_data.txt

#		FLOOR		HOLE		HEIGHT	
# TIME	SPEED	MIN	MAX	MIN	MAX	MIN	MAX
0.0	5.0	9	10	1	2	0	0
15.0	6.0	8	9	3	5	0	0
30.0	7.0	7	8	7	9	-4	4
45.0	9.0	5	6	12	15	1	1
60.0	8.0	6	7	10	12	2	2

=> 텍스트 파일로 정리해둘 것

* 텍스트 데이터 게임에 반영

각 요소를 관리하는 새로운 클래스를 만들어줌

LevelControl.cs

```
//중간에 추가
public class LevelData {
    public struct Range { // 범위를 표현하는 구조체.
    public int min; // 범위의 최솟값.
    public int max; // 범위의 최댓값.
};
public float end_time; // 종료 시간.
public float player_speed; // 플레이어의 속도.
public Range floor_count; // 발판 블록 수의 범위.
public Range hole_count; // 구멍의 개수 범위.
public Range height_diff; // 발판의 높이 범위.
```

```
public LevelData()
{
    this.end_time = 15.0f; // 종료 시간 초기화.
    this.player_speed = 6.0f; // 플레이어의 속도 초기화.
    this.floor_count.min = 10; // 발판 블록 수의 최솟값을 초기화.
    this.floor_count.max = 10; // 발판 블록 수의 최댓값을 초기화.
    this.hole_count.min = 2; // 구멍 개수의 최솟값을 초기화.
    this.hole_count.max = 6; // 구멍 개수의 최댓값을 초기화.
    this.height_diff.min = 0; // 발판 높이 변화의 최솟값을 초기화.
    this.height_diff.max = 0; // 발판 높이 변화의 최댓값을 초기화.
}
```

* Level Data를 List로 다루기

LevelControl.cs

```
// 스크립트 시작 부분에 써준다.
// xx 안에 정의된 이름을 사용할 거에요 라고 선언
using System.Collections.Generic;


//List형 멤버변수를 추가, 각각의 최댓값 최솟값을 넣어준다.
public class LevelControl : MonoBehaviour {
    private List<LevelData> level_datas = new List<LevelData>();
    public int HEIGHT_MAX = 20;
    public int HEIGHT_MIN = -4;

...
}
```

level_data.txt

#		FLOOR		HOLE		HEIGHT	
# TIME	SPEED	MIN	MAX	MIN	MAX	MIN	MAX
0.0	5.0	9	10	1	2	0	0
15.0	6.0	8	9	3	5	0	0
30.0	7.0	7	8	7	9	-4	4
45.0	9.0	5	6	12	15	1	1
60.0	8.0	6	7	10	12	2	2

* text data를 그대로 읽어들여 해석하기

그림 7-6 데이터를 잘게 나누어 반영한다

level_texts

#		FLOOR		HOLE		HEIGHT	
# TIME	SPEED	MIN	MAX	MIN	MAX	MIN	MAX
0.0	5.0	9	10	1	2	0	0
15.0	6.0	8	9	3	5	0	0
30.0	7.0	7	8	7	9	−4	4
45.0	9.0	5	6	12	15	1	1
60.0	8.0	6	7	10	12	2	2

① 각 행 단위로 나눈다

lines

0.0	5.0	9	10	1	2	0	0

② 각 워드 단위로 나눈다

words

0.0	5.0	9	10	1	2	0	0

③ 각각 대응하는 변수에 넣는다

* text data를 그대로 읽어들여 해석하기

LevelControl.cs

```
public void loadLevelData(TextAsset level_data_text){
    // 텍스트 데이터를 문자열로 가져온다.
    string level_texts = level_data_text.text;
    // 개행 코드 '₩'마다 분할해서 문자열 배열에 넣는다.
    string[] lines = level_texts.Split('₩n');
    // lines 내의 각 행에 대해서 차례로 처리해 가는 루프.
    foreach(var line in lines) {
    if(line = ="") { // 행이 빈 줄이면.
        continue; // 아래 처리는 하지 않고 반복문의 처음으로 점프한다.
    };
    Debug.Log(line); // 행의 내용을 디버그 출력한다.
    string[] words = line.Split(); // 행 내의 워드를 배열에 저장한다.
    int n = 0;
    // LevelData형 변수를 생성한다.
    // 현재 처리하는 행의 데이터를 넣어 간다.
    LevelData level_data = new LevelData();
    // words내의 각 워드에 대해서 순서대로 처리해 가는 루프.
    foreach(var word in words) {
    if(word.StartsWith("#")) { // 워드의 시작문자가 #이면.
        break;} // 루프 탈출.
    if(word = = "") { // 워드가 텅 비었으면.
        continue;} // 루프의 시작으로 점프한다.
    // n 값을 0, 1, 2,...7로 변화시켜 감으로써 8항목을 처리한다.
    // 각 워드를 플롯값으로 변환하고 level_data에 저장한다.
    switch(n) {
        case 0: level_data.end_time = float.Parse(word); break;
        case 1: level_data.player_speed = float.Parse(word); break;
        case 2: level_data.floor_count.min = int.Parse(word); break;
        case 3: level_data.floor_count.max = int.Parse(word); break;
        case 4: level_data.hole_count.min = int.Parse(word); break;
        case 5: level_data.hole_count.max = int.Parse(word); break;
        case 6: level_data.height_diff.min = int.Parse(word); break;
        case 7: level_data.height_diff.max = int.Parse(word); break; }
    n++;
```

* text data를 그대로 읽어들여 해석하기

LevelControl.cs

```
if(n >= 8) { // 8항목(이상)이 제대로 처리되었다면.
    // List 구조의 level_datas에 level_data를 추가한다.
    this.level_datas.Add(level_data);
}
else { // 그렇지 않다면(오류의 가능성이 있다).
    if(n = = 0) { // 1워드도 처리하지 않은 경우는 주석이므로.
    // 문제없다. 아무것도 하지 않는다.
    } else { // 그 이외이면 오류다.
    // 데이터 개수가 맞지 않다는 것을 보여주는 오류 메시지를 표시한다.
    Debug.LogError("[LevelData] Out of parameter.₩n");
    }
}}
```

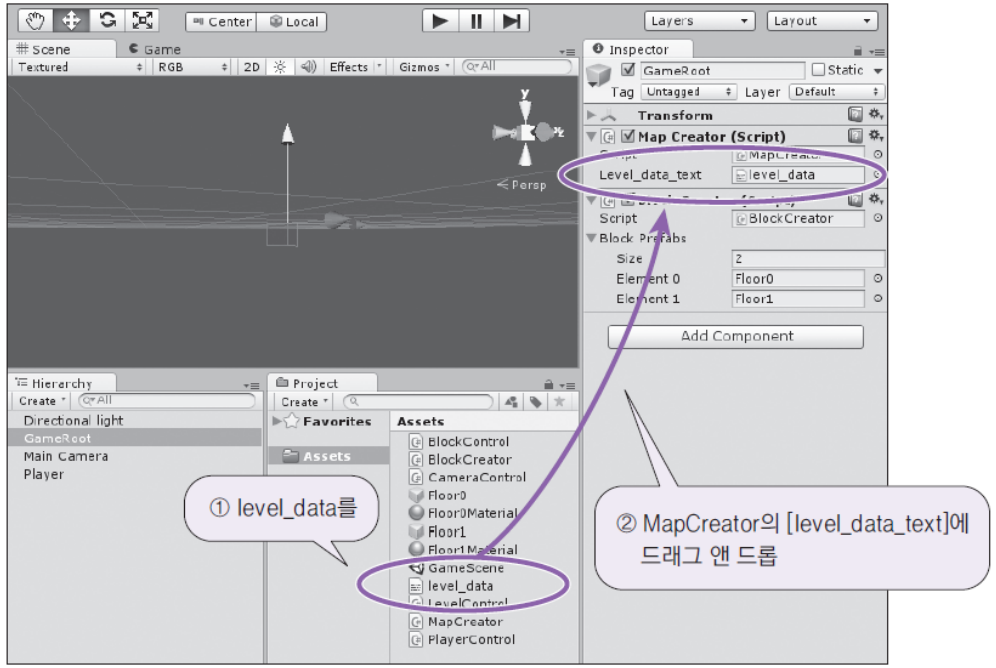
```
// level_datas에 데이터가 하나도 없으면.
if(this.level_datas.Count == 0) {
    // 오류 메시지를 표시한다.
    Debug.LogError("[LevelData] Has no data.₩n");
    // level_datas에 기본 LevelData를 하나 추가해 둔다.
    this.level_datas.Add(new LevelData());
    }
}
```

* loadLevelData() 메서드 호출

LevelControl.cs

```
public TextAsset level_data_text = null;
```

그림 7-7 level_data, MapCreator를 알려준다

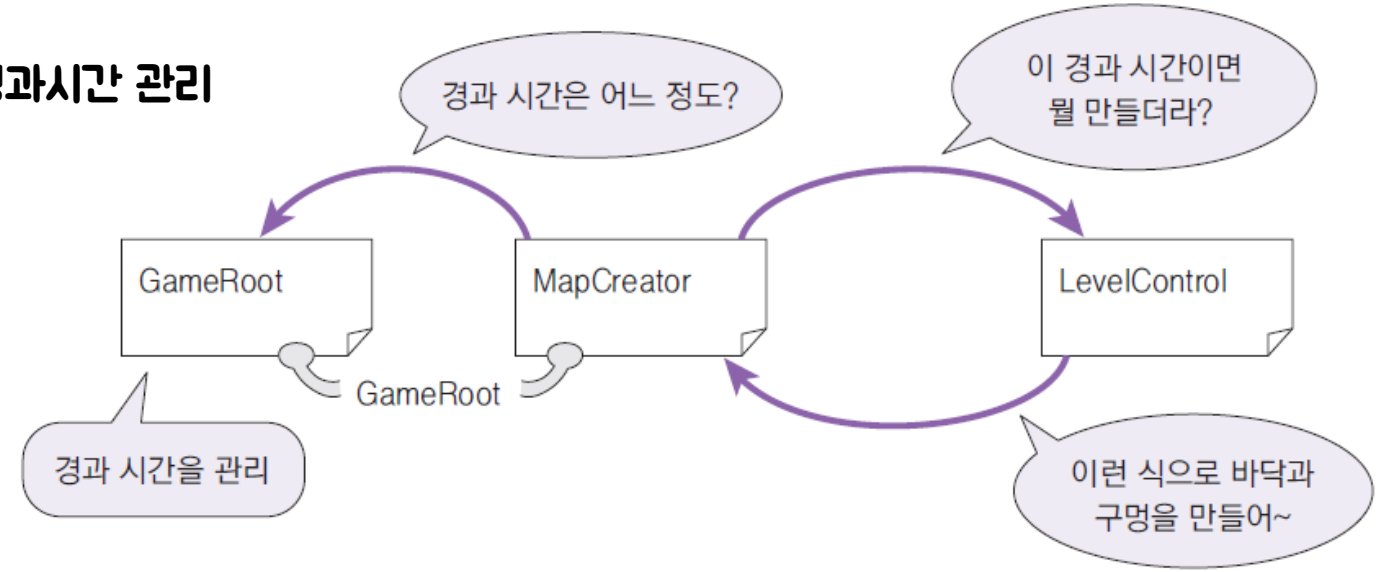



```
void Start() {
    this.player = GameObject.FindGameObjectWithTag("Player"
    ).GetComponent<PlayerControl>();
    this.last_block.is_created = false;
    this.block_creator = this.gameObject.GetComponent<BlockCreator>();
    this.level_control = new LevelControl();
    this.level_control.initialize();
    this.level_control.loadLevelData(this.level_data_text); //이 구문을 추가한다.
}
```

* 발판과 구멍에 데이터 반영

읽어온 text data 중 발판과 구멍에 관한 것을 반영시킨다.

그림 7-10 스크립트 연계



Gameroot 스크립트 = 경과시간 관리

① [Assets] → [Create] → [C# Script]로 스크립트를 만들고 'GameRoot'로 이름을 변경한다.
② Hierarchy의 GameRoot에 GameRoot 스크립트를 드래그 앤 드롭한다.

* 발판과 구멍에 데이터 반영

GameRoot.cs

```
public float step_timer = 0.0f; // 경과 시간을 유지한다.
void Update() {
    this.step_timer += Time.deltaTime; // 경과 시간을 더해 간다.
}
public float getPlayTime(){
    float time;
    time = this.step_timer;
    return(time); // 호출한 곳에 경과 시간을 알려준다.
}
```

MapCreator.cs

```
private GameRoot game_root = null;

void Start() {
...
this.game_root = this.gameObject.GetComponent<GameRoot>();
}

void create_floor_block() {
...
// this.level_control.update();
this.level_control.update(this.game_root.getPlayTime());
...
}
```

GameRoot 스크립트의 getPlayTime() 메서드를 사용하기 위해서 MapCreator 클래스를 수정해준다.

* level_data에서 읽어온 데이터 반영

LevelControl.cs

```
private void update_level(
ref CreationInfo current, CreationInfo previous, float passage_time){
    // 새 인수 passage_time으로 플레이 경과 시간을 받는다.
    // 레벨 1~레벨 5를 반복한다.
    float local_time = Mathf.Repeat(passage_time,
    this.level_datas[this.level_datas.Count - 1].end_time);
    // 현재 레벨을 구한다.
    int i;
    for(i = 0; i < this.level_datas.Count - 1; i++) {
        if(local_time <= this.level_datas[i].end_time) {
        break;}}
    this.level = i;
    current.block_type = Block.TYPE.FLOOR;
    current.max_count = 1;
if(this.block_count >= 10) {
    // 현재 레벨용 레벨 데이터를 가져온다.
    LevelData level_data;
    level_data = this.level_datas[this.level];
switch(previous.block_type) {
    case Block.TYPE.FLOOR: // 이전 블록이 바닥인 경우.
        current.block_type = Block.TYPE.HOLE; // 이번엔 구멍을 만든다.
        // 구멍 크기의 최솟값~최댓값 사이의 임의의 값.
        current.max_count = Random.Range(
        level_data.hole_count.min, level_data.hole_count.max);
        current.height = previous.height; // 높이를 이전과 같이 한다.
        break;
    case Block.TYPE.HOLE: // 이전 블록이 구멍인 경우.
        current.block_type = Block.TYPE.FLOOR; // 이번엔 바닥을 만든다.
        // 바닥 길이의 최솟값~최댓값 사이의 임의의 값.
        current.max_count = Random.Range(
        level_data.floor_count.min, level_data.floor_count.max);
```

* level_data에서 읽어온 데이터 반영

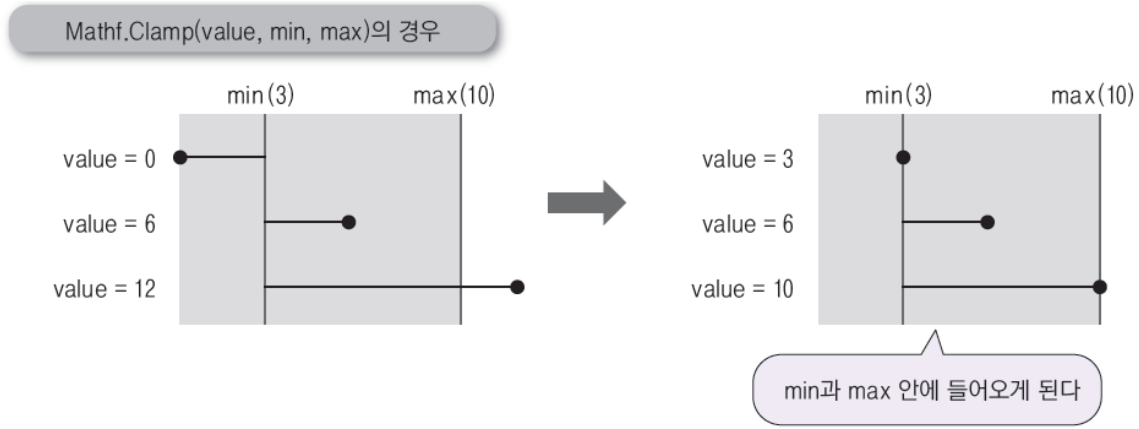
LevelControl.cs

```
// 바닥 높이의 최솟값과 최댓값을 구한다.
int height_min = previous.height + level_data.height_diff.min;
int height_max = previous.height + level_data.height_diff.max;
height_min = Mathf.Clamp(height_min, HEIGHT_MIN, HEIGHT_MAX);
height_max = Mathf.Clamp(height_max, HEIGHT_MIN, HEIGHT_MAX);
// 바닥 높이의 최솟값~최댓값 사이의 임의의 값.
current.height = Random.Range(height_min, height_max);
break; }}
```

// 지난 번 update_level()과 달라진 점 중 하나는
// 플레이 경과 시간을 인수 passage_time으로 받음

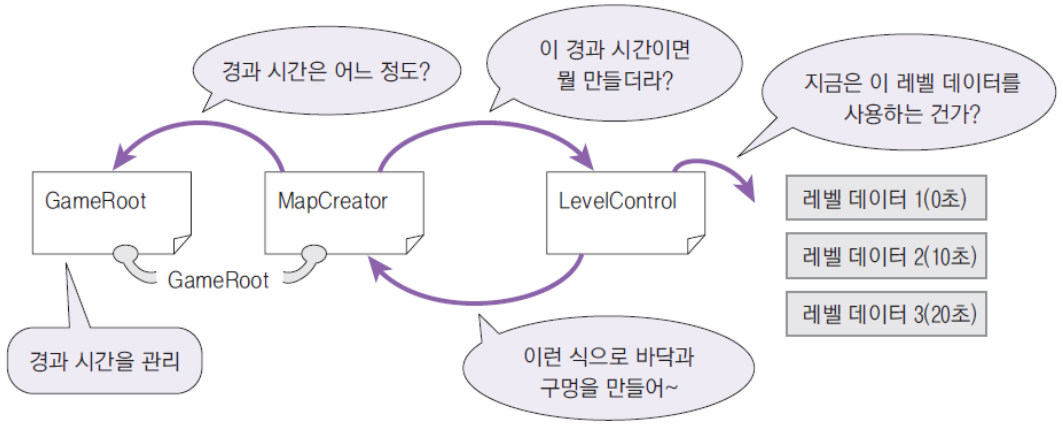
// 최솟값과 최대값 사이의 값을 강제로 넣기 위해 사용, 인수는 3개

그림 7-12 Mathf.Clampl() 메서드



* level_data에서 읽어온 데이터 반영

그림 7-13 스크립트의 연계



update() 메서드는 update_level()에 경과 시간을 알려줘야 한다.
경과 시간을 인수로 받도록 변경한다.

LevelControl.cs

```
// public void update()
public void update(float passage_time){
    this.current_block.current_count++;
    if(this.current_block.current_count >= this.current_block.max_count) {
        this.previous_block = this.current_block;
        this.current_block = this.next_block;
        this.clear_next_block(ref this.next_block);
        // this.update_level(ref this.next_block, this.current_block);
        this.update_level(
        ref this.next_block, this.current_block, passage_time);
}
this.block_count++;
}
```

* 플레이어에도 데이터 반영

<수정해야 할 사항>

- LevelControl 클래스에 getPlayerSpeed() 메서드 추가
- PlayerControl 클래스에 멤버 변수 두 개 추가
- PlayerControl 클래스의 Update() 메서드 수정
- MapCreator 클래스의 Start() 메서드에 한 행 추가

* 플레이어에도 데이터 반영

● LevelControl 클래스에 getPlayerSpeed() 메서드 추가

LevelControl.cs

```
public float getPlayerSpeed(){
return(this.level_datas[this.level].player_speed); }
```

● PlayerControl 클래스에 멤버 변수 두 개 추가

PlayerControl.cs

```
private float click_timer = -1.0f; // 버튼이 눌린 후의 시간.
private float CLICK_GRACE_TIME = 0.5f; // 점프하고 싶은 의사를 받아들일 시간.
```

* 플레이어에도 데이터 반영

● PlayerControl 클래스의 Update() 메서드 수정

PlayerControl.cs

```
void Update() {
……
    this.step_timer += Time.deltaTime;

    if(Input.GetMouseButtonDown(0)) { // 버튼이 눌렸으면.
        this.click_timer = 0.0f; // 타이머를 리셋.
    } else {
        if(this.click_timer >= 0.0f) { // 그렇지 않으면.
            this.click_timer += Time.deltaTime; // 경과 시간을 더한다.
    }}

if(this.next_step = = STEP.NONE) {
…
```

* 플레이어에도 데이터 반영

● PlayerControl 클래스의 Update() 메서드 수정

PlayerControl.cs

```
if(this.next_step = = STEP.NONE) {
    switch(this.step) {
    case STEP.RUN:
        if(! this.is_landed) {
        } else {
            if(Input.GetMouseButtonDown(0)) {
            this.next_step = STEP.JUMP; }}
break;
```

```
if(this.is_landed) { // 착지했다면.
    this.click_timer = -1.0f; // 버튼이 눌리지 않은 상태를 나타내는 -1.0f로.
    this.next_step = STEP.JUMP; // 점프 상태로 한다.
}
```

* 플라스틱 러너의 시퀀스 연결

- Title 씬을 만들고 프로그램을 준비.

TitleScript.cs

```
void Update() {
    if(Input.GetMouseButtonDown(0)) {
    Application.LoadLevel("GameScene");
}}

void OnGUI() {
    GUI.Label(new Rect(Screen.width / 2, Screen.height / 2, 128, 32),
    "PlasticRunner");
}
```

* 플라스틱 러너의 시퀀스 연결

- 사용할 씬을 프로젝트에 알려주어야 한다.

① [File] → [Building Settings...]를 실행

② [Building Settings] 창의 [Scenes In Build]에 TitleScene을 드래그 앤 드롭

③ 마찬가지로 GameScene을 드래그 앤 드롭

playerControl.cs

```
public bool isPlayEnd() // 게임이 끝났는지 판정.
{
    bool ret = false;
    switch(this.step) {
    case STEP.MISS: // MISS 상태라면.
        ret = true; // '죽었어요'(true)라고 알려줌.
        break;
    }
return(ret);
}
```

* 플라스틱 러너의 시퀀스 연결

- 구멍에 빠져서 게임 타이틀로 돌아가게 한다.

GameRoot.cs

```
public class GameRoot : MonoBehaviour {
public float step_timer = 0.0f;
private PlayerControl player = null;
void Start(){
    this.player = GameObject.FindGameObjectWithTag(
    "Player").GetComponent<PlayerControl>();
}
void Update() {
    this.step_timer += Time.deltaTime;
    if(this.player.isPlayEnd()) {
    Application.LoadLevel("TitleScene");
}}
public float getPlayTime()
…
```

유니티
게임 제작 입문

다음 페이지에서 보아요

길벗