C++ 프로그래밍 및 실습

SaveSome

최종 보고서

제출일자: 2024.12.22
제출자명: 김동민
제출자학번: 185478

1. 제안 동기 및 필요성

- 모바일 게임이 유행한 후 몇 년간 현실의 재화가 게임 내의 강력함으로 연결되어 그것이 곧 난이도가 되는 기조가 더욱 강해졌습니다. 또한 별 노력을 요하지 않는 방치형 게임들도 큰 인기를 얻었으며 이용자를 모으기 위해 자극적인 부분을 강조하거나 인게임과는 다른 영상을 어필하는 경우도 많아졌다고 생각합니다. 그런 요소들을 모두 뺀 게임을 만들고 싶었습니다.
- 비 오는 날 날아다니는 잠자리를 보며 우산이 필요 없을까 궁금했습니다. 그 때 다양한 동물과 곤충을 집까지 바래다주는 내용의 게임을 기획해 보고 싶었습니다. 처음에는 게임에 있어서 영광의 시대이자 몰락의 상징인 아타리의 게임을 변형시켜서 만들고자 했으나, 벅차더라도 새로운 시도를 해 보고 싶어서 특정 오브젝트를 구하는 내용의 게임을 기획했습니다.
- 지금은 색의 상성으로 인해 각 스테이지의 내용과 공략 방법이 달라지지만, 추후 언리얼 엔진을 배워 각 오브젝트를 동물과 플레이어(사람)으로 확장시킨다면 해당 색은 더 다양화될 것이며, 색은 동물 혹은 곤충이 가진 속성이 될 것입니다. 장애물은 자연재해나 위험한 쓰레기와 같이 생태계를 위협하는 물체로 확장될 것입니다.

2. 게임 개요

테마

플랫포머 게임: 플랫포머 장르는 공격 방식에 따라 합앤밥, 런앤건, 보블보블, 퍼즐 플랫 포머로 구분된다. 이들의 공통적인 특징은 플랫폼 위를 자유롭게 이동하며 게 임을 진행한다는 점이며, 위에 언급된 레벨 에디터 콘텐츠 사례에서 사용자는 게임 구성요소를 자유롭게 배치하며 게임을 제작한다는 점을 비추어보았을 때, 플랫포머 장르의 맵을 자유롭게 이동하며 게임을 플레이하는 특성은 레벨 에디터를 통한 게임 콘텐츠 창작 장르에 적합한 장르라고 판단된다. (이진우, 2017)

대상자

본 게임의 대상자는 10 대, 20 대이며, 자극적이고 파괴적인 게임에 지친 사람, 어떠한 변수도 없이 전략과 컨트롤로 승부를 보는 걸 좋아하는 승부욕이 넘치는 사람이 대상입니다. 추후 확장시에는 동물을 좋아하는 사람도 대상자입니다.

이용환경

추후 언리얼을 활용하여 확장하기 위해서 PC 로 설정했습니다.

재미요소

A. 숙달-어려운 도전

본 게임은 여러 번 시도할수록 숙달되며, 각 스테이지에 따라 달라지는 지형과 재해 패턴, 위협 요소를 파악해서 점차 수월하게 공략하는 재미가 있습니다. 약점도 매 스테이지 변하기에 헷갈리지 않게 주의해야 하며 다양한 아이템을 스테이지 내에 배치해서 플레이어가 만들 수 있는 변수를 다양화하면 숙달될수록 더욱 다양한 방식을 시도하며 재미를 느낄 것입니다.

B. 달성-지키는 보람

무언가를 파괴하고 재화를 얻기 위함이 아닌, 순순히 대상을 지키며 보람을 느낄 수 있습니다. 현재 다양한 게임은 짧은 시간, 짧은 단계별로 연속적인 보상을 얻을 수 있도록 기획하며, 그것으로 이용자를 붙잡아 둡니다. 하지만 게임 밖의 세계에서는 그런 것들이 의미가 없는 경우가 많습니다. 게임처럼 행동해도 얻을 수 없는 것들이 대부분입니다. 마치 무언가 얻었다고 생각하지만, 사실 아무것도 없을 때 공허함을 느낄 수 있습니다. 하지만 무언가 지키는 행위는 현실에서도 보람을 주기에 이렇게 기획했습니다.

C. 교감

굳이 말로 하지 않아도, 지켜야 하는 대상과 유대감을 느낄 수 있도록 간단하지만 귀여운 외형과 행동을 디자인해보고 싶습니다.

비즈니스 모델

정가 판매: 제품에 정해진 가격이 부여되며, 단 한번의 비용 지불로 게임을 소유할 수 있습니다.

컨텐츠 다운로드(DLC): 더 다양한 적과 다양한 환경을 추가해서 DLC 로 묶어 판매할 수 있습니다.

3. 게임 스토리

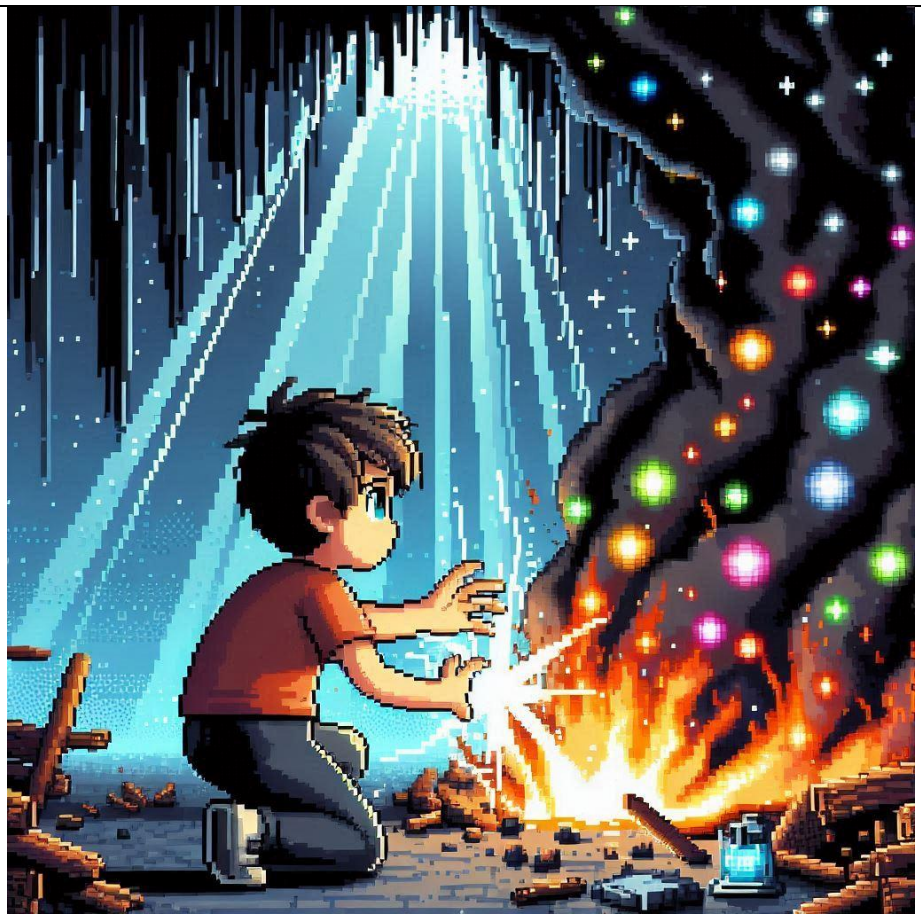
게임 스토리

2.1.1. 제한된 색만이 허용되는 픽셀의 세계에 떨어진 주인공, 해당 세계는 오직 관리자만이 다양한 색을 독점하고 있다. 관리자는 색을 통해 모든 존재를 관리하기에 이질적인 색이 들어오면 적극적으로 제거하려 한다.

2.1.2. 주인공은 본인의 색을 마음대로 바꿀 수 있기에 생존의 위협은 느끼지 못했지만, 색이 적은 세상에 질려버리고 만다. 그러던 중 공격당하던 색들을 충동적으로 구하게 되고, 거기서부터 다채로운 세상을 만들고자 돌아다니며 색을 숨겨준다.

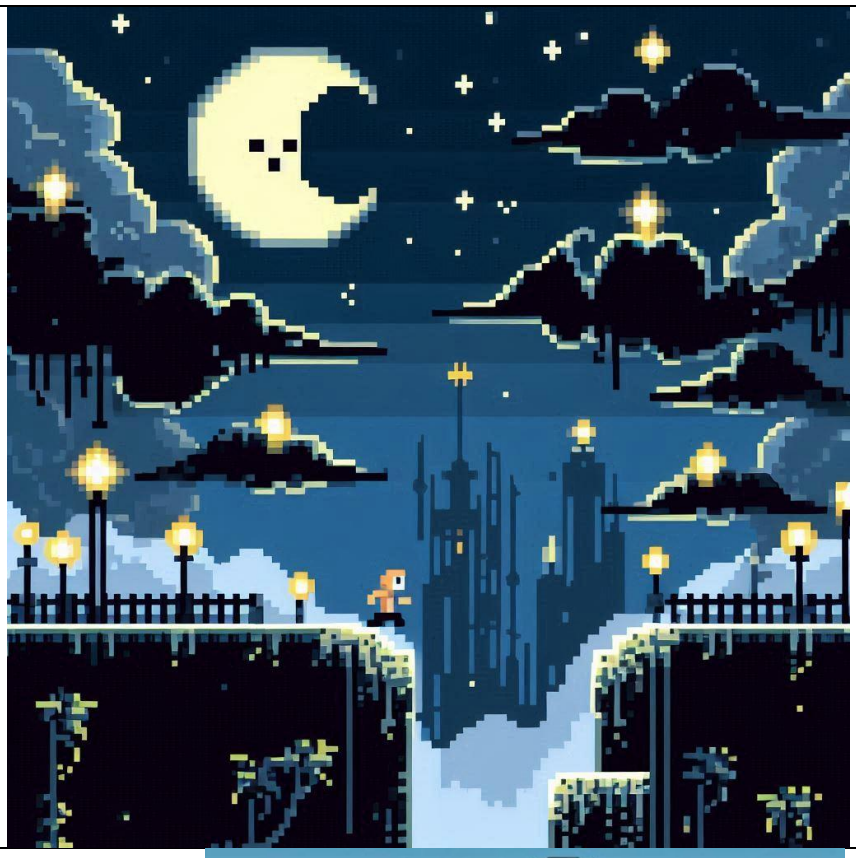
2.1.3. 1차적으로 주어진 스테이지를 모두 클리어 하게 되면 엔딩을 보는 방식입니다. 끊임없이 기록을 세우며 경쟁하는 게임보다는 엔딩을 정해두고, 엔딩 시간을 기록한 후 추후 재도전 가능한 방식으로 기획했습니다.

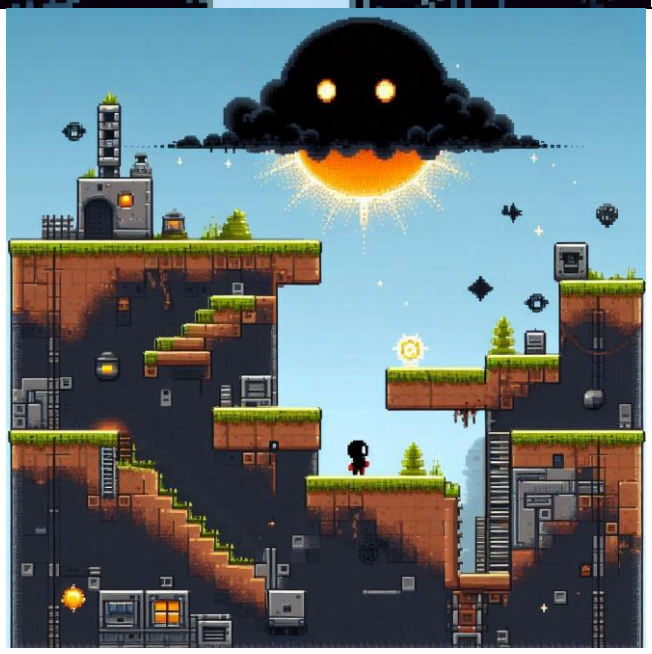
4. 스토리 컨셉 아트



5. 맵 디자인<희망편>

본 맵 디자인은 지형과 테마만 참고로 봐주시면 감사드리겠습니다.





6. 게임 매커니즘

게임 목표
플레이어가 목표 오브젝트를 장애물로부터 지키며 골인 지점까지 데려다 주면 스테이지 클리어입니다.

플레이어
플레이어는 목표 오브젝트를 유인과 동시에 수호하며 목표 지점에 도달하는 것이 목표입니다.

목표 오브젝트(Shining paint/ SP로 요약)
초기색은 흰색으로 시작함, 플레이어가 스테이지 내에서 아이템을 먹어서 색 변화 가능
SP는 일정 AI 패턴을 가지고 움직임

장애물
빨간색: 초록색 픽셀을 파괴합니다.
파란색: 빨간색 픽셀을 파괴합니다.
초록색: 파란색 픽셀을 파괴합니다.
검정색: 모든 걸 파괴합니다.
장애물에 닿을 때마다 플레이어와 SP의 해당 부위 픽셀이 사라집니다.
플레이어 혹은 목표 오브젝트의 픽셀이 모두 깎이면 패배합니다.

아이템
색상 변경 아이템
SP의 색상을 바꾸어 특정 장애물의 영향을 덜 받게 합니다.
장애물 상성에 맞춰 아이템을 사용할 수 있는 동선을 짜야 합니다.

속도 증가 아이템
SP의 속도가 증가하여 플레이어와 더 빠르게 이동합니다.
심각한 공격 패턴을 앞두고 은폐 엄폐가 필요할 때, 혹은 SP가 올바른 길을 가고 있을 때 활용 가능합니다.

플레이어 선택 요소
본인 색 선택: 개개인이 어려움을 느끼는 장애물이나 스테이지를 고려해 색을 선택합니다.
스테이지 내 동선 설정: 장애물 패턴과 특성, SP의 행동을 고려하여 효율적인 동선을 구상해야 합니다.
아이템 사용: 아이템을 습득하면 바로 발동하기에 일부러 먹지 않도록 주의하며 필요한 타이밍에 사용해야 합니다.

7. 게임 플로우

균형 루프 (Balanced Loop)
플레이어와 목표 오브젝트의 픽셀이 깎일수록 장애물을 피하기 쉽기에 위험할수록 도전 기회가 커지도록 설계되었습니다. 따라서 위기 상황에서도 클리어 가능성을 유지하게 하는 위험-보상 루프가 형성됩니다.

게임 루프
매 프레임마다 반복되며, 모든 오브젝트(플레이어, SP, 장애물, 아이템)의 상태를 업데이트하고 충돌을 감지하며, 목표 달성 여부를 체크하는 역할을 합니다. 승리 또는 패배 조건이 충족될 때까지 루프가 지속됩니다.

색상 파괴 루프
플레이어나 SP가 장애물과 충돌 시에 반복됩니다. 특정 색상 장애물이 다른 색상 픽셀을 파괴하는 규칙을 통해 지속적인 색상 기반 상호작용 루프가 만들어집니다. 이와 같은 구조로 루프들이 상호작용하며 게임의 흥미로운 난이도를 유지하게 됩니다.

8. 진척사항

기능 구현

1

```cpp
#include <SFML/Graphics.hpp>
#include <iostream>
int main()
{
    sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
    sf::CircleShape shape(100.f);
    shape.setFillColor(sf::Color::Green);

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        window.clear();
        window.draw(shape);
        window.display();
    }

    return 0;
}
```

설명

SFML 설치 및 visual studio 환경 설정

32비트용 SFML을 다운받은 후, visual studio의 project 속성 메뉴에서 경로를 지정해주고, 각 모듈의 라이브러리를 추가 종속성에 추가했습니다. 위 코드를 통해서 정상 작동을 확인했습니다.

```cpp
#ifndef PLAYER_H
#define PLAYER_H

#include <SFML/Graphics.hpp>

class Player {
public:
    void initialize();
    void update(float deltaTime);
    void render(sf::RenderWindow& window);
    void changeColor(sf::Color newColor);
    void losePixel();
    sf::FloatRect getBounds() const;

private:
    sf::RectangleShape shape;
    sf::Color color;
    float speed = 200.f; //좌우 이동 속도
    float jumpVelocity = 0.f;// 점프 속도
    float gravity = 400.f;// 중력 가속도
    bool isJumping = false;// 점프 여부
    float groundY = 500.f; //바닥의 Y좌표
    int pixels = 100; //플레이어의 체력, 공격당하면 픽셀이 깎인다.
};

#endif // PLAYER_H
```

2

멤버변수	함수	설명 및 배운 내용
speed: 좌우 이동 속도를 설정합니다. 초당 이동 픽셀 수. jumpVelocity: 점프의 초기 속도 및 점프 중 가속도. gravity: 중력 가속도를 설정합니다. 점프 후 플레이어가 아래로 떨어지도록 합니다. isJumping: 플레이어가 점프 중인지 여부를 나타냅니다. 점프 중에는 중력의 영향을 받아 내려가게 됩니다. groundY: 바닥의 Y 좌표를 설정합니다. 이 값은 플레이어가 떨어지지 않고 바닥에 서 있을 수 있게 합니다 pixels: 플레이어의 체력을 나타내며, 픽셀 감소와 관련이 있습니다..	initialize(): 플레이어를 초기화하는 함수로, 플레이어의 크기와 색상, 위치를 설정합니다. update(float deltaTime): 매 프레임마다 호출되는 함수로, 플레이어의 이동 및 점프 처리, 중력 적용 등을 담당합니다. 매개변수로 deltaTime(시간차이)를 받습니다. render(sf::RenderWindow& window): 화면에 플레이어를 그립니다. 매개변수로 window(렌더링 창)을 받습니다. getBounds(): 플레이어의 크기를 나타내는 sf::FloatRect를 반환하여 충돌 감지에 사용됩니다. changeColor(sf::Color newColor) 플레이어의 색상을 변경합니다 매개변수로 newColor(변경할	Player class 플레이어 캐릭터를 정의하고 플레이어의 행동(이동, 아이템 사용)과 상태(HP, 현재 위치)를 관리합니다. 배운 내용 클래스, 접근 제한자, const 멤버 함수(안정성)

		색상)을 받습니다. losePixel() Obstacle과 충돌시 플레이어 크기를 줄임으로써 체력 감소를 나타낼 함수입니다. getBounds() sf::FloatRect (플레이어 경계)를 반환합니다.	

```
#include "Player.h"
using namespace sf;

void Player::initialize() {
    shape.setSize(Vector2f(20.f, 20.f));//플레이어 크기 설정
    shape.setFillColor(Color::Red);
    shape.setPosition(100.f, groundY); //플레이어 초기 위치 설정
    color = Color::Red;
}
```

	입출력	설명	배운내용
3	Vector2f: 플레이어의 크기 Color::Red : 플레이어의 색상 100.f, groundY: 플레이어의 초기 위치	Player 픽셀 구현 shape.setSize(sf::Vector2f(20.f, 20.f))로 플레이어를 20x20 픽셀 크기의 사각형으로 설정합니다. shape.setFillColor(sf::Color::Red)로 플레이어의 색을 빨간색으로 설정합니다. shape.setPosition(100.f, groundY)로 초기 위치를 화면에서 x = 100과 y = groundY로 설정합니다	

4

```
void Player::update(float deltaTime) {
    Vector2f movement(0.f, 0.f);

    // 좌우 이동
    if (Keyboard::isKeyPressed(Keyboard::Left))
        movement.x -= speed * deltaTime;
    if (Keyboard::isKeyPressed(Keyboard::Right))
        movement.x += speed * deltaTime;
```

입출력	설명	배운내용
Keyboard :: Left : 키보드의 왼쪽 화살표의 입력 확인 Keyboard :: Right : 키보드의 오른쪽 화살표의 입력 확인	플레이어 좌우 이동 구현 sf::Keyboard::isKeyPressed(sf::Keyboard::Left)와 sf::Keyboard::isKeyPressed(sf::Keyboard::Right)를 사용해 왼쪽 및 오른쪽 화살표 키 입력을 확인합니다. 그에 따라 플레이어의 shape가 왼쪽 또는 오른쪽으로 이동합니다.	조건문

5

```
// 점프 처리
if (Keyboard::isKeyPressed(Keyboard::Space) && !isJumping) {
    jumpVelocity = -250.f; // 점프 속도
    isJumping = true;      // 점프 상태로 전환
}

// 중력 적용
jumpVelocity += gravity * deltaTime;
movement.y += jumpVelocity * deltaTime;
```

입출력	설명	배운내용

	sf::Keyboard::isKeyPressed(sf::Keyboard::Space)를 통해 스페이스바 입력.	플레이어 점프 구현 점프할 때 jumpVelocity가 초기화되어 플레이어가 위로 이동하며, 중력이 적용되어 아래로 떨어지게 됩니다.	

6

```
// 화면 경계 체크 (화면 바깥으로 나가지 않도록 위치 조정)
Vector2f position = shape.getPosition();
if (position.x < 0) {
    position.x = 0; // 왼쪽 경계
}
if (position.x + shape.getSize().x > 800) { // 화면 너비(800)를 기준으로 경계 체크
    position.x = 800 - shape.getSize().x; // 오른쪽 경계
}
shape.setPosition(position);
```

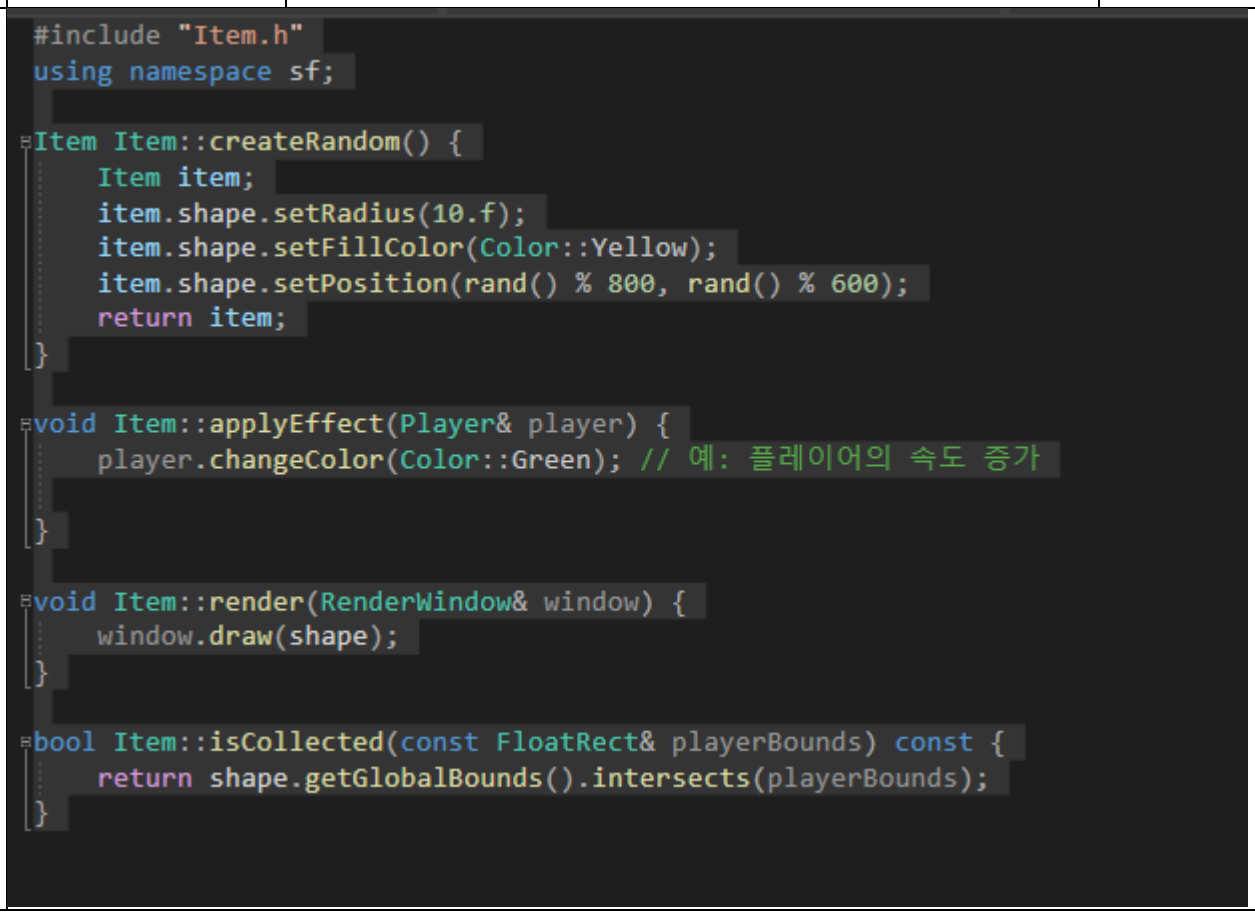
입출력	설명	배운내용

	sf::Vector2f position = shape.getPosition();를 통해 플레이어의 현재 위치를 얻고, 화면의 왼쪽 및 오른쪽 경계를 체크하여 플레이어가 화면을 벗어나지 않도록 제한합니다.
	
	if (position.x < 0):
	왼쪽 경계를 벗어날 경우, position.x = 0으로 설정하여 왼쪽 경계에서 멈추도록 합니다.
	
	if (position.x + shape.getSize().x > 800):
	오른쪽 경계를 벗어날 경우, position.x = 800 - shape.getSize().x로 설정하여 오른쪽 경계에서 멈추도록 합니다.
	이로써, 플레이어는 화면 밖으로 나가지 않게 제한됩니다.
		플레이어 이동 제한 구현

7

```cpp
#include "GameManager.h"

using namespace sf;

GameManager& GameManager::getInstance() {
    static GameManager instance;
    return instance;
}

void GameManager::initialize() {
    window.create(VideoMode(800, 600), "Game with Features");
    player.initialize();
    map.initialize(20, 15); // 20x15 맵 생성
}

void GameManager::processEvents() {
    Event event;
    while (window.pollEvent(event)) {
        if (event.type == Event::Closed) {
            window.close();
        }
    }
}

void GameManager::update(float deltaTime) {
    player.update(deltaTime);
    for (auto& item : items) {
        if (item.isCollected(player.getBounds())) { // 경계 체크
            item.applyEffect(player);
        }
    }
    for (auto& obstacle : obstacles) {
        obstacle.update(deltaTime);
        if (obstacle.collidesWith(player.getBounds())) { // 경계 체크
            player.losePixel();
        }
    }
    map.update(player.getBounds()); // 맵 충돌 체크

}

void GameManager::render() {
    window.clear();
    map.render(window);
    player.render(window);
    for (auto& item : items) {
        item.render(window);
    }
    for (auto& obstacle : obstacles) {
        obstacle.render(window);
    }
    window.display();
}

void GameManager::spawnItem() {
    items.emplace_back(Item::createRandom());
}

void GameManager::spawnObstacle() {
    obstacles.emplace_back(Obstacle::createFallingObstacle(window.getSize().x));
}
```

멤버변수	함수	설명 및 배운 내용

	window: 게임 창 (sf::RenderWindow) 게임의 모든 렌더링과 이벤트를 처리. player: Player 객체 플레이어 데이터를 관리. map: Map 객체 맵 데이터와 충돌, 렌더링을 관리. items: std::vector<Item> 맵에 생성된 아이템 목록. obstacles: std::vector<Obstacle> 맵에 생성된 장애물 목록.	getInstance() 정적 지역 변수로 GameManager 인스턴스를생성하고 GameManager& (싱글턴 객체 참조)를 반환합니다. initialize() 게임 창, 플레이어, 맵을 초기화합니다. processEvents() 이벤트 루프를 실행, 닫기 및 이벤트 처리를 합니다. update(float deltaTime) deltaTime (시간 차이)을 매개변수로 받아 플레이어, 아이템, 장애물, 맵 상태를 업데이트 합니다. render() 창을 새로 그리며 플레이어, 맵, 아이템, 장애물을 렌더링 합니다. spawnItem() 아이템을 생성하고 items 벡터에 추가할 예정입니다. spawnObstacle() 장애물을 생성하고 obstacles 벡터에 추가할 예정입니다.	GameManager 각 서브 시스템에서 발생한 상태 변화를 멤버 변수의 변경을 통해 관리하는 핵심 클래스입니다. 게임 흐름, 데이터, 이벤트를 관리합니다. 배운 내용 싱글턴 패턴, auto 키워드, 객체 참조
8			

```cpp
#include "Item.h"
using namespace sf;

Item Item::createRandom() {
    Item item;
    item.shape.setRadius(10.f);
    item.shape.setFillColor(Color::Yellow);
    item.shape.setPosition(rand() % 800, rand() % 600);
    return item;
}

void Item::applyEffect(Player& player) {
    player.changeColor(Color::Green); // 예: 플레이어의 속도 증가

}

void Item::render(RenderWindow& window) {
    window.draw(shape);
}

bool Item::isCollected(const FloatRect& playerBounds) const {
    return shape.getGlobalBounds().intersects(playerBounds);
}
```

	변수	함수	설명 및 배운 내용
	shape: sf::CircleShape	createRandom() 랜덤 위치와 크기의 아이템 생성하고 Item 객체를 반환합니다. applyEffect(Player& player) player (플레이어 참조)를 매개변수로 받아서 아이템 효과를 플레이어에게 적용합니다. 현재는 임시로 색상 변경으로 정해뒀습니다. render(sf::RenderWindow& window) window (렌더링 창)을 매개변수로 받아 아이템을 화면에 렌더링합니다. isCollected(const FloatRect& playerBounds) playerBounds (플레이어 경계)를 매개변수로 받고 아이템이 플레이어와 충돌했는지 확인하여 bool값을 반환합니다.	Item class 아이템 종류, 위치, 효과를 관리합니다. 배운 내용 객체 참조 및 const

9

```cpp
#include "Obstacle.h"

using namespace sf;

Obstacle::Obstacle(Vector2f position, Vector2f size, Color color) {
    shape.setSize(size);
    shape.setFillColor(color);
    shape.setPosition(position);
}

void Obstacle::render(RenderWindow& window) {
    window.draw(shape);
}

void Obstacle::update(float deltaTime) {
    // 아래로 낙하
    shape.move(0, fallSpeed * deltaTime);
}

bool Obstacle::collidesWith(const FloatRect& targetBounds) const {
    return shape.getGlobalBounds().intersects(targetBounds);
}

FloatRect Obstacle::getBounds() const {
    return shape.getGlobalBounds();
}

Obstacle Obstacle::createFallingObstacle(float windowWidth) {
    // 랜덤 위치에서 생성되는 장애물
    float x = static_cast<float>(rand() % static_cast<int>(windowWidth - 20.f)); // 20.f는 장애물
    Vector2f position(x, 0.f); // 화면 상단에서 시작
    Vector2f size(20.f, 20.f); // 기본 크기
    return Obstacle(position, size);
}

```

	변수	함수	설명 및 배운 내용
	shape: sf::RectangleShape 장애물의 그래픽 표현과 위치. fallSpeed: float 장애물이 낙하하는 속도 (기본값: 100.f)...	Obstacle(Vector2f position, Vector2f size, Color color) 위치, 크기, 색상을 매개변수로 받아 장애물을 초기화합니다. update(float deltaTime) deltaTime (시간 차이)를 매개변수로 받아서 장애물을 아래로 낙하시킵니다. collidesWith(const FloatRect& targetBounds) targetBounds (플레이어 경계)를 매개변수로 받아서 장애물이 플레이어와 충돌했는지 확인후 bool값을 반환합니다.	Obstacle class 장애물 위치, 크기, 속성을 관리합니다.

10

```cpp
#include "Map.h"
#include <cstdlib>
using namespace sf;

void Map::initialize(int width, int height) {

    mapData = std::vector<std::vector<int>>(height, std::vector<int>(width, 0));
    tileShape.setSize(Vector2f(tileSize, tileSize));
    tileShape.setOutlineColor(Color::Black);
    tileShape.setOutlineThickness(1);

    loadMap(); // 맵 데이터 생성
}

void Map::loadMap() {
    int width = mapData[0].size();
    int height = mapData.size();

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            // 벽으로 맵 테두리 생성
            if (x == 0 || x == width - 1 || y == 0 || y == height - 1) {
                mapData[y][x] = 1; // 1은 충돌 가능 블록
            } else if (rand() % 10 == 0) {
                mapData[y][x] = 2; // 2는 색상 타일 (랜덤)
            }
        }
    }
}

void Map::update(const FloatRect& playerBounds) {
    int width = mapData[0].size();
    int height = mapData.size();

    // 충돌 처리 (예시)
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            if (mapData[y][x] == 1) {
                FloatRect tileBounds(x * tileSize, y * tileSize, tileSize, tileSize);
                if (playerBounds.intersects(tileBounds)) {
                    // 충돌 발생 시 플레이어 위치를 보정
                }
            }
```

```cpp
void Map::render(RenderWindow& window) {
    int width = mapData[0].size();
    int height = mapData.size();

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            if (mapData[y][x] > 0) {
                tileShape.setPosition(x * tileSize, y * tileSize);

                if (mapData[y][x] == 1)
                    tileShape.setFillColor(Color::Magenta); // 벽
                else if (mapData[y][x] == 2)
                    tileShape.setFillColor(Color::Green); // 특수 타일

                window.draw(tileShape);
            }
        }
    }
}

bool Map::isCollidable(int x, int y) const {
    if (x >= 0 && y >= 0 && y < mapData.size() && x < mapData[0].size())
        return mapData[y][x] == 1;
    return false;
}
```

변수	함수	설명 및 배운 내용
mapData: std::vector<std::vector<int>> 맵의 타일 정보를 저장. 1: 충돌 가능, 2 특수 타일. tileShape: sf::RectangleShape 개별 타일의 그래픽 표현. tileSize: int 각 타일의 크기 (기본값: 40).	initialize(int width, int height) width, height (맵 크기)를 매개변수로 받아서 맵 데이터를 생성하고 타일을 초기화합니다. update(const FloatRect& playerBounds) playerBounds (플레이어 경계)를 매개변수로 받아서 플레이어와 타일의 충돌 여부를 확인합니다. render(sf::RenderWindow& window) window (렌더링 창)을 매개변수로 받아서 타일을 화면에 렌더링합니다. isCollidable(int x, int y) x, y (좌표)를 매개변수로 받아서 주어진 좌표가 충돌 가능한지 확인하고 boo값을 반환합니다.	Map 맵의 크기, 타일 정보, 경계 등을 관리합니다. 충돌 처리와 특수 타일 연산으로 게임 플레이를 개선합니다. 플레이어와 타일 간의 상태 동기화 유지합니다.

11

```cpp
#include "ColorSelectionScene.h"

ColorSelectionScene::ColorSelectionScene()
    : colorSelected(false), selectedColor(sf::Color::White) {
    redArea.setSize({ 800 / 3.f, 600 });
    redArea.setPosition(0, 0);
    redArea.setFillColor(sf::Color::Red);

    greenArea.setSize({ 800 / 3.f, 600 });
    greenArea.setPosition(800 / 3.f, 0);
    greenArea.setFillColor(sf::Color::Green);

    blueArea.setSize({ 800 / 3.f, 600 });
    blueArea.setPosition(2 * 800 / 3.f, 0);
    blueArea.setFillColor(sf::Color::Blue);
}

void ColorSelectionScene::handleInput(sf::RenderWindow& window) {
    if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {
        sf::Vector2i mousePos = sf::Mouse::getPosition(window);
        if (redArea.getGlobalBounds().contains(static_cast<sf::Vector2f>(mousePos))) {
            selectedColor = sf::Color::Red;
            colorSelected = true;
        }
        else if (greenArea.getGlobalBounds().contains(static_cast<sf::Vector2f>(mousePos))) {
            selectedColor = sf::Color::Green;
            colorSelected = true;
        }
        else if (blueArea.getGlobalBounds().contains(static_cast<sf::Vector2f>(mousePos))) {
            selectedColor = sf::Color::Blue;
            colorSelected = true;
        }
    }
}

void ColorSelectionScene::render(sf::RenderWindow& window) {
    window.draw(redArea);
    window.draw(greenArea);
    window.draw(blueArea);
}

bool ColorSelectionScene::isColorSelected() const {
    return colorSelected;
}

sf::Color ColorSelectionScene::getSelectedColor() const {
    return selectedColor;
}
```

변수 (멤버,생성자)	함수	설명 및 배운 내용

	sf::RectangleShape redArea 기능: 빨간색 선택 영역을 나타냅니다. 접근 제어: private sf::RectangleShape greenArea 기능: 초록색 선택 영역을 나타냅니다. 접근 제어: private sf::RectangleShape blueArea 기능: 파란색 선택 영역을 나타냅니다. 접근 제어: private sf::Color selectedColor 기능: 현재 선택된 색상을 저장합니다. 접근 제어: private bool colorSelected 기능: 색상이 선택되었는지 여부를 나타냅니다. 접근 제어: private	ColorSelectionScene() 기능: 객체를 초기화하며, 선택 영역 및 초기 상태를 설정합니다. 접근 제어: public void handleInput(sf::RenderWindow& window) 기능: 사용자의 입력을 처리하여 색상 선택 상태를 업데이트합니다. 입력: sf::RenderWindow& (렌더링 창). 출력: 없음. 접근 제어: public void render(sf::RenderWindow& window) 기능: 빨강, 초록, 파랑 선택 영역을 화면에 렌더링합니다. 입력: sf::RenderWindow& (렌더링 창). 출력: 없음. 접근 제어: public bool isColorSelected() const 기능: 색상이 선택되었는지 여부를 반환합니다. 입력: 없음. 출력: bool (colorSelected). 접근 제어: public sf::Color getSelectedColor() const 기능: 선택된 색상을 반환합니다. 입력: 없음. 출력: sf::Color (selectedColor). 접근 제어: public	ColorSelectionScene는 색상 선택 화면을 관리하는 클래스입니다. 사용자 입력을 처리하여 색상을 선택하며, 선택된 상태와 색상을 반환하는 메소드가 제공됩니다.

```cpp
#include "Obstacle.h"

Obstacle::Obstacle(sf::Vector2f position, sf::Vector2f size, sf::Color color) {
    shape.setSize(size);
    shape.setPosition(position);
    shape.setFillColor(color);
}

void Obstacle::render(sf::RenderWindow& window) const{
    window.draw(shape);
}

void Obstacle::update(float deltaTime) {
    shape.move(0, fallSpeed * deltaTime); // 아래로 낙하
}

bool Obstacle::collidesWith(const sf::FloatRect& targetBounds) const {
    return shape.getGlobalBounds().intersects(targetBounds);
}

sf::FloatRect Obstacle::getBounds() const {
    return shape.getGlobalBounds();
}

Obstacle Obstacle::createFallingObstacle(float windowWidth) {
    float x = static_cast<float>(std::rand() % static_cast<int>(windowWidth));
    return Obstacle({ x, 0 }, { 20, 20 }, sf::Color::Red);
}
```

12

```cpp
#include "ObstacleManager.h"

void ObstacleManager::update(float deltaTime, const sf::FloatRect& playerBounds) {
    for (auto it = obstacles.begin(); it != obstacles.end();) {
        it->update(deltaTime);

        if (it->collidesWith(playerBounds)) {
            // 충돌 로직 (플레이어 피해 등)
            it = obstacles.erase(it);
        }
        else if (it->getBounds().top > 600) {
            it = obstacles.erase(it);
        }
        else {
            ++it;
        }
    }

    // 랜덤하게 새로운 장애물 생성
    if (std::rand() % 100 < 10) {
        obstacles.push_back(Obstacle::createFallingObstacle(800));
    }
}

void ObstacleManager::render(sf::RenderWindow& window) {
    for (const auto& obstacle : obstacles) {
        obstacle.render(window);
    }
}
```

변수(멤버,생성자)	함수	설명 및 배운 내용

	Obstacle(sf::Vector2f position, sf::Vector2f size, sf::Color color) 장애물의 초기 위치, 크기, 색상을 설정합니다. 기본 색상은 빨간색(sf::Color::Red)으로 지정되어 있습니다.	void render(sf::RenderWindow& window) 장애물을 지정된 창(window)에 렌더링합니다. void update(float deltaTime) 장애물이 아래로 낙하하도록 위치를 업데이트합니다. 낙하 속도는 fallSpeed로 제어되며, 시간(deltaTime)에 따라 변화합니다. bool collidesWith(const sf::FloatRect& targetBounds) const 대상의 경계(targetBounds)와 충돌했는지 확인합니다. sf::FloatRect getBounds() const 장애물의 현재 경계 정보를 반환합니다. static Obstacle createFallingObstacle(float windowWidth) 랜덤한 가로 위치에서 생성되는 장애물입니다. 화면의 폭(windowWidth)을 기반으로 장애물의 초기 위치를 설정합니다.	ObstacleManager는 다수의 장애물을 관리하는 클래스로, 장애물의 업데이트와 렌더링을 책임지고 있습니다. 게임 플레이 중 장애물의 생성, 제거, 충돌 검사를 진행합니다.

13

13번째 표 부터는 리펙토링 이후의 코드입니다.

<Render, Bound, draw 관련 함수는 한번만 설명했습니다>

```
#ifndef TILE_H
#define TILE_H

#include <SFML/Graphics.hpp>

/*이 클래스는 타일(Tile) 객체를 표현하는 데 사용되며, 주로
게임의 맵을 구성하는 기본적인 요소로 사용될 수 있습니다. 각 타일은 사각형 형태의 그래픽을 가지며,
장애물 여부에 따라 색상이 달라집니다. SFML을 사용해 타일을 그리며,
충돌을 감지할 때 getBounds() 메서드를 사용하여타일의 경계를 알 수 있습니다.
render() 메서드는 타일을 화면에 출력하는 역할을 합니다.*/

class Tile {
public:
    sf::RectangleShape shape;
    bool isObstacle; // 장애물 여부

    Tile(sf::Vector2f position, sf::Vector2f size, bool isObstacle)
        : isObstacle(isObstacle) {
        shape.setSize(size);
        shape.setPosition(position);
        shape.setFillColor(isObstacle ? sf::Color::Red : sf::Color::White);
    }

    sf::FloatRect getBounds() const {
        return shape.getGlobalBounds();
    }

    void render(sf::RenderWindow& window) const {
        window.draw(shape);
    }
};

#endif
```

변수(멤버,생성자)	함수	설명 및 배운 내용
sf::RectangleShape shape: 타입: sf::RectangleShape 기능: 사각형 형태의 그래픽 객체로, 게임의 타일을 나타냅니다. 이 객체는 화면에 그려지는 물리적인 모양을 정의합니다. . bool isObstacle: 타입: bool 기능: 타일이 장애물인지 아닌지를 나타내는 변수입니다. true이면 장애물, false이면 장애물이 아닙니다. 생성자 생성자는 Tile 객체를 초기화하는 역할을 하며, 초기화된 객체는 Tile	sf::FloatRect getBounds() const: 입력: 없음 출력: sf::FloatRect 이 함수는 타일의 경계(shape의 GlobalBounds)를 반환합니다. sf::FloatRect는 사각형의 좌표와 크기를 나타내는 클래스입니다. 이 함수는 주로 타일의 크기나 충돌을 감지하는 데 사용될 수 있습니다. 기능: shape.getGlobalBounds()는 shape의 전체 영역을 sf::FloatRect로 반환합니다. 이 영역은 타일이 차지하는 사각형 범위를 정의합니다. 게임에서 충돌 처리를 할 때	배열, 2차원 배열: Tile 객체는 게임 내에서 그리드 형태로 배치될 가능성이 크므로, Tile 객체들이 2D 배열로 관리될 수 있습니다. 예를 들어, Tile 객체들의 집합을 2D 배열로 관리하면서, 각 타일에 대해 장애물 여부를 확인하고 렌더링하는 작업이 이루어질 수 있습니다. 조건문: Tile 클래스의 생성자 내에서 isObstacle 값을 이용해 setFillColor() 함수에 조건문을 사용하여 색상을 설정합니다. 이 부분은 조건문을 이용해 true일 때는 빨간색, false일 때는 흰색을 설정하는 로직입니다.

	클래스의 멤버로서 shape, isObstacle 값을 가집니다.	유용합니다.	객체지향 프로그래밍 (OOP): Tile 클래스는 객체지향적인 방식으로 설계되었습니다. Tile 클래스는 shape라는 private 멤버 변수와 getBounds(), render()와 같은 public 메서드를 제공합니다. 이러한 구성은 타일 객체를 추상화하고, 게임 내에서 타일을 독립적인 객체로 관리할 수 있게 해줍니다. 이 클래스는 객체지향적인 설계로, 게임의 타일을 독립적으로 관리하며, 장애물과 비장애물 타일을 구분하여 처리할 수 있습니다 .

14

```cpp
#include "Player.h"

// Player 생성자: 모든 속성 초기화
Player::Player(sf::Vector2f position, sf::Vector2f size)
    : health(1000), speed(200.f), velocityY(0.f), gravity(300.f), jumpStrength(-400.f), isJumping(false) {
    shape.setSize(size);
    shape.setPosition(position);
    shape.setFillColor(sf::Color::White);
}


void Player::update(float deltaTime, const Map& map) {
    sf::Vector2f movement(0, 0);
    // 키 입력 처리 (좌우 화살표 로 이동)

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left)) {
        movement.x -= 200 * deltaTime;
    }
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right)) {
        movement.x += 200 * deltaTime;
    }

    // 점프 처리
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space) && !isJumping) {
        velocityY = jumpStrength;
        isJumping = true;
    }

    // 중력 적용 (매 프레임마다 속도에 중력을 더함)
    velocityY += gravity * deltaTime;
    movement.y += velocityY * deltaTime;

    // 이동 처리
    shape.move(movement);

    // 충돌 검사
    const auto& tiles = map.getTiles(); // 타일 벡터 가져오기
    bool onGround = false;
    for (const auto& tile : tiles) {
        if (tile.isObstacle && shape.getGlobalBounds().intersects(tile.getBounds())) {
            // 충돌 발생
            if (movement.y > 0) { // 플레이어가 아래로 이동 중
                shape.setPosition(shape.getPosition().x, tile.getBounds().top - shape.getSize().y);
                velocityY = 0;
                isJumping = false;
                onGround = true;
            }
        }
    }
    if (!onGround) {
        isJumping = true;
    }
```

```cpp
//렌더링
void Player::render(sf::RenderWindow& window) {
    window.draw(shape);
}

//체력 감소
void Player::takeDamage() {
    health = std::max(0, health - 1);
    shape.setFillColor(health > 0 ? sf::Color::Yellow : sf::Color::Black);
}

//파괴 확인
bool Player::isDestroyed() const {
    return health <= 0;
}


//충돌 경계 반환
sf::FloatRect Player::getBounds() const {
    return shape.getGlobalBounds();
}

//색상 변경
void Player::changeColor(const sf::Color& newColor) {
    shape.setFillColor(newColor);  // shape는 Player의 RectangleShape
}

void Player::setColor(sf::Color color) {
    shape.setFillColor(color);
}

void Player::increaseSpeed(float multiplier) {
    speed *= multiplier; // 기존 속도에 곱셈 연산
}

// 점프 메서드: 점프를 시작하는 순간 Y 속도를 설정
void Player::jump() {
    if (!isJumping) {
        velocityY = jumpStrength; // 점프 힘을 위쪽 방향으로 설정
        isJumping = true;
    }
}
```

변수(멤버,생성자)	함수	설명 및 배운 내용

	sf::RectangleShape shape: 타입: sf::RectangleShape 기능: 플레이어를 나타내는 사각형 형태의 그래픽 객체입니다. 이 객체는 게임 화면에 플레이어를 그리는데 사용됩니다. int health: 플레이어의 체력을 나타냅니다. 체력은 0에서 1000까지의 값으로 설정되어 있으며, takeDamage() 메서드를 통해 감소할 수 있습니다. float speed: 타입: float 기능: 플레이어의 이동 속도를 나타냅니다. update() 메서드에서 키 입력에 따라 플레이어의 이동을 제어하는 데 사용됩니다. float velocityY: 타입: float 기능: 플레이어의 Y축 방향 속도(중력과 점프에 의한 변화)를 나타냅니다. float gravity: 타입: float 기능: 중력의 영향을 정의합니다. 매 프레임마다 플레이어의 velocityY에 중력 값이 추가됩니다. float jumpStrength: 타입: float 기능: 점프의 강도를 정의합니다. velocityY에 이 값이 설정되어 점프를 시작할 때 위로 상승하는 힘을 제공합니다. bool isJumping: 타입: bool 기능: 플레이어가 점프 중인지 여부를 나타냅니다. 점프 중이라면 true, 그렇지 않으면 false입니다.	void Player::update(float deltaTime, const Map& map): 입력(Parameters): deltaTime (float): 프레임 간 시간 차이로, 이동 속도나 물리 계산을 프레임 독립적으로 유지하는 데 사용됩니다. map (const Map&): 게임 맵을 나타내는 객체로, 타일을 포함한 맵 정보를 제공합니다. 출력(Output): 없음: 이 함수는 Player 객체의 상태를 업데이트합니다. 기능: 이동 처리: 키 입력에 따라 플레이어의 좌우 이동을 처리합니다. 좌우 화살표 키가 눌리면, 이동 벡터 movement.x가 변경되고, 이 값은 deltaTime을 곱하여 프레임 속도에 영향을 받지 않도록 합니다. 점프 처리: 스페이스바가 눌리면, 플레이어가 점프하도록 velocityY를 jumpStrength로 설정하고, isJumping을 true로 설정합니다. 중력 적용: 매 프레임마다 velocityY에 중력 값을 더하여 플레이어가 내려가도록 합니다. 이동 처리: shape.move(movement)로 플레이어를 이동시킵니다. 충돌 처리: 맵의 타일들과 충돌 검사를 진행합니다. 충돌이 발생하면, shape의 위치를 타일의 상단에 맞추고, Y축 속도를 0으로 설정하여 점프를 멈추고 isJumping을 false로 설정합니다. void Player::takeDamage(): 입력: 없음 출력: 없음 기능: 플레이어가 피해를 입을 때 체력을 감소시킵니다. bool Player::isDestroyed() const: 입력: 없음	조건문: 여러 곳에서 조건문을 사용하여 플레이어의 상태를 업데이트합니다. 예를 들어, 점프 상태, 충돌 여부 등을 처리할 때 사용됩니다. 배열 및 벡터: map.getTiles() 함수에서 타일 벡터를 가져와 플레이어와 타일의 충돌 여부를 검사합니다. 이는 벡터 자료구조를 활용한 반복문으로 타일들을 하나씩 검사하는 방식입니다. Player 클래스는 객체지향 설계로, 플레이어의 상태와 동작을 객체로 표현하고 있습니다.

	생성자 position (sf::Vector2f): 플레이어의 초기 위치를 나타내는 sf::Vector2f 타입의 벡터입니다. size (sf::Vector2f): 플레이어의 크기를 정의하는 sf::Vector2f 타입의 벡터입니다. 출력(Output): 없음: 생성자는 Player 객체의 초기 상태를 설정하는 역할을 합니다. 기능: health를 1000으로 초기화하여 플레이어의 체력을 설정합니다. speed를 200으로 설정하여 플레이어의 기본 이동 속도를 설정합니다. velocityY를 0으로 설정하여 플레이어의 Y축 속도를 초기화합니다. gravity를 300으로 설정하여 중력의 영향을 정의합니다. jumpStrength를 -400으로 설정하여 플레이어가 점프할 때 적용되는 힘을 설정합니다. isJumping을 false로 설정하여 처음에는 점프 중이지 않다고 설정합니다. shape는 size와 position으로 초기화되고, 기본 색상은 흰색(sf::Color::White)으로 설정됩니다.	출력: bool 기능: 플레이어의 체력이 0 이하이면 true를 반환하여 플레이어가 파괴되었는지를 확인합니다. void Player::changeColor(const sf::Color& newColor): 입력: newColor (sf::Color): 플레이어의 색상을 변경할 색상입니다. 출력: 없음 기능: shape 객체의 색상을 newColor로 변경합니다. void Player::setColor(sf::Color color): 입력: color (sf::Color): 변경할 색상입니다. 출력: 없음 기능: shape 객체의 색상을 color로 변경합니다. void Player::increaseSpeed(float multiplier): 입력: multiplier (float): 속도를 증가시킬 배율입니다. 출력: 없음 기능: 플레이어의 speed에 multiplier를 곱하여 이동 속도를 증가시킵니다. void Player::jump(): 입력: 없음 출력: 없음 기능: 플레이어가 점프할 때 velocityY에 점프 강도를 적용하고, isJumping을 true로 설정하여 점프 상태로 전환합니다.	

15

```cpp
#include "ObstacleManager.h"
#include <cstdlib>
#include<random>

float ObstacleManager::spawnCooldown = 0.4f; // 기본값 설정

void ObstacleManager::spawnObstacle(float windowWidth) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::uniform_real_distribution<float> dis(0.f, windowWidth);
    float x = dis(gen);
    obstacles.emplace_back(sf::Vector2f(x, 0.f), sf::Vector2f(20.f, 20.f), sf::Color::Magent
   // 생성자 호출로 Obstacle 생성
}


void ObstacleManager::update(float deltaTime, Player& player) {
    for (auto it = obstacles.begin(); it != obstacles.end();) {
        it->update(deltaTime);

        if (it->collidesWith(player.getBounds())) {
            player.takeDamage(); // 충돌 시 플레이어 피해 처리
            it = obstacles.erase(it);
        }
        else if (it->getBounds().top > 600) { // 화면 아래로 사라지면 제거
            it = obstacles.erase(it);
        }
        else {
            ++it;
        }
    }

    // 장애물 생성 로직
    spawnCooldown -= deltaTime;
    if (spawnCooldown <= 0.f) {
        spawnObstacle(800.f);
        spawnCooldown = 1.f; // 1초 쿨다운
    }
}




void ObstacleManager::render(sf::RenderWindow& window) {
    for (const auto& obstacle : obstacles) {
        obstacle.render(window);
    }
}

std::vector<Obstacle>& ObstacleManager::getObstacles() {
    return obstacles;
}
```

변수(멤버,생성자)	함수	설명 및 배운 내용

	static float spawnCooldown: 타입: float 기능: 장애물이 생성되는 쿨다운 시간입니다. 초기값은 0.4초로 설정되어 있지만, update() 함수에서 1초로 재설정됩니다. 장애물을 생성하는 주기를 제어하는 변수입니다. 정적 변수: static 키워드가 붙어 있어 클래스의 모든 인스턴스에서 공통적으로 사용되는 변수입니다. std::vector<Obstacle> obstacles: 타입: std::vector<Obstacle> 기능: 장애물 객체들을 저장하는 동적 배열입니다. Obstacle 객체들이 이 벡터에 추가되며, 각 장애물은 update()와 render() 함수에서 처리됩니다.	void ObstacleManager::spawnObstacle(float windowWidth): 입력(Parameters): windowWidth (float): 게임 창의 가로 크기입니다. 이 값을 사용해 장애물이 화면의 가로 범위 내에서 무작위로 생성됩니다. 출력(Output): 없음: 장애물을 생성하고 벡터에 추가하는 역할을 합니다. 기능: std::random_device와 std::mt19937를 사용해 랜덤 수를 생성합니다. std::uniform_real_distribution<float> dis(0.f, windowWidth)를 사용해 0부터 windowWidth까지의 값 중에서 임의의 x 좌표를 생성합니다. obstacles.emplace_back()으로 새 장애물을 생성하여 obstacles 벡터에 추가합니다. 장애물의 크기, 색상, 초기 위치는 sf::Vector2f(x, 0.f)와 같이 설정됩니다. void ObstacleManager::update(float deltaTime, Player& player): 입력(Parameters): deltaTime (float): 프레임 간 시간 차이로, 시간에 의존하는 움직임을 프레임 독립적으로 처리하기 위한 값입니다. player (Player&): 플레이어 객체로, 장애물과의 충돌을 처리하기 위해 사용됩니다. 출력(Output): 없음: 장애물들을 업데이트하고, 플레이어와의 충돌을 처리합니다. 기능: obstacles.begin()부터 obstacles.end()까지 반복하면서 각	std::vector<Obstacle>: 장애물들은 동적 배열인 std::vector에 저장됩니다. std::vector는 크기가 동적으로 조정되는 배열로, 장애물이 추가되거나 제거될 때 유용하게 사용됩니다. 이 자료구조는 빠른 삽입과 삭제에 효율적입니다. std::random_device, std::mt19937, std::uniform_real_distribution: 이들은 C++의 난수 생성기 관련 라이브러리입니다. std::random_device는 시드를 생성하고, std::mt19937은 난수 생성기의 엔진이며, std::uniform_real_distribution은 지정된 범위 내에서 균등 분포로 실수 난수를 생성합니다. 해당 클래스는 게임의 중요한 요소 중 하나인 장애물을 관리합니다. 장애물들은 플레이어와 충돌하여 피해를 주고, 화면 아래로 사라지거나 제거됩니다. 이는 게임의 진행을 어렵게 만드는 요소로 작용할 수 있습니다.

		장애물의 update(deltaTime)을 호출하여 장애물을 이동시킵니다. 각 장애물이 플레이어와 충돌했는지 확인하기 위해 collidesWith(player.getBounds())를 호출합니다. 충돌 시 플레이어는 피해를 입고(player.takeDamage()), 충돌한 장애물은 제거됩니다. 장애물이 화면 아래로 벗어나면 getBounds().top > 600 조건에 의해 해당 장애물이 제거됩니다. spawnCooldown을 감소시키고, 쿨다운이 0 이하일 경우 spawnObstacle(800.f)를 호출하여 새로운 장애물을 생성합니다. 이후 쿨다운 시간을 1초로 리셋합니다. std::vector<Obstacle>& ObstacleManager::getObstacles(): 입력(Parameters): 없음 출력(Output): std::vector<Obstacle>&: 장애물들의 벡터를 반환합니다. 기능: 장애물 목록을 반환하는 메서드로, 다른 객체가 장애물들을 참조할 수 있게 합니다.	

16

```cpp
#include "Obstacle.h"

Obstacle::Obstacle(sf::Vector2f position, sf::Vector2f size, sf::Color color, int damage)
    : damage(damage) {
    shape.setSize(size);
    shape.setPosition(position);
    shape.setFillColor(color);
}

void Obstacle::render(sf::RenderWindow& window) const{
    window.draw(shape);
}

void Obstacle::update(float deltaTime) {
    shape.move(0, fallSpeed * deltaTime); // 아래로 낙하
}


sf::FloatRect Obstacle::getBounds() const {
    return shape.getGlobalBounds();
}


bool Obstacle::collidesWith(const sf::FloatRect& targetBounds) const {
    return shape.getGlobalBounds().intersects(targetBounds);
}

Obstacle Obstacle::createFallingObstacle(float windowWidth) {
    float x = static_cast<float>(std::rand() % static_cast<int>(windowWidth));
    return Obstacle({ x, 0 }, { 20, 20 }, sf::Color::Red, 10);
}
```

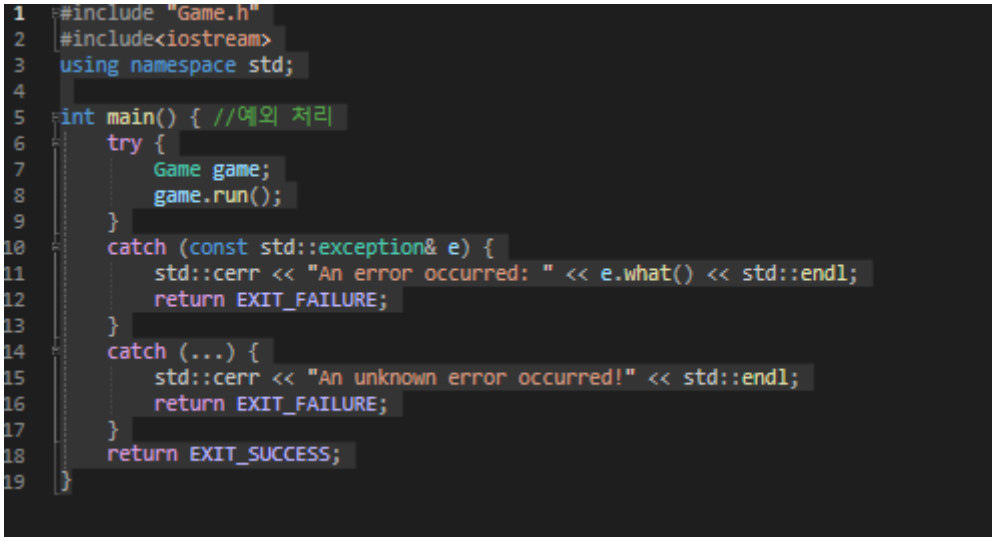
변수(멤버,생성자)	함수	설명 및 배운 내용
sf::RectangleShape shape: 타입: sf::RectangleShape 기능: Obstacle 객체의 시각적 표현을 담당하는 사각형 모양의 도형입니다. 게임에서 장애물은 이 shape 객체로 나타내어집니다. int damage: 타입: int 기능: 장애물이 플레이어와 충돌할 때 플레이어에게 가하는 피해의 크기입니다. 이 값은 객체 생성 시 설정됩니다. float fallSpeed: 타입: float 기능: 장애물이 떨어지는 속도입니다.	void Obstacle::update(float deltaTime): 입력(Parameters): deltaTime (float): 게임 루프에서 프레임 간 경과 시간을 나타냅니다. 이 값은 낙하 속도를 조정하는 데 사용됩니다. 출력(Output): 없음: 장애물이 화면에서 아래로 떨어지게 합니다. 기능: shape.move(0, fallSpeed * deltaTime)를 호출하여 장애물이 아래로 이동합니다. fallSpeed 값에 따라 장애물의 낙하 속도가 결정됩니다. deltaTime을 곱하여 프레임 독립적인 움직임을	Obstacle 클래스는 장애물 객체의 속성과 동작을 모두 포함하고 있어 객체지향적으로 설계된 클래스입니다.

	Obstacle::Obstacle(sf::Vector2f position, sf::Vector2f size, sf::Color color, int damage): 입력(Parameters): position (sf::Vector2f): 장애물의 위치입니다. 생성 시 좌표를 지정합니다. size (sf::Vector2f): 장애물의 크기입니다. 크기도 생성 시 설정됩니다. color (sf::Color): 장애물의 색상입니다. 색상은 sf::Color 객체로 지정됩니다. damage (int): 장애물이 플레이어와 충돌할 때 주는 피해의 양입니다. 출력(Output): 없음: 객체를 초기화하는 생성자입니다. .	구현합니다. Obstacle Obstacle::createFallingObstacle(float windowWidth): 입력(Parameters): windowWidth (float): 게임 화면의 너비로, 장애물의 생성 위치를 결정하는 데 사용됩니다. 출력(Output): Obstacle: 새로 생성된 낙하 장애물 객체를 반환합니다. 기능: rand()를 사용해 0에서 windowWidth 사이의 임의의 x 좌표를 생성합니다. 새 장애물을 Obstacle 클래스의 생성자를 호출하여 생성하고 반환합니다.	
17			

```cpp
#include "Map.h"
#include <cstdlib>
#include <ctime>

Map::Map() {
    tileSize = { 40.f, 40.f };  // 기본 타일 크기 설정
}

void Map::initialize(int width, int height) {
    widthCount = width;
    heightCount = height;
    std::srand(static_cast<unsigned>(std::time(nullptr)));
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            bool isObstacle = (std::rand() % 4 != 0); // 75% 확률로 장애물 생성
            tiles.emplace_back(sf::Vector2f(x * tileSize.x, y * tileSize.y), tileSize, isObstacle);
        }
    }
}

void Map::render(sf::RenderWindow& window) {
    for (const auto& tile : tiles) {
        if (tile.isObstacle) {
            tile.render(window);
        }
    }
}

void Map::update(const sf::FloatRect& playerBounds) {
    for (const auto& tile : tiles) {
        if (tile.isObstacle && tile.getBounds().intersects(playerBounds)) {
            // 충돌 처리 로직
        }
    }
}

sf::FloatRect Map::getBounds() const {
    if (tiles.empty()) {
        return sf::FloatRect(0, 0, 0, 0);
    }

    // 맵의 크기를 계산
    float width = tileSize.x * static_cast<float>(widthCount);
    float height = tileSize.y * static_cast<float>(heightCount);
    return sf::FloatRect(0, 0, width, height);
}

    // 플레이어와 맵 타일 간의 충돌 확인
bool Map::checkCollision(const sf::FloatRect& playerBounds) const {
    for (const auto& tile : tiles) {
        if (tile.isObstacle && tile.getBounds().intersects(playerBounds)) {
            return true;
        }
    }
    return false;
};

const std::vector<Tile>& Map::getTiles() const {
    return tiles;
}
```

변수(멤버,생성자)	함수	설명 및 배운 내용

	f::Vector2f tileSize: 타입: sf::Vector2f 기능: 각 타일의 크기를 나타냅니다. 기본값은 {40.f, 40.f}로 설정됩니다. 즉, 타일의 너비와 높이가 40픽셀입니다. int widthCount 및 int heightCount: 타입: int 기능: 맵의 너비와 높이를 타일 단위로 나타냅니다. 이 값들은 맵의 크기를 결정하는 데 사용됩니다. std::vector<Tile> tiles: 타입: std::vector<Tile> 기능: 맵의 모든 타일을 저장하는 벡터입니다. 각 타일은 Tile 객체로, 타일의 위치, 크기, 장애물 여부 등을 포함합니다. Map::Map(): 기능: 생성자입니다. 기본적으로 tileSize를 {40.f, 40.f}로 설정합니다.	oid Map::initialize(int width, int height): 입력(Parameters): width (int): 맵의 너비(타일의 수). height (int): 맵의 높이(타일의 수). 출력(Output): 없음: 맵을 초기화하여 tiles 벡터에 타일들을 추가합니다. 기능: 이 메서드는 width와 height를 바탕으로 맵의 크기를 설정하고, 각 타일을 생성합니다. 각 타일은 위치와 크기, 장애물 여부를 설정하며, 장애물은 약 75% 확률로 생성됩니다 (std::rand() % 4 != 0). void Map::update(const sf::FloatRect& playerBounds): 입력(Parameters): playerBounds (const sf::FloatRect&): 플레이어의 경계입니다. 충돌을 검사하는 데 사용됩니다. 출력(Output): 없음: 충돌 처리 로직이 추가될 수 있습니다. 현재는 실제 충돌 처리는 없지만, 이 메서드는 플레이어와 맵 타일의 충돌을 확인할 수 있는 곳입니다. 기능: 각 타일에 대해 플레이어의 경계(playerBounds)와 충돌하는지 확인합니다. 충돌이 발생하면 처리 로직을 실행할 수 있습니다(현재는 비어있음). . bool Map::checkCollision(const sf::FloatRect& playerBounds) const: 입력(Parameters): playerBounds (const sf::FloatRect&): 플레이어의 경계입니다. 출력(Output): bool: 충돌 여부를 반환합니다.	Map 클래스는 게임에서 맵을 구성하고, 타일들을 관리하는 중요한 역할을 합니다. 각 타일에 장애물을 설정하고, 맵의 크기와 충돌 여부를 확인하는 기능을 통해 게임의 전반적인 맵 관리가 이루어집니다. 개선할 점으로는 충돌 처리와 장애물 생성 로직을 더욱 세밀하게 다듬을 수 있습니다.

		기능: tiles 벡터를 순회하면서 각 타일이 장애물인지 확인하고, 플레이어의 경계와 충돌하는지 검사합니다. 충돌하면 true를 반환하고, 충돌하지 않으면 false를 반환합니다. const std::vector<Tile>& Map::getTiles() const: 입력(Parameters): 없음 출력(Output): const std::vector<Tile>&: tiles 벡터의 참조를 반환합니다. 기능: tiles 벡터의 참조를 반환하여 다른 객체가 타일에 접근할 수 있도록 합니다. 이 메서드는 타일을 조회하거나 처리하는 데 사용됩니다.	
18	 		
	변수(멤버,생성자)	함수	설명 및 배운 내용

		try-catch 블록 try: 예외가 발생할 수 있는 코드를 try 블록 안에 넣습니다. 이 코드에서는 Game 객체를 생성하고 game.run()을 호출하여 게임을 실행합니다. catch (const std::exception& e): std::exception으로 파생된 예외를 처리합니다. 게임 실행 중 예외가 발생하면 e.what()을 사용하여 예외의 설명을 출력합니다. catch (...): 이 블록은 예기치 않은 예외를 처리합니다. 일반적으로 std::exception에 해당되지 않는 예외들이 발생할 경우 이 블록이 실행됩니다.	

19

```cpp
#include "ItemManager.h"
#include <cstdlib>
#include <ctime>
#include <random>
#include<iostream>

void ItemManager::spawnItems(sf::FloatRect mapBounds, int itemCount) {
    std::random_device rd;
    std::mt19937 gen(rd());
    // 최소값과 최대값이 유효한지 확인
    float maxX = std::max(0.f, mapBounds.width - 20.f);
    float maxY = std::max(0.f, mapBounds.height - 20.f);

    std::uniform_real_distribution<float> disX(0, maxX);
    std::uniform_real_distribution<float> disY(0, maxY);

    for (int i = 0; i < itemCount; ++i) {
        Type type = (i % 2 == 0) ? Type::SpeedBoost : Type::ColorChange;
        int value = (type == Type::SpeedBoost) ? 2 : 0;
        items.emplace_back(sf::Vector2f(disX(gen), disY(gen)), type, value);
    }
}

void ItemManager::update(Player& player) {
    for (auto it = items.begin(); it != items.end();) {
        if (player.getBounds().intersects(it->shape.getGlobalBounds())) {
            it->applyEffect(player);
            it = items.erase(it);
        }
        else {
            ++it;
        }
    }
}

void ItemManager::render(sf::RenderWindow& window) const {
    for (const auto& item : items) {
        item.render(window);
    }
}

const std::vector<Item>& ItemManager::getItems() const {
        return items;
    }
```

변수(멤버,생성자)	함수	설명 및 배운 내용
	spawnItems() 목적: 주어진 맵 영역 내에서 아이템을 생성합니다. 입력: int itemCount: 생성할 아이템의 개수. 출력: 없음. items 벡터에 아이템을 추가. 로직: std::random_device와 std::mt19937을 사용하여 난수를	ItemManager 클래스는 게임 내 아이템을 관리하는 역할을 합니다. 주된 기능은 아이템을 생성하고, 업데이트하며, 렌더링하는 것입니다. 이 클래스는 아이템을 생성하고, 플레이어와의 충돌을 감지하여 효과를 적용하며, 화면에 아이템을 그리는 등의 작업을 합니다.

		생성합니다. mapBounds에 맞는 범위 내에서 아이템의 위치를 랜덤하게 생성합니다. 아이템의 타입은 SpeedBoost와 ColorChange 중 하나로 결정됩니다. (짝수 번째 아이템은 SpeedBoost, 홀수 번째는 ColorChange 타입) 생성된 아이템은 items 벡터에 추가됩니다. 2. update() 목적: 플레이어와 아이템이 충돌하는지 확인하고, 충돌 시 아이템의 효과를 적용한 후 아이템을 삭제합니다. 입력: Player& player: 플레이어 객체. 출력: 없음. 로직: items 벡터의 각 아이템에 대해 player.getBounds()와 it->shape.getGlobalBounds()를 비교하여 충돌을 확인합니다. 충돌 시 applyEffect()를 호출하여 아이템의 효과를 플레이어에게 적용하고, 해당 아이템을 items 벡터에서 삭제합니다. 아이템이 삭제되지 않으면, 다음 아이템으로 넘어갑니다. 3. getItems() 목적: items 벡터에 저장된 아이템들을 반환합니다. 입력: 없음. 출력: const std::vector<Item>&: 아이템 목록을 반환합니다. 로직: items 벡터를 반환합니다. 아이템 생성에 대한 설명 아이템은 주어진 맵 영역 내에서 랜덤 위치에 생성됩니다. 각 아이템은 SpeedBoost 또는 ColorChange 중 하나의 타입을 가집니다. SpeedBoost	

		아이템은 플레이어의 속도를 증가시키고, ColorChange 아이템은 플레이어의 색상을 변경하는 등의 효과를 가질 것으로 예상됩니다.	
20			

```cpp
#include "GameManager.h"
#include<iostream>

GameManager& GameManager::getInstance() {
    static GameManager instance;
    return instance;
}

void GameManager::initialize() {
    window.create(sf::VideoMode(800, 600), "Game with Features");
    Player player({ 400.f, 300.f }, { 50.f, 50.f }); // 초기 위치와 크기를 바로 설정;
    map.initialize(20, 15); // 맵 초기화
    itemManager.spawnItems(map.getBounds()); // 아이템 생성
    // 기타 초기화 코드
}



GameManager::GameManager()
    :window(sf::VideoMode(800, 600), "Game"),
    currentState(GameState::ColorSelection),
    isGameOver(false),
    player({ 400, 300 }, { 20, 20 }),
    map() {
    setupStateHandlers();
    itemManager.spawnItems(map.getBounds());
}

void GameManager::setupStateHandlers() {
    updateState = [this](float deltaTime) {
        switch (currentState) {
        case GameState::ColorSelection:
            updateColorSelection(deltaTime);
            break;
        case GameState::MainGame:
            updateMainGame(deltaTime);
            break;
        case GameState::GameOver:
            updateGameOver(deltaTime);
            break;
        }
    };

    renderState = [this]() {
        switch (currentState) {
        case GameState::ColorSelection:
            renderColorSelection();
            break;
        case GameState::MainGame:
            renderMainGame();
            break;
        case GameState::GameOver:
            renderGameOver();
            break;
        }
    };
}
```

```
void GameManager::run() {
    sf::Clock clock;

    while (window.isOpen()) {
        float deltaTime = clock.restart().asSeconds();
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                window.close();
            }
        }
        if (!isGameOver) {
            updateState(deltaTime);
        }
        renderState();

    }
}

void GameManager::updateColorSelection(float deltaTime) {
    colorSelectionScene.handleInput(window);
    if (colorSelectionScene.isColorSelected()) {
        player.setColor(colorSelectionScene.getSelectedColor());
        currentState = GameState::MainGame;
    }
}

void GameManager::updateMainGame(float deltaTime) {
    player.update(deltaTime, map);
    obstacleManager.update(deltaTime, player);
    itemManager.update(player);
    if (player.isDestroyed()) {
        currentState = GameState::GameOver;
    }
}
```

```cpp
void GameManager::updateGameOver(float deltaTime) {
    // 게임 오버 상태의 추가 처리 로직
}

void GameManager::renderColorSelection() {
    window.clear();
    colorSelectionScene.render(window);
    window.display();
}

void GameManager::renderMainGame() {
    window.clear();
    map.render(window);
    player.render(window);
    obstacleManager.render(window);
    itemManager.render(window);
    window.display();
}

void GameManager::renderGameOver() {
    window.clear();
    sf::Font font;
    font.loadFromFile("arial.ttf");
    sf::Text gameOverText("Game Over", font, 50);
    gameOverText.setPosition(200, 250);
    gameOverText.setFillColor(sf::Color::Red);
    window.draw(gameOverText);
    window.display();
}
```

변수(멤버,생성자)	함수	설명 및 배운 내용
생성자:게임 시작 시 필요한 초기화 작업을 수행합니다. 로직: window 객체를 생성하여 화면 크기와 제목을 설정합니다. currentState는 초기 상태를 ColorSelection으로 설정합니다. player와 map을 초기화합니다. itemManager.spawnItems(map.getBounds())로 맵에 아이템을 배치합니다. setupStateHandlers() 메서드를 호출하여 상태별 업데이트 및 렌더링 핸들러를 설정합니다.	getInstance(): GameManager 인스턴스를 하나만 생성하고 전역에서 접근할 수 있도록 하는 싱글턴 패턴을 구현합니다. 로직: static GameManager instance를 통해 GameManager의 인스턴스를 한 번만 생성하며, 이 인스턴스를 반환합니다. 상태 핸들러 함수들: updateState와 renderState는 람다 표현식을 사용하여 상태에 따른 업데이트 및 렌더링 함수를 할당합니다. currentState 값에 따라 해당 상태에 맞는 함수가 호출됩니다. run():목적: 게임의 메인 루프를 실행하여 게임을 지속적으로	GameManager 클래스는 게임의 전반적인 흐름을 제어하는 주요 클래스입니다. 이 클래스는 게임 상태에 따라 상태를 전환하고, 각 상태에 따라 적절한 업데이트 및 렌더링을 담당합니다. 또한, 싱글턴 패턴을 적용하여 전역에서 하나의 인스턴스를 공유합니다.

		갱신합니다. 로직: sf::Clock을 사용하여 deltaTime을 계산하고, 이를 각 상태 업데이트 함수에 전달합니다. window.pollEvent()로 발생한 이벤트를 처리하고, sf::Event::Closed 이벤트가 발생하면 창을 종료합니다. 게임이 종료되지 않았다면 updateState(deltaTime)로 상태에 맞는 업데이트를 수행하고, renderState()로 상태에 맞는 화면을 렌더링합니다. render 함수들 : 각각 다른 상태의 게임 화면을 렌더링	

21

```cpp
#include "Game.h"


Game::Game()
    : gameManager(GameManager::getInstance()) {
    // GameManager 초기화
    gameManager.initialize();
}

void Game::run() {
    // GameManager 실행
    gameManager.run();
}
```

변수(멤버,생성자)	함수	설명 및 배운 내용

		GameManager::getInstance()를 호출하여 GameManager의 인스턴스를 가져옵니다. gameManager.initialize()를 호출하여 GameManager 내부에서 필요한 초기화 작업을 수행합니다.	Game 객체가 생성될 때, GameManager의 싱글턴 인스턴스를 가져와 초기화합니다. Game 클래스는 게임의 핵심 로직을 GameManager에 위임합니다. Game 클래스는 GameManager 인스턴스를 생성하고 초기화한 뒤, 단순히 run() 메서드를 호출하여 게임을 실행하는 구조입니다. 이 클래스는 GameManager의 세부 적인 업데이트 및 렌더링 로직을 직접 처리하지 않고, 대신 GameManager에 책임을 맡깁니다. 이 구조는 GameManager를 중심으로 한 게임 흐름을 효과적으로 관리하고, Game 클래스는 초기화 및 게임 루프 실행에 집중함으로써 클래스 간 역할을 명확히 구분하는 데 도움이 됩니다..

22

```cpp
#include "Item.h"
#include <cstdlib> // for rand()

using namespace sf;


Item::Item(sf::Vector2f position, Type type, int value)
    : itemType(type), value(value) {
    shape.setRadius(10.f);
    shape.setFillColor(type == Type::SpeedBoost ? sf::Color::Cyan : sf::Color::Yellow);
    shape.setPosition(position);
}

// 플레이어 효과 적용
void Item::applyEffect(Player& player) {
    // 플레이어 속도를 1.5배 가속
    if (itemType == Type::SpeedBoost) {
        player.increaseSpeed(value);
    }
    // 색상 변경
    else if (itemType == Type::ColorChange) {
        player.changeColor(sf::Color::Green);
    }

}

// 아이템 렌더링
void Item::render(sf::RenderWindow& window) const {
    window.draw(shape);
}

sf::FloatRect Item::getBounds() const {
    return shape.getGlobalBounds();
}

void Item::setPosition(float x, float y) {
    shape.setPosition(x, y);
}
```

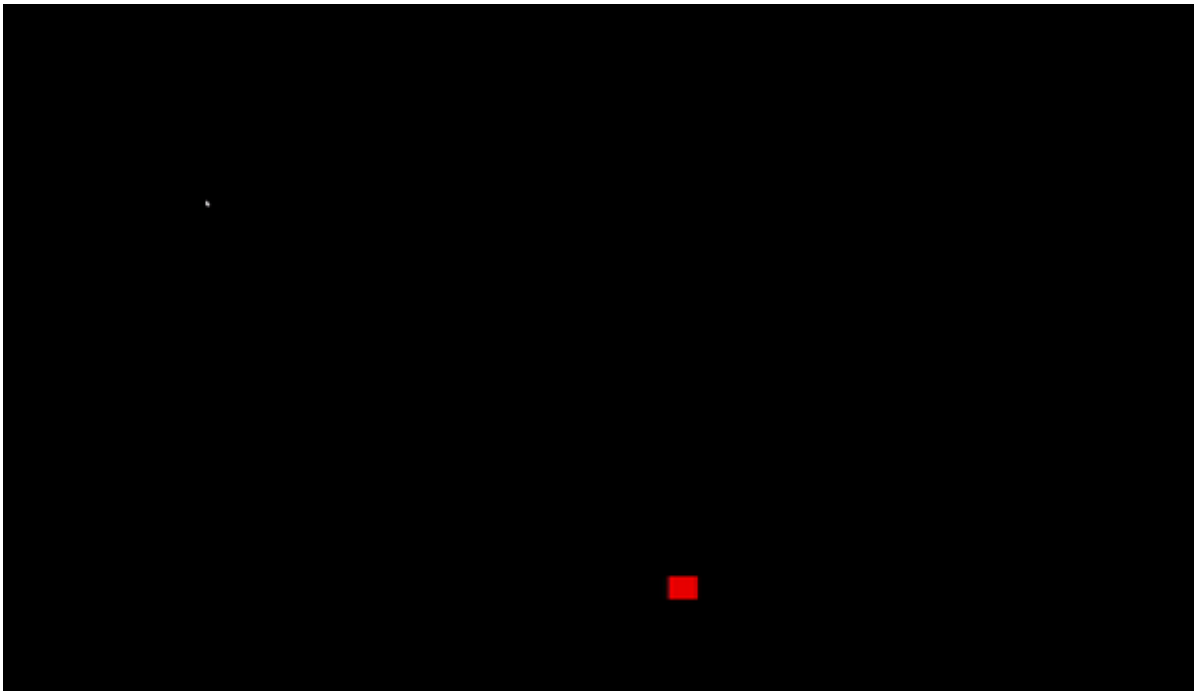
변수(멤버,생성자)	함수	설명 및 배운 내용
생성자 입력: sf::Vector2f position: 아이템의 위치. Type type: 아이템의 타입 (예: SpeedBoost 또는 ColorChange). int value: 아이템의 값, 속도 증가와 같은 효과를 제어하는 값. 로직: 아이템의 타입에 따라 shape의 색상을 설정합니다 (SpeedBoost는 Cyan, ColorChange는 Yellow). shape는 아이템을 표현하는 원 모양(sf::CircleShape)으로, setRadius(10.f)로 크기를 설정하고 setPosition()을 사용하여 위치를 설정합니다.	applyEffect : 목적: 플레이어에게 아이템의 효과를 적용합니다. 입력: Player& player: 아이템 효과를 적용할 플레이어 객체. 로직: itemType이 SpeedBoost인 경우, player.increaseSpeed(value)로 플레이어의 속도를 증가시킵니다. itemType이 ColorChange인 경우, player.changeColor(sf::Color::Green)으로 플레이어의 색상을 변경합니다. 확장 가능성: itemType에 따라 더 많은 효과를 추가할 수 있습니다. 예를 들어, ColorChange가 다양한 색으로 변경될 수 있도록 하고,	Item 클래스는 게임 내에서 아이템을 정의하고, 아이템의 효과를 플레이어에게 적용하며, 렌더링하는 기능을 제공합니다. 아이템은 두 가지 타입(SpeedBoost, ColorChange)으로 분류되어 플레이어에게 효과를 미칩니다.

		SpeedBoost에 특정 시간 제한을 둘 수 있습니다..	

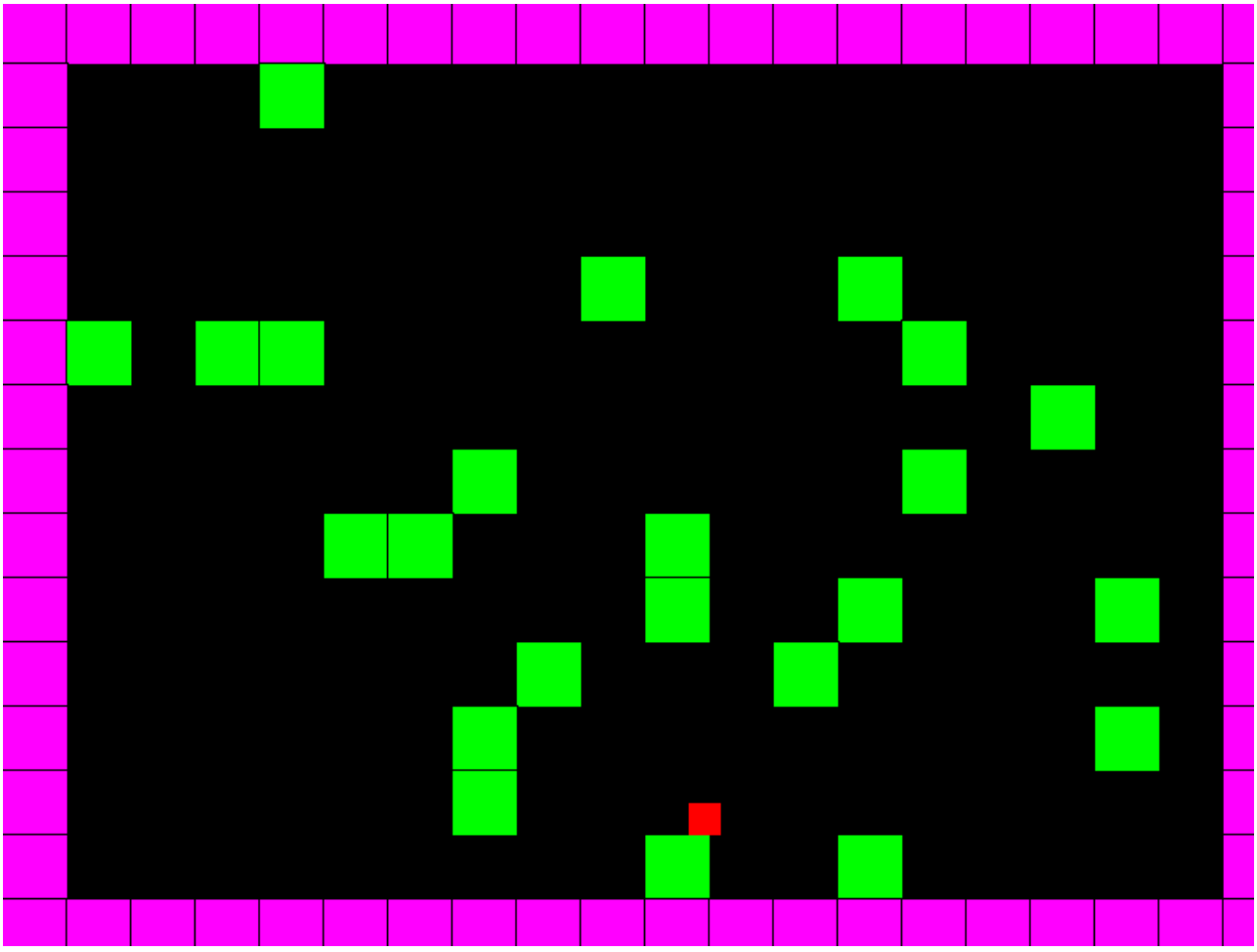
9. 테스트 결과

(1) 플레이어 픽셀 구현

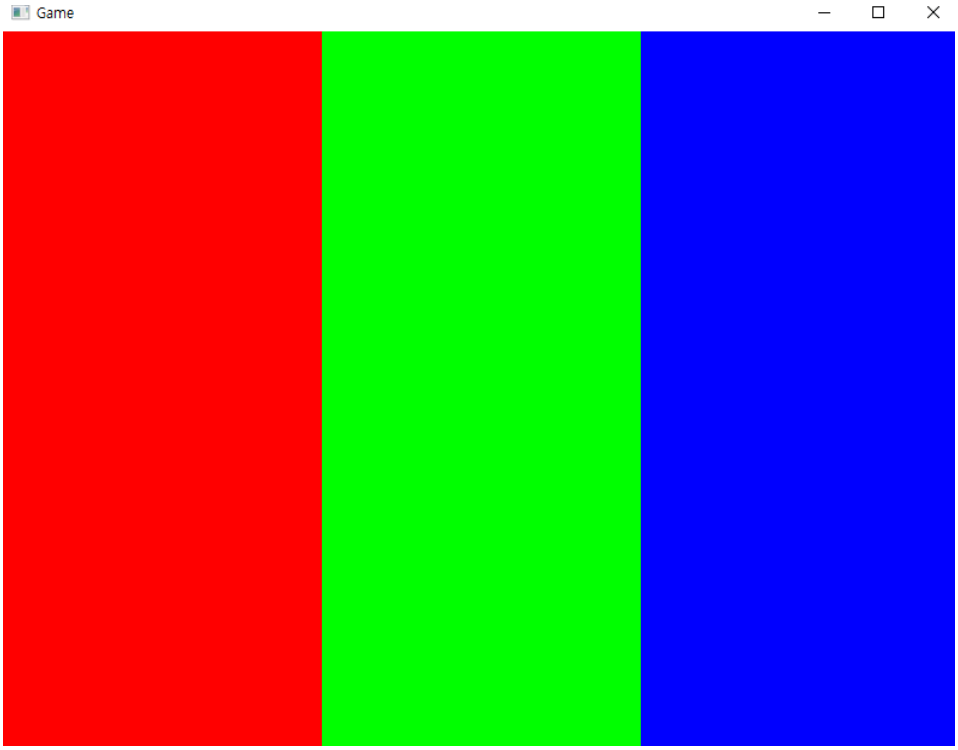
(2) 플레이어 좌우 이동, 점프, 이동제한 구현-



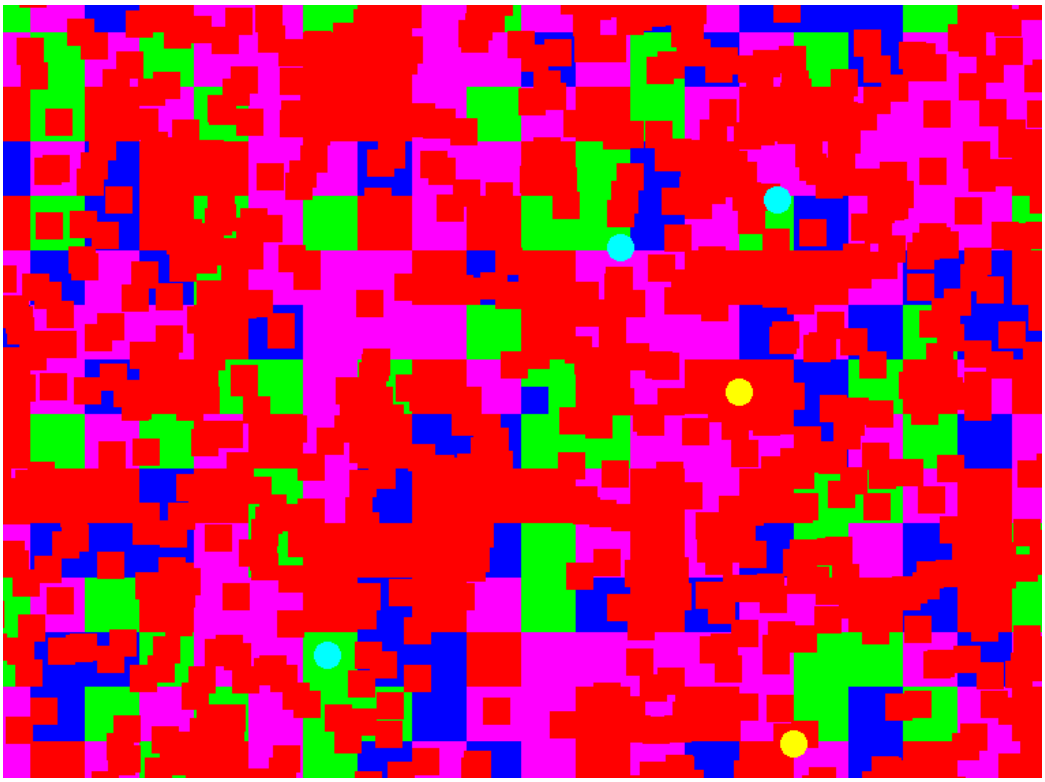
(3) 맵 생성 구현



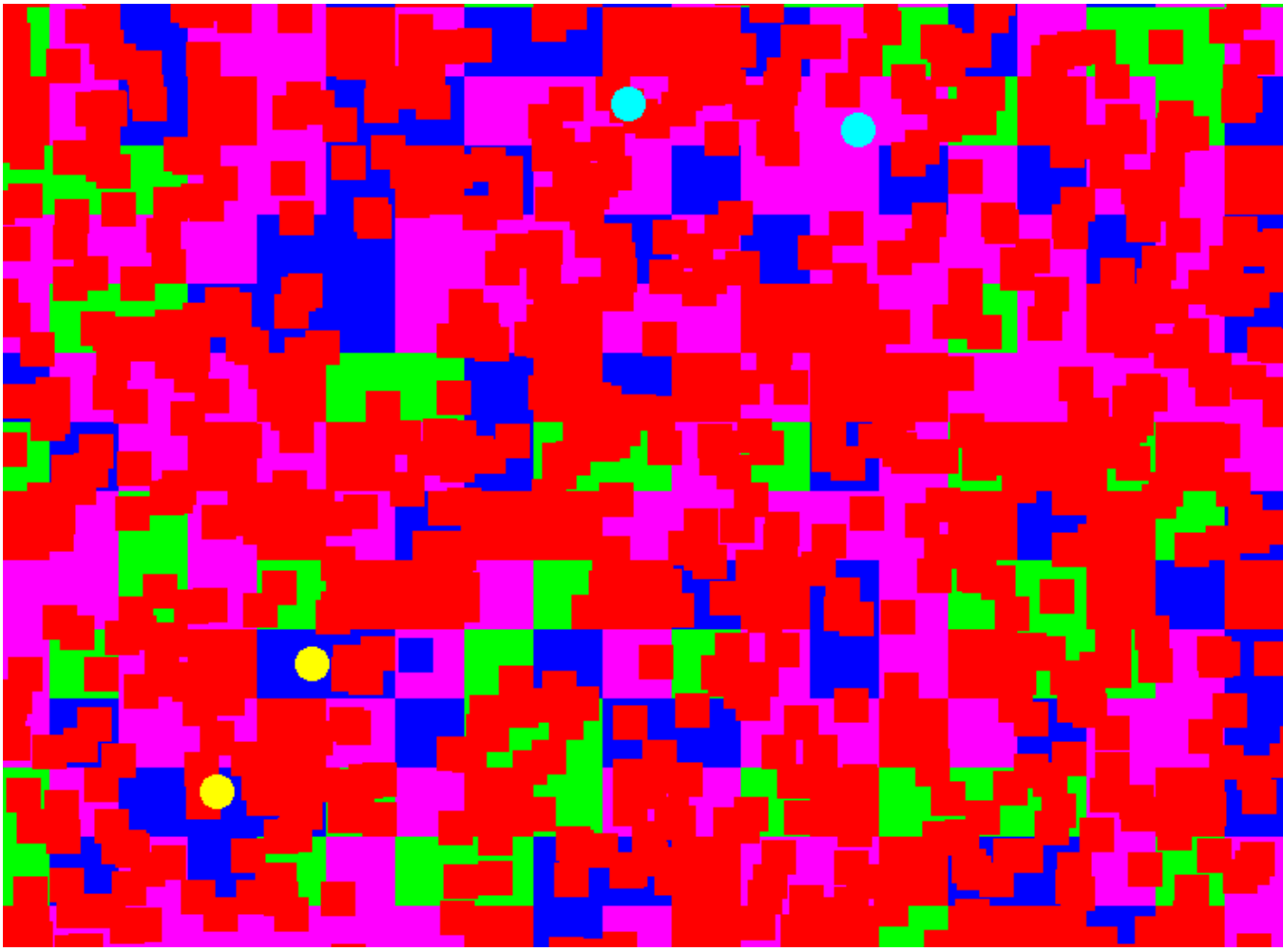
(4)플레이어 색상 선택창

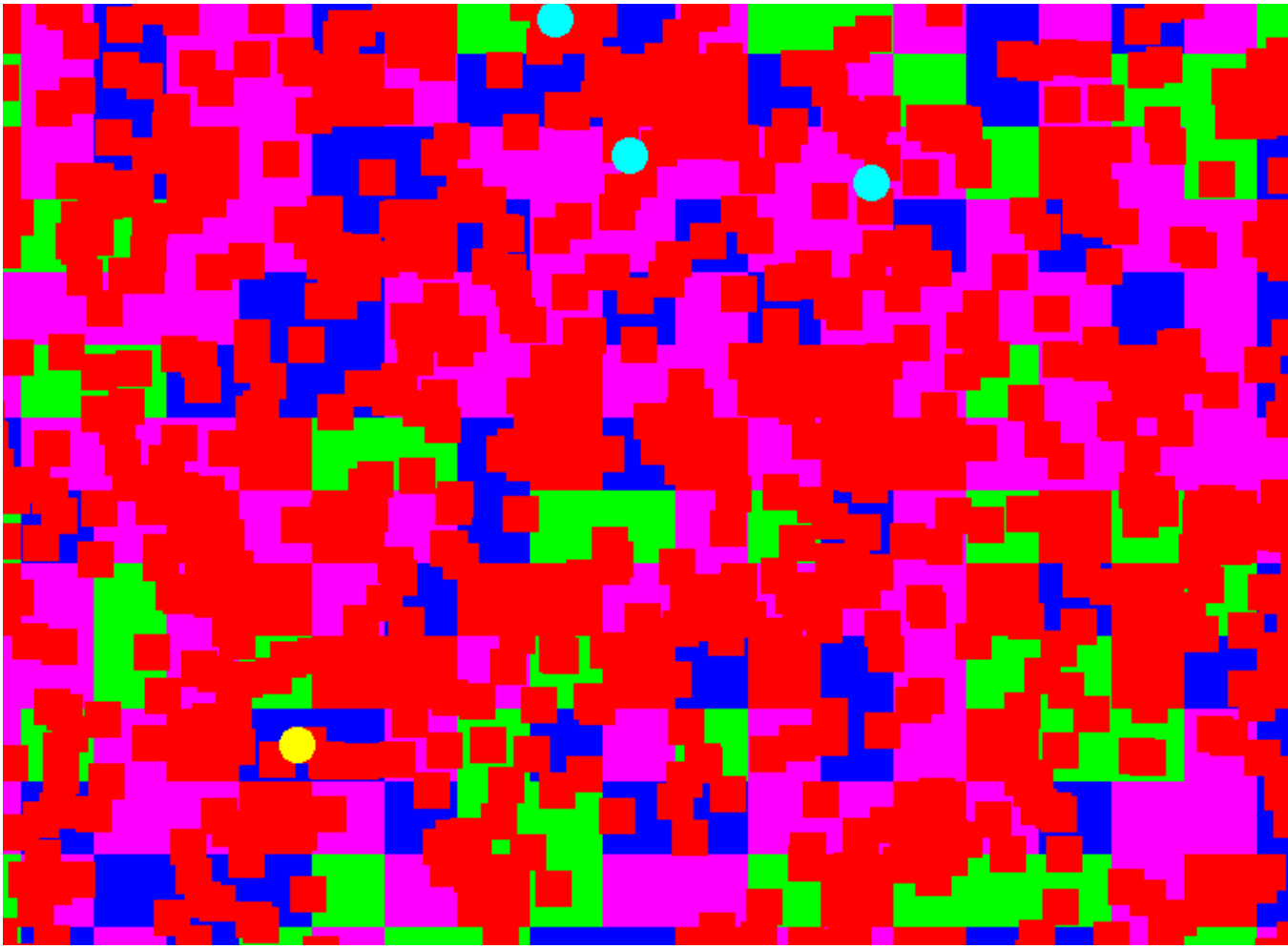


(5) 장애물 및 아이템 스폰 구현(장애물: 붉은색 사각형, 아이템: 원형)

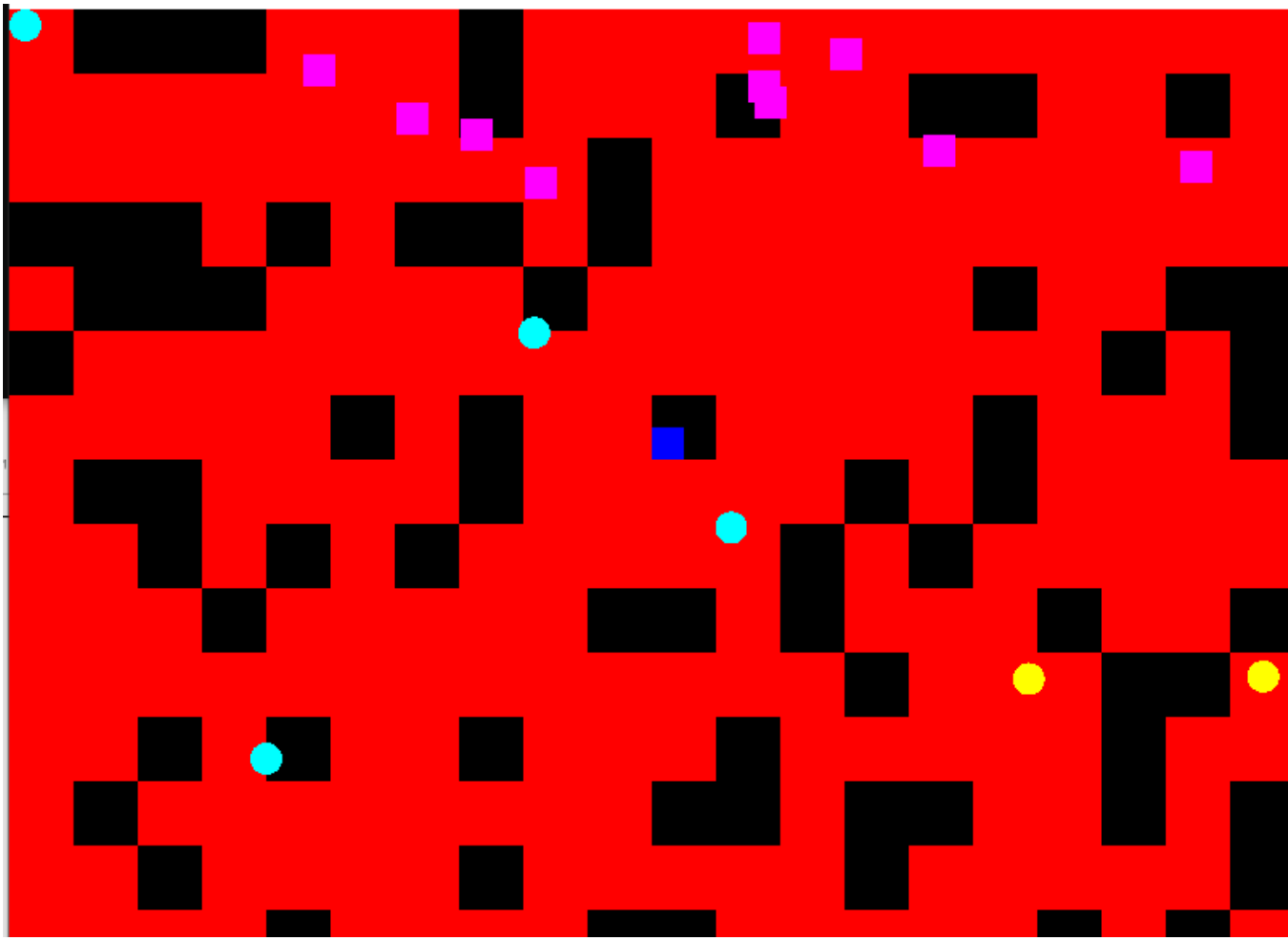


(6)아이템 및 장애물 충돌 구현

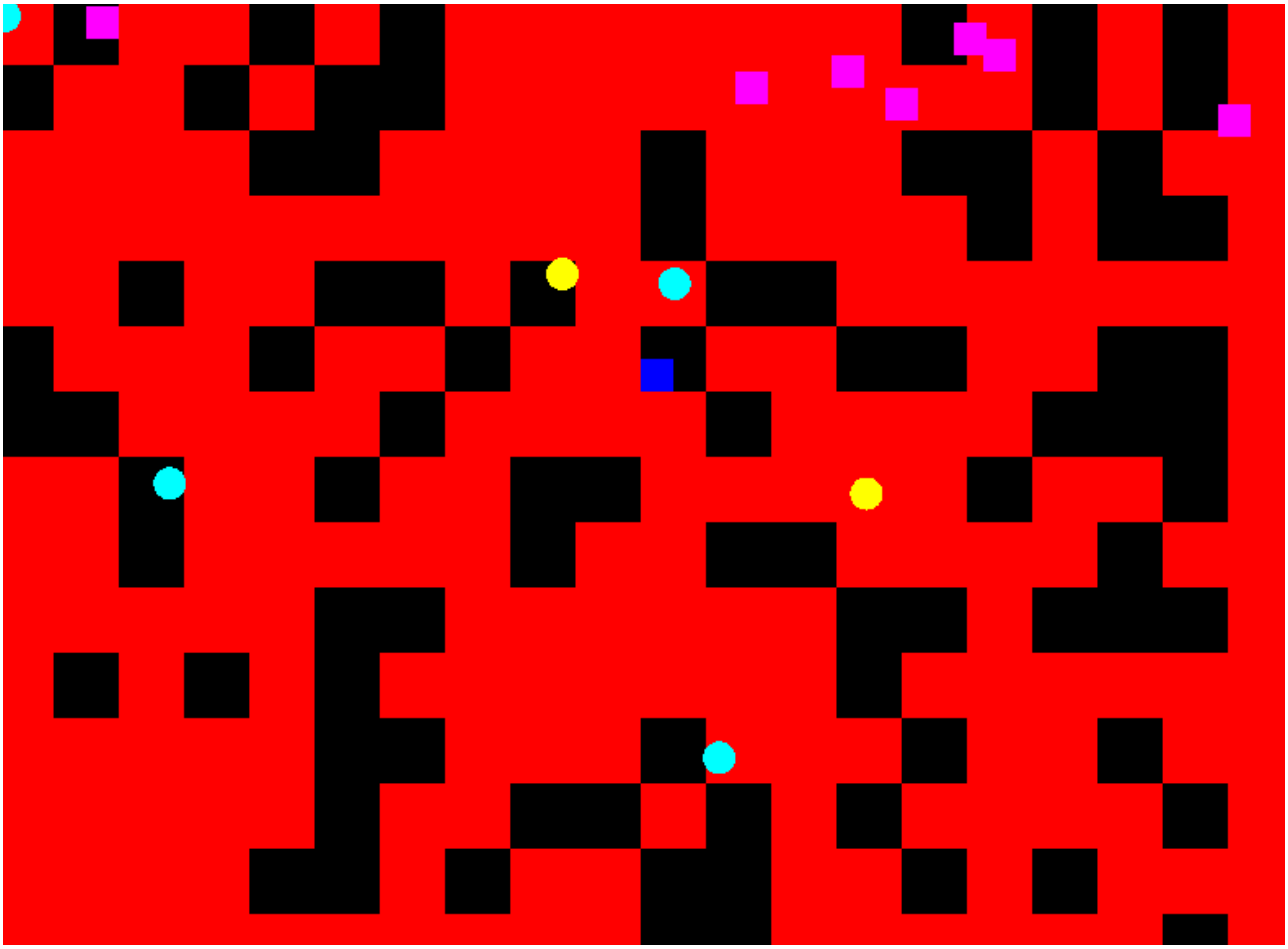


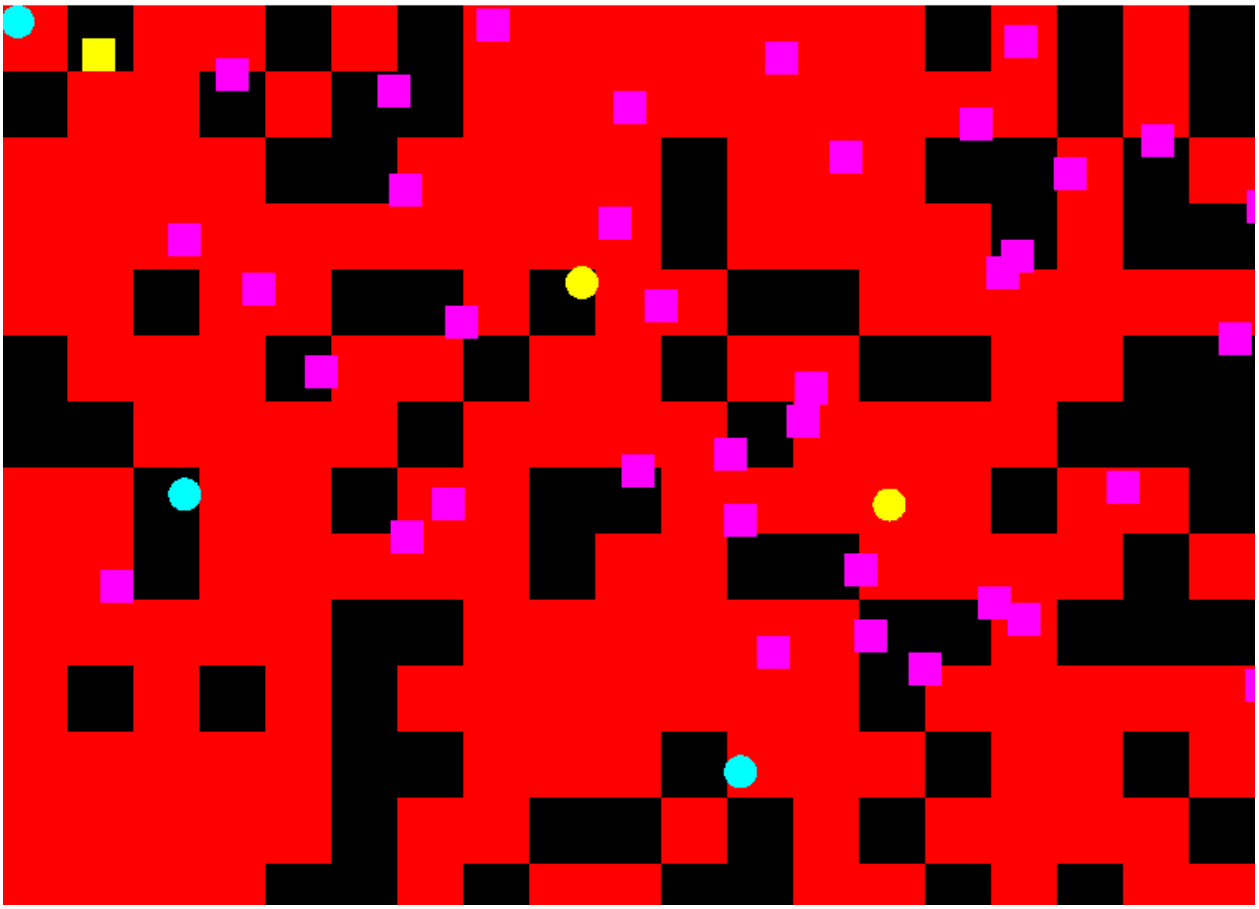


〈파란색 픽셀의 캐릭터가 노란색 아이템을 먹은 후의 모습〉



〈빨간색 타일 위에 서있는 파란색 플레이어 캐릭터 위로 분홍색 장애물이 내려오고 있습니다, 원형은 아이템들입니다.〉





〈아이템을 먹고 색이 파란색에서 노란색으로 바뀐 모습〉

10. 계획 대비 변경 사항

1차 변경: 아이템별 아이콘 준비, 색상 선택, 파괴 효과 설정, 아이템 종류와 효과 정의

연기사유

추후 게임의 볼륨을 생각해 RAM제한이 넉넉한 64bit 버전의 SFML을 설치했으나 어떤 방법으로도 호환이 되지 않아 결국 모두 지우고 32bit로 돌아가는 과정에서 예상치 못한 시간 소모가 있었습니다. Unity나 unreal engine과 같은 툴을 생각하며 계획을 수립했으나 SFML은 충분한 기능이 있지만 시각화 되어있는 툴이 없기에 기능을 숙지하는 데 예상치 못한 시간 비용이 소모되었습니다. 또한 적절한 image resource를 찾는 과정에서 게임의 컨셉은 색이 적은 세상이기에 흑백에 가까운 픽셀 기반의 resource를 찾고 싶었으나 그 과정에서도 예상치 못한 시간이 많이 소모되어 연기하게 되었습니다. 보다 견고한 계획과 함께 넉넉한 시간을 확보해두고 잦은 빈도로 프로젝트를 진행해야 할 필요성을 느꼈습니다. 그에 따라 아이템은 아이템끼리, 장애물은 장애물끼리 단기 목표를 묶어 구현 계획을 수정했습니다.

2차 변경:

처음에는 타 과목의 시험 일정 및 프로젝트 일정이 시험 주간에 몰려 있을 것으로 예상하여 최종 점검 일자를 넉넉하게 짰지만, 예상과 달리 타 과목의 시험 일정 및 프로젝트 일정이 12월 초에 몰려 있기에 3,4,5주차의 날짜를 변경했습니다. 또한 중간제출은 늘 작동이 되는 코드를 올려야 하기에 사운드 및 디버깅을 최종 점검에 하기보다 한 기능이 구현될 때마다 보고서에 추가하고 테스트하며 하는 것으로 변경했습니다.

3차 변경 :

기존 코드의 확장성이 컬러 선택 Scene 및 아이템, 장애물 및 게임 종료, 시작 관리에 적합하지 않다고 생각해서 전체적으로 단일 책임 원칙을 따르려 노력하며 코드를 다시 짰기에 일정이 늦어졌습니다.

4차 변경 :

리펙토링을 지속하며 Game 클래스와 GameManager Calss의 책임을 명확히 분리하고자 했고, 생성자와 Initialize의 중복을 최소화하려고 했으나, 시간이 오래 걸렸고 디버깅하는 과정에서 다시 쓰게 되었습니다.

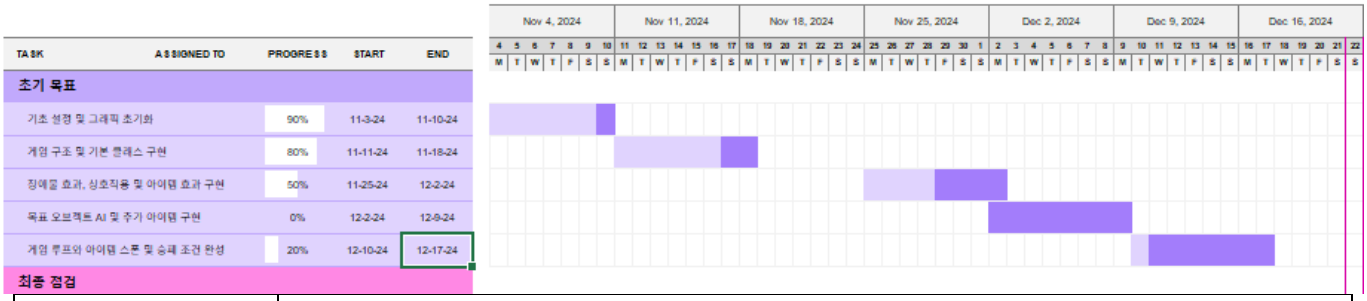
11. 프로젝트 일정

Save Some

VanArsdel, Ltd. Project lead

Project start: Sun, 11-3-2024

Display week: 1



완료	**1주차: 기초 설정 및 그래픽 초기화** Visual Studio Code와 C++ 개발 환경 설정. SFML 설치 및 기본 창 띄우기.
	2주차: 게임 구조 및 기본 클래스 구현 이동, 색상 선택. 초기 상태 및 픽셀 속성 정의
	2주차: 게임 구조 및 기본 클래스 구현 아이템, 장애물 생성 및 플레이어와의 충돌 효과 구현 소스 코드 리팩토링
진행중	**3주차: 장애물 효과, 상호작용 및 아이템 효과 구현** 장애물 종류 및 상성에 따른 파괴 효과 설정. 장애물의 상성에 따른 플레이어 픽셀 파괴 구현. 목표 오브젝트가 목적지에 도달했을 때 승리, 픽셀 수가 0일 때 패배 처리. 아이템 효과 구현: 색상 변경 아이템, 방어막 아이템, 회복 아이템, 속도 증가 아이템 추가.
예정	**4주차: 목표 오브젝트 AI 및 추가 아이템 구현** 아이템 종류와 효과 정의 아이템별 아이콘 준비 및 색상 출력 테스트. 목표 오브젝트가 플레이어를 따라가는 유도 패턴, 회피 패턴 구현.
	5주차: 게임 루프와 아이템 스폰 및 승패 조건 완성 아이템의 지속 시간과 여러 아이템이 동시에 활성화될 경우의 우선순위 처리. 장애물과 아이템의 출현 빈도 조절.
	최종 점검: UI, 사운드 UI와 사운드 추가, 픽셀 수, 점수, 아이템 지속 시간 등 게임 정보 UI 표시. 아이템 획득, 장애물 충돌 등의 사운드 효과 추가.

12. 느낀 점

우선 아쉬웠습니다. 더 꾸준히 하지 않은 제 자신에게 실망했습니다. SFML이 처음에는 무슨 소린지 알아듣기도 어려웠고, 생각보다 코딩으로 구현해야 하는 부분이 예상보다 많아서 계획을 잘못 수립했다는 것을 알아차렸습니다. 그냥 하드코딩보다는 최대한 확장성 있게 구성해서 나중에 난이도를 아이템과 타일, 장애물의 지표들로 측정하여 여러 스테이지를 만들고 싶었기에 최종 목적인 난이도별 여러 스테이지 구현에 있어서 소스 코드의 구조가 걸림돌이 될 것이라 예상했기 때문에 나름 다시 엎는다고 엎었지만 완성해놓고 나니 크게 달라진 게 없는 것 같기도 해서 아쉬웠습니다.

initialize method와 생성자의 통합을 시도했지만 player class에만 적용이 가능했고 다른 클래스들의 initialize method도 다 없앴는데 결국 디버깅하는 과정에서 다시 살리게 되었습니다. 처음부터 잘 하면 편하다는 것을 몸소 느꼈고, Clean Code라는 책을 가이드삼아 겨울방학에 더 정진할 계획을 세웠습니다.

단일 책임의 원칙과 캡슐화, 헤더 파일 등을 최대한 활용해보려고 했지만 정작 상속은 활용하지 못해서 아쉬웠습니다. 추후 개발이 더 진행되면 상속도 활용해보고 싶습니다.

코드를 다시 짜려는 과정에서는 어떤 타이밍에 테스트를 해야 할지 막막했습니다. 코드 수정이 모든 클래스에 대해서 끝나야 정상 작동할 것이라 생각해서 마지막에 디버깅을 시작했는데, 돌이켜 보니 비효율적인 것 같았습니다. 그렇다고 중간에 진행하기에는 다른 클래스도 그에 맞게 바뀐 게 아니라 오작동할 게 예상되었습니다. 또한 오류 코드나 메시지, try-catch문을 통해서 디버깅을 합리적으로 하는 방법도 배우고 싶었습니다. 디버깅에 집중해본 경험이 없다 보니 큰 오류가 아님에도 쏟는 시간이 많았습니다. 그래서 강의에서 디버깅을 다루는 시간이 있다면 참 좋겠다고 생각했습니다.

그래도 SFML을 더 잘 다루고 싶다는 생각이 들었고, C++도 매력적인 언어라고 생각합니다. 언젠가 저 원들이 동물이 되고, 저 네모가 살람이, 빨간 블록은 지형이, 하늘에서 내리는 분홍 블록은 보스 패턴이 되기를 바라면서 끝까지 완성해보고 싶습니다.