C++ 프로그래밍 및 실습 진척 보고서 #3

2024. 12. 15 / 지능실감미디어 융합전공 185478 김동민

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

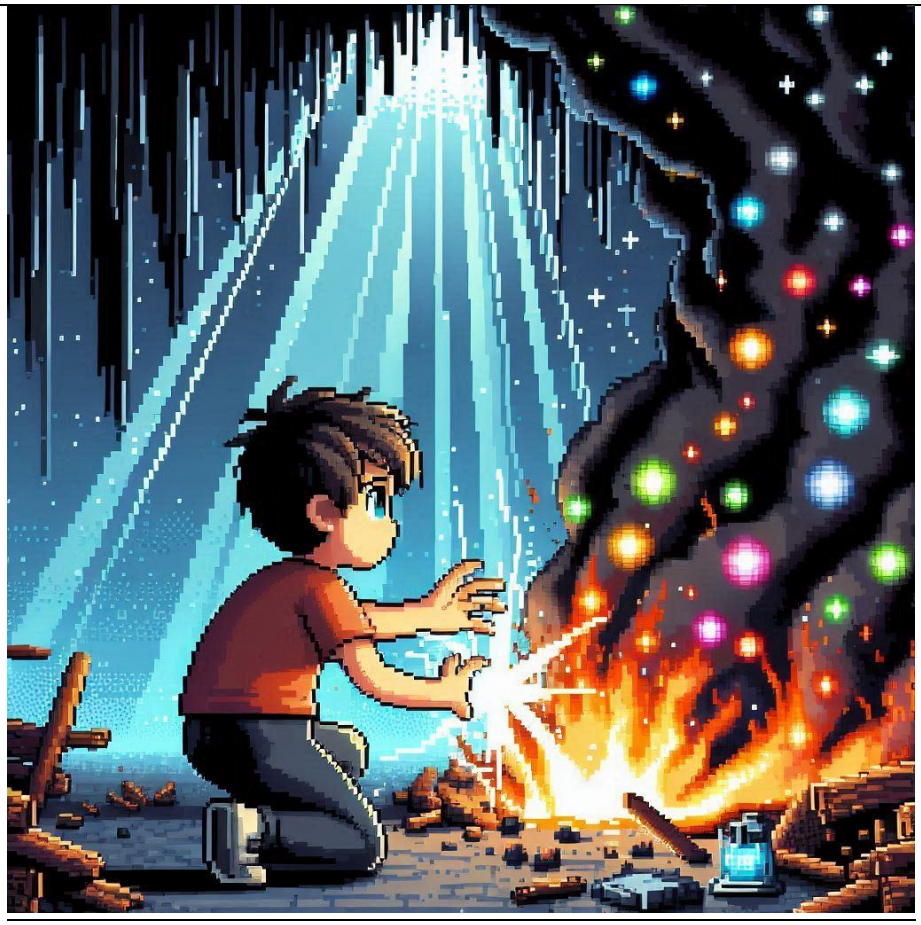
게임 스토리

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.

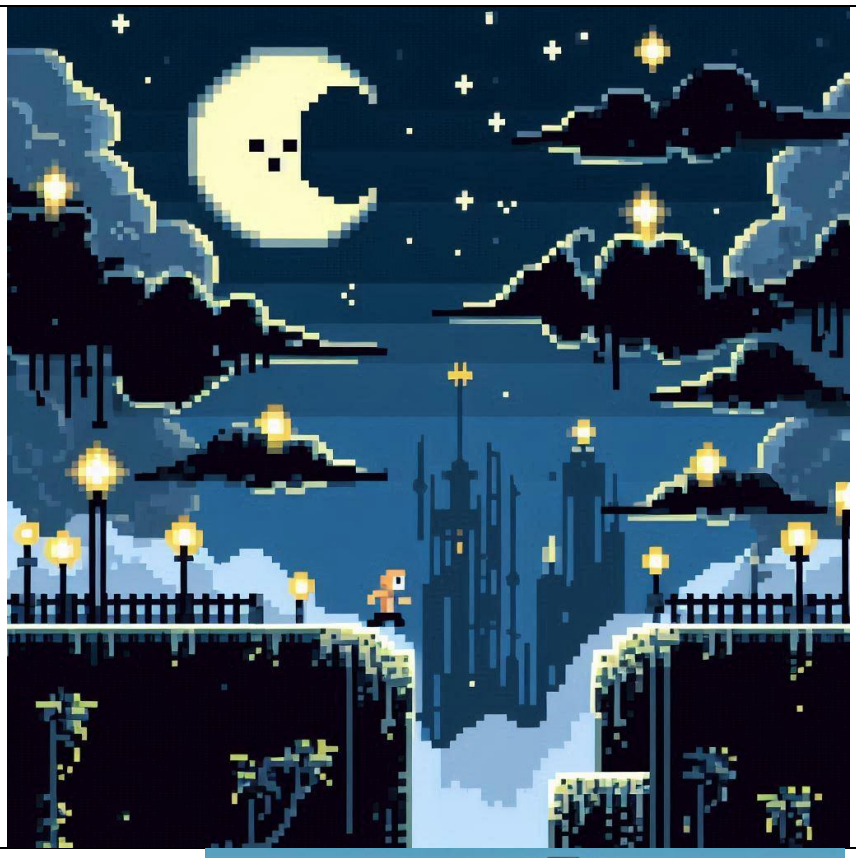
2.1.3. 1차적으로 주어진 스테이지를 모두 클리어 하게 되면 엔딩을 보는 방식입니다. 끊임없이 기록을 세우며 경쟁하는 게임보다는 엔딩을 정해두고, 엔딩 시간을 기록한 후 추후 재도전 가능한 방식으로 기획했습니다.

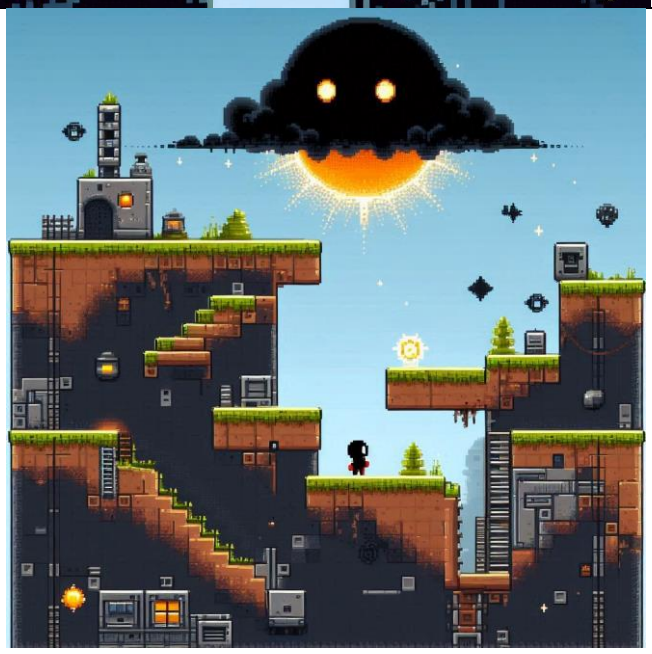
4. 스토리 컨셉 아트



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

```
#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);
```

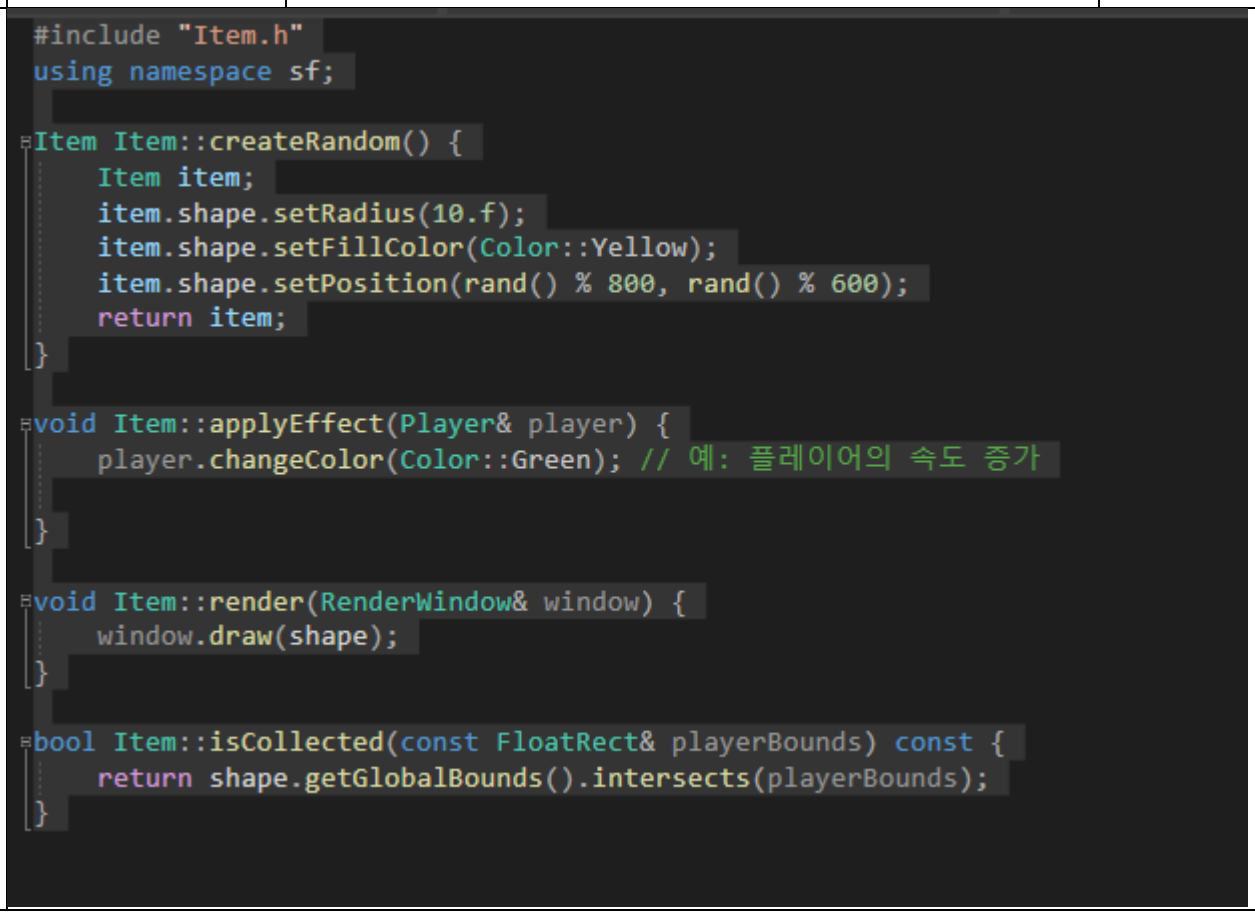
입출력	설명	배운내용

	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);
    }
}
```

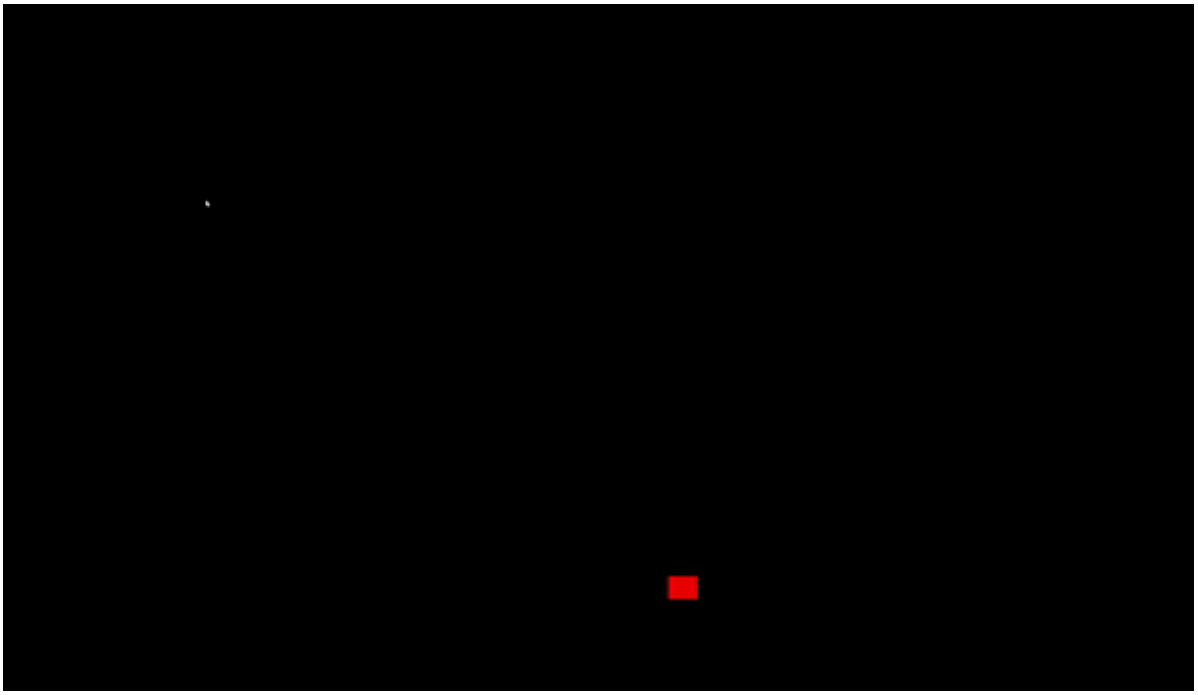
변수(멤버,생성자)	함수	설명 및 배운 내용

	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.

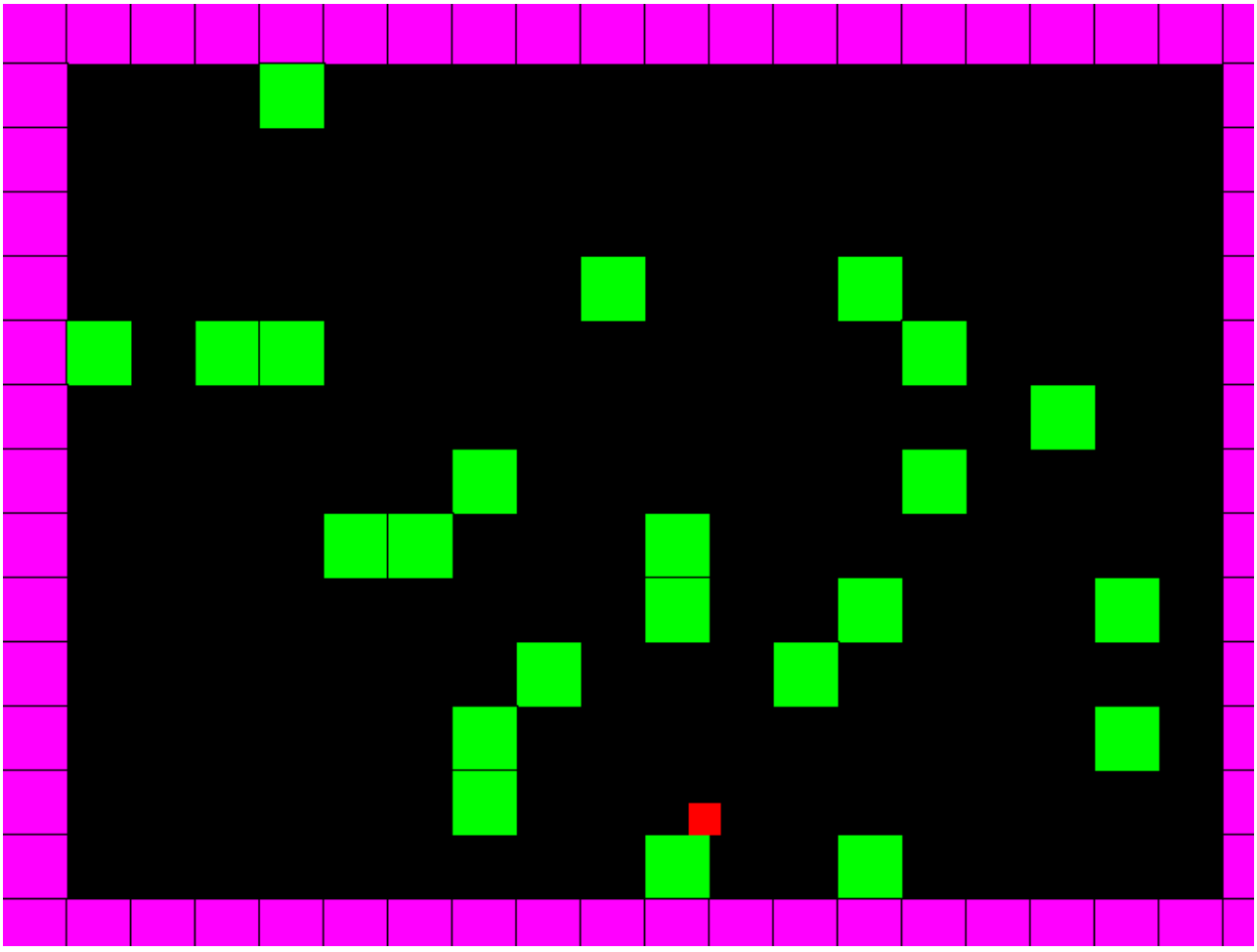
9. 테스트 결과

(1) 플레이어 픽셀 구현

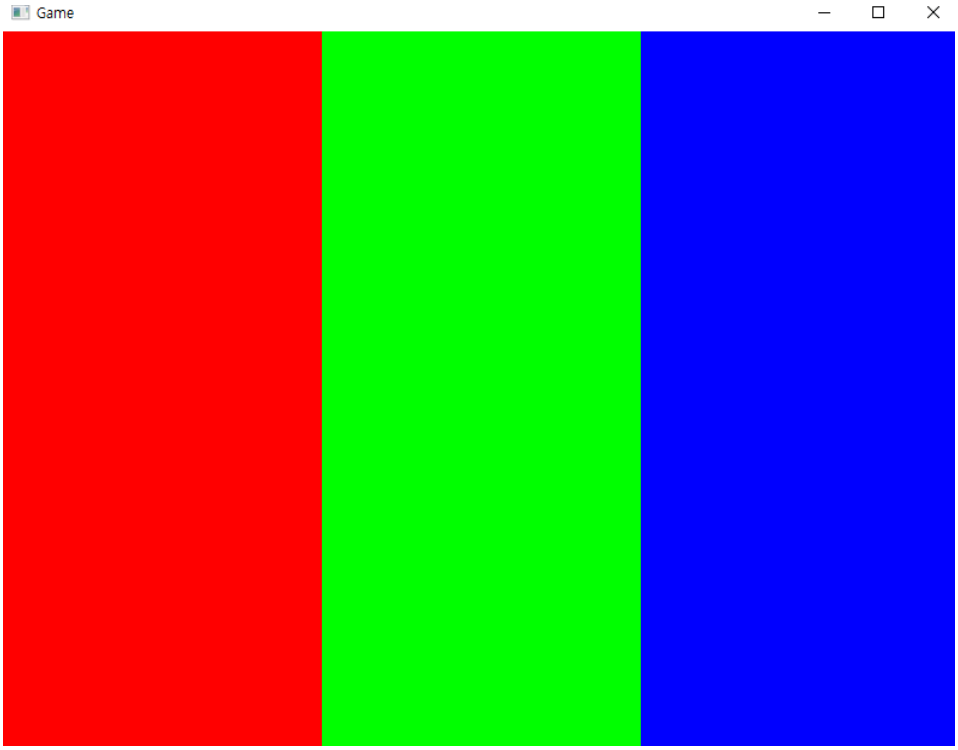
(2) 플레이어 좌우 이동, 점프, 이동제한 구현-



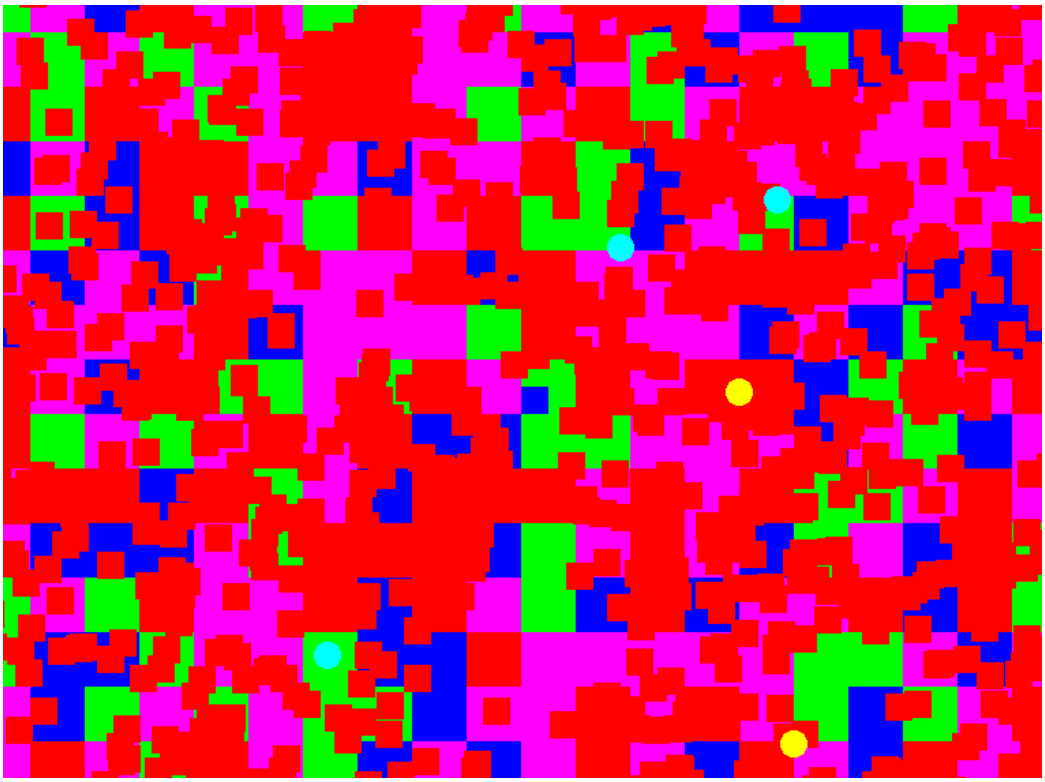
(3) 맵 생성 구현



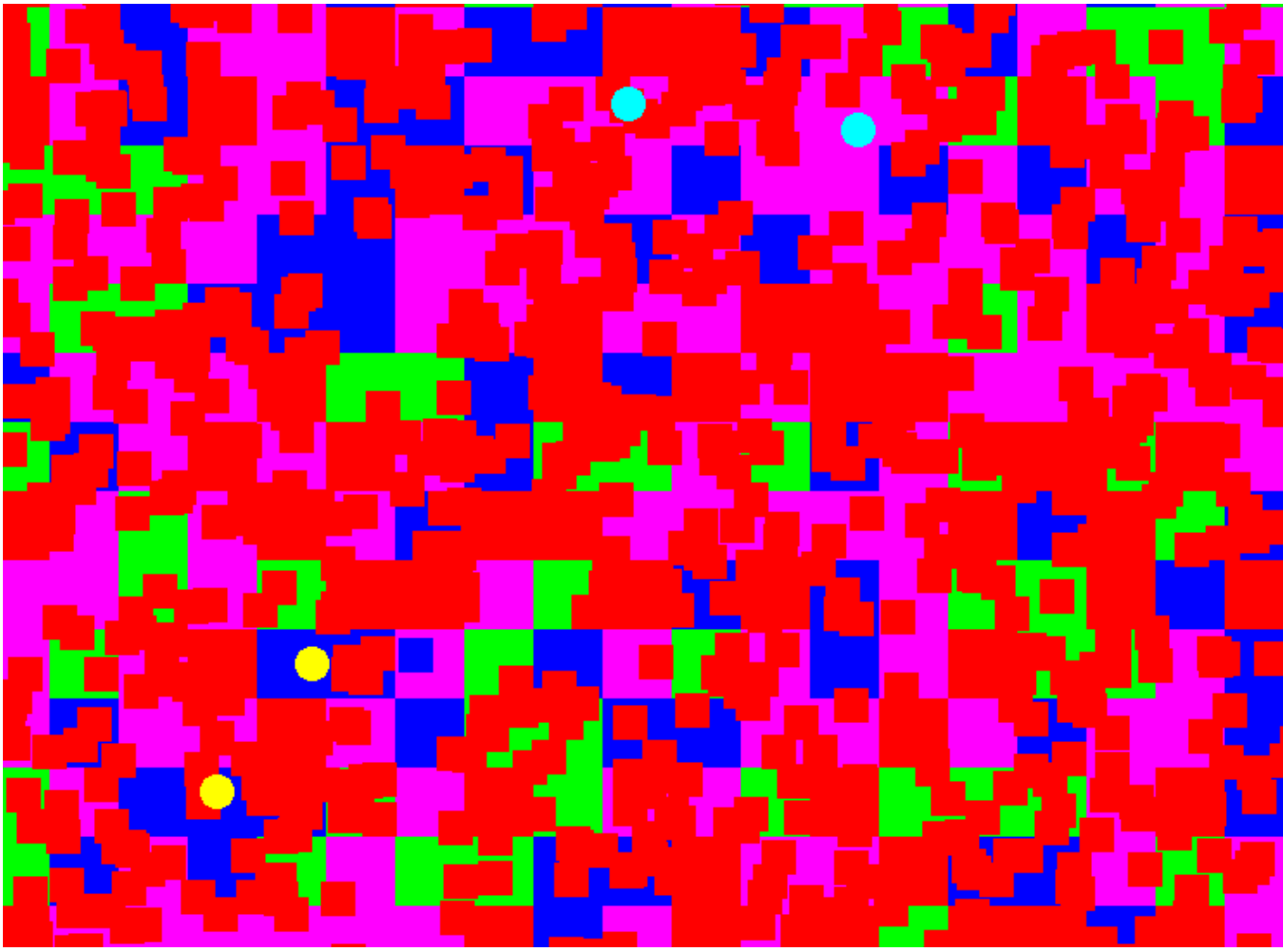
(4)플레이어 색상 선택창

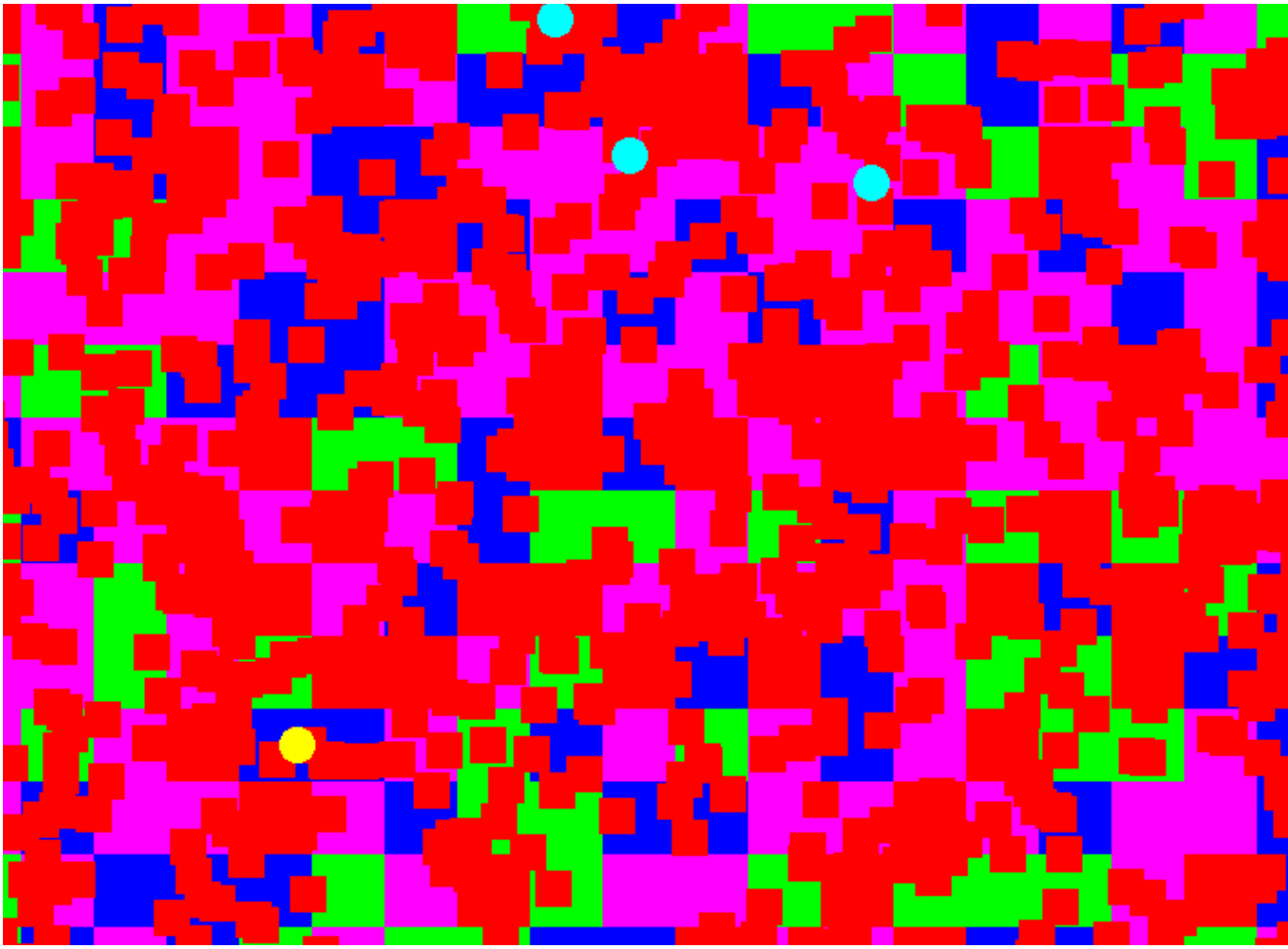


(5) 장애물 및 아이템 스폰 구현(장애물: 붉은색 사각형, 아이템: 원형)



(6)아이템 및 장애물 충돌 구현





〈파란색 픽셀의 캐릭터가 노란색 아이템을 먹은 후의 모습〉

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 늘어졌습니다.

11. 프로젝트 일정

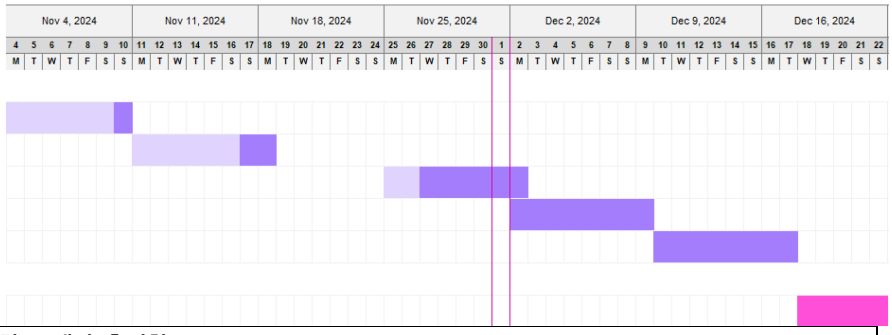
Save Some

VanArsdel, Ltd. Project lead

Project start: Sun, 11-3-2024

Display week: 1

TASK	ASSIGNED TO	PROGRESS	START	END
초기 목표				
기초 설정 및 그래픽 초기화		90%	11-3-24	11-10-24
게임 구조 및 기본 클래스 구현		80%	11-11-24	11-18-24
장애물 효과, 상호작용 및 아이템 효과 구현		30%	11-25-24	12-2-24
목표 오브젝트 AI 및 추가 아이템 구현		0%	12-2-24	12-9-24
게임 루프와 아이템 스폰 및 승패 조건 완성		0%	12-10-24	12-17-24
최종 점검				
UI, 사운드		0%	12-18-24	12-22-24



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.
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. 아이템 효과 구현: 색상 변경 아이템, 방어막 아이템, 회복 아이템, 속도 증가 아이템 추가.
	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.