01. KUNAI

게임명 : KUNAI
제작 인원 : 1인
플랫폼 : Window API
제작 도구 : VisualStudio2019
제작 기한 : 4주
게임 장르 : 플래포머

게임 선정 이유

WindowAPI 환경에서 다양한 요소를 보여줄 수 있는 게임을 찾던 도중 게임의 특성상 화려한 이펙트, 시원한 타격감을 중심으로 게임의 기승전결을 구현 하면서 제작 능력을 향상시킬 수 있을 것으로 판단하여 KUNAI를 선정하였습니다.

Used Skill

동영상 클릭

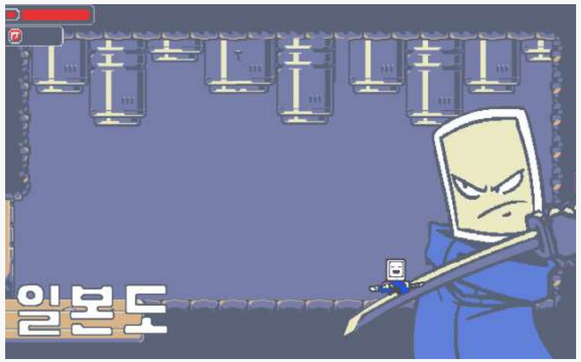
타격감과 액션, 이펙트에 중점을 두며 구현

Name 김유민
Moblie 010 7480 0851
E-mail rladbals0129@naver.com

게임 시작

타이틀 화면에서 게임 진행을 위한
게임시작과 테스트를 위한 튜토리얼
에 진입할 수 있습니다.

모험



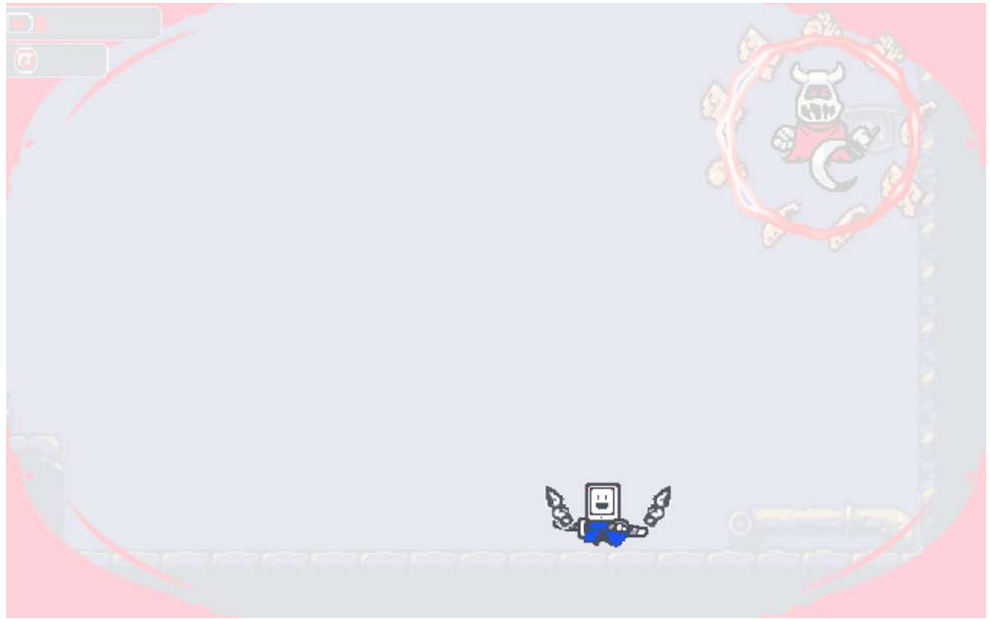
맵을 탐험하며 무기를 획득하고
성장한 캐릭터가 적과 조우하며
맵을 탐험할 수 있습니다.

보스전

모험을 통해 얻은 무기들을 활용해
보스전투를 진행합니다.

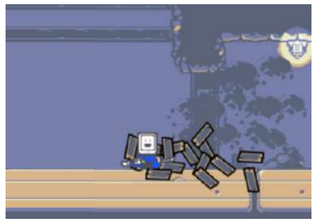
01 게임 디테일 - 오브젝트와 파편

화려한 연출과 시각적인 즐거움, 타격감을 주고 디테일을 올리기 위해 타격감과 이펙트에 중점을 두고 구현하였습니다. 사용하고 있는 GDI 의 특성상 회전이 어려워 회전이 필요한 부분은 GDI+를 사용해 구현했습니다.



캐릭터 등장

유리관을 깨고 나오는 캐릭터의 등장



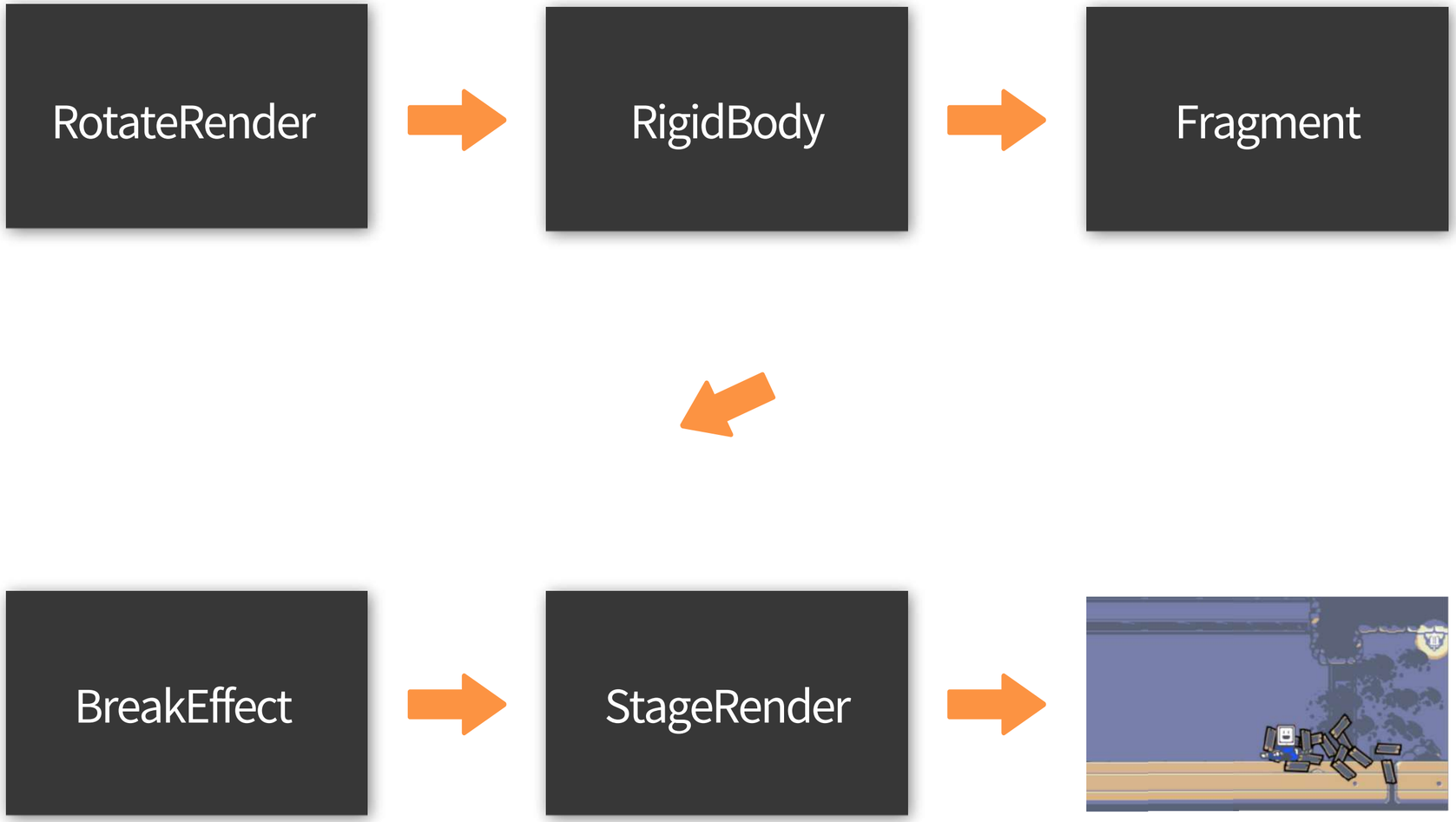
오브젝트 파괴

맵을 가로막고 있는 상자를 파괴하면 생성되는 연기와 파편

적 처치

적 처치시 로봇 파편이 생성

파편의 클래스 구조


RotateRender
RigidBody
Fragment
BreakEffect
StageRender

RotationRender

```h
#include "GameNode.h"
#include <gdiplus.h>
#pragma comment(lib, "gdiplus.lib")

using namespace Gdiplus;

class RotationRender : public GameNode
{
private:
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    Image* image;
    REAL _angle;

public:
    HRESULT init(void);
    void release(void);
    void LoadImage(wchar_t* filePath);

    void RotateRender(int x, int y, int width, int height, bool render);
    void rotateImage(REAL delta_angle,bool isColliding);

};
```

GDI 기반으로 화면을 렌더링 해주고 있어서 이미지에 회전을 주는 효과를 넣기 어려운 부분이 있었습니다. 편하게 사용할 다른 렌더링 포맷이 필요했고 파편을 생성해 주는 부분에서만 GDI+를 활용해 프레임 안정성을 확보하고 회전에 대한 이미지 처리를 하기 위해 GDI+를 불러오는 코드입니다.

```cpp
HRESULT RotationRender::init(void)
{
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, nullptr);
    _angle = 0;

    return S_OK;
}

void RotationRender::release(void)
{
    delete image;
    GdiplusShutdown(gdiplusToken);
}


void RotationRender::LoadImage(wchar_t* filePath)
{
    if (image != nullptr)
    {
        delete image;
    }
    image = new Image(filePath);
}
```

초기화를 진행해 준 후 이미지의 파일 경로를 입력해 이미지를 불러오는 LodaImage입니다.

```cpp
void RotationRender::RotateRender(int x, int y, int width, int height, bool render)
{
    Gdiplus::Graphics graphics(getMemDC());
    graphics.SetSmoothingMode(SmoothingModeAntiAlias);


    int centerX = image->GetWidth() / 2;
    int centerY = image->GetHeight() / 2;
    graphics.TranslateTransform(x, y);
    graphics.RotateTransform(_angle);
    graphics.TranslateTransform(-x, -y);
    if (render)
    {
        graphics.DrawImage(image, x - centerX, y - centerY, width, height);
    }
}

void RotationRender::rotateImage(REAL delta_angle, bool isColliding)
{
    if (!isColliding)
    {
        _angle += delta_angle;
        if (_angle >= 360)
            _angle -= 360;
    }
}
```

이미지를 회전시켜 주기 위해 pivot의 좌표를 옮겨 준 후 이미지를 회전시켜줍니다.
실제 회전에 들어가는 rotateImage 함수는 바닥에 붙지 않았을때만 이미지가 회전하기 위해 isColliding 변수를 통해 관리해주었습니다. 충돌검출방식은 픽셀충돌을 사용하였습니다.

발생했던 문제점과 해결방안

GDI+의 특성상 이미지를 그냥 회전 시켜주니까 회전 pivot이 이미지의 0,0좌표로 부터 회전하는 문제가 있었습니다. 이를 해결하기 위해 pivot의 좌표를 이미지의 중앙 위치로 옮겨 준 후 이미지를 회전 시켜 준 후 이미지를 그려주었습니다.

Rigidbody

```h
#pragma once
#include "GameNode.h"
struct Vec2
{
    float x, y;

    Vec2() : x(0), y(0) {}
    Vec2(float x, float y) : x(x), y(y) {}
};

class RigidBody
{
private:
    float m_gravity;
    Vec2 m_position;
    Vec2 m_velocity;
    Vec2 m_acceleration;

public:
    RigidBody() : m_gravity(9.81f) {}
    ~RigidBody() {}
    inline float getGravity()
    {
        return m_gravity;
    }
    void SetPosition(float x, float y) { m_position.x = x; m_position.y = y; }
    void SetVelocity(float x, float y) { m_velocity.x = x; m_velocity.y = y; }

    void SetGravity(float x) { m_gravity = x; }
    void SetPosTop(int y) { m_position.y += y; }
    void SetPosBottom(int y) { m_position.y -= y; }
    void SetAcceleration(float x, float y) { m_acceleration.x = x; m_acceleration.y = y; }

    const Vec2& GetPosition() const { return m_position; }
    Vec2 GetVelocity() const { return m_velocity; }
    Vec2 GetAcceleration() const { return m_acceleration; }

    void Update(float deltaTime);
};
```

- Rigidbody를 상속받는 하위 클래스에서 바닥과 충돌했는지 여부를 판단하기 위해 다양한 프로퍼티 함수를 준비

```cpp
#include "Stdafx.h"
#include "RigidBody.h"

void RigidBody::Update(float deltaTime)
{
    // 중력 적용
    m_acceleration.y = m_gravity;

    // 속도 업데이트
    m_velocity.x += m_acceleration.x * deltaTime;
    m_velocity.y += m_acceleration.y * deltaTime;

    // 위치 업데이트
    m_position.x += m_velocity.x * deltaTime;
    m_position.y += m_velocity.y * deltaTime;
}
```

- 게임 프레임워크에 맞는 deltaTime을 매개변수로 받아 중력을 적용
- 속도와 위치의 값을 deltaTime에 맞게 update 해줄 수 있게 구조를 작성

Fragment

```h
#pragma once
#include "RotationRender.h"
#include "RigidBody.h"

class Fragment : public RigidBody
{
private:
    RotationRender m_rotationRender;
    float m_rotation;
    float m_rotationSpeed;

    bool m_shouldRotate;
    bool m_render;

    int _cnt;
    bool _initCnt;
    bool _goUpdate;

public:
    Fragment() : m_rotation(0.0f),m_shouldRotate(true), m_render(true),_cnt(0),_goUpdate(true){}

    void SetRotation(float rotation) { m_rotation = rotation; }
    float GetRotation() const { return m_rotation; }

    void LoadImage(wchar_t* filePath) { m_rotationRender.LoadImage(filePath); }

    void SetRotationSpeed(float rotationSpeed) { m_rotationSpeed = rotationSpeed; }
    float GetRotationSpeed() const { return m_rotationSpeed; }

    void RotateRender(int x, int y, int width, int height)
    {
        m_rotationRender.rotateImage(m_rotation, m_shouldRotate);    // 파편의 회전
        m_rotationRender.RotateRender(x, y, width, height, m_render); // 파편 렌더링
    }

    void Update(float deltaTime,HDC hdc,int offsetX,int offsetY);


};
```

- 회전을 위해 준비한 RotationRender를 불러옴
- 오브젝트가 파괴될 때 힘을 적용받고 중력의 영향을 받아야 하기해 Rigidbody를 상속 받음

```cpp
void Fragment::Update(float deltaTime , HDC hdc, int offsetX, int offsetY)
{
    if (_goUpdate)
    {
        float currentGravity = RigidBody::getGravity();
        SetAcceleration(0, currentGravity);

        Vec2 nextPosition;
        nextPosition.x = GetPosition().x + GetVelocity().x * deltaTime;
        nextPosition.y = GetPosition().y + GetVelocity().y * deltaTime;

        int colOBJ = GetRValue(GetPixel(hdc, static_cast<int>(nextPosition.x) + offsetX,
            static_cast<int>(nextPosition.y) + offsetY));

        const float friction = 0.87f;
        const float bounciness = 0.7f;
        const float speedLimit = 0.1f;

        if (colOBJ == 131)
        {

            SetVelocity(GetVelocity().x * friction, -GetVelocity().y * bounciness);
            SetAcceleration(0, 0);
            SetRotationSpeed(GetRotationSpeed() * friction);
            m_shouldRotate = true;

        }
        else
        {
            m_shouldRotate = false;
            RigidBody::Update(deltaTime);

        }
        SetPosition(nextPosition.x, nextPosition.y);

        if (colOBJ == 131)
        {

            SetVelocity(GetVelocity().x * friction, GetVelocity().y);

            if (abs(GetVelocity().x) < speedLimit && abs(GetVelocity().y) < speedLimit)
            {
                SetVelocity(0, 0);
                SetRotationSpeed(0);
                m_shouldRotate = true;
                if (!_initCnt)
                {
                    _cnt++;
                }
                if (_cnt < 80)
                {
                    if (_cnt % 20 == 0)
                    {
                        m_render = !m_render;
                    }

                }
                else if (_cnt >= 80 && _cnt < 140)
                {
                    if (_cnt % 5 == 0)
                    {
                        m_render = !m_render;
                    }
                }
                if (_cnt > 140)
                {
                    m_render = false;
                    _goUpdate = false;
                    _initCnt = true;
                }
            }
            else
            {
                m_shouldRotate = false;
                SetAcceleration(0, currentGravity);
            }
        }
        m_rotation += (m_rotationSpeed * deltaTime);
    }
}
```

- 지면의 충돌을 확인하기 위해 픽셀충돌 사용
- 충돌시 position X값은 마찰력을 적용 받고 Y값은 bounciness의 값만큼 저항을 받아 통통 튀다 멈추는 효과를 줌
- 회전값에도 마찰력에 영향을 줘 서서히 회전을 멈춤

- 파편이 중력과 마찰력의 영향을 받아 일정 수준까지 감소하면 깜빡거림 효과를 넣어줌
- 일정 시간이 지나면 파편 삭제

발생했던 문제점과 해결방안

현제 구성되어있는 프레임워크의 게임 update주기가 0.01초여서 일반적인 60프레임을 지향하는 delatime 값인 0.016f을 집어넣으니 힘이 너무 쌔게 들어가는 등의 화면에 이상하게 출력되는 문제가 발생했습니다. 값을 적절히 조절하여 이 문제를 해결하였습니다.

파편 생성

```
//박스
std::vector<size_t> breakIndices;
if (IntersectRect(&_collider, &PLAYER->getATKRange(), &_obj[i].rc))
{
    SOUNDMANAGER->play("상자파괴");
    SOUNDMANAGER->setVolume("상자파괴", 0.2f);
    SOUNDMANAGER->play("칼피격");
    SOUNDMANAGER->setVolume("칼피격", 0.2f);
    applyShake(_initialShakeDuration);
    createBoxEF = true;
    if (createBoxEF)
    {
        for (int k = 0; k < 8; k++)
        {
            float EFX = RND->getFromIntTo(_obj[i].rc.left, _obj[i].rc.right);
            float EFY = RND->getFromIntTo(_obj[i].rc.bottom - 40, _obj[i].rc.bottom);
            _slashEffect.addEffect(EFX, EFY, 1, "상자깨지기이펙트검은색");
        }
        for (int j = 0; j < 3; j++)
        {
            float EFX = RND->getFromIntTo(_obj[i].rc.left, _obj[i].rc.right);
            float EFY = RND->getFromIntTo(_obj[i].rc.bottom - 40, _obj[i].rc.bottom);
            _slashEffect.addEffect(EFX, EFY, 1, "상자깨지기이펙트");
        }
        for (int j = 0; j < 3; j++)
        {
            float EFX = RND->getFromIntTo(_obj[i].rc.left, _obj[i].rc.right);
            float EFY = RND->getFromIntTo(_obj[i].rc.bottom - 40, _obj[i].rc.bottom);
            _slashEffect.addEffect(EFX, EFY, 1, "베기먼지");
        }

        float slashX = _obj[i].rc.left;
        float slashY = (_obj[i].rc.top + _obj[i].rc.bottom) / 2;
        _slashEffect.addSlashEffect(slashX, slashY, 1,"베기");
    }
createBoxEF = false;

breakIndices.push_back(i);
```

- 상자가 피격할 시 이펙트를 생성하며 vector로 파편 관리

```
for (size_t i : breakIndices)
{
    POINT position = { _obj[i].rc.left, _obj[i].rc.top };
    createFragments(m_fragments, position, L"Resources/Images/Stage1/Object/BoxBreak.png", 5);
}

for (auto it = breakIndices.rbegin(); it != breakIndices.rend(); ++it)
{
    _obj.erase(_obj.begin() + *it);
}

for (auto& fragment : m_fragments)
{
    fragment.Update(0.16f, IMAGEMANAGER->findImage("스테이지1픽셀")->getMemDC(),_offsetX,_offsetY);
}
```

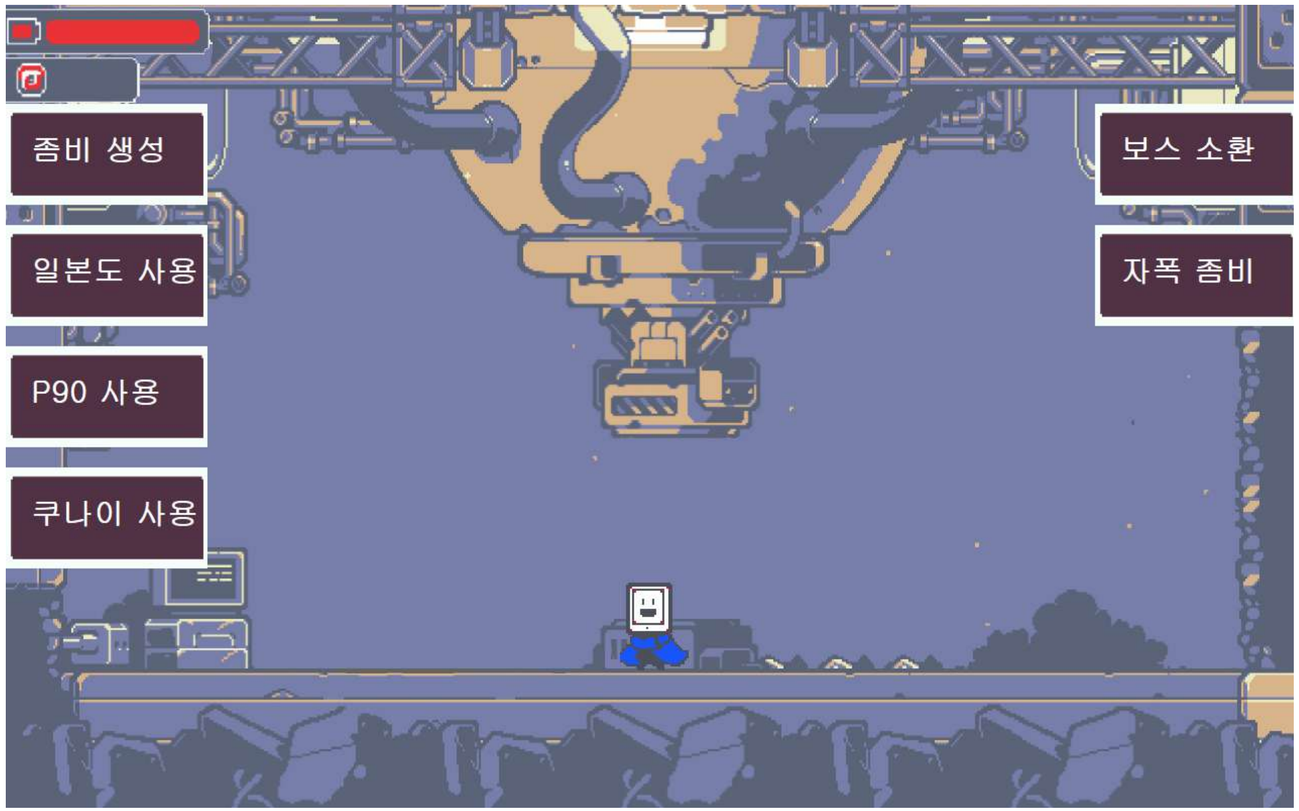
- 파편 생성에 필요한 인자 함수에 넘겨주기
- 파편의 지면 충돌을 위해 매개변수까지 넘겨줌

```
void Stage1::createFragments(std::vector<Fragment>& fragments, const POINT& position, wchar_t* imagePath, int numFragments)
{
    const float gravity = 9.81f;
    for (int i = 0; i < numFragments; i++)
    {
        Fragment fragment;
        fragment.SetPosition(static_cast<float>(position.x), static_cast<float>(position.y));
        float velocityX = static_cast<float>(rand() % 21 - 10) * 2.0f;
        float velocityY = static_cast<float>(rand() % 16 - 15) * 2.0f;
        fragment.SetVelocity(velocityX, velocityY);
        fragment.SetAcceleration(0, gravity);
        fragment.SetRotation(RND->getFloat(10.f));
        fragment.LoadImage(imagePath);
        fragments.push_back(fragment);
    }
}
```

- 이미지 경로를 입력받아 불러오고 속도와 가속도를 입력받고 세팅해 준 후 vector를 통해 관리

결과 화면

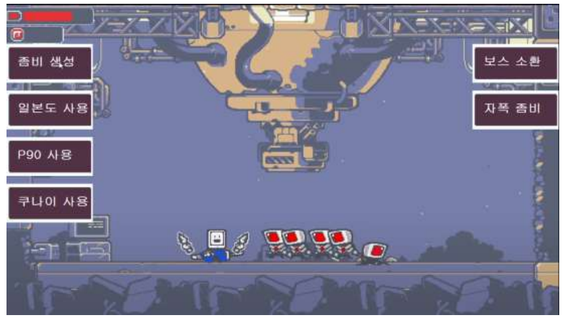
02 테스트 씬



게임을 진행하기 앞서 기능 테스트

구현되어있는 무기와 적, 파티클을 미리 테스트해 볼 수 있는 튜토리얼 스테이지를 만들어 개발 과정에서 테스트가 필요하거나 프레임 안정성을 체크하는데 도움이 되는 튜토리얼 씬을 제작하였습니다.
이 덕분에 게임을 테스트를 원활하게 진행할 수 있었습니다.

결과 화면



Button 구성

```
struct Button
{
    RECT rect;
    string text;

    Button(int left, int top, int width, int height, const string& buttonText)
    {
        rect.left = left;
        rect.top = top;
        rect.right = left + width;
        rect.bottom = top + height;
        text = buttonText;
    }
};

class Tutorial : public GameNode
{
private:
    std::vector<Button> _buttons;
    const int _buttonWidth = 200;
    const int _buttonHeight = 100;
    ZombieManager _zombieManager;
```

- 버튼을 구조체로 정의하고 Vector로 관리해 줄 준비

Button 추가

```
//버튼//
const vector<string> buttonTexts = {
"좀비 생성", "일본도 사용" ,"P90 사용" , "쿠나이 사용","보스 소환", "자폭 좀비"};
for (int i = 0; i < buttonTexts.size(); i++)
{
    int x = (i < 4) ? 0 : 1280 - _buttonWidth;
    int y = 100 + i % 4 * (_buttonHeight + 20);
    _buttons.emplace_back(x, y, _buttonWidth, _buttonHeight, buttonTexts[i]);
}
```

- 기능이 추가될 시 buttonTexts의 인덱스에 요소 추가 시 버튼 자동 구성

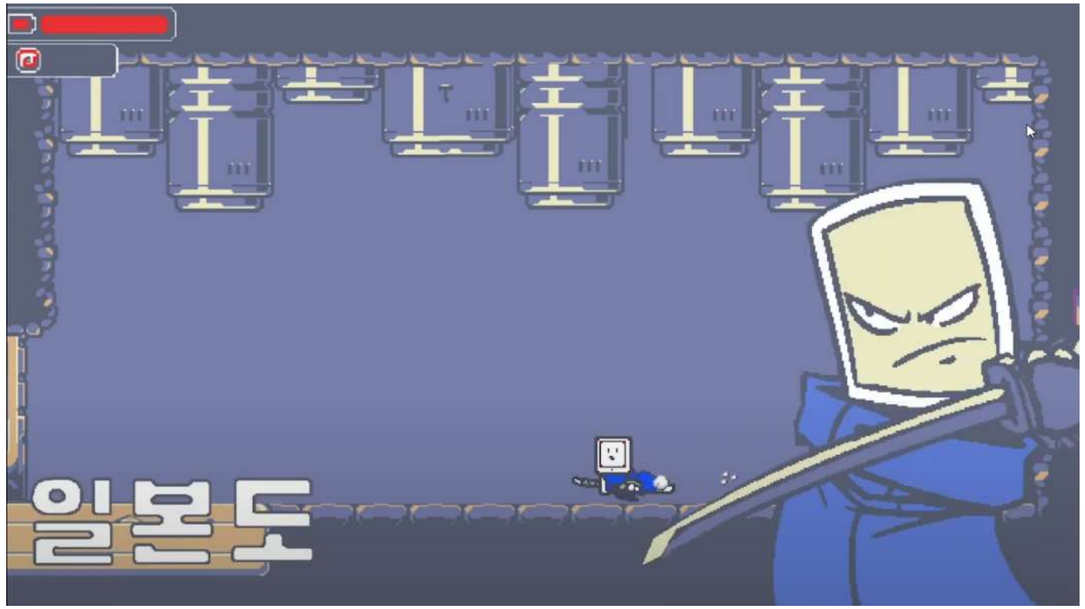
버튼의 기능

```
void Tutorial::onButtonClick(int buttonIndex)
{
    switch (buttonIndex)
    {
    case 0:
        _zombieManager.createZombie(800, 600);

        break;
    case 1:
        PLAYER->setUsingKnife(!PLAYER->getUsingKnife());
        break;
    case 2:
        PLAYER->setUsingGun(!PLAYER->getUsingGun());
        break;
    case 3:
        PLAYER->setUsingKunai(!PLAYER->getUsingKunai());
        break;
    case 4:
        PLAYER->setCreateBoss(!PLAYER->getCreateBoss());
        break;
    case 5:
        createLipo(300, 580);
        break;
    }
}
```

- 개발 과정에서 기능구현
- 튜토리얼 씬 필요한 버튼 추가

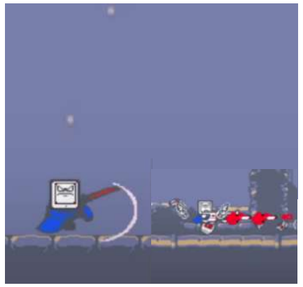
03 플레이어와 UI

처음 시작하면 플레이어는 아무것도 없는 상태로 시작합니다. 맵을 탐험하며 무기를 획득하고 기능들이 하나하나 추가됩니다. 원작과 유사성과 타격감에 신경을 쓰며 구현하였습니다. 호흡을 하는듯하는 효과는 삼각함수를 이용해 구현했습니다.
UI는 각각에 상황에 맞게 화면에 출력해주었습니다.



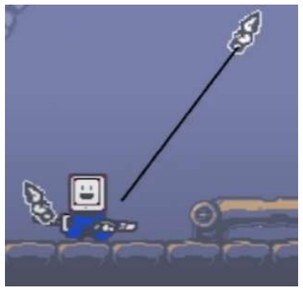
이동

플레이어가 이동 또는 점프동작을 하면 먼지가 생성됩니다. 플레이어의 위치와현제 맵에 따라 카메라가 플레이어의 위치를 추적할지, 맵을 변경할지를 결정합니다.



일본도와 P90

모험을 통해 일본도를 획득하게 되면 플레이어의 외관에 일본도 관련 애니메이션이 나오고 공격동작이 추가됩니다. P90을 획득시 원거리 공격이 가능해집니다.



쿠나이

쿠나이를 획득하게 되면 벽을 타고 오를 수 있는 행동이 추가됩니다.

피격과 UI상호작용

피격시 플레이어가 넉백되며 HP가 감소합니다. 체력이 얼마 남지 않았을 경우 화면에 경고UI가 표기됩니다.

쿠나이 경로 그리기

```
radian = angle * (3.14159265f / 180.0f);

if (KEYMANAGER->isStayKeyDown('Q'))
{
    isLeftFlying = false;
    createL_Rope = true;
    inputQ = true;
    angle = -135.0f;
}
else
{
    inputQ = false;
    isLeftFlying = true;
    isLeftStuck = false;
    leftAnimFinished = false;

    createL_Rope = false;
    EfLeft = false;
    L_pathPoints.clear();
}
```

- Q 버튼 입력시 쿠나이가 정해진 각도로 던져지며 선은 vector<pair<int,int>>로 관리
- 키 입력을 땠을땐 R 또는 L_pathPoints를 clear해줘서 선 경로를 삭제

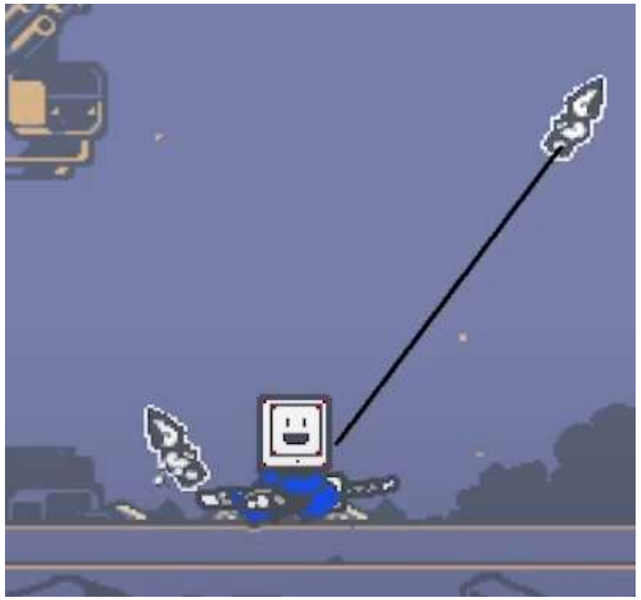
```
if (!isRightFlying && !isRightStuck)
{

    _rc2.left += speed * cos(radian);
    _rc2.right += speed * cos(radian);
    _rc2.top += speed * sin(radian);
    _rc2.bottom += speed * sin(radian);
    R_pathPoints.push_back(make_pair(_rc2.left+20, _rc2.top+60));
}
else
{
    _rc2 = RectMake(PLAYER->getPlayerPos().left + 70,
        PLAYER->getPlayerPos().top - 20, 60, 160);
}
```

- _rc2는 쿠나이 이미지
- 그 뒤를 L또는R_pathPoints로 위치값을 저장해둠

```
if (createL_Rope)
{
    for (int i = 0; i + 1 < L_pathPoints.size(); ++i)
    {
        MoveToEx(hdc, L_pathPoints[i].first, L_pathPoints[i].second, NULL);
        LineTo(hdc, L_pathPoints[i + 1].first, L_pathPoints[i + 1].second);
    }

}
```

- 쿠나이의 렌더링 함수 일부분
- 경로 포인트만큼 WindowAPI에서 지원하는 MoveToEx, LineTO로 렌더링

결과 화면



쿠나이 벽 충돌

```
void KunaiCollision::kunaiCollision(HDC hdc, int offsetX, int offsetY)
{
    int LKT[60];
    for (int i = 0; i < 60; i++)
    {
        LKT[i] = GetGValue(GetPixel(hdc,
            PLAYER->getKunai()->getLeftKuniaPos().left + offsetX + i,
            PLAYER->getKunai()->getLeftKuniaPos().top + offsetY));
    }

for (int i = 0; i < 60; i++)
{
    if (LKT[i] == 134)
    {

        PLAYER->getKunai()->setLeftStuck(true);
        POINT leftCollision;
        leftCollision.x = PLAYER->getKunai()->getLeftKuniaPos().left + i + 90;
        leftCollision.y = PLAYER->getKunai()->getLeftKuniaPos().top + 20;
        PLAYER->getKunai()->setLeftWallCollision(leftCollision);
        break;
    }
}
```

- 벽 충돌은 픽셀기반 충돌 검출방식 사용
- 정밀도를 위해 60개의 픽셀값을 얻고 카메라 offset값을 더해줌

쿠나이 벽 충돌

```
if (!PLAYER->getKunai()->getLeftFlying())
{
    if (PLAYER->getKunai()->getLeftStuck())
    {
        float targetPosX = PLAYER->getKunai()->getLeftWallCollision().x;
        float targetPosY = PLAYER->getKunai()->getLeftWallCollision().y;
        PLAYER->setPosX(PLAYER->getPosX()
            + (targetPosX - PLAYER->getPosX()) * _lerpSpeed);
        PLAYER->setPosY(PLAYER->getPosY()
            + (targetPosY - PLAYER->getPosY()) * _lerpSpeed);
        PLAYER->getKunai()->eraseLeftPathPoints(3);
        if (abs(PLAYER->getPosX() - targetPosX) < 1.f &&
            abs(PLAYER->getPosY() - targetPosY) < 1.f)
        {
            PLAYER->setPosX(targetPosX);
            PLAYER->setPosY(targetPosY);
            PLAYER->getKunai()->setLeftStuck(false);

        }

        if (!_soundL)
        {
            SOUNDMANAGER->play("쿠나이벽충돌");
            SOUNDMANAGER->setVolume("쿠나이벽충돌", 0.2f);
            _soundL = true;
        }

    }
    else
    {
        _soundL = false;
    }
}
```

- 충돌한 벽의 위치를 targetPos로 저장
- lerpSpeed를 정해줘 부드럽게 이동
- 플레이어의 위치를 끌어당기며 경로를 지워줌
- 타겟과 플레이어의 위치가 가까워지면 플레이어 위치 고정

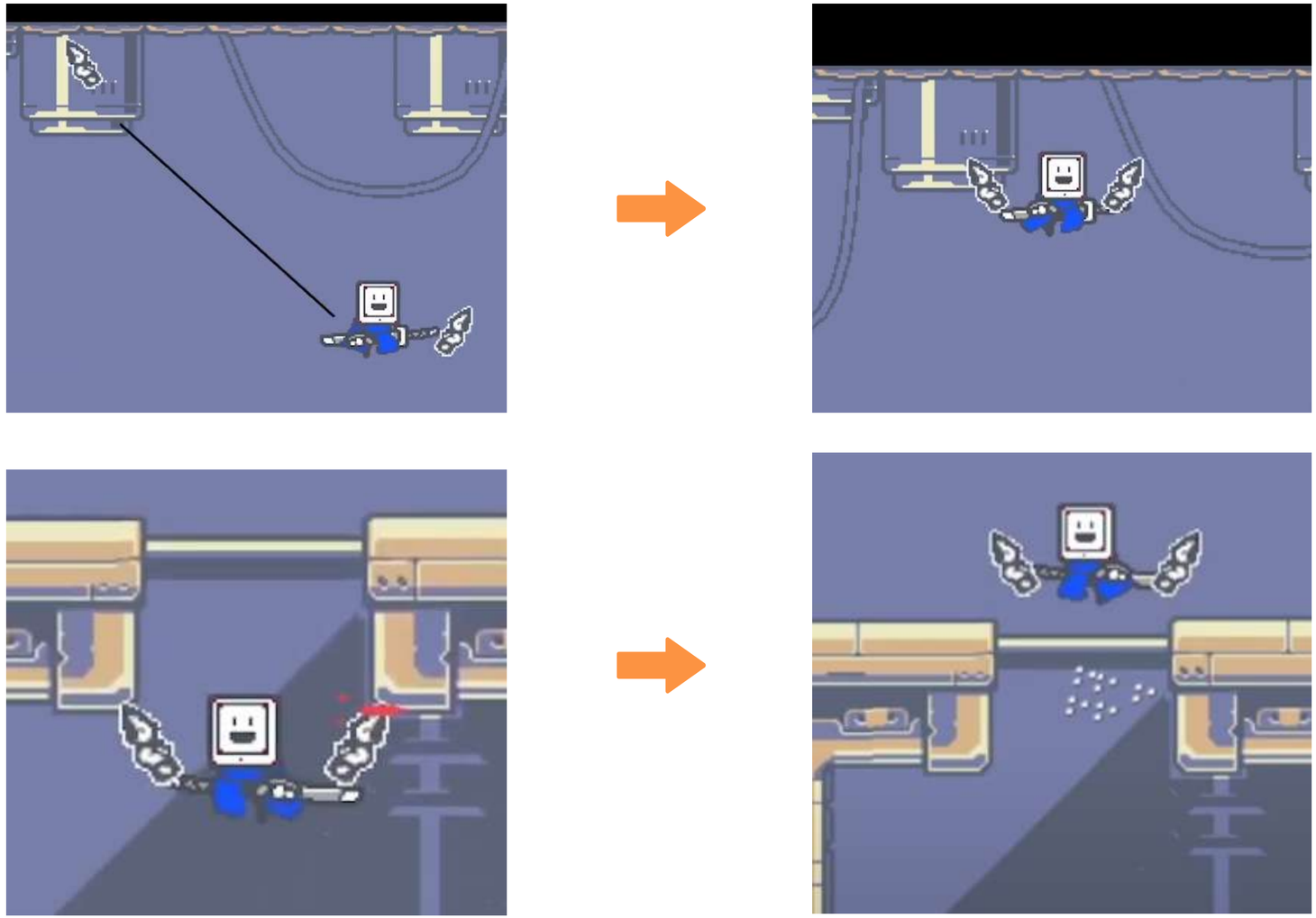
쿠나이 벽 충돌

```
void Kunai::eraseLeftPathPoints(size_t count)
{

    if (count > L_pathPoints.size())
    {
        L_pathPoints.clear();
    }
    else
    {
        L_pathPoints.erase(L_pathPoints.begin(), L_pathPoints.begin() + count);
    }
}
```

```
if (!PLAYER->getKunai()->getLeftFlying() &&
    PLAYER->getKunai()->getLeftStuck())
{

    PLAYER->setIsCanLeftWallJump(true);
    PLAYER->setGravity(0);
    PLAYER->setIsJumping(false);
}
else
{
    PLAYER->setIsCanLeftWallJump(false);
    PLAYER->setGravity(8);
    PLAYER->setIsJumping(true);
}
```

- 벽에 고정되어 있는 상태라면 중력을 0으로
- 고정되어 있는 상태에선 점프키 활성화

결과 화면



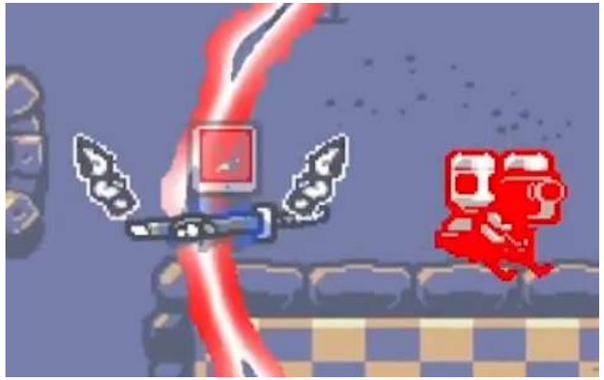
넉백

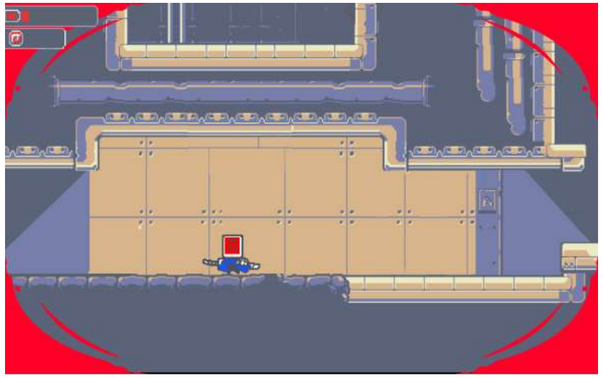
```
_rc.left += _knockbackSpeedX;
_rc.right += _knockbackSpeedX;
_rc.top += _knockbackSpeedY;
_rc.bottom += _knockbackSpeedY;
_knockbackDistanceX += abs(_knockbackSpeedX);
_knockbackDistanceY += abs(_knockbackSpeedY);

if (_knockbackDistanceX >= _maxKnockbackDistance ||
    _knockbackDistanceY >= _maxKnockbackDistance)
{
    _knockbackSpeedX = 0.0f;
    _knockbackSpeedY = 0.0f;
}
```

- Player의 update함수
- 넉백 속도로 최대 넉백거리만큼 밀어주어 부드럽게 넉백

```
if (IntersectRect(&_collider, &PLAYER->getPlayerPos(), &_Fzm[i]->getPos()))
{
    if (!_hitDelay)
    {
        PLAYER->setDmg(_Fzm[i]->getAtk());
        PLAYER->setHit(true);
        _hitDelay = true;
        //넉백
        float knockBackX = PLAYER->getPlayerCenter() >
            _Fzm[i]->getCenter() ? _knockBackMagnitude : -_knockBackMagnitude;
        float knockBackY = 0; ;
        PLAYER->setKnockback(knockBackX, knockBackY);
        //======
        applyShake(_initialShakeDuration);
    }
}
```

- 넉백이 필요한 상황이 발생하면 부딪힌 적의 center값을 구함
- 왼쪽으로 밀릴지 오른쪽으로 밀릴지 결정해 플레이어에게 전달해주고 **화면 쉐이크**





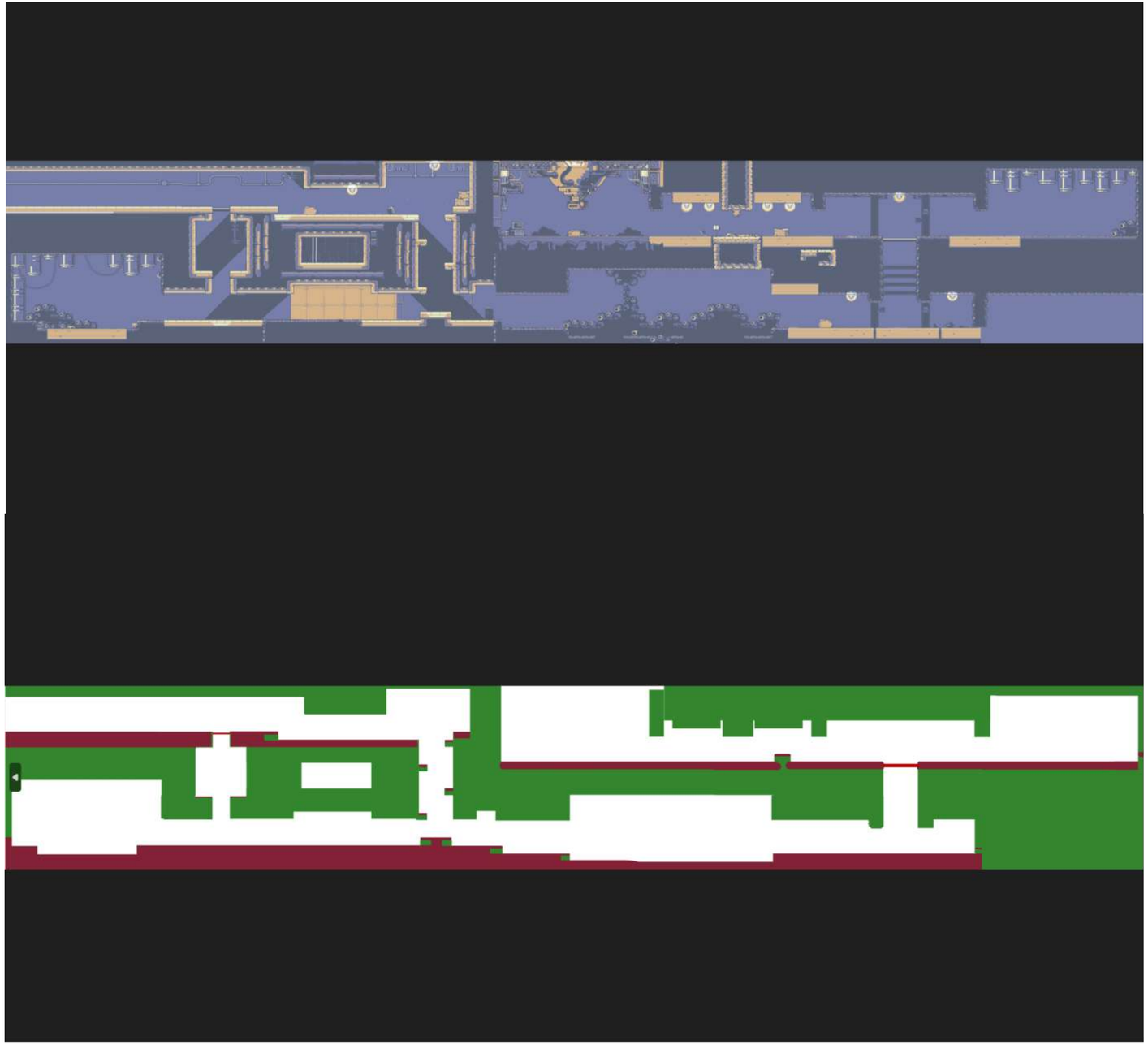
플레이어를 구현함에 있어 문제점과 해결방법

단순히 값을 넣어봤더니 뚝뚝 끊기는 느낌이 들었습니다. 점프, 쿠나이, 넉백 등등 최대한 부드러운 렌더링 방식에 대해 고민했고 또한 넉백의 경우 밀리는 방향 역시 고민해본 결과 디테일을 살리는 로직을 작성해 보자 해서 그것에 중점을 두고 많은 고민을하며 구현했습니다.

맵과 카메라

클리핑을 통한 맵 렌더링

큰 맵에서 offset값을 정하여 윈도우 화면에 맞게 화면을 랜더링했습니다. 충돌방식은 픽셀기반 충돌 검출방식을 사용하였습니다.



카메라 랜더링

```
IMAGEMANAGER->render("스테이지1", getMemDC(), 0 - _shakeOffsetX, 0 - _shakeOffsetY, _offsetX, _offsetY, 8960, 1600);
```

- Offset 값을 이용한 카메라 위치 렌더링
- ShakeOffset값을 이용한 화면 흔들기로 타격감 증가

좌우

```
if (PLAYER->getPlayerPos().left > RcameraOffsetX && _offsetX < RmaxOffsetX)
{
    _offsetX += 8;
    PLAYER->_dustEffect.setLeft(8.f);
    PLAYER->_bullet.setLeft(8.f);
    _slashEffect.setLeft(8.f);
    if (_currentMap == map1)
    {
        for (int i = 0; i < _Fzm.size(); i++)
        {
            _Fzm[i]->setPosRight(8);
        }
        for (auto& fragment : _Zfragments)
        {
            fragment.SetPosition(fragment.GetPosition().x - 8, fragment.GetPosition().y);
        }
    }

    PLAYER->setPlayerPosRight(8);
}
```

offset과 반대가 되는 값으로 플레이어와 맵에 있는 오브젝트, 적 등을 밀어주어 카메라 효과를 구현하였습니다.

상하

```
if (PLAYER->getPlayerPos().top < cameraOffsetY && _offsetY > 0)
{

    _offsetY -= cameraOffsetY - PLAYER->getPlayerPos().top;
    _slashEffect.setBottom(cameraOffsetY - PLAYER->getPlayerPos().top);
    if (_currentMap == map1)
    {
        for (int i = 0; i < _Fzm.size(); i++)
        {
            _Fzm[i]->setPosTop(cameraOffsetY - PLAYER->getPlayerPos().top);
        }
        for (auto& fragment : _Zfragments)
        {
            fragment.SetPosTop(cameraOffsetY - PLAYER->getPlayerPos().top);
        }
    }

    PLAYER->setPlayerPosTop(cameraOffsetY - PLAYER->getPlayerPos().top);
}
```

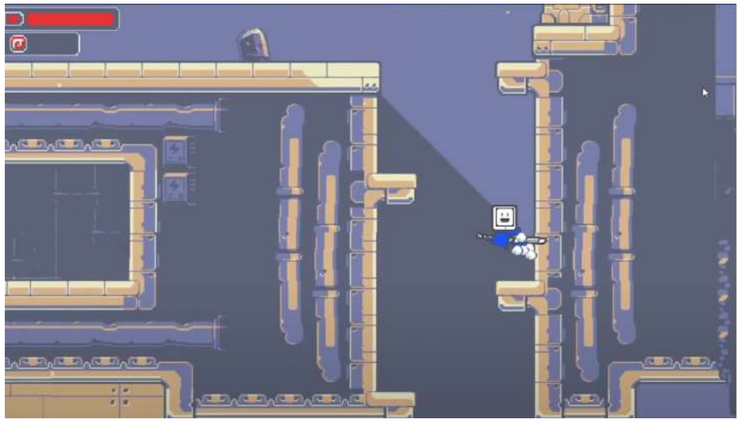
상하 역시 마찬가지로 반대가되는 Y값으로 밀어주었습니다.

화면 흔들기

```
void Stage1::updateShakeEffect(float& shakeDuration, float& shakeOffsetX, float& shakeOffsetY)
{
    if (shakeDuration > 0)
    {
        shakeDuration -= 0.016f;

        shakeOffsetX = (rand() % (int)(_initialShakeMagnitude * 2 + 1) - _initialShakeMagnitude) *
            sin(2 * PI * shakeDuration / _initialShakeDuration);
        shakeOffsetY = (rand() % (int)(_initialShakeMagnitude * 2 + 1) - _initialShakeMagnitude) *
            sin(2 * PI * shakeDuration / _initialShakeDuration);
    }
    else
    {
        shakeOffsetX = 0.0f;
        shakeOffsetY = 0.0f;
    }
}

void Stage1::applyShake(float shakeDuration)
{
    _shakeDuration = shakeDuration;
}
```

- updateShakeEffect함수를 이용해 **지속시간동안 랜덤한 방향으로 카메라가 흔들릴 수 있게 구현**
- 필요한 시점에 applyShake 함수를 호출 해 지속시간을 입력함

결과 화면



문제점과 해결방법

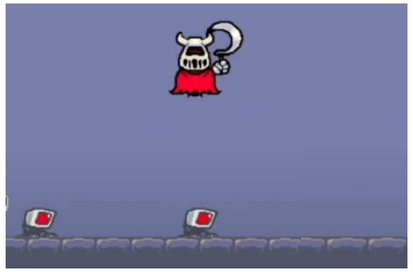
enum문의 map이 넘어가야 하는 상황에선 빠른 속도로 맵을 밀어주며 enum문의 값을 바꿔주면 되지만 길이가 길거나 높이가 높은 맵의 경우 플레이어를 추적하는 카메라가 필요했습니다. 처음엔 맵의 offset값을 이동시키며 플레이어만 반대로 밀어주는 방식을 사용해 보았는데 그 결과 적, 오브젝트 등 다른 요소들이 못따라오는 현상이 생겨 그에 대한 예외처리를 하고 구현하였습니다.

05 보스

최종적으로 만나게되는 보스입니다. 처음 맵에 입장하면 잠들어있으며 플레이어와 상호작용 시 대사와 함께 등장하게 됩니다. 3가지의 패턴을 가지고있으며 플레이어에게 피해를 입으면 다음 패턴을 시전합니다. 총 9회 피격시 이펙트와 함께 사망하며 엔딩씬으로 전환됩니다.

패턴 1

포물선을 그리며 맵 왼쪽 오른쪽을 이동합니다. 플레이어와 충돌시 플레이어에게 피해를 주며 이동중이지 않을땐 무적상태입니다.



패턴 2

보스의 위치가 서서히 정 중앙을 향해 움직이며 일반 몬스터인 좀비 봇을 3마리 생성합니다.

패턴 3

맵에 바위를 생성하며 높은곳으로 올라갑니다.

패턴

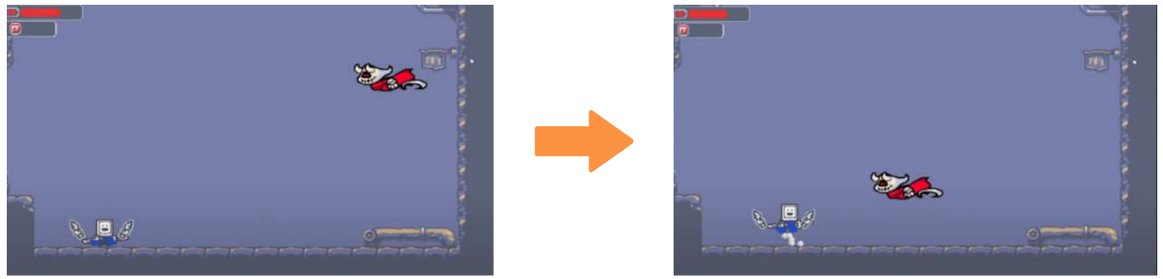
```
void Boss::pattern1(void)
{
    const float pi = 3.14159265;


    float t = static_cast<float>(_movementTime) / static_cast<float>(_maxMovementTime);

    float startX = _isLeft ? 1000 : 200;
    float endX = _isLeft ? 200 : 1000;

    float startY = 110;
    float endY = 110;

    int amplitude = 400;


    float newX = startX + t * (endX - startX);


    float newY = startY + t * (endY - startY);


    int newTop = static_cast<int>(newY + amplitude * sin(pi * t));


    int width = _rc.right - _rc.left;
    int height = _rc.bottom - _rc.top;

    _rc.left = static_cast<int>(newX);
    _rc.right = static_cast<int>(newX + width);
    _rc.top = newTop;
    _rc.bottom = newTop + height;
}
```

```
if (_currentState == ATTACK1)
{
    if (_movementTime < _maxMovementTime)
    {
        if (!_patternSound1)
        {
            SOUNDMANAGER->play("보스패턴1");
            SOUNDMANAGER->setVolume("보스패턴1", 0.2f);
            _patternSound1 = true;
        }
        pattern1();
        _movementTime++;
    }
    else
    {

        _isLeft = !_isLeft;
        _currentState = IDLE;
    }

}
```

- 보스의 패턴 구현 코드

1. t는 보스 캐릭터가 움직임을 시작한 후 흐른 시간 비율 (0과 1 사이의 값)
2. isLeft의 상태에 따라 Start와 End위치 지정
3. **진폭을 설정한 후 다음 위치값을 계산해 선형 보간**
4. newTop 변수를 이용해 y축 위치값을 계산 해 주기적인 상하 움직임을 만들고 진폭적용

결과 화면과 발생했던 문제점

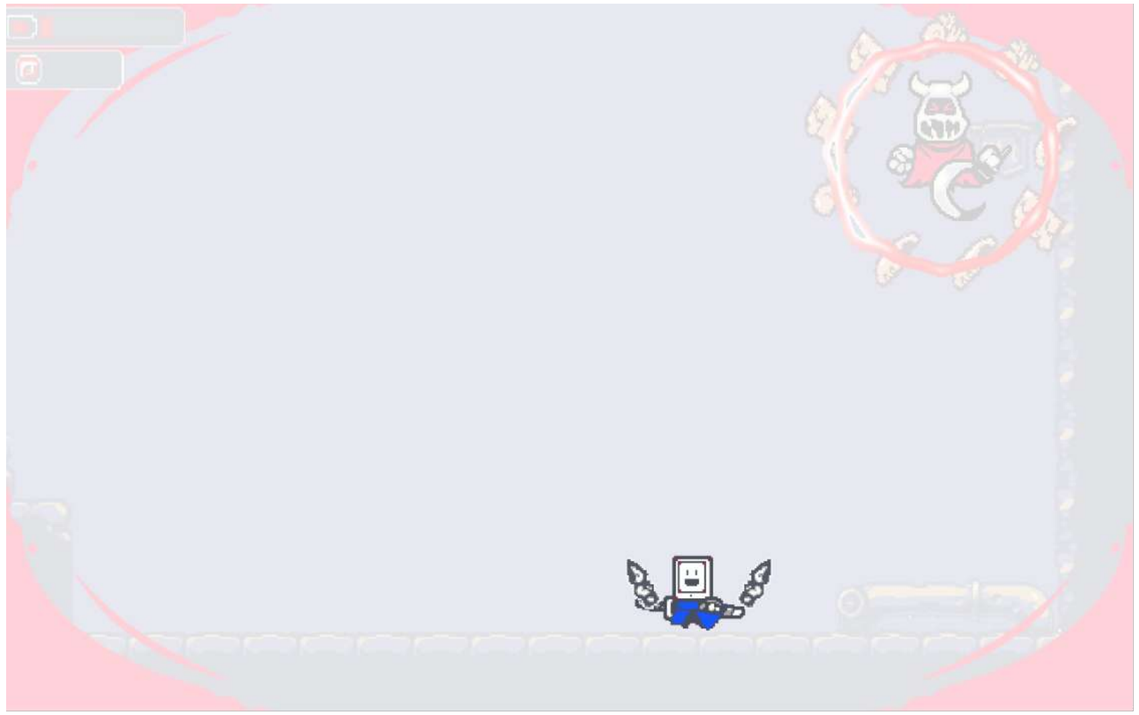


결과와 어려웠던점

보스가 초기 위치에서 목표 위치까지 선형적으로 이동하는 동안 사인 곡선을 따라 상하로 움직입니다. 이동 경로 계산과 시간비율 t를 생각하는데 어려움을 겪었습니다. 계산한 시간 비율을 선형 이동과 사인 곡선의 계산에 사용했는데 두가지 이상의 사고력을 합치는데 어려움을 겪었습니다.

이 코드를 작성하면서 단순한 이동 경로를 구하는 것도 어려운데 복잡한 패턴을 구현하려면 더 많은 노력을 해야겠구나를 느꼈습니다. 이번 경험은 선형보간, 사인함수, 시간관리에 대해 숙달될 수 있는 경험이였습니다.

보스 처치



총 9회 타격을 입히면 이펙트와 함께 보스가 사망하게 되며 엔딩씬으로 전환됩니다.

보스를 마치며 느낀점과 배운점

원작의 특성과 패턴을 동일하게 살린 보스입니다. 패턴1에 비해 2 ,3이 비교적 구현 난이도가 적었습니다. 하지만 AI구현 및 처음 보스몬스터 제작 경험이였기 때문에 아쉬움도 많이 남지만 이러한 경험 덕분에 효율적인 구조에 대해 추후 많이 생각해 보게 되었고 팀 포트폴리오 제작을 함에 있어 선 적, 보스에 대해 더 효율적으로 구현할 수 있었습니다.

결과/회고

첫 개인 프로젝트를 마치며

이번 프로젝트를 통해 기능을 하나씩 추가하면서 초기 클래스 설계와 효율적인 구조의 중요성을 다시 한 번 깨달았습니다.

1. 하나씩 기능을 추가하면서, **초기에 더욱 체계적으로 구조를 설계했다면 얼마나 좋았을까 하는 아쉬움이 들었습니다.** 특히 적과 보스에 대한 AI 구현에 있어 완벽하게 만족스럽지 못한 결과를 얻었습니다, 또한 **클래스 전방선언 및 스마트 포인터활용 등등 메모리 구조**에 대해 좀 더 생각하며 설계했었다면 더 좋은 프로젝트가 탄생할 수 있었을거란 아쉬움이 남는 프로젝트였습니다

2. 하지만 이 프로잭트 후에 진행한 팀 프로젝트에서는 프로젝트 경험을 바탕으로 클래스 구조에 대해 더 많이 생각했고 적과 보스에 대한 AI를 효율적인 구조로 설계하고 구현하는 데 이번 프로젝트가 많이 기여했습니다. 팀프로젝트에서는 팀원과의 협업, 메모리 간의 소통, 클래스 전방선언 등등 적극적으로 설계에 힘썼습니다

3. 해당 프로젝트에 대한 경험으로 C++의 설계와 메모리구조 파악의 중요성을 생각하고 코드를 작성해야겠다는 것을 알게해주는 좋은 경험이였습니다.

영상을 보고싶으시면 여기를 클릭해주세요

Name 김유민
Moblie 010 7480 0851
E-mail rladbals0129@naver.com