VIRUS STRIKER

Direct X 포트폴리오 정리

Dev: leeinhwan0421

포트폴리오 가이드

- 본 포트폴리오는 1인 개발 이후, GIGDC 출품을 위해 TEAM SWTH 에서 리팩토링 및 기능 추가를 함께 진행하였음을 밝힙니다.
- 직접 작성 및 수정한 내용만 서술하였습니다.

- 사용 기술은 아래와 같습니다.
  - Directx 9 API
  - C++ (C++ 17)
  - Photoshop (간단한 이미지 해상도, 배경 투명화 작업)

- 프로젝트 코드 및 자세한 구성을 확인하고 싶으신 분들은 여기 에서 확인해주세요.
- 추가적인 질문이 있으신 경우, lsnan421@naver.com 이메일로 문의를 남겨주시면 최대한 빠른 시일 내 답변하겠습니다.

목차

1 프레임워크 구조

1. Main.cpp

의도 | WinAPI를 사용해 Direct3D 기반의 윈도우 애플리케이션을 생성하고, 키 입력 및 게임 루프를 처리하도록 하였습니다.

방식 | 메시지 루프 내에서 입력, 고정 업데이트, 프레임 업데이트, 렌더링을 순차적으로 수행합니다.

효과 | 키 입력에 반응하고, 일정한 주기로 게임 로직을 갱신하며 화면을 렌더링할 수 있는 Direct3D 기반의 애플리케이션이 실행된다.

전체 코드: 바로가기

키 입력 처리 MsgProc

```
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
    case WM_KEYDOWN:
        Input::GetInstance().IfKeyPressed = true;
        Input::GetInstance().SetKeyState(wParam, InputState::DOWN);
        return 0;
    case WM_KEYUP:
        Input::GetInstance().IfKeyPressed = false;
        Input::GetInstance().SetKeyState(wParam, InputState::UP);
        return 0;
    case WM_DESTROY:
        PostQuitMessage( 0 );
        return 0;
    }
    return DefWindowProc( hWnd, msg, wParam, lParam );
}
```

게임 루프 처리

```
{
    static DWORD prevTime = timeGetTime();
    if (SceneManager::GetInstance().IsResetDeltaTime())
        prevTime = timeGetTime();

    DWORD curTime = timeGetTime();

    float deltaTime = (curTime - prevTime) / 1000.0f;
    static float fixedUpdateTime = 1.0f / 50.0f;
    static float fixedDeltaTime = 0.0f;

    fixedDeltaTime += deltaTime;
    if (fixedDeltaTime >= 2.0f) fixedDeltaTime = 2.0f;
    while (fixedDeltaTime > fixedUpdateTime)
    {
        fixedDeltaTime -= fixedUpdateTime;
        Direct3D::GetInstance().FixedUpdate(fixedUpdateTime);
    }

    static int fps = 0;
    static float fpsDeltaTime = 0.0f;
    ++fps;

    fpsDeltaTime += deltaTime;

    if (fpsDeltaTime > 1.0f)
    {
        Direct3D::GetInstance().fps = fps;
        wprintf(TEXT("%d\n"), fps);
        fps = 0;
        fpsDeltaTime = 0.0f;
    }

    Direct3D::GetInstance().Update(deltaTime);
    Direct3D::GetInstance().Render();
    Input::GetInstance().KeyUpdate();
    prevTime = curTime;
}
```

2. Direct3D 클래스

의도 | Direct3D 환경을 초기화하고, 씬(scene) 시스템과 함께 실시간으로 렌더링되는 게임 화면을 구성하려는 목적

방식 | 디바이스 생성, Z-buffer 설정, 알파 블렌딩을 활성화 한 상태로 Direct3D를 초기화 합니다. 또한 카메라 시점과 행렬을 설정했습니다.

효과 | 카메라 트랜지션 및 줌, 씬 관리, UI 렌더링 등을 포함한 Direct3D 클래스를 완성하였습니다. 또한 Scene 기반의 구조로 다양한 화면 전환 및 상태 관리 유연성을 확보하였습니다.

전체 코드: 바로가기

Direct3D 초기화 및 Device 생성

```
if (NULL == (pD3D = Direct3DCreate9(D3D_SDK_VERSION)))
    return E_FAIL;

if (FAILED(pD3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
    D3DCREATE_SOFTWARE_VERTEXPROCESSING,
    &d3dpp, &pd3dDevice)))
{
    return E_FAIL;
}
```

카메라 뷰 및 투영 행렬 설정

```
D3DXVECTOR3 vEyePt(0.0f, 0.0f, -10.0f);
D3DXVECTOR3 vLookatPt(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 vUpVec(0.0f, 1.0f, 0.0f);
D3DXMatrixLookAtLH(&matView, &vEyePt, &vLookatPt, &vUpVec);
pd3dDevice->SetTransform(D3DTS_VIEW, &matView);

D3DXMatrixOrthoLH(&matProj, gameWidth, gameHeight, 0.0f, 1000.0f);
pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj);
```

3. Scene, SceneManager 클래스

의도 | 씬 기반 구조를 사용해 게임 로직과 UI를 분리하고 상태에 따른 전환을 유연하게 처리하도록 하였습니다. 각 씬마다 오브젝트 관리 및 UI 렌더링을 독립적으로 처리할 수 있도록 하였으며 전역적인 씬 관리 시스템을 통해 씬의 등록, 삭제, 전환을 제어할 수 있도록 설계하였습니다.

방식 | Scene 클래스를 추상화하여, 각 씬은 공통된 순수 가상 함수(Update, Render …)를 상속받아 개별적으로 구현합니다. SceneManager는 내부에 Map 자료 구조를 사용해 씬을 관리합니다. 씬 전환 시 알파 값을 이용한 페이드 효과를 넣어 부드러운 화면 전환을 구현하였습니다.

효과 | 씬 별 로직이 깔끔하게 정리되었으며, 씬 전환이 자연스럽게 이루어집니다.

전체 코드: 바로가기

씬 등록 및 전환

```cpp
void SceneManager::AddScene(const std::wstring& _name, Scene* _newScene)
{
    auto sf = sceneMap.find(_name);
    if (sf != sceneMap.end())
        RemoveScene(_name);
    sceneMap.insert(make_pair(_name, _newScene));
}

void SceneManager::ChangeScene(const std::wstring& _name)
{
    auto sf = sceneMap.find(_name);
    if (sf != sceneMap.end())
        nextScene = sf->second;
}
```

씬 전환 시 페이드 효과 및 씬 전환 로직

```cpp
if (nextScene)
{
    spriteLoadingRI.a += _deltaTime * 2.0f;
    if (spriteLoadingRI.a >= 1.0f)
    {
        spriteLoadingRI.a = 1.0f;
        curScene = nextScene;
        curScene->OnActiveScene();
        isResetDeltaTime = true;
        nextScene = nullptr;
    }
}
```

4. Object 클래스

의도 | 게임 내 오브젝트를 통합적으로 관리하고, 프레임별 업데이트 및 충돌 처리를 쉽게 구성하고 하였고, 박스 및 원형 형태의 간단한 콜라이더 구조를 직접 구현하여 충돌 판정을 효율적으로 수행하고자 하였습니다. 또한 디버깅 시 충돌 영역을 빨간색으로 출력하여 조정하기 편하게 하였습니다.

방식 | 위치, 충돌 정보, 렌더링을 포함한 추상 클래스를 상속받아 구현하는 방식입니다. ObjectManager를 통해 객체 생명주기, 업데이트 루프, 충돌 검사, 렌더링을 수행합니다. 또한 CollisionBody를 통해 박스 및 원형 형태의 간단한 콜라이더 충돌을 지원합니다.

효과 | ObjectManager를 통해 게임 씬 내 수많은 오브젝트를 쉽고 간편하게 처리할 수 있습니다.

전체 코드: 바로가기

씬 등록 및 전환

```cpp
bool IsCollision(const CollisionBody& lhs, const CollisionBody& rhs)
{
    D3DXVECTOR2 dp = lhs.owner->pos - rhs.owner->pos;
    float dl = D3DXVec2Length(&dp);
    if (dl < lhs.circle.radius + rhs.circle.radius)
        return true;

    D3DXVECTOR2 min1 = lhs.owner->pos + lhs.aabb.min;
    D3DXVECTOR2 max1 = lhs.owner->pos + lhs.aabb.max;
    D3DXVECTOR2 min2 = rhs.owner->pos + rhs.aabb.min;
    D3DXVECTOR2 max2 = rhs.owner->pos + rhs.aabb.max;
    if (max1.x > min2.x && max1.y > min2.y &&
        max2.x > min1.x && max2.y > min1.y)
        return true;

    return false;
}
```

충돌 및 삭제 처리 루프

```cpp
for (auto it = objects.begin(); it != objects.end(); )
{
    (*it)->FixedUpdate(_fixedDeltaTime);
    if ((*it)->destroy)
    {
        delete* it;
        it = objects.erase(it);
    }
    else
        ++it;
}
```

디버그용 충돌 시각화

```cpp
if (body.circle.radius != 0.0f)
{
    D3DXVECTOR3 lines[37] = {};
    for (int i = 0; i < 37; ++i)
    {
        D3DXVECTOR2 vp = MathUtility::DegreeToVector2(i *
                         body.circle.radius) + body.owner->pos;

        lines[i] = D3DXVECTOR3(vp.x, vp.y, 0.0f);
    }
    Direct3D::GetInstance().p
}
```

5. Sprite, SpriteRI 클래스

의도 | 2D 게임이기 때문에, 이미지를 렌더링할 수 있게 Sprite 클래스를 제작하였습니다. 색상, 회전, 위치, 스케일, 피벗을 조절할 수 있는 RenderInfo(RI) 클래스를 추가적으로 구현하였으며, 마스킹 텍스쳐를 구현하여 특정 지역에 효과를 주도록 하였습니다.

방식 | 지정된 폴더의 텍스쳐를 프레임 단위로 불러오고, 애니메이션을 재생합니다. SpriteRI 클래스를 통해 스프라이트의 렌더 정보를 쉽게 조절할 수 있습니다.

효과 | SpriteRI 클래스를 수정해 다양한 트랜스폼 연출이 가능하며, 연출 및 애니메이션 작업의 유연성이 비약적으로 상승하였습니다.

전체 코드: 바로가기

스프라이트 애니메이션 처리

```
void Sprite::Update(float _deltaTime)
{
    aniDeltaTime += _deltaTime;
    if (aniDeltaTime > aniTime)
    {
        aniDeltaTime -= aniTime;
        if (++scene >= szScene)
        {
            if (aniLoop) scene = 0;
            else --scene;
        }
    }
}
```

충돌 및 삭제 처리 루프

```
void Sprite::UpdateMaskTexture()
{
    D3DLOCKED_RECT rect;
    if (FAILED(maskSrc->LockRect(0, &rect, nullptr, D3DLOCK_DISCARD))) return;

    unsigned char* pBits = static_cast<unsigned char*>(rect.pBits);
    for (int i = 0; i < maskHeight; ++i)
    {
        memcpy(pBits, &maskBit[i * maskWidth], sizeof(D3DCOLOR) * maskWidth);
        pBits += rect.Pitch;
    }
    maskSrc->UnlockRect();
}
```

스프라이트 애니메이션 처리

```
D3DXMATRIX matrix;

D3DXMatrixTransformation2D(&matrix,
                           &_ri.scaleCen,
                           0.0f,
                           &_ri.scale,
                           &_ri.rotateCen,
                           -D3DXToRadian(_ri.rotate),
                           &_ri.pos);

pd3dDevice->SetTransform(D3DTS_WORLD, &matrix);
pd3dDevice->SetTexture(0, pTexture->src);
pd3dDevice->SetTexture(1, maskSrc);
pd3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
```

2 유틸리티 코드

1. Sound, SoundManager 클래스

의도 | 게임 내 효과음 및 배경음을 지원하기 위해 사운드 시스템을 직접 구현하였습니다.

방식 | Windows MCI를 사용하여 .wav 등의 사운드 파일을 직접 로드하고 재생합니다. SoundManager를 통해 사운드 생성/재생/정지/볼륨 설정 등을 전역적으로 관리할 수 있습니다.

효과 | 충돌음, 공격음 등 여러 효과음들이 재생되더라도 큰 문제 없이 재생됩니다.

전체 코드: 바로가기

사운드 인스턴스 로드 및 초기화

```
for (int i = 0; i < _szSound; ++i)
{
    WCHAR buffer[256];
    swprintf_s(buffer, TEXT("open %s alias %d_%d"), _pathWithName.c_str(), soundIDX, i);
    mciSendString(buffer, 0, 0, 0);
    swprintf_s(buffer, TEXT("play %d_%d from 0"), soundIDX, i);
    mciSendString(buffer, 0, 0, 0);
    swprintf_s(buffer, TEXT("pause %d_%d"), soundIDX, i);
    mciSendString(buffer, 0, 0, 0);
}
```

반복 재생 및 순차 재생 처리

```
void Sound::Play(bool _loop)
{
    WCHAR buffer[255];
    swprintf_s(buffer, L"play %d_%d from 0%s", soundIDX, curSound, _loop ? L" repeat" : L"");
    mciSendString(buffer, 0, 0, 0);
    if (++curSound >= szSound) curSound = 0;
}
```

2. Input 클래스

의도 | 키보드 입력 상태를 “눌림”, “유지“, “뗌”을 구분하여 관리하는 시스템을 구축하려는 의도로 제작하였습니다.

방식 | InputState 열거형을 사용하여 각 키의 상태를 DOWN, PRESS, UP, NONE으로 구분하며 Main.cpp의 MsgProc 메서드에서 매 프레임 업데이트 됩니다.

효과 | 입력 이벤트에 따라 정확한 상태 판별이 가능하며, 프레임 단위로 입력이 갱신됩니다.

전체 코드: 바로가기

입력 상태 갱신

```cpp
void Input::KeyUpdate()
{
    for (int i = 0; i < 256; ++i)
    {
        if (keyStates[i] == InputState::DOWN)
            keyStates[i] = InputState::PRESS;

        if (keyStates[i] == InputState::UP)
            keyStates[i] = InputState::NONE;
    }
}
```

실제 사용 사례

```cpp
if (Input::GetInstance().GetKeyState(VK_F4) == InputState::DOWN)
{
    SceneManager::GetInstance().ChangeScene(TEXT("TitleScene"));
}

if (Input::GetInstance().GetKeyState(VK_F5) == InputState::DOWN)
{
    SoundManager::GetInstance().Play(TEXT("Assets/SoundFX/menu_click.mp3"));
    if (stage == 1)
    {
        SceneManager::GetInstance().AddScene(TEXT("GameScene"), new GameScene(0));
        SceneManager::GetInstance().ChangeScene(TEXT("GameScene"));

}
```

3 맵 코드

1. Map 클래스

의도 | 게임의 필드를 픽셀 단위로 구성하고, 감염/치유 영역을 표현하기 위해 각 픽셀마다 데이터를 직접 집어 넣었습니다. 또한 마스크 텍스처 조작을 사용해 치유되는 효과를 가시적으로 보이게 하였고 FloodFill 알고리즘을 사용해서 감염 구역을 점령해 나가는 시스템을 만드려는 의도입니다.

방식 | Sprite 클래스의 마스크 텍스쳐 기능을 사용하였으며, PixelState, BoundaryPixelState 열거형을 기반으로 각각의 상태에 대한 색상을 설정해두고 직접 그 픽셀을 수정합니다. FloodFill 알고리즘을 멀티 스레드로 분산 처리하여 더욱 빠르게 작동할 수 있도록 하였습니다.

효과 | 갈스 패닉(Gals Panic) 모작 게임에 알맞은 영역 수복 / 경로 기반 색칠 / 경계 충돌 등 여러 기능을 사용할 수 있습니다.

전체 코드: 바로가기

영역 설명 이미지

Flood Fill 핵심 로직

```
while (!nodes.empty())
{
    auto node = nodes.front();
    nodes.pop();

    if (int(scene->pBoss->pos.x) == node.x && -int(scene->pBoss->pos.y) == node.y)
        return;

    auto fPosition = D3DXVECTOR2(float(node.x), float(node.y));
    if (IsOutBound(fPosition))
        continue;

    if (_colors[node.y * mapWidth + node.x] == colors[int(PixelState::OPEN)])
    {
        if (_boundaryColors[node.y * mapWidth + node.x] == boundaryColors[int(BoundaryPixelState::BOUNDARY)] ||
            _boundaryColors[node.y * mapWidth + node.x] == boundaryColors[int(BoundaryPixelState::BOUNDARYEND)])
        {
            _boundaryColors[node.y * mapWidth + node.x] = boundaryColors[int(BoundaryPixelState::NONE)];
        }
        continue;
    }

    ++fillCount;
    _colors[node.y * mapWidth + node.x] = colors[int(PixelState::OPEN)];
    for (int i = 0; i < _countof(DirectionWay8); ++i)
        nodes.push({ DirectionWay8[i].x + node.x, DirectionWay8[i].y + node.y });
}
```

4 엔티티 코드

1. Entity 클래스

의도 | 맵 위에서 동작하는 모든 Entity를 정의하는 클래스로, 픽셀 한 칸 한 칸 각각 이동 처리를 진행하도록 하였으며 이 클래스를 상속받아 플레이어, 적, 오브젝트, NPC 등 다양한 Entity를 쉽게 정의할 수 있도록 하였습니다.

방식 | 한 칸씩 이동을 처리하게 하였으며, 맵 경계 검사, 픽셀 상태 및 경계 상태를 기반으로 이동 가능 여부를 파악하도록 설계하였습니다. 또한 1 픽셀씩 움직이게하여 예상치 못한 동작을 방지하였습니다.

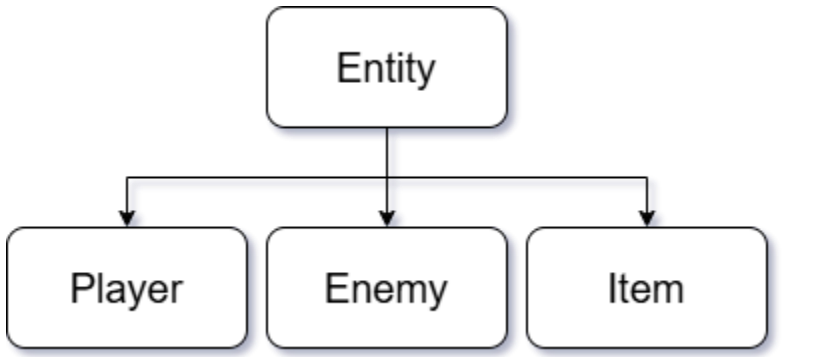
효과 | Entity를 상속하는 적, 오브젝트, NPC 등 다양한 Entity를 더욱 쉽고 체계적으로 구현할 수 있게 되었습니다.

전체 코드: 바로가기

방향 기반 한 칸 이동 및 충돌 검사

```cpp
void Entity::FrameMove(const D3DXVECTOR2& _direction)
{
    auto scene = static_cast<GameScene*>(SceneManager::GetInstance().GetActiveScene());
    for (int i = 0; i < MinFrameInterval; ++i)
    {
        FrameMoveResult ret;
        ret.np = pos + _direction;

        for (auto& offset : moveCollisionOffsets)
        {
            auto np = ret.np + offset;
            if (scene->pMap->IsOutBound({ np.x, -np.y }))
                return;

            FrameMoveCondition condition;
            condition.cpbps = scene->pMap->GetBoundaryPixelState({ pos.x, -pos.y });
            condition.cpps = scene->pMap->GetPixelState({ pos.x, -pos.y });
            condition.npbps = scene->pMap->GetBoundaryPixelState({ np.x, -np.y });
            condition.npps = scene->pMap->GetPixelState({ np.x, -np.y });
            condition.np = np;

            if (IsFrameMove(condition, ret) == false)
            {
                return;
            }
        }

        OnFrameMove(ret);
        pos = ret.np;
    }
}
```

Entity 상속 구조



2. Player 클래스

의도 | 플레이어가 경계선을 따라 이동하며 감염을 치유, 안전한 구역을 확장해야하는 게임의 핵심 메커니즘을 구현하려고 하였으며, 그 외 아이템 효과 및 무적 상태 또한 이 클래스에서 관리할 수 있도록 하였습니다.

방식 | 맵 위 픽셀 단위로 이동하며, 현재 위치와 다음 위치의 맵 상태에 따라서 이동 허용 여부를 판단합니다.아이템 효과에 따라 스프라이트 연출이 일정 시간 유지되도록 하였습니다.

효과 | 플레이어의 입력에 맞추어 플레이어 조작을 자연스럽게 진행할 수 있습니다. 또한 아이템 효과에 따른 연출을 스프라이트를 통해 플레이어가 직관적으로 효과를 알아차릴 수 있습니다.

전체 코드: 바로가기

Flood Fill 트리거

```
if (_condition.npbps == Map::BoundaryPixelState::BOUNDARYEND ||
    _condition.npbps == Map::BoundaryPixelState::BOUNDARY)
{
    scene->pMap->CalcFloodFill();
    return true;
}
```

효과에 따른 스프라이트 연출

5 후기

‘Virus Striker’ 게임은 고등학교에 입학하여 처음으로 C, C++ 언어를 배우며 만들어 본 게임입니다. 이 프로젝트는 단순한 경진대회 출품작이 아닙니다.
저에게 있어서는 적어도 게임 개발이라는 세계에 처음으로 발을 들이게 해준 프로젝트입니다.

DirectX API를 사용해 2D 렌더링부터 충돌 처리, 애니메이션 구현까지 직접 처리하며 게임이 구동되기 위한 하부 시스템과 구조를
직접 구현해보며 게임 프레임워크, 게임 엔진이라는 개념에 대해 실질적으로 체험할 수 있었습니다. 이후 유니티 엔진을 공부할 때 크게 도움이 되었다고 생각합니다.

지금 돌아보면 ‘Virus Striker’는 비록 짧은 시간 내에 제작할 수 있는 작품 중 하나이지만, 그 당시 저의 순수한 열정과 도전 정신이 가득 담긴 작품입니다.
가끔 이 작품을 제작했을 무렵을 상상하면 당시의 감정이 아직도 새록새록 떠올라 묘한 기분을 느끼게 됩니다.

VIRUS STRIKER

포트폴리오에 대한 모든 의견을 존중하며, 오늘도 사랑스런 하루 보내세요!

Email: lsnan421@naver.com
깃허브: 바로가기
유튜브: 바로가기
노　션: 바로가기

작성 일지: 2025. 03. 31