포인트 블랭크는 미리 컴파일된 바이너리 파일을 배포하여 라이브 서비스 중에 있다. Uber 셰이더 방식을 사용하기 때문에 바이너리 파일에는 셰이더들이 조합되어 저장되어 있다.

미리 컴파일되어 있기 때문에 속도적으로 굉장한 이득이 있다. 반면 경우의 수대로 조합되어야 하기 때문에 어떤 경우에는 누락되어 호출하지 못하는 상황이 발생한다. 이 경우 실시간으로 셰이더를 컴파일 시도하고 실패할 경우 조합 상황을 줄여 호출을 시도한다.

코드를 살펴보니 실시간으로 셰이더를 컴파일하는 경우 대상 타깃은 HLSL 파일이 아닌 소스 코드에 박혀 있는 16진수 값 배열이다. 아마도 보안 문제 때문에 이런 방식을 사용한 것으로 보인다. 꼭 이렇게 해야 했나. 개인적으로는 HLSL 파일도 리소스로 간주하고 배포했으면 하는 아쉬움이 있다.

코드에 박혀 있는 16진수 값 배열을 살펴 본다. 초기 개발 때 넣었던 일부 셰이더들만 보이고 최신으로 업데이트 되어 있지도 않은 상태이다. 그리고 컴파일도 되지 않는다. 컴파일 되지 않는 이유는 #include 문 때문이다. 아래에 자세히 설명하도록 하겠다.

```
static SBIN_TABLE s_ShaderBinTable[] =
{
    {   g_i3Common,             sizeof(g_i3Common)},
    {   g_i3Reflection,         sizeof(g_i3Reflection) },
    {   g_i3ShadowMap,          sizeof(g_i3ShadowMap) },
    {   g_i3DirectionalLight,   sizeof(g_i3DirectionalLight)},
    {   g_i3LuxMap,             sizeof(g_i3LuxMap)},
    {   g_i3Phong_LightModel,   sizeof(g_i3Phong_LightModel)},
    {   g_i3HSL_LightModel,     sizeof(g_i3HSL_LightModel) },
    {   g_i3GI1_LightModel,     sizeof(g_i3GI1_LightModel) },
    {   g_i3Custom_LightModel,  sizeof(g_i3Custom_LightModel) },
    {   g_i3PointLight,         sizeof(g_i3PointLight)},
    {   g_i3SpotLight,          sizeof(g_i3SpotLight)},
    {   g_i3VertexShader_PPL,   sizeof(g_i3VertexShader_PPL) },
    {   g_i3VertexShader_PVL,   sizeof(g_i3VertexShader_PVL) },
    {   g_i3PixelShader_PPL,    sizeof(g_i3PixelShader_PPL) },
    {   g_i3PixelShader_PVL,    sizeof(g_i3PixelShader_PVL) },
    {   g_i3SuperShader,        sizeof(g_i3SuperShader)},

    {   NULL,                   0   }
};
```

실제 값은 HLSL 파일을 파싱한 암호화된 16진수 데이터로 저장되어 있다.

static const unsigned char g_i3SuperShader[] = {0x1B, 0x4B, 0x31, 0x01, 0x0B ... };

많이 황당했지만 기존 로직을 따르되 좀 더 자동화 할 수 있는 방법으로 개선을 시도했다. 별도 툴을 만들어 CPP 파일을 자동으로 뽑아낸다. 툴은 HLSL 파일을 검색하여 결과를 도출한다. 우선 암호화된 코드값들은 CPP 변수로 출력된다.

```
namespace Ocean
{
    static const BYTE g_encryptedCode[] = {0x1B, 0x23, 0x2B, 0x33, 0x4B, 0x73, 0x29, 0x02, 0x49, 0x9A, 0x6
}

namespace Ocean_Flow
{
    static const BYTE g_encryptedCode[] = {0x1B, 0x23, 0x2B, 0x33, 0x4B, 0x73, 0x29, 0x02, 0x49, 0x9A, 0x6
}

namespace Wallhack_Protect
{
    static const BYTE g_encryptedCode[] = {0x1B, 0x4B, 0x31, 0x01, 0x0B, 0x23, 0x2B, 0x33, 0x4B, 0x73, 0x2
}
```

이 변수들은 별도 저장 및 검색을 위해 별도 클래스에서 관리한다.

```
_Register("i3BitBlit.hlsl", i3BitBlit::g_encryptedCode, sizeof(i3BitBlit::g_encrypte
_Register("i3Blend.hlsl", i3Blend::g_encryptedCode, sizeof(i3Blend::g_encryptedCode)
_Register("i3BloomCombine.hlsl", i3BloomCombine::g_encryptedCode, sizeof(i3BloomComb
_Register("i3BrightPathExtract.hlsl", i3BrightPathExtract::g_encryptedCode, sizeof(i
_Register("i3CalculateAdaptedLumPS.hlsl", i3CalculateAdaptedLumPS::g_encryptedCode,
_Register("i3CelShading.hlsl", i3CelShading::g_encryptedCode, sizeof(i3CelShading::g
_Register("i3Common.hlsl", i3Common::g_encryptedCode, sizeof(i3Common::g_encryptedCo
_Register("i3Custom_LightModel.hlsl", i3Custom_LightModel::g_encryptedCode, sizeof(i
_Register("i3DefaultChainShader.hlsl", i3DefaultChainShader::g_encryptedCode, sizeof
_Register("i3DirectionalLight.hlsl", i3DirectionalLight::g_encryptedCode, sizeof(i3D
_Register("i3DownScale.hlsl", i3DownScale::g_encryptedCode, sizeof(i3DownScale::g_en
_Register("i3EdgeDetection.hlsl", i3EdgeDetection::g_encryptedCode, sizeof(i3EdgeDet
_Register("i3Gamma.hlsl", i3Gamma::g_encryptedCode, sizeof(i3Gamma::g_encryptedCode)
_Register("i3GaussianBlur.hlsl", i3GaussianBlur::g_encryptedCode, sizeof(i3GaussianB
```

셰이더 컴파일 시점에는 원본 값이 필요하기 때문에 복호화하여 스트링으로 반환한다.

```
I3_EXPORT_GFX
i3::string GetShaderMemoryCode(const char* shaderFileName)
{
    return g_ctx.GetDecryptedCode(shaderFileName);
}
```

이 스트링을 D3DXCompileShader 함수에 전달하여 컴파일을 시도한다(pszHLSL).

```
            &pInst, &pErrorMsg, &pConstTable);*/

            hRv = D3DXCompileShader(pszHLSL, sz, pDef, GetShaderSourceCodeMgr()->GetD3DXInclude(), "VS_Def",
                (LPCSTR)s_szProfile[i], flags,
                &pInst, &pErrorMsg, &pConstTable);
            break;

        case I3G_SHADER_TYPE_PIXEL :
            /*hRv = ::D3DXCompileShader( pszHLSL, sz, pDef, NULL, "PS_Def",
            (LPCSTR) s_szProfile[ i], flags,
            &pInst, &pErrorMsg, &pConstTable);*/

            hRv = D3DXCompileShader(pszHLSL, sz, pDef, GetShaderSourceCodeMgr()->GetD3DXInclude(), "PS_Def",
                (LPCSTR)s_szProfile[i], flags,
                &pInst, &pErrorMsg, &pConstTable);
```

pszHLSL 코드는 include 문이 들어있다. 아래와 같은 식으로.

#include “i3Common.hlsl”

#include “i3ShadowMap.hlsl”

…

DirectX 는 include 문을 인식하지 못하기 때문에 컴파일을 실패하게 된다. 따라서 해당 include 줄을 실제 셰이더 코드로 대체해야 한다. ID3DXInclude 가 이런 일을 가능하게 한다. D3DXCompileShader 함수의 4번째 인자이다. ID3DXInclude 인터페이스에 따라 구현한다.

```
class IShaderSourceCodeMgr;
struct i3ShaderDXInclude : ID3DXInclude
{
    i3ShaderDXInclude(IShaderSourceCodeMgr* pOwner) : m_pOwner(pOwner) {}
    virtual ~i3ShaderDXInclude() {}

    virtual COM_DECLSPEC_NOTHROW HRESULT STDMETHODCALLTYPE Open(D3DXINCLUDE_TYPE type, LPCSTR pszName, LPCVOID pParentData, LPCVOID* ppData, UINT* pBytes
    virtual COM_DECLSPEC_NOTHROW HRESULT STDMETHODCALLTYPE Close(LPCVOID pData);

private:
    IShaderSourceCodeMgr* m_pOwner;
};
```

Open 함수와 Close 함수를 오버라이딩하여 구현한다. Open 함수에서는 include 문의 셰이더 실 코드를 메모리에 할당하여 전달한다. 할당한 메모리는 Close 함수에서 삭제 처리한다.

```
COM_DECLSPEC_NOTHROW HRESULT STDMETHODCALLTYPE i3ShaderDXInclude::Open(D3DXINCLUDE_TYPE type, LPCSTR pszName, LPCV
{
    i3::string code = m_pOwner->GetSourceCode(pszName);

    const size_t size = code.size();
    if (size == 0)
    {
        *ppData = nullptr;
        *pBytes = 0;
        return S_OK;
    }

    BYTE* pData = new BYTE[size];
    if (!pData)
        return E_OUTOFMEMORY;

    i3::copy(code.begin(), code.end(), pData);

    *ppData = pData;
    *pBytes = size;

    return S_OK;
}

COM_DECLSPEC_NOTHROW HRESULT STDMETHODCALLTYPE i3ShaderDXInclude::Close(LPCVOID pData)
{
    BYTE* pBuff = (BYTE*)pData;
    if (pBuff)
        delete[] pBuff;

    return S_OK;
}
```

실시간 컴파일 시 속도가 약간 끊기긴 하지만 게임에 지장을 줄 정도의 수준은 아니다. 크래시 발생하는 것보다는 훨씬 낫다.