DirectX Effect Tool

포트폴리오 : DirectX API와 MFC를

이용한 이펙트 툴 제작

제작 기간: 25일

작 성 자 : 오민규

< Contents >

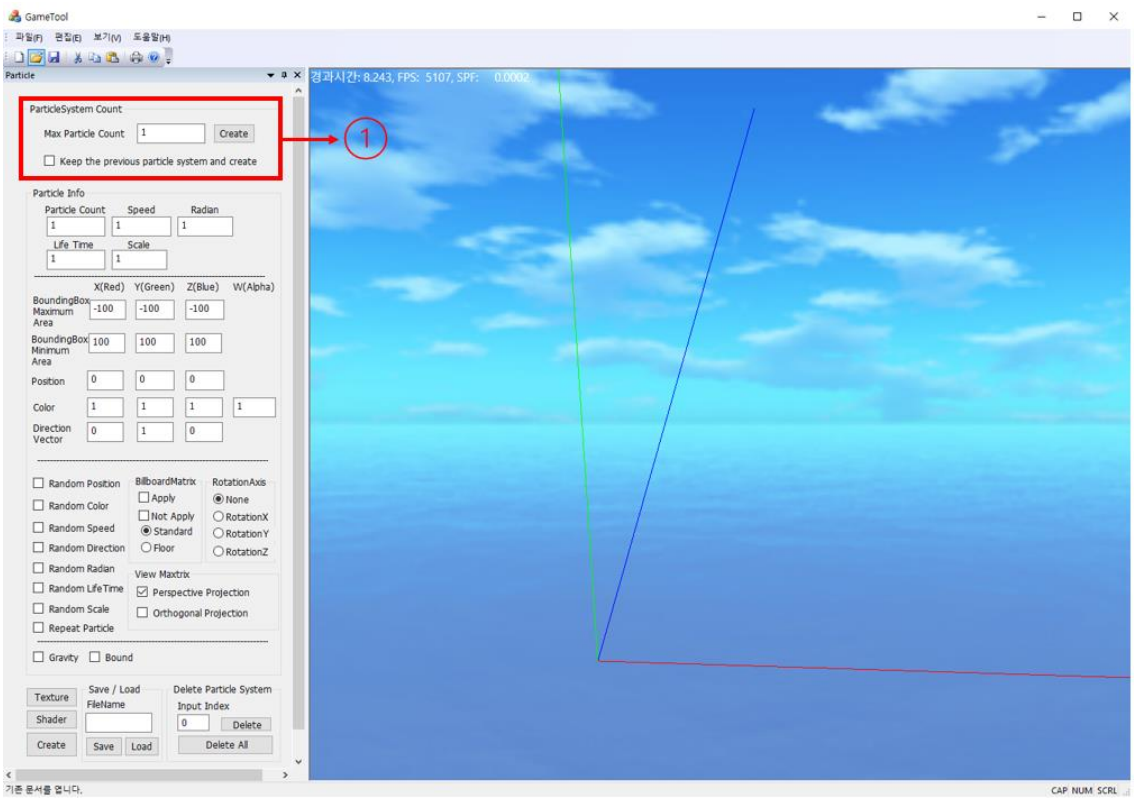
1. 이펙트 툴 개요

1) 이펙트 툴 소개

DirectX 3D 게임을 제작할 때, 게임 프로젝트에서 직접 이펙트를 생성하는 것이 아닌 **게임에서 사용되는 이펙트 리소스를 기획에 맞춰서 제작하기 위하**여 **DirectX API와 MFC를 이용하여 이펙트 툴을 제작**한다. **위치, 방향, 속도 등 특징을 버튼과 텍스트 입력 박스에 값을 넣어 이펙트를 제작**한다. 텍스처는 단일 이미지, 다중 이미지(최대 4개), 스프라이트 이미지 모두 적용이 가능하다. 다양한 연출을 위해 **중력, 충돌, 반복, 회전** 등 다양한 기능 탑재되어있다. 툴을 이용하여 원하는 **이펙트 리소스**를 제작했다면 텍스트 파일로 데이터를 **저장**할 수 있으며, 추후 변경 사항이 필요할 때 툴로 **데이터 파일을 읽어와 이펙트를 변경할 수 있다.**

2) 이펙트 툴 사용방법

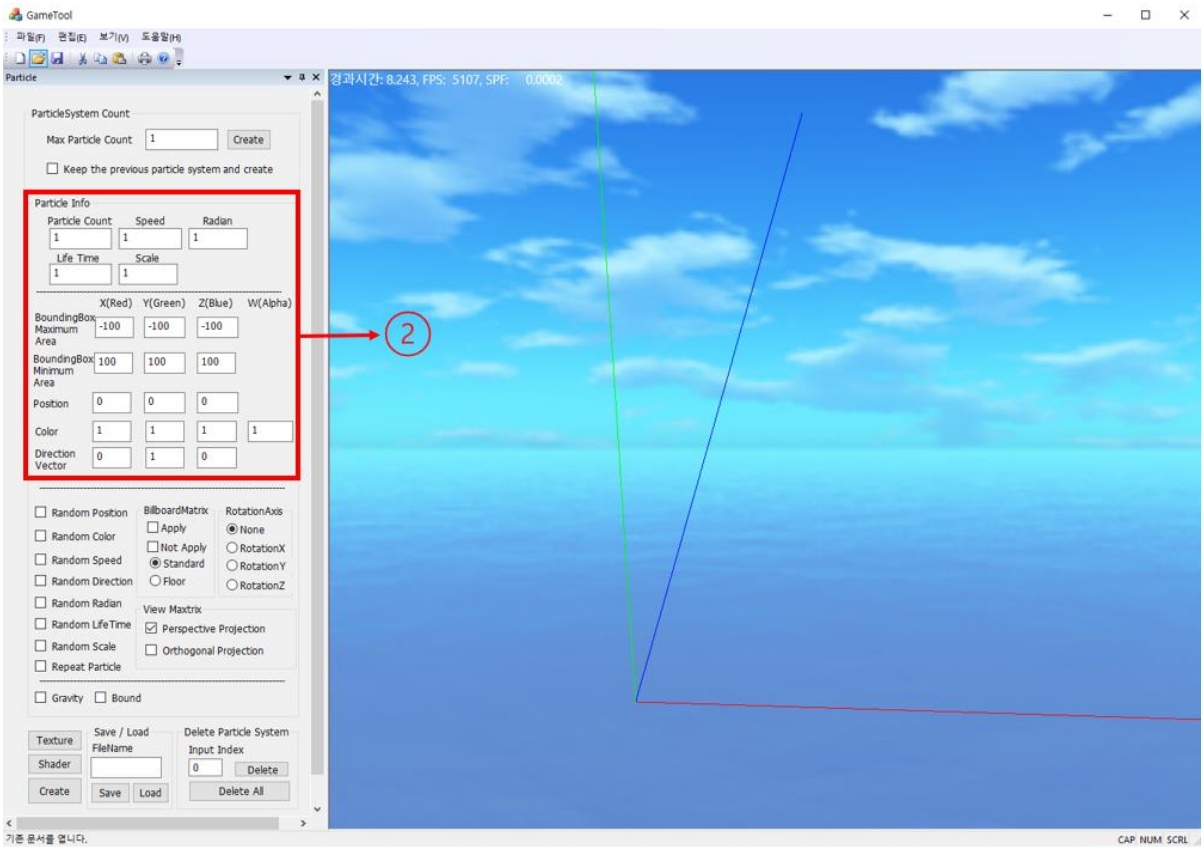
① 파티클 시스템 개수 지정



[그림 1-1] 파티클 시스템 생성

생성할 파티클 시스템의 개수를 입력받고 생성한다. **파티클 시스템은 최소 30 프레임으로 유지하기 위해 최대 1000 개까지 생성할 수 있도록 제한**한다. 만약 10 개의 파티클 시스템을 생성하였고 하단 **체크박스를 체크**한다면 **기존에 생성되어 있던 파티클 시스템은 유지**되며 **(10 – 기존 파티클 시스템 개수) 개의 파티클 시스템을 추가로 생성할 수 있다.**

② 이펙트 기본 속성



[그림 1-2] 이펙트 기본 속성

해당 부분은 파티클 시스템 생성에 가장 중요한 속성값을 지정한다. 한 개의 파티클 시스템에서 **파티클 개수, 속도, 위치, 파티클 생존 시간, 파티클 크기, 색상, 방향, 바운딩박스 크기(최소, 최대)**를 정할 수 있다.

- **Particle Info**

  파티클 정보 입력 속성에서는 **파티클 수, 속도, 이동 반경(반지름), 생존 시간, 크기**를 정할 수 있다. 여기서 지정한 파티클 속성값은 아래의 **다양한(랜덤) 파티클 속성을 적용할 때 기준 수치**가 된다. 예를 들어 크기 값을 10 으로 적용하고 아래의 Random Scale 을 선택하면, 1 부터 10 사이의 크기가 각각

파티클에 랜덤으로 적용된다. 생존 시간(LifeTime)의 경우는 -1 일 때, 프로세스가 종료되거나 파티클을 삭제하지 않는다면, 생성된 파티클이 계속 실행된다. N(임의의 양의 수)를 입력한다면, N 초 만큼 실행된다.

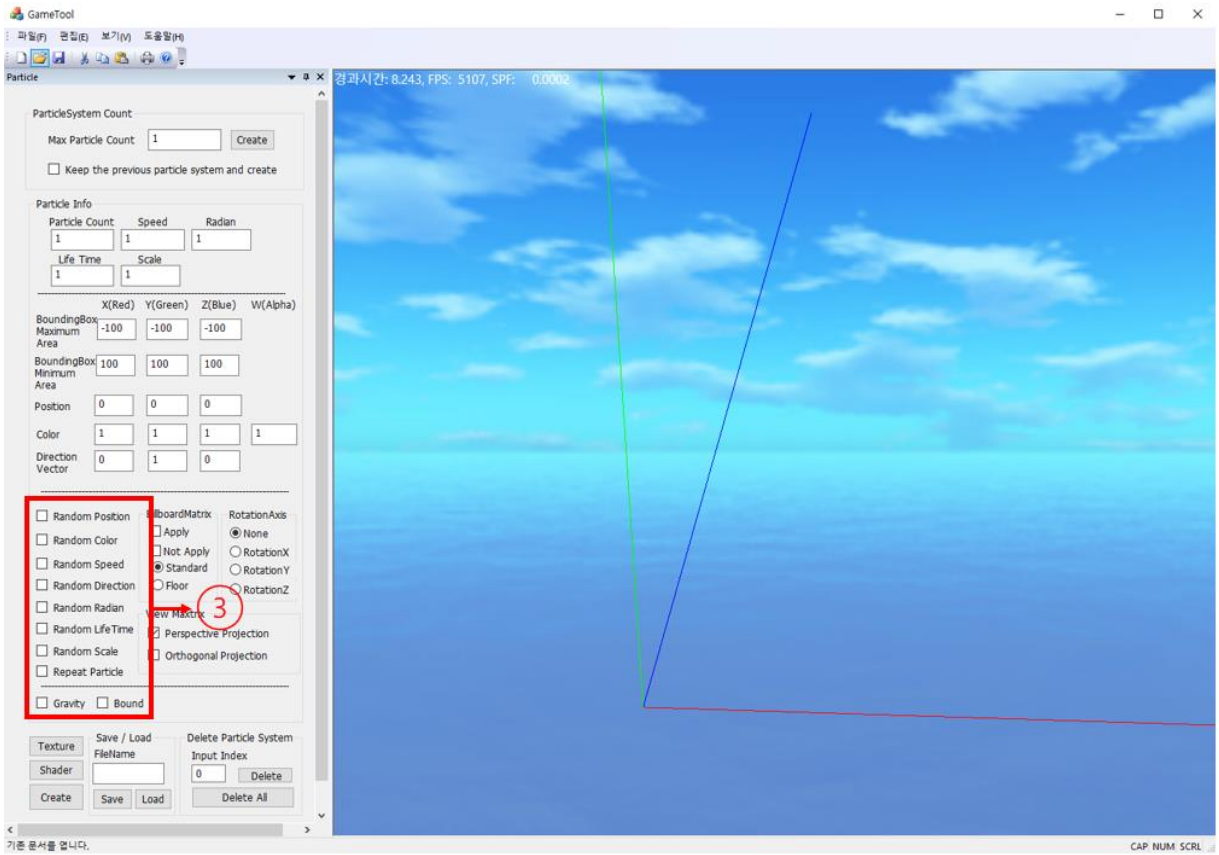
- **BoundingBox Maximum/Minimum Area**

  파티클이 **생성되는 범위와 존재할 수 있는 범위**를 지정한다. 파티클이 생성된 바운딩 박스를 벗어나게 되면 삭제된다. 만약 특정 시간 동안 **반복되는 파티클 속성이 적용되어 있다면 생성된 위치로 다시 복귀하여 파티클이 실행**된다.

- **Position, Color, Direction**

  파티클이 생성되는 **위치, 색상, 방향**을 정한다. 위치의 경우 파티클의 개수가 1000 개일 때, 별다른 속성값이 존재하지 않는다면 1000 개의 파티클은 모두 같은 위치에서 생성된다.

③ 이펙트 랜덤 속성 및 특성



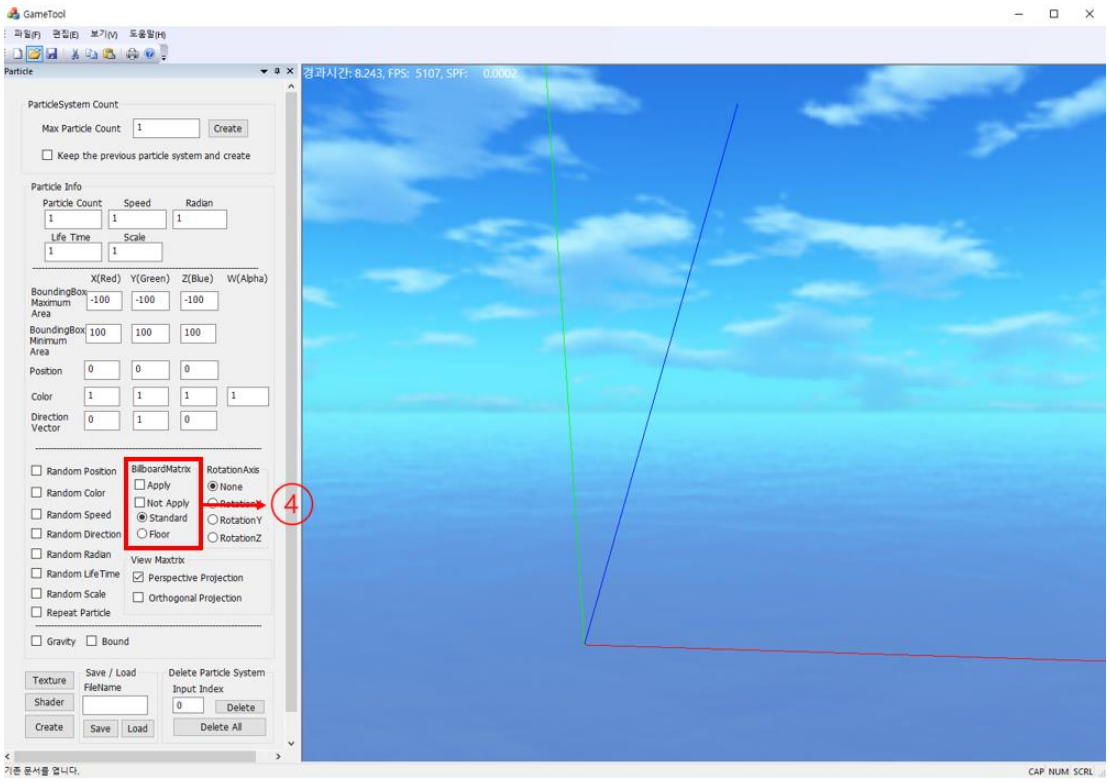
[그림 1-3] 이펙트 랜덤 속성 설정 및 특성

파티클의 속성인 **위치, 색상, 속도, 방향, 이동 반경(반지름), 생존 시간, 크기를 랜덤으로 설정**할 수 있다. 속도, 이동 반경(반지름), 크기 랜덤값의 **최솟값**은 **1** 이고, **최댓값**은 **위에서 적용한 속성 값**이다. 랜덤 위치를 선택한 경우 파티클의 위치는 바운딩 박스 최소, 최대값 사이의 위치에서 생성된다. 이외의 색상은 RGB(0~255, 0~255, 0~255)로, 방향은 (x,y,z) = (0~1, 0~1, 0~1)의 값이 적용된다. 다음은 파티클 특성을 나타낸다.

특성	내용
Repeat Particle	어떤 운동을 하며 움직이는 파티클이 있을 때, 이 설정을 **비활성화**한다면, **바운딩 박스에서 벗어나게 되었을 때, 파티클은 삭제**된다. 하지만 이 특성을 **활성화**한다면 바운딩 박스에서 벗어나도 파티클은 삭제되지 않고 **원래 생성되었던 위치로 돌아간다.**
Gravity	이펙트에 **중력**을 적용한다.
Bound	운동하는 파티클이 **바운딩 박스와 충돌이 발생하면 현재 방향의 반대 방향으로 운동을 진행**한다.

[표 1-1] 이펙트 특성

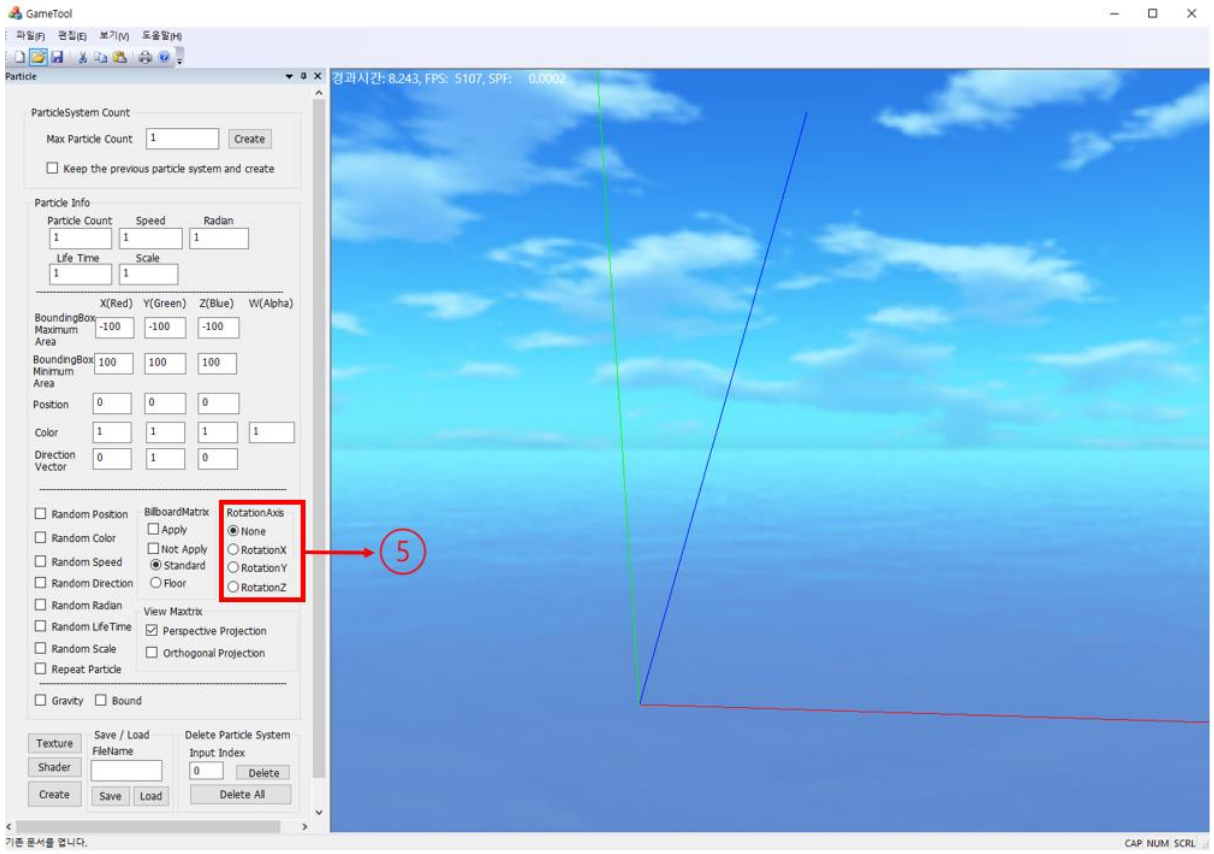
④ 이펙트 렌더링



[그림 1-4] 이펙트 랜덤 렌더링 타입 설정

이펙트 렌더링 방식을 선택한다. **빌보드 행렬을 적용 유무**를 선택하며, 만약 빌보드 행렬을 적용하지 않는다면, 바닥에 붙어 있는 이펙트를 생성할지 스크린 위에 정면으로 렌더링 되는 이펙트를 생성할지 선택한다.

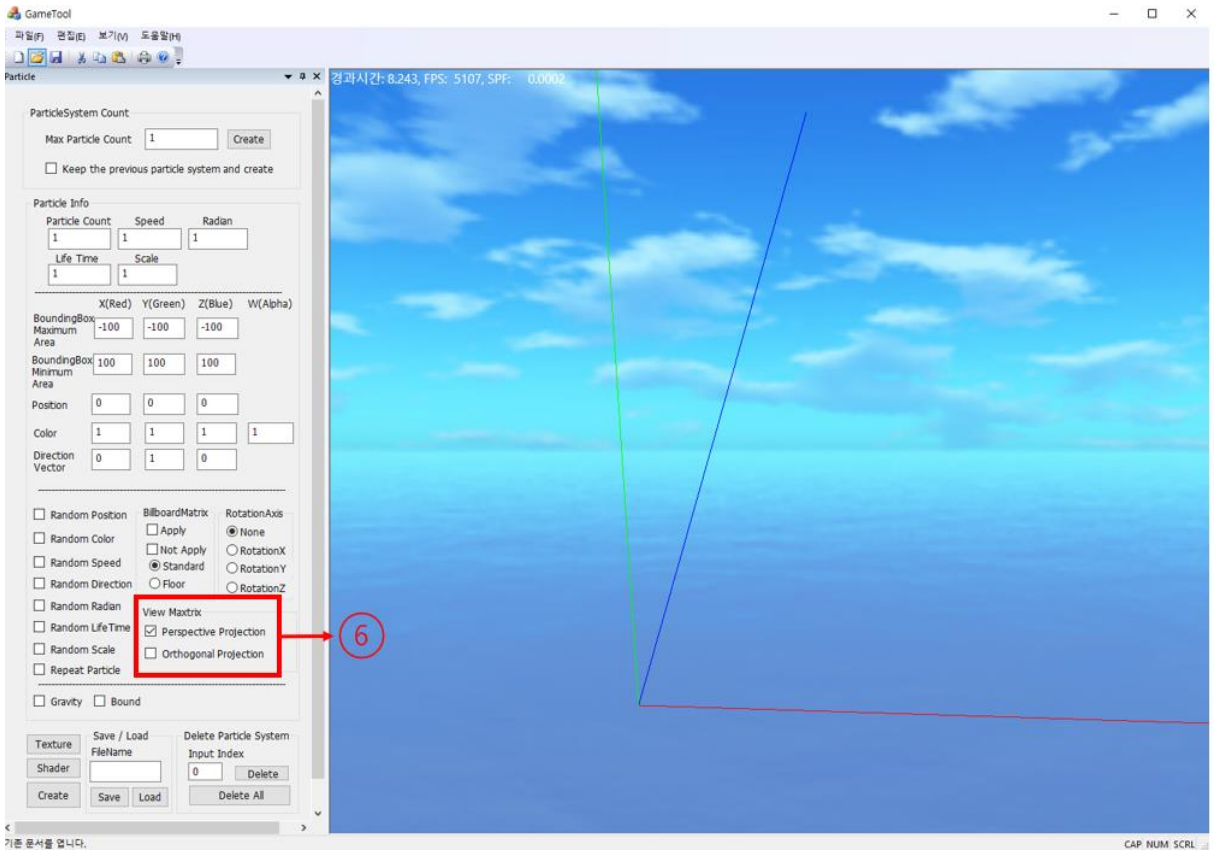
⑤ 이펙트 회전



[그림 1-5] 이펙트 회전 설정

회전이 필요한 이펙트를 제작할 때, 이 속성을 설정하여 **회전**을 **적용**할 수 있다. **현재 툴에서는 한 축(x,y,z) 회전만 선택**할 수 있도록 강제하였다. 추후 업데이트를 통해 사용자가 원하는 방향으로 회전할 수 있도록 적용하고자 한다.

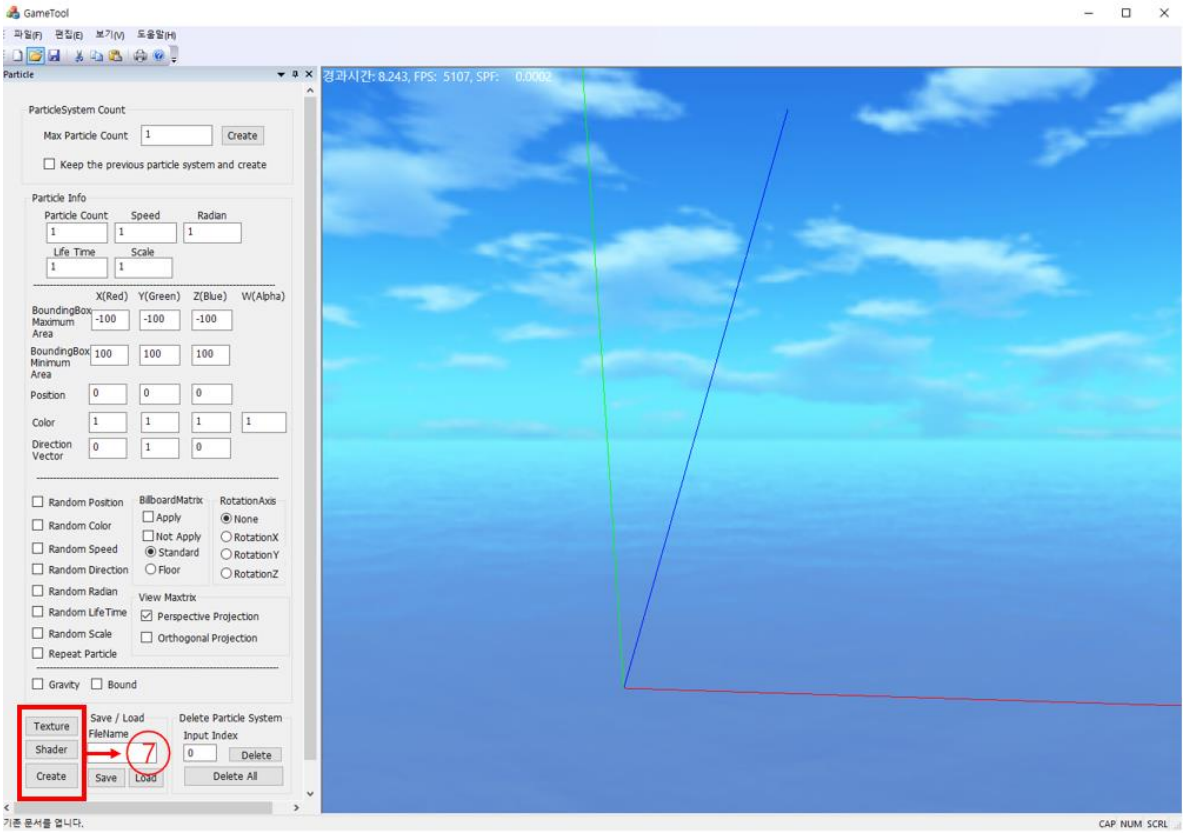
⑥ 이펙트 타입(월드 이펙트, 화면 이펙트) 설정



[그림 1-6] 이펙트 타입 설정

게임 세계에서만 이펙트가 존재하는 것은 아니다. 보통 많은 이펙트가 게임 세계에서 존재하지만 게임 플레이어가 사용하는 **모니터 전체에 효과를 내는 이펙트**도 존재한다. 따라서, 툴의 기본 설정은 월드 공간의 이펙트를 생성하도록 설정되어 있지만, **Othogonal Projection 속성을 선택하게 되면, 화면 이펙트를 제작**할 수 있다.

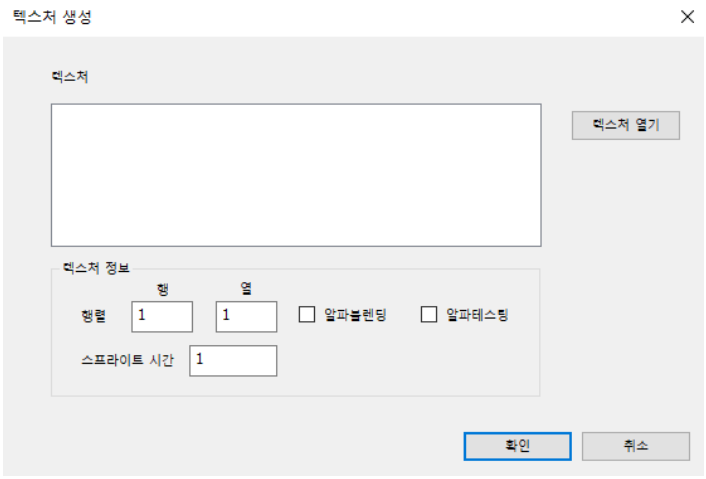
⑦ 이펙트 텍스처/셰이더 적용 및 생성



[그림 1-7] 이펙트 텍스처 및 셰이더 적용 및 생성

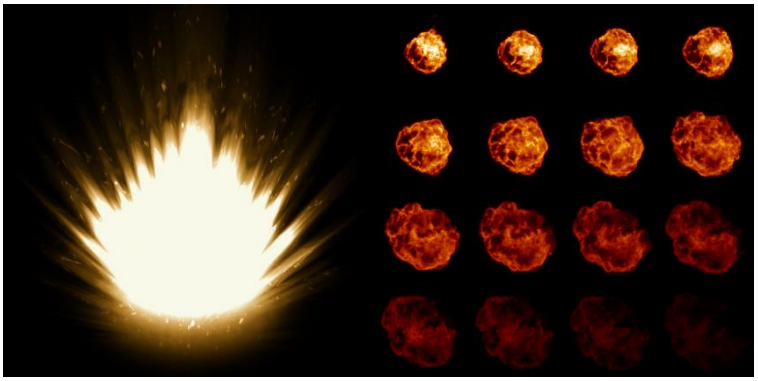
생성하고자 하는 이펙트의 모든 속성을 정하고 난 후 화면에 렌더링 될 **이미지**와 사용할 **셰이더**를 적용해야 한다. 텍스처와 셰이더 버튼을 클릭하면 모달이 생성된다.

- Texture



[그림 1-7-1] 텍스처 모달

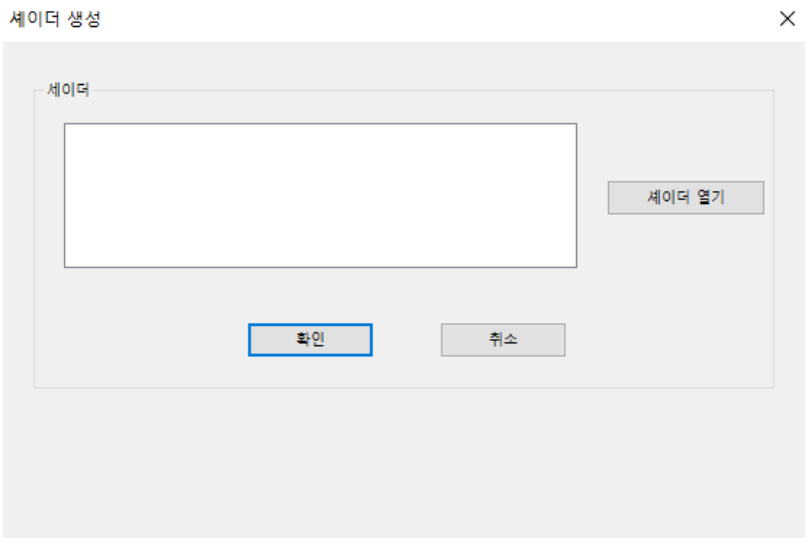
텍스처 모달에서 파일에 존재하는 이미지를 선택할 수 있다. 텍스처 기능 중에는 **멀티 텍스처(최대 4 개), 스프라이트 이미지 적용**이 **가능**하다. 하나의 이펙트에 최대 4 개의 텍스처가 적용이 가능하므로 여러 개의 이미지로 더욱 화려한 이펙트 1 개를 제작할 수 있다. 스프라이트 이미지의 경우 가로, 세로 개수를 입력해야한다. 다음은 1 개의 그림과 16 개의 그림이 존재하는 텍스처가 있다.



[그림 1-7-2] 텍스처 및 스프라이트

이펙트에 첫 번째 텍스처를 적용한다면 행렬 정보에 (1,1)을 입력하여 사용하고, 두 번째 텍스처를 적용한다면 (4, 4)를 입력하여 사용한다. 텍스처의 배경이 알파 값이 존재하지 않다면 행렬 입력란의 우측에 보이는 **알파 블렌딩 또는 알파 테스팅을 적용하여 사용하면 텍스처의 배경은 투명 처리되어 생성**된다. 마지막으로 **스프라이트 시간 속성**은 사용자가 **스프라이트 애니메이션 속도를 원하는 속도로 제어**할 수 있도록 제공한다. (1 = 1 초 동안 16 장의 스프라이트가 애니메이션 된다.)

- Shader



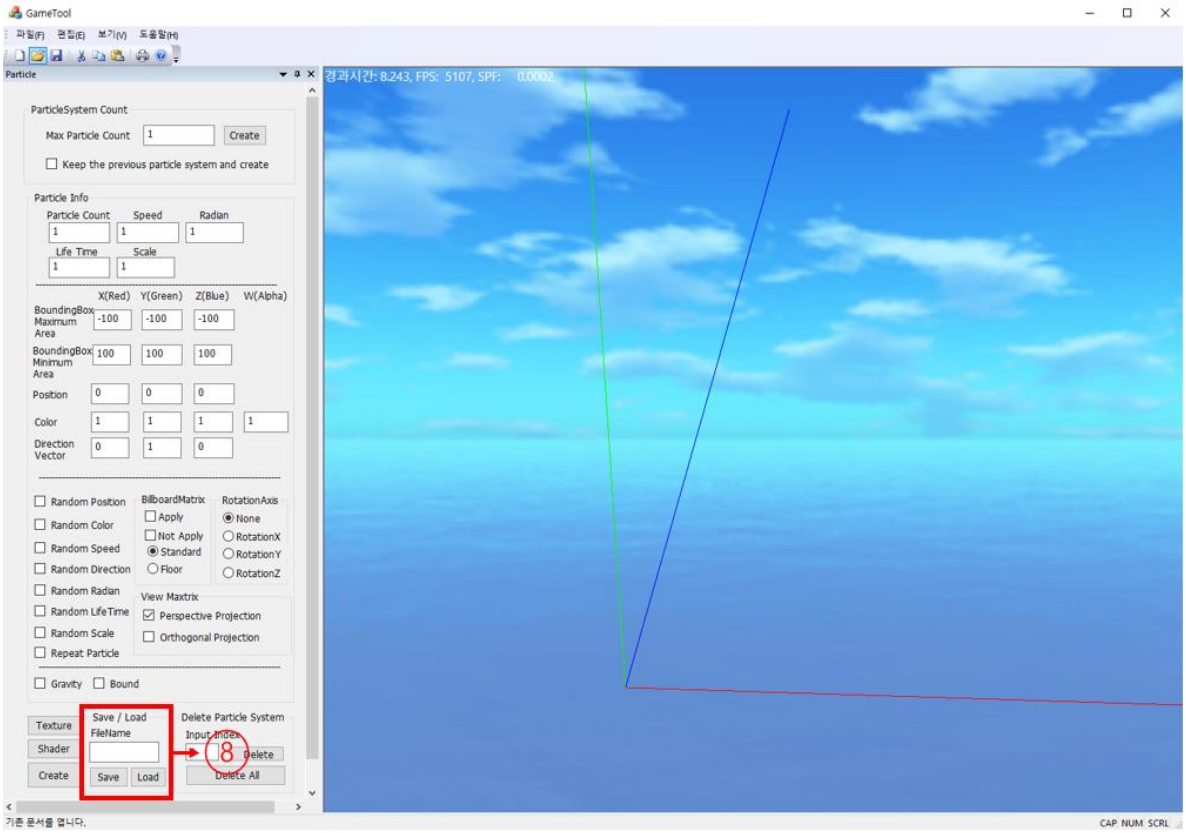
[그림 1-7-3] 셰이더 모달

사용자가 원하는 이펙트 셰이더를 적용할 수 있도록 폴더에서 셰이더 파일을 선택하여 적용한다.

- Create

툴에 이펙트 속성과 텍스처, 셰이더 설정을 모두 적용하였으면, Create 버튼을 눌러 [그림 17]의 우측 화면에 이펙트를 생성한다.

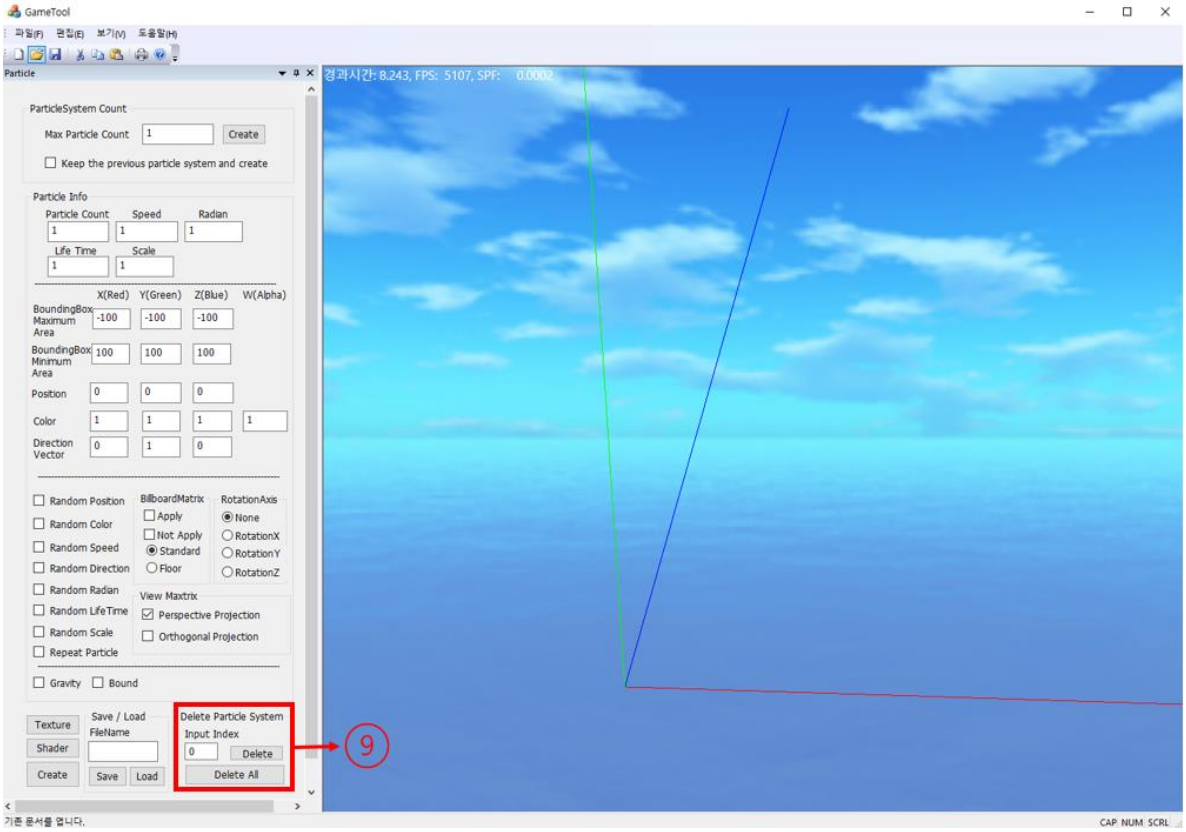
⑧ 이펙트 저장 및 로드 기능



[그림 1-8] 이펙트 저장 및 로드

생성한 이펙트를 텍스트 파일로 **저장**하는 기능이 있다. 이펙트를 생성하는데 **필요한 정보들만 저장하여 저장용량을 줄이고** 이후에 수정이 필요할 때, 저장된 파일을 **로드하여 이펙트를 수정**할 수 있다. 툴에서 만든 이펙트를 로드하면 속성값 또한 툴에 적용된다. (단, 툴이 업데이트되었다면 이전에 저장한 이펙트는 사용 불가능하며, 사용 시 예외를 발생시킨다.)

⑨ 이펙트 삭제 기능



[그림 1-9] 이펙트 삭제

단일 이펙트를 생성하였을 때, 전체 삭제 버튼만 클릭하면 삭제된다. 하지만 파티클 시스템을 여러 개 생성하여 이펙트를 제작하였을 때, 제작 도중 마음에 들지 않는 이펙트를 삭제하고 싶을 수 있다. 예를 들어 사용자가 **4 개의 이펙트를 생성하였는데 3 번째 이펙트를 삭제하고 싶어 한다.** 모든 이펙트를 삭제하고 다시 만드는 작업은 시간 낭비이며 상당히 귀찮은 작업이 될 것이다. 따라서 **전체 이펙트 중 일부 이펙트만 삭제할 수 있도록 인덱스 삭제 기능을 적용**하였고 **제거하고 싶은 이펙트 인덱스를 입력하고 삭제 버튼을 누르면 해당 이펙트만 제거**할 수 있다.

2. 이펙트 툴 주요 기술

- **알파블렌딩(AlphaBlend)**

  알파블렌딩이란 **텍스처의 알파값**을 사용하여 **이미 렌더링 되어 있는 대상(보통 백 버퍼 픽셀의 컬러 및 렌더 타겟 텍스처 컬러)**과 현재 렌더링 하려고 하는 **소스 컬러와의 연산을 통한 컬러 혼합**을 말한다. 이 연산은 알파블렌딩 공식에 의해 혼합된다.

  Final Color = DestColor * (1.0f – SourceAlpha) + SourceColor * SourceAlpha

  DestColor 는 대상 컬러이며 백 버퍼에 직접 렌더링을 하게 될 때는 **백 버퍼의 컬러**가 된다. **SoureColor 는 현재 렌더링하려는 컬러**이며 이펙트를 제작할 때 **평면에 매핑 될 텍스처의 컬러**이다. **SourceAlpha** 는 **소스 컬러값에 곱해지는 알파값**이며 **DestColor** 에는 **Invert(1.0f – SourceAlpha) 된 알파 값**을 **사용**하게 된다. DestColor = (1.0f, 1.0f, 1.0f, 1.0f) 일 때, 세가지 경우를 예를들어 보자면

  ① **소스 텍스처 알파 = 0.0f** 일 경우(완전 투명),

  흰색 = 흰색 * (1.0f – 0.0f) + 소스 컬러 * 0.0f

  이므로, 소스 컬러는 보이지 않게 된다.

  ② **소스 텍스처의 알파 = 1.0f** 일 경우(완전 불투명),

  소스컬러 = 흰색 * (1.0f – 1.0f) + 소스 컬러 * 1.0f

  이므로, 대상 컬러는 혼합되지 못하고 **소스 컬러만 남게 된다.**

  ③ **소스 텍스처 알파 = 0.5f 일 경우(반투명),** 두 컬러가 1:1 로 혼합되면 소스 컬러가 검정색(0.0f)라고 할 때,

  회색 = 흰색 * (1.0f – 0.5f) + 검정색 * 0.5f

  이다.

```
D3D11_BLEND_DESC bd;
bd.IndependentBlendEnable = FALSE;
bd.AlphaToCoverageEnable = FALSE;
bd.RenderTarget[0].BlendEnable = TRUE;
bd.RenderTarget[0].RenderTargetWriteMask = D3D11_COLOR_WRITE_ENABLE_ALL;
bd.RenderTarget[0].SrcBlend = D3D11_BLEND_ONE;
bd.RenderTarget[0].DestBlend = D3D11_BLEND_ONE;
bd.RenderTarget[0].BlendOp = D3D11_BLEND_OP_ADD;
bd.RenderTarget[0].SrcBlendAlpha = D3D11_BLEND_ONE;
bd.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ZERO;
bd.RenderTarget[0].BlendOpAlpha = D3D11_BLEND_OP_ADD;
if (FAILED(hr = g_pd3dDevice->CreateBlendState(&bd, m_pAlphaBlend.GetAddressOf())))
{
    return false;
}
```

[그림 1-10] 알파블랜딩 블랜드 스테이트 생성

- **알파테스팅(AlphaTesting)**

  알파 테스팅은 **256 단계의 알파 채널의 단계 중 일정 부분을 정하여 알파 값을 0 또는 1 로 정의**한다. 즉 알파 테스팅에서 반투명은 존재하지 않는다. 예를 들어 0.3f 를 기준값으로 정하여 픽셀의 알파 값이 기준값보다 작다면 알파 값을 0 크거나 같으면 알파 값을 1 로 정한다. 기준점을 정하지 않고 알파 테스팅을 적용하였을 때, 알파 값이 있으면 1, 없으면 0 으로 정해진다. 이러한 연산으로 퀄리티가 안 좋아지긴 하지만 **Z 버퍼에 기제하지 않으므로 연산속도가 매우 빠르다는 장점**이 있다. 만약 **캐릭터의 머리카락 하나하나가 오브젝트**라면 머리카락 이미지를 사용하여 헤어를 제작하였을 때, 알파 테스팅을 사용한다면 **알파 정렬(Alpha sorting: 앞뒤 판정)을 하지 않기 때문에 이미지가 겹겹이 있는 객체를 알파 블렌딩 보다 훨씬 저렴한 방법으로 사용할 수 있다.**

```
D3D11_BLEND_DESC bdTesting;
bdTesting.IndependentBlendEnable = FALSE;
bdTesting.AlphaToCoverageEnable = TRUE;
bdTesting.RenderTarget[0].BlendEnable = TRUE;
bdTesting.RenderTarget[0].RenderTargetWriteMask = D3D11_COLOR_WRITE_ENABLE_ALL;
bdTesting.RenderTarget[0].SrcBlend = D3D11_BLEND_ONE;
bdTesting.RenderTarget[0].DestBlend = D3D11_BLEND_ONE;
bdTesting.RenderTarget[0].BlendOp = D3D11_BLEND_OP_ADD;
bdTesting.RenderTarget[0].SrcBlendAlpha = D3D11_BLEND_ONE;
bdTesting.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ZERO;
bdTesting.RenderTarget[0].BlendOpAlpha = D3D11_BLEND_OP_ADD;
if (FAILED(hr = g_pd3dDevice->CreateBlendState(&bdTesting, m_pAlphaTesting.GetAddressOf())))
{
    return false;
}
```

[그림 1-11] 알파테스팅 블랜드 스테이트 생성

- **멀티텍스처(Multi-Texture)**

  멀티텍스처는 **2 장 이상의 텍스처를 하나의 오브젝트에 적용**하는 것이다. 이는 이펙트에서 아주 중요한 역할을 한다. 예를 들어, 총을 발사할 때 총구에서 총구 화염 이펙트가 발생하는 데 하나의 이미지만 사용하면 밋밋한 이펙트가 생성될 것이다. 하지만 **여러 장의 이미지를 겹쳐서 이펙트를 생성한다면 역동적인 이펙트를 제작**할 수 있다. 3 장의 텍스처가 다음과 같이 있다.

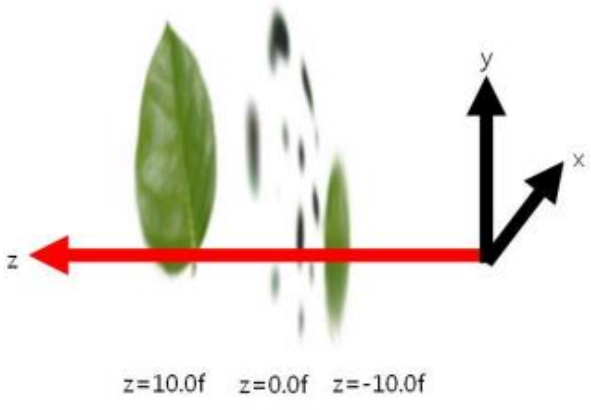
[그림 1-12] 3 개의 서로 다른 이미지(왼쪽부터 0 번,1 번 2 번)

평면에 세개의 텍스처를 다음과 같이 Z 값을 적용하여 렌더링 하면 다음과 같다.

0 번 텍스처 렌더링 : Z 값 = 0.0f

1 번 텍스처 렌더링 : Z 값 = 10.0f;

2 번 텍스처 렌더링 : Z 값 = -10.0



[그림 1-13] Plane 위치와 매핑 텍스처

알파 블랜딩, 알파 테스팅, 깊이 버퍼를 적용하고 하나의 평면에 세 개의 텍스처를 혼합하여 적용하면 다음과 같은 결과가 생성된다.

[그림 1-14] 멀티텍스처 결과

위의 개념을 적용하기 위해서는 응용프로그램에서 텍스처를 로드하여 픽셀 셰이더로 텍스처를 전달한다. **프로젝트에서는 셰이더로 최대 4 개의 텍스처를 전달**할 수 있으며 넘겨받은 텍스처를 모두 합산하여 렌더링하게 된다.

```
bool ParticleSystem::Render(ID3D11DeviceContext* pContext)
{
    pContext->VSSetShader(m_Effect.m_pVS, NULL, 0);
    pContext->PSSetShader(m_Effect.m_pPS, NULL, 0);
    //m_pImmediateContext->GSSetShader(m_Effect->m_pGS, NULL, 0);
    pContext->IASetInputLayout(m_pLayout.Get());

    ID3D11Buffer* pBuffer[2] = { m_Effect.m_pVertexBuffer, m_pInstanceBuffer.Get() };

    UINT stride[2] = { sizeof(PNCT_VERTEX), sizeof(Instance) };
    UINT offset[2] = { 0, 0 };

    pContext->PSSetShaderResources(0, m_Effect.m_pTexture->m_pTextureSRV.size(), &m_Effect.m_pTexture->m_pTextureSRV.at(0));
    pContext->UpdateSubresource(m_Effect.m_pConstantBuffer, 0, NULL, &m_Effect.m_cbData, 0, 0);
    pContext->UpdateSubresource(m_pInstanceBuffer.Get(), 0, NULL, m_pInstData, 0, 0);
    // 두개의 버퍼가 넘어간다.
    pContext->IASetVertexBuffers(0, 2, pBuffer, stride, offset);
    pContext->IASetIndexBuffer(m_Effect.m_pIndexBuffer, DXGI_FORMAT_R32_UINT, 0);
    pContext->VSSetConstantBuffers(0, 1, &m_Effect.m_pConstantBuffer);
    pContext->PSSetConstantBuffers(0, 1, &m_Effect.m_pConstantBuffer);
    pContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

    pContext->DrawIndexedInstanced(m_Effect.m_IndexList.size(), m_iEffectCount, 0, 0, 0);

    return true;
}
```

픽셀 셰이더로 텍스처 전달

[그림 1-15] 응용프로그램에서 셰이더로 텍스처 전달

```
float4 GetPSOutput(uint itextCount, float2 t)
{
    float4 vColor = float4(1.0f, 1.0f, 1.0f, 1.0f);
    switch (itextCount)
    {
        case 1:
        {
            vColor = g_txDiffuse[0].Sample(sample0, t);
        }break;
        case 2:
        {
            vColor = g_txDiffuse[0].Sample(sample0, t)
                + g_txDiffuse[1].Sample(sample0, t);
        }break;
        case 3:
        {
            vColor = g_txDiffuse[0].Sample(sample0, t)
                + g_txDiffuse[1].Sample(sample0, t)
                + g_txDiffuse[2].Sample(sample0, t);
        }break;
        case 4:
        {
            vColor = g_txDiffuse[0].Sample(sample0, t)
                + g_txDiffuse[1].Sample(sample0, t)
                + g_txDiffuse[2].Sample(sample0, t)
                + g_txDiffuse[3].Sample(sample0, t);
        }break;
        default:
        {
            vColor = g_txDiffuse[0].Sample(sample0, t);
        }break;
    }
    return vColor;
}
```

[그림 1-16] 픽셀셰이더에서 텍스처 처리

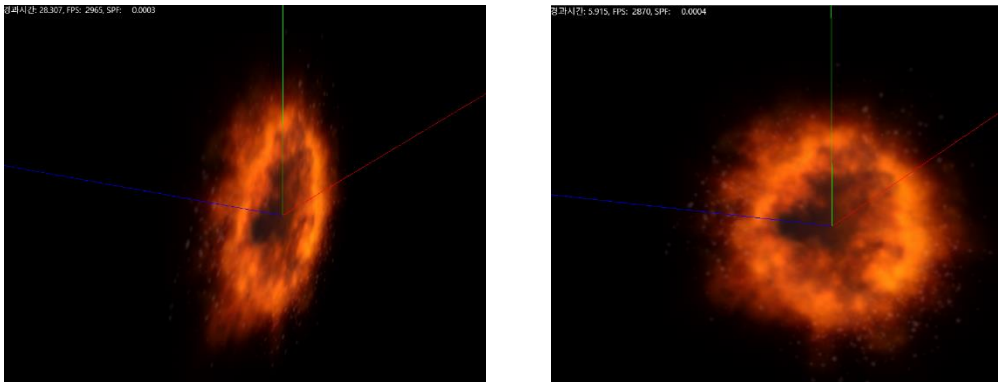
- **빌보드(Billboard) 행렬**

빌보드(Billboard) 행렬은 **카메라 행렬의 (x, y, z) 축 중에서 1 개의 축 내지는 모든 축의 역행렬을 사용하여 오브젝트의 월드 행렬에 결합하는 것**을 말한다. 만약 평면의 법선 노말이 Z 축과 나란하게 배치되어 있다면 모든 축(x,y,z) 빌보드 행렬은 **평면의 법선노말이 카메라가 회전에 상관없이 항상 카메라의 위치를 바라보게 된다.** 따라서 **빌보드 행렬**은 **카메라의 역행렬**을 가지게 된다.

```
TMatrix matWorld, matBillboard;
TMatrix matView, matProj;
D3DXMatrixIdentity(&matBillboard);
// 빌보드행렬 적용
if (m_bBillboard == TRUE)
{
    matBillboard = m_pCamera->m_matView;
    D3DXMatrixInverse(&matBillboard, NULL, &matBillboard);
    matBillboard._41 = 0;
    matBillboard._42 = 0;
    matBillboard._43 = 0;
    matBillboard._44 = 1;
}
```

[그림 1-17] 빌보드 행렬 생성

위와 같이 빌보드 행렬을 생성하면 이펙트의 월드 행렬과 곱한 행렬을 이펙트의 월드 행렬로 사용한다.



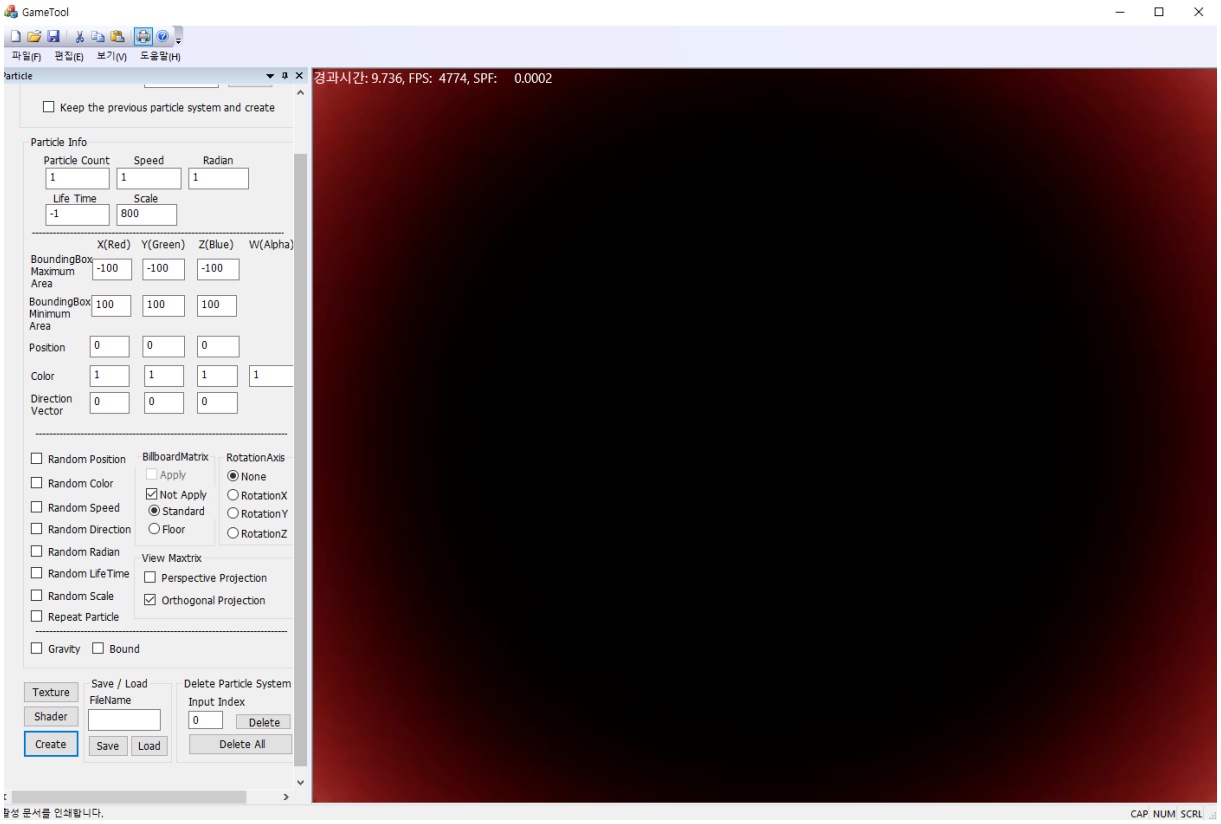
[그림 1-18] 빌보드 행렬 미적용/적용(왼쪽: 미적용, 오른쪽: 적용)

- **화면 이펙트**

일인칭 게임에서 캐릭터가 몬스터에게 피격당했을 때, **캐릭터 출혈 효과가 플레이어에게 보이지 않는다.** 이럴 때 **스크린**에 **피격 효과**를 주는데 화면 이펙트는 월드 이펙트를 제작하는 방식과는 다르다. 월드에서 나타나는 이펙트의 경우, 카메라를 기준으로 뷰 행렬과 투영 행렬을 카메라의 뷰, 투영 행렬을 사용하여 적용한다. 하지만, 화면 이펙트의 경우는 투영 행렬을 다시 생성하여 적용해야 한다. **오브젝트가 스크린에 항상 붙어있어야 하기 때문에 투영 크기를 화면 크기로 적용해야 항상 스크린 위치에서 렌더링이 된다.**

```cpp
if (m_bOrthogonal == TRUE)
{
    TVector3 vEye = TVector3(0.0f, 0.0f, -10.0f);
    TVector3 vAt = TVector3(0.0f, 0.0f, 0.0f);
    TVector3 vUp(0.0f, 1.0f, 0.0f);
    D3DXMatrixLookAtLH(&matView, &vEye, &vAt, &vUp);
    int iRectWidth = g_iWindowWidth / 2;
    int iRectHeight = g_iWindowHeight / 2;
    // 화면 중앙이 원점으로 계산되기 때문에 넓이 및 높이가 -1 ~ 1 범위로 직교투영된다.
    D3DXMatrixOrthoOffCenterLH(&matProj, -iRectWidth, iRectWidth, -iRectHeight, iRectHeight, 0.0f, 1000.0f);
}
```

[그림 1-19] 스크린 이펙트 적용



[그림 1-20] 스크린 이펙트 적용 결과

- **파티클 및 파티클 시스템**

파티클(Particle)이란 **게임 프로그램에서 게임 효과를 처리하기 위해서 다수의 평면 내지는 정점으로 특정한 형상을 표현**하는 것을 의미한다. 파티클은 다음과 같은 속성을 갖는다.

```
class Particle
{
public:
    TVector3        m_vPos;
    TVector4        m_vColor;
    TVector3        m_vSpeed;
    TVector3        m_vDirection;
    TVector3        m_vScale;
    TVector3        m_vCenter;
    TVector3        m_vFirstPos;
    float           m_fRadian;
public:
    Particle();
    ~Particle();
};
```

[그림 1-21] 파티클 속성

각 파티클은 서로 다른 속성값을 가지며 같은 운동(이동 및 회전)을 하더라도 모두 다르게 움직일 수 있다. **다수의 파티클들을 하나의 효과로 사용하기 위해서는 파티클들을 관리**하는 **파티클 시스템(Particle System)**을 적용한다.

```
class ParticleSystem
{
public:
    Sprite                      m_Effect;               // Real Effect(this object have plane)
    vector<Particle>            m_Particle;             // Particle Elements Infomation
    Camera*                     m_pCamera;              // Camera
    TMatrix                     m_matWorld;             // World Matrix
    ComPtr<ID3D11BlendState>    m_pAlphaBlend;          // Alpha Blend Device
    ComPtr<ID3D11BlendState>    m_pAlphaTesting;        // Alpha Testing Device
    BOOL                        m_bAlphaBlending;       // Alpha Blend apply to Effect
    BOOL                        m_bAlphaTesting;        // Alpha Testing apply to Effect

    ComPtr<ID3D11Buffer>        m_pInstanceBuffer;      // Instancing Buffer
    ComPtr<ID3D11InputLayout>   m_pLayout;              // Instancing Layout
    Instance*                   m_pInstData;            // Instancing Data

    int                         m_iEffectCount;         // Effect Count
    EFFECT_TYPE                 m_EffectType;           // Effect Type
    COORDINATE_TYPE             m_CoordinateType;       // If Effect type is circle exercise, this type determine how to rotate circle effect
    float                       m_fLifeTime;            // LifeTime
    int                         m_iRow;                 // Matrix(Row) - Use Sprites Texture
    int                         m_iCol;                 // Matrix(Col) - Use Sprites Texture
    TCHAR*                      m_szEffectName;         // Effect Name
    TCHAR*                      m_szParticleName;       // ParticleName (Using ParticleSystemManager)
    int                         m_iIndex;               // Particle System Manager Index
    bool                        m_bBillboard;           // Billboard Matrix apply to Effect
    bool                        m_bPerspective;         // Perspective Projection apply to Effect
    bool                        m_bOrthogonal;          // Orthogonal Projection Matrix apply to Effect
    bool                        m_bRepeatEffect;        // Variables that distinguish repetitive effects
```

[그림 1-22] 파티클 시스템 속성

파티클 시스템은 위 그림과 같은 속성을 가지고 있으며, 스프라이트, 파티클 리스트, 블랜딩 스테이트, 인스턴스 버퍼를 가지고 있다. 파티클 시스템에 존재하는 하나의 스프라이트 정보(텍스처, 정점, 인덱스 버퍼) 통해 수많은 파티클을 **인스턴싱**을 통해 **낮은 부하로 파티클을 렌더링**할 수 있다.

```
//--------------------------------------------------------------------------------
// 파티클 업데이트 함수
//--------------------------------------------------------------------------------
void ParticleSystem::ParticleUpdate(ID3D11DeviceContext* pContext)
{
    m_Effect.Update();

    m_Effect.m_VertexList[0].t = m_Effect.GetDrawRect(0);
    m_Effect.m_VertexList[1].t = m_Effect.GetDrawRect(1);
    m_Effect.m_VertexList[2].t = m_Effect.GetDrawRect(2);
    m_Effect.m_VertexList[3].t = m_Effect.GetDrawRect(3);
    pContext->UpdateSubresource(m_Effect.m_pVertexBuffer, 0, NULL, &m_Effect.m_VertexList.at(0), 0, 0);

    TMatrix matWorld, matBillboard;
    TMatrix matView, matProj;
    D3DXMatrixIdentity(&matBillboard);
    // 빌보드행렬 적용
    if (m_bBillboard == TRUE)
    {
        matBillboard = m_pCamera->m_matView;
        D3DXMatrixInverse(&matBillboard, NULL, &matBillboard);
        matBillboard._41 = 0;
        matBillboard._42 = 0;
        matBillboard._43 = 0;
        matBillboard._44 = 1;
    }
```

[그림 1-23] 파티클 시스템 업데이트(1)

```
// 직교투영 적용
if (m_bOrthogonal == TRUE)
{
    TVector3 vEye = TVector3(0.0f, 0.0f, -1.0f);
    TVector3 vAt = TVector3(0.0f, 0.0f, 0.0f);
    TVector3 vUp(0.0f, 1.0f, 0.0f);
    D3DXMatrixLookAtLH(&matView, &vEye, &vAt, &vUp);
    int iRectWidth = g_iWindowWidth / 2;
    int iRectHeight = g_iWindowHeight / 2;
    // 화면 중앙이 원점으로 계산되기 때문에 넓이 및 높이가 -1 ~ 1 범위로 직교투영된다.
    D3DXMatrixOrthoOffCenterLH(&matProj, -iRectWidth, iRectWidth, -iRectHeight, iRectHeight, 0.0f, 1000.0f);
}

for (int iCnt = 0; iCnt < m_Particle.size(); iCnt++)
{
    TVector3 vPos = { 0.0f, 0.0f, 0.0f };
    D3DXMatrixIdentity(&matWorld);

    if (m_EffectType == EFFECT_TYPE::Straight)
    {
        // 방향벡터와 속도로부터 위치 갱신
        vPos = m_Particle[iCnt].m_vPos + (m_Particle[iCnt].m_vDirection * (m_Particle[iCnt].m_vSpeed * g_fSecPerFrame));
    }
    else if(m_EffectType == EFFECT_TYPE::Circle)
    {
        vPos = m_Particle[iCnt].m_vPos
            + ((Circle(m_Particle[iCnt].m_fRadian, g_fElapseTime * m_Particle[iCnt].m_vSpeed.x)
                + m_Particle[iCnt].m_vDirection) * (m_Particle[iCnt].m_vSpeed * g_fSecPerFrame));
    }
    m_Particle[iCnt].m_vPos = vPos;
```

[그림 1-24] 파티클 시스템 업데이트(2)

```
    // 이펙트 크기 적용
    D3DXMatrixScaling(&matWorld, m_Particle[iCnt].m_vScale.x, m_Particle[iCnt].m_vScale.y, m_Particle[iCnt].m_vScale.z)
    matWorld = matWorld * matBillboard;

    matWorld._41 = vPos.x;
    matWorld._42 = vPos.y;
    matWorld._43 = vPos.z;

    // 상수버퍼 갱신
    if(m_bOrthogonal == TRUE)
        m_Effect.SetMatrix(&matWorld, &matView, &matProj);
    else
        m_Effect.SetMatrix(&matWorld, &m_pCamera->m_matView, &m_pCamera->m_matProj);

    m_Effect.m_cbData.vColor.w = m_Effect.m_pTexture->m_pTextureSRV.size();

    // 인스턴스 상수버퍼 갱신
    m_pInstData[iCnt].m_vColor = m_Particle[iCnt].m_vColor;
    D3DXMatrixTranspose(&m_pInstData[iCnt].matWorld, &matWorld);
}
}
```

[그림 1-25] 파티클 시스템 업데이트(3)

- **인스턴싱(Instancing)**

  인스턴싱(Instancing)이란 **한 번의 객체 렌더링을 통하여 여러 개의 객체를 동시에 렌더링하는 방법**이다. **렌더링한 객체의 데이터**를 **재사용**하여 여러 객체에 각각 사용될 **인스턴스 버퍼를 사용하여 렌더링**한다. 여러 번 렌더링 메소드를 사용하지 않을 수 있으므로 렌더링 성능이 향상된다. 오브젝트를 렌더링할 때, PNCT(P: position, N: normal, C: color, T: texture) 구조를 사용하여 정점 레이아웃을 사용하였다. 하지만 인스턴싱을 위해서는 인스턴싱 버퍼를 생성하고 정점 레이아웃에 추가해야 한다.

  ① **인스턴스 버퍼 생성**

  인스턴스 버퍼는 파티클 시스템 클래스에서 생성된다. **인스턴스 버퍼를 생성하는 방법은 일반 정점 버퍼를 생성하는 방법과 같다.** 파티클 시스템에 존재하는 모든 이펙트(파티클)의 위치와 색상 값을 인스턴스 데이터 리스트에 추가하고 버퍼를 생성한다.

```
//--------------------------------------------------------------------------------
// 인스턴스 버퍼 생성
//--------------------------------------------------------------------------------
bool ParticleSystem::CreateInstanceBuffer()
{
    SAFE_NEW_ARRAY(m_pInstData, Instance, m_iEffectCount);
    for (int iCnt = 0; iCnt < m_iEffectCount; iCnt++)
    {
        D3DXMatrixTranslation(&m_pInstData[iCnt].matWorld, m_Particle[iCnt].m_vPos.x, m_Particle[iCnt].m_vPos.y, m_Particle[iCnt].m_vPos.z);
        D3DXMatrixTranspose(&m_pInstData[iCnt].matWorld, &m_pInstData[iCnt].matWorld);
        m_pInstData[iCnt].m_vColor = m_Particle[iCnt].m_vColor;
    }

    D3D11_BUFFER_DESC bd;
    bd.Usage = D3D11_USAGE_DEFAULT;
    bd.ByteWidth = sizeof(Instance) * m_iEffectCount;
    bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
    bd.CPUAccessFlags = 0;
    bd.MiscFlags = 0;

    D3D11_SUBRESOURCE_DATA InitData;
    InitData.pSysMem = (void*)m_pInstData;
    g_pd3dDevice->CreateBuffer(&bd, &InitData, m_pInstanceBuffer.GetAddressOf());
    return true;
}
```

[그림 1-26] 인스턴스 버퍼 생성

  ② **인스턴스 레이아웃 생성**

  인스턴스 버퍼를 생성하면 레이아웃을 생성해야한다. 기존 정점 버퍼를 구성하는 PNCT 구조는 그대로 적용하고, 스프라이트의 월드 행렬과 색상 값을 새로 정의하여 생성한다.

```
const D3D11_INPUT_ELEMENT_DESC layout[] =
{
    { "POSITION",  0, DXGI_FORMAT_R32G32B32_FLOAT,    0,  0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL" ,   0, DXGI_FORMAT_R32G32B32_FLOAT,    0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "COLOR",     0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 24, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD",  0, DXGI_FORMAT_R32G32_FLOAT,       0, 40, D3D11_INPUT_PER_VERTEX_DATA, 0 },

    // 인스턴스 버퍼
    { "mTransform",      0, DXGI_FORMAT_R32G32B32A32_FLOAT,    1,  0, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
    { "mTransform",      1, DXGI_FORMAT_R32G32B32A32_FLOAT,    1, 16, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
    { "mTransform",      2, DXGI_FORMAT_R32G32B32A32_FLOAT,    1, 32, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
    { "mTransform",      3, DXGI_FORMAT_R32G32B32A32_FLOAT,    1, 48, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
    { "PARTICLECOLOR",   0, DXGI_FORMAT_R32G32B32A32_FLOAT,    1, 64, D3D11_INPUT_PER_INSTANCE_DATA, 1 },
};

UINT numElements = sizeof(layout) / sizeof(layout[0]);

if (FAILED(hr = g_pd3dDevice->CreateInputLayout(layout, numElements, pVSBuf->GetBufferPointer(), pVSBuf->GetBufferSize(), m_pLayout.GetAddressOf())))
{
    return false;
}
```

[그림 1-27] 인스턴스 레이아웃 생성

③ 인스턴스 셰이더

이렇게 만들어진 **인스턴스 버퍼는 프레임마다 인스턴스 객체에 적용할 행렬값을 계산하여 갱신**하게 된다. 응용프로그램에서 인스턴싱을 위한 모든 작업이 완료되었다면 셰이더 코드에서 이를 받아 사용할 수 있도록 해야 한다.

```
struct VS_INPUT
{
    float3 p : POSITION;
    float3 n : NORMAL;
    float4 c : COLOR;
    float2 t : TEXCOORD;

    float4x4    matInstance   : mTransform;
    float4      instanceColor : PARTICLECOLOR;
    uint        InstanceId    : SV_InstanceID;
};

struct VS_OUTPUT
{
    float4 p : SV_POSITION;
    float3 n : NORMAL;
    float4 c : COLOR0;
    float2 t : TEXCOORD0;
};
```

```
VS_OUTPUT VS(VS_INPUT Input)
{
    VS_OUTPUT output = (VS_OUTPUT)0;
    output.p = float4(Input.p, 1.0f);
    output.p = mul(output.p, Input.matInstance);
    output.p = mul(output.p, g_matView);
    output.p = mul(output.p, g_matProj);
    output.n = Input.n;
    output.c = Input.instanceColor;
    output.t = Input.t;
    return output;
}
```

[그림 1-28] 인스턴스 셰이더 메소드

정점 버퍼로 전달된 각 정점의 구조는 정점 레이아웃의 구조와 동일하며 VS_INPUT 구조체의 uint instanceID 는 각 인스턴스의 고유한 인덱스를 의미한다. 이 값은 인스턴싱 될 때 0 부터 렌더링 순서에 따라 인덱스가 1 씩 증가한다.

3. 최종 결과

- **툴 시연 영상**

  https://youtu.be/GfTfRzlZWFo

- **툴 풀 소스코드**

  https://github.com/MingyuOh/EffectTool