# REPORT

## Final Project – Filter Program

| | |
|---|---|
| 과 목 : | 게임 프로그래밍 |
| 분 반 : | A |
| 담당 교수 : | 조윤식 교수님 |
| 학 번 : | 1971222 |
| 이 름 : | 백종훈 |

https://github.com/100jnghn/DirectX11-Filter-Program

## [프로젝트 개요]

DirectX11을 활용해 구현한 Filter 프로그램입니다.

일부 게임에서는 물체에 일정 시간 열을 가하거나 물과 닿으면 물체가 변하게 되는 기능을 사용합니다. 대표적인 예시로 마인크래프트에서는 Cube로 된 흙 Object를 화로에서 일정 시간 열을 가하면 돌 Object로 변하게 됩니다. 이렇게 물체에 열이 가해지는 현상을 숫자나 게이지가 점점 차오르는 UI로 표현하지 않고 Object의 Texture에 변화를 줌으로 직관적으로 기능을 관찰할 수 있도록 프로젝트를 구성했습니다.

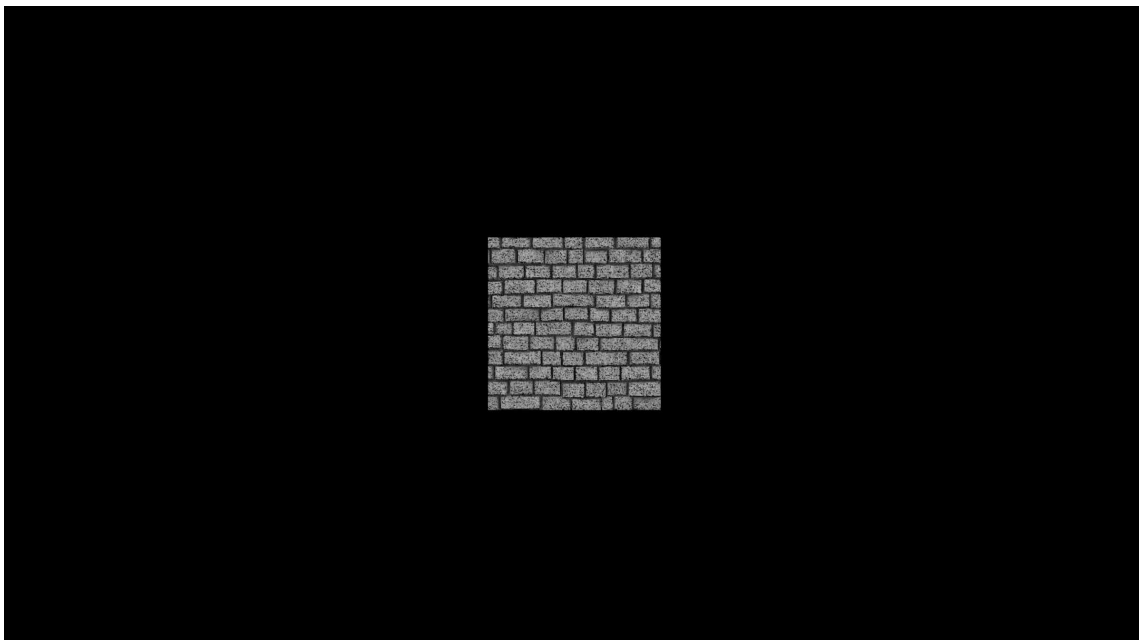
이 프로젝트에서는 물체에 Ice Filter와 Fire Filter 기능을 적용하고, 필터에 따른 물체의 색상 변화를 Render To Texture를 통해 관찰할 수 있습니다.

## [기능 소개]

- 방향키 ← →를 눌러 Cube 모델을 좌우로 이동시킬 수 있습니다.
- 숫자 1을 눌러 Cube의 앞에 Ice 필터를 띄울 수 있습니다.
- 숫자 2를 눌러 Cube의 앞에 Fire 필터를 띄울 수 있습니다.
- 숫자 0을 눌러 Cube에 가해지는 필터를 제거할 수 있습니다.
- 필터1이나 필터2가 켜진 상태에서 방향키 ↑ ↓를 눌러 Filter 효과를 키우거나 줄일 수 있습니다.
- Cube 모델의 위치가 필터에서 크게 벗어나면 필터 효과가 가해지지 않습니다.
- 우측 상단 Render Texture를 통해 모델에 가해지는 Filter 효과를 직관적으로 관찰할 수 있습니다.

## [구현 기능 상세 설명]

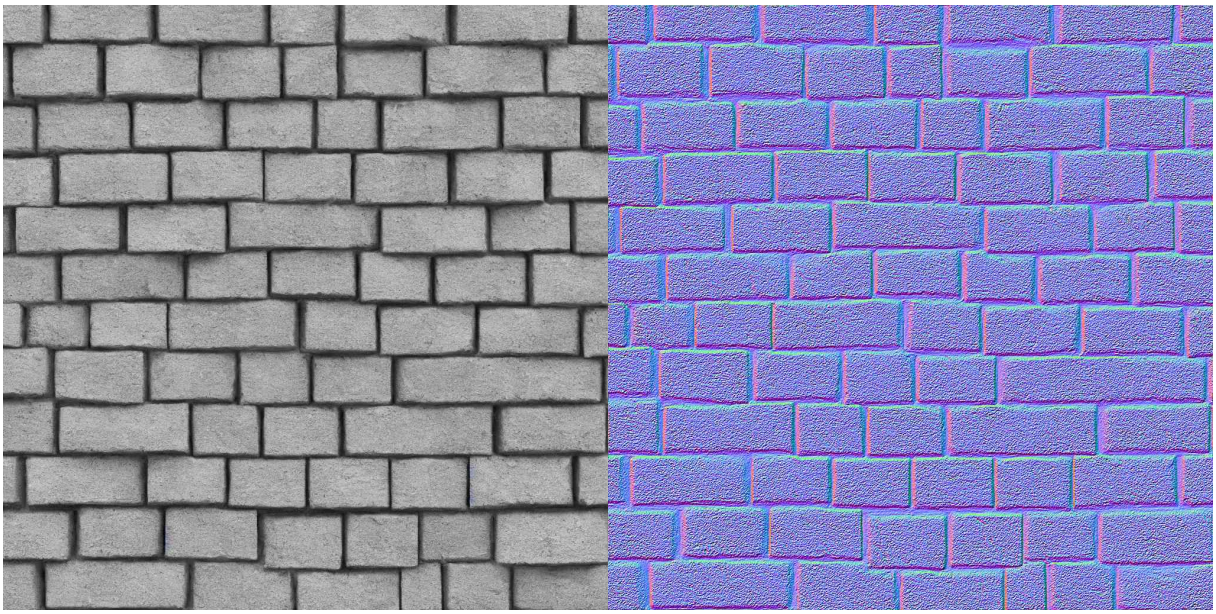
### (01) - Cube 모델 Render



Chap8-NormalMap에서 진행한 대면 수업 실습 프로젝트를 베이스로 시작했습니다.

cube.txt를 사용해 네모난 큐브 모델을 로드했습니다. 큐브 모델의 Texture는 block.jpg 리소스를 저장해 프로젝트에 포함했고, normal map 변환 사이트를 이용해 blocknormal.jpg를 만들어 프로젝트에 포함시켰습니다. 아래는 원본 텍스쳐 이미지입니다.

(좌) block.jpg / (우) blockNormal.jpg



## (02) - Cube Normal Texture 적용

가져온 두 이미지를 사용해 PixelShader를 작성했습니다.

blockNormal과 bumpMap을 샘플링을 통해 texture의 Value를 가져온 후 bumpMap의 normal 계산을 위해 값의 범위를 –1~1로 변환합니다. 이 데이터를 통해 bumpMap의 최종 normal 값을 계산합니다.

[ 코드 – normalPS.hlsl ]

```
// 텍스쳐 픽셀 Sampling
textureColor = shaderTexture1.Sample(SampleType, input.tex);
bumpMap = shaderTexture2.Sample(SampleType, input.tex);

// normal Value 범위 지정
bumpMap = (bumpMap *2.0f) -1.0f;

// Normal map 데이터를 사용해 Normal 계산
bumpNormal = (bumpMap.x * input.tangent) + (bumpMap.y * input.binormal) + (bumpMap.z * input.normal);
bumpNormal = normalize(bumpNormal);
```

## (03) - 사용자 입력을 통한 Cube 모델 이동

방향키 ← → 버튼을 눌러서 모델을 좌우로 이동시킬 수 있습니다.

applicationclass에 m_cubePosX 변수를 선언하고 inputclass에 좌우 방향키가 눌렸다면 true를 반환하는 IsLeftArrowPressed(), IsRightArrowPressed() 함수를 작성했습니다. applicationclass에서 해당 함수의 true 값을 받으면 m_cubePosX의 값을 증가 감소시키고, 이 값을 Render() 함수에 전달하여 worldMatrix의 x값을 이동시켜 모델 움직임을 구현했습니다.

(좌) 왼쪽으로 이동 / (우) 오른쪽으로 이동

[ 코드 – inputclass.cpp ]

```
// 왼쪽 방향키
bool InputClass::IsLeftArrowPressed() {
	if (m_keyboardState[DIK_LEFT]) {
		return true;
	}
	return false;
}

// 오른쪽 방향키
bool InputClass::IsRightArrowPressed() {
	if (m_keyboardState[DIK_RIGHT]) {
		return true;
	}
	return false;
}
```

[ 코드 – applicationclass.cpp ]

```
// Frame 함수 내에서 매 프레임 호출됨 // InputMove(Input, 0.1f)
void ApplicationClass::InputMove(InputClass* Input, float value) {

	// cube 모델 좌우 이동
	if (Input->IsLeftArrowPressed()) {
		m_cubePosX -= value;
	}

	if (Input->IsRightArrowPressed()) {
		m_cubePosX += value;
	}
}
```

### (04) - Cube에 가해지는 두 개의 조명 구현

Cube 모델에 두 개의 조명이 비춰지도록 m_Light1과 m_Light2를 선언 후 초기화했습니다.
Light1은 붉은색을 띄고 ↙방향을 비추도록 Direction을 설정했습니다.
Light2는 푸른색을 띄고 ↘방향을 비추도록 Direction을 설정했습니다.

두 조명은 Cube에 적용된 shader인 m_NormalMapShader를 통해 normalPS의 상수 버퍼로 전달되며 전달된 상수 버퍼 값을 사용해 Pixel의 최종 색상을 결정합니다.

아래는 Cube가 왼쪽으로 이동하여 우측 면에 붉은 조명이 보이고 Cube가 오른쪽으로 이동하여 좌측면에 푸른 조명이 보이는 이미지입니다. 중간 부분은 두 조명이 합쳐져 보라색 색상이 나타납니다.

[ 코드 – normalmapshaderclass.cpp ] - 조명1도 조명2와 동일하게 Set

```
// ----- 조명2 ----- //
result = deviceContext->Map(m_lightBuffer2, 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedResource);
if (FAILED(result))
{
        return false;
}

LightBufferType* dataPtr3 = (LightBufferType*)mappedResource.pData;
dataPtr3->diffuseColor = diffuseColor2;                    // 색상
dataPtr3->lightDirection = lightDirection2;          // 방향
dataPtr3->padding =0.0f;

deviceContext->Unmap(m_lightBuffer2, 0);
bufferNumber =1;
deviceContext->PSSetConstantBuffers(bufferNumber, 1, &m_lightBuffer2);
```

[ 코드 – normalPS.hlsl ] - 두 조명에 대한 상수 버퍼 선언

```
cbuffer LightBuffer1
{
    float4 diffuseColor1;
    float3 lightDirection1;
    float padding;
};

cbuffer LightBuffer2
{
    float4 diffuseColor2;
    float3 lightDirection2;
    float padding2;
};
```

[ 코드 – normalPS.hlsl ] - 두 조명값을 연산하여 최종 색상 반환

```
// Invert the light direction for calculations.
lightDir1 =-lightDirection1;
// Calculate the amount of light on this pixel based on the normal map value.
lightIntensity1 = saturate(dot(bumpNormal, lightDir1));

lightDir2 =-lightDirection2;
lightIntensity2 = saturate(dot(bumpNormal, lightDir2));

// Combine the final light color with the texture color.
float4 color1 = saturate(diffuseColor1 * lightIntensity1);
float4 color2 = saturate(diffuseColor2 * lightIntensity2);
color = saturate(color1 + color2);
color = color * textureColor * 1.3;      // 1.3은 임의의 값

return color;
```

위의 조명 코드에서 Diffuse를 결정할 때 normal 값은 texture의 일반적인 normal이 아닌 bumpMap의 bumpNormal을 사용하여 입체적이고 거친 표면에 조명이 가해지는 장면을 구현했습니다.

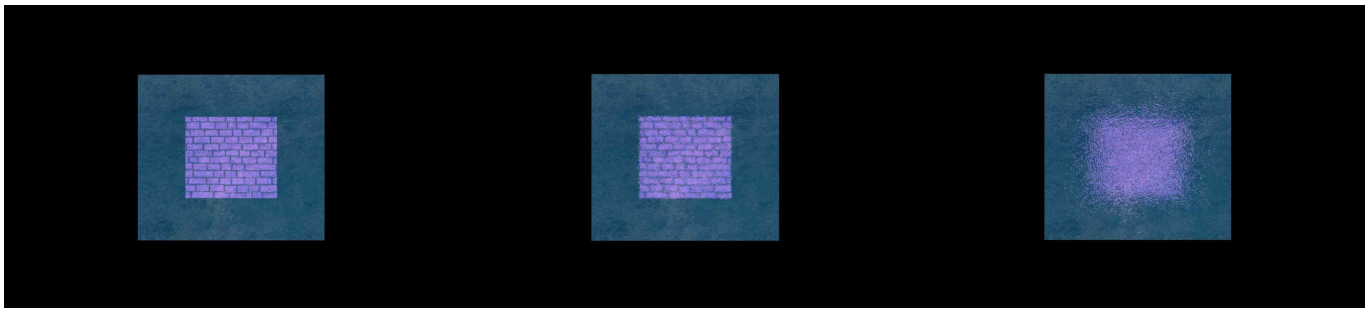
각 조명의 연산 값을 color1, color2 변수에 적절한 보간 함수를 통해 저장하고 두 색상과 texture의 색상을 연산한 후, 임의의 값을 곱해줌으로 밝기를 조절하여 최종 색상을 반환합니다.

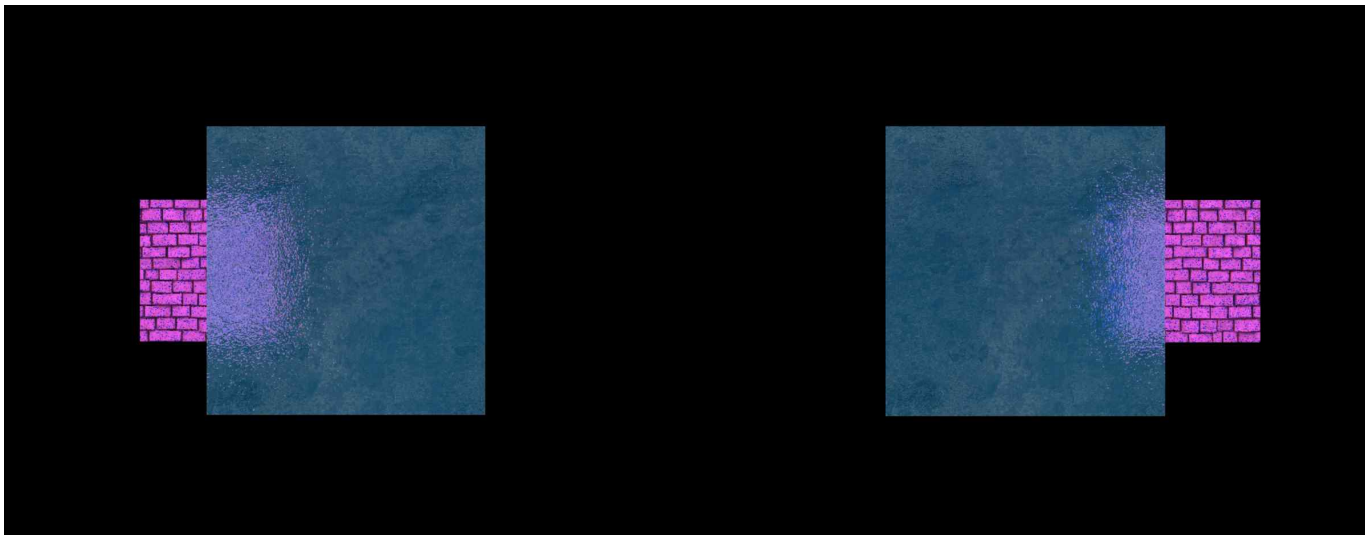
## (05) - 사용자 입력을 통한 Filter 효과(1) - Ice Filter (Render To Texture)

숫자 1을 눌러 Ice Filter를 화면에 띄우고 모델에 효과를 적용할 수 있습니다.

Ice Filter가 적용된 상태에서 방향키 ↑↓를 눌러 필터의 강도를 높이고 줄일 수 있습니다. Refraction Scale은 0.0~0.3 범위로 제한됩니다(기본 0.02). 또한, 필터를 적용한 상태에서도 모델을 좌우로 움직일 수 있으며 필터 밖으로 벗어난 Cube 모델은 필터의 효과를 받지 않습니다.

(좌) refraction scale = 0.0 / (중) refraction scale = 0.02(기본) / (우) refraction scale = 0.3





Ice Filter는 Chap9의 Ice 효과를 참고하여 구현하였기에 Render To Texture를 사용했습니다. 사용자 입력에 따라 변화하는 Refraction Scale은 상수 버퍼를 사용해 값을 전달받도록 작성했으며, 이 값에 따라 icebump.jpg을 Sampling할 때 값의 범위를 연산합니다.

[ 코드 – applicationclass.cpp:: Initialize() ] - Ice 효과를 입힐 모델, Shader, Render Texture 생성 및 초기화

```
// ----- Ice RTT 초기화 ----- //
strcpy_s(modelFilename, "data/square.txt");

// 모델 생성
m_IceModel =new ModelClass;   // Ice를 입힐 모델 (네모)

result = m_IceModel->Initialize(m_Direct3D->GetDevice(), m_Direct3D->GetDeviceContext(),
modelFilename, (WCHAR*)L"data/ice01.jpg", (WCHAR*)L"data/icebump01.jpg", NULL);
if (!result)
{
        MessageBox(hwnd, L"Could not initialize the window model object.", L"Error", MB_OK);
        return false;
}

// Render Texture 생성
m_RenderTextureIce =new RenderTextureClass;

result = m_RenderTextureIce->Initialize(m_Direct3D->GetDevice(), screenWidth, screenHeight,
SCREEN_DEPTH, SCREEN_NEAR, 1);
if (!result)
{
        MessageBox(hwnd, L"Could not initialize the render texture object.", L"Error", MB_OK);
        return false;
}

// Ice Shader 생성
m_GlassShader =new GlassShaderClass;

result = m_GlassShader->Initialize(m_Direct3D->GetDevice(), hwnd);
if (!result)
{
        MessageBox(hwnd, L"Could not initialize the glass shader object.", L"Error", MB_OK);
        return false;
}
```

[ 코드 – applicationclass.cpp ] - Input에 따른 refractionScale 변화

```
void ApplicationClass::InputFilterScale(InputClass* Input, float value, float shiftValue) {
        if (Input->IsUpArrowPressed()) {
                switch (m_filterMode) {
                        case 1:
                                m_refractionScale += value;
                                // max refraction scale: 0.3f
                                if (m_refractionScale >=0.3f) {
                                        m_refractionScale =0.3f;
                                }
```

[ 코드 – applicationclass.cpp ] - Filter Mode가 바뀜에 따른 Render To Texture 렌더

```
bool ApplicationClass::ChangeFilter() {
        bool result;
        switch (m_filterMode) {
                case 1:
                        result = RenderSceneToTextureIce(m_cubePosX);

                        if (!result)
                        {
                                return false;
                        }
```

[ 코드 – applicationclass.cpp ] - Render To Texture를 사용한 Ice Filter 효과 함수

```
// ----- Ice RTT 함수 정의 ----- //
bool ApplicationClass::RenderSceneToTextureIce(float cubePosX) {
	XMMATRIX worldMatrix, viewMatrix, projectionMatrix;
	bool result;

	// Set the render target to be the render to texture and clear it.
	m_RenderTextureIce->SetRenderTarget(m_Direct3D->GetDeviceContext());
	m_RenderTextureIce->ClearRenderTarget(m_Direct3D->GetDeviceContext(), 0.0f, 0.0f, 0.0f, 1.0f);

	// Get the world, view, and projection matrices from the camera and d3d objects.
	m_Direct3D->GetWorldMatrix(worldMatrix);
	m_Camera->GetViewMatrix(viewMatrix);
	m_Direct3D->GetProjectionMatrix(projectionMatrix);

	// ----- 방향키 Input에 따른 worldMatrix 변경 ----- //
	worldMatrix = XMMatrixTranslation(cubePosX, 0.0f, 0.0f);
	// ------------------------------------------- //

	// Render the cube model using the texture shader.
	m_Model->Render(m_Direct3D->GetDeviceContext());
	result = m_NormalMapShader->Render(m_Direct3D->GetDeviceContext(),
				m_Model->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,
				m_Model->GetTexture(0), m_Model->GetTexture(1),
				m_Light1->GetDirection(), m_Light1->GetDiffuseColor(),
				m_Light2->GetDirection(), m_Light2->GetDiffuseColor());

	if (!result)
	{
		return false;
	}

	m_Direct3D->SetBackBufferRenderTarget();
	m_Direct3D->ResetViewport();
	return true;
}
```

Input에 따른 Cube의 위치 변화를 Render To Texture 함수의 인자로 전달하고 worldMatrix에 Translation 연산을 적용함으로 Cube의 움직임을 Render Texture를 적용한 상태에서도 관찰할 수 있도록 작성했습니다. Ice Filter 효과의 Render To Texture에서는 생성한 두 개의 조명을 Shader에 모두 전달해 필터 뒤의 모델에는 빨간색과 파란색 조명 모두 가해집니다.

[ 코드 – applicationclass.cpp ] - Render()에서의 filter 변화 코드

```
// Ice 모델 렌더
if (m_filterMode == 1) {
        // ----- Ice Model Render ----- //
        worldMatrix = XMMatrixScaling(2.0f, 2.0f, 2.0f);
        worldMatrix *= XMMatrixTranslation(0.0f, 0.0f, -1.1f);

        m_IceModel->Render(m_Direct3D->GetDeviceContext());

        result = m_GlassShader->Render(m_Direct3D->GetDeviceContext(), m_IceModel->GetIndexCount(),
                                worldMatrix, viewMatrix, projectionMatrix,
                                m_IceModel->GetTexture(0), m_IceModel->GetTexture(1),
                                m_RenderTextureIce->GetShaderResourceView(), m_refractionScale);

        if (!result)
        {
                return false;
        }
```

[ 코드 – glassPS.hlsl ] - Ice 효과의 Shader. Input에 따른 refractionScale 변화를 위한 상수 버퍼 선언

```
cbuffer GlassBuffer
{
    float refractionScale;
    float3 padding;
};
```

[ 코드 – glassPS.hlsl ] - refractionScale, texture, normal texture를 사용한 Pixel Value 연산

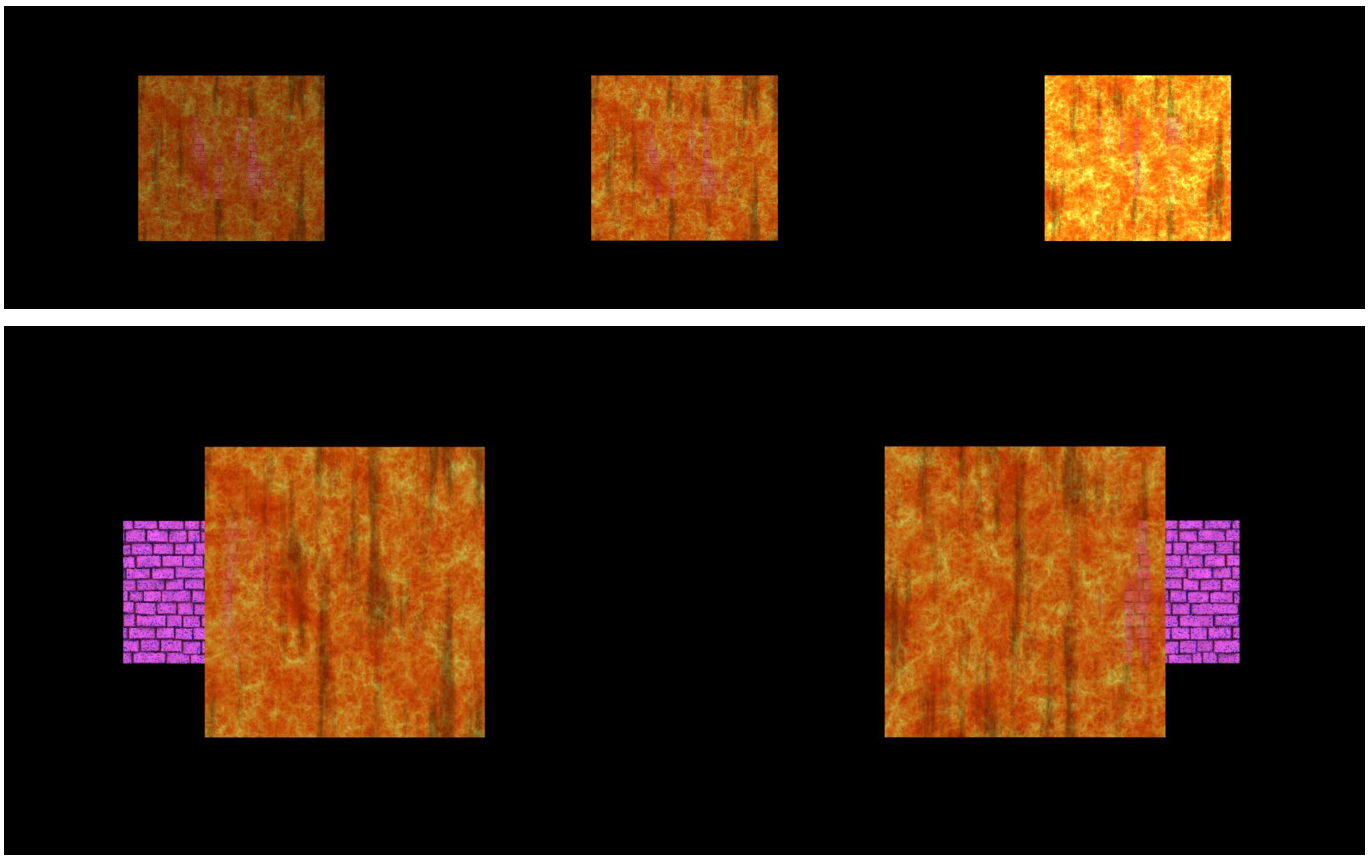
```
float4 GlassPixelShader(PixelInputType input) : SV_TARGET
{
    float2 refractTexCoord;     float4 normalMap;     float3 normal;
    float4 refractionColor;     float4 textureColor;     float4 color;

    // refraction texture 좌표 계산
    refractTexCoord.x = input.refractionPosition.x / input.refractionPosition.w /2.0f +0.5f;
    refractTexCoord.y =-input.refractionPosition.y / input.refractionPosition.w /2.0f +0.5f;

    // Normal map 샘플링
    normalMap = normalTexture.Sample(SampleType, input.tex);

    // 범위 재지정: (0,1) to (-1,+1).
    normal = (normalMap.xyz *2.0f) -1.0f;

    refractTexCoord = refractTexCoord + (normal.xy * refractionScale);

    // 계산한 refraction texture 좌표를 사용한 refractionTexture 샘플링
    refractionColor = refractionTexture.Sample(SampleType, refractTexCoord);

    // Ice Texture 샘플링
    textureColor = colorTexture.Sample(SampleType, input.tex);

    // Ice, Refraction을 보간한 최종 색상
    color = lerp(refractionColor, textureColor, 0.5f);

    return color;
}
```

## (06) - 사용자 입력을 통한 Filter 효과(2) - Fire Filter (Alpha Texture)

숫자 2를 눌러 Fire Filter를 화면에 띄우고 모델에 효과를 적용할 수 있습니다.

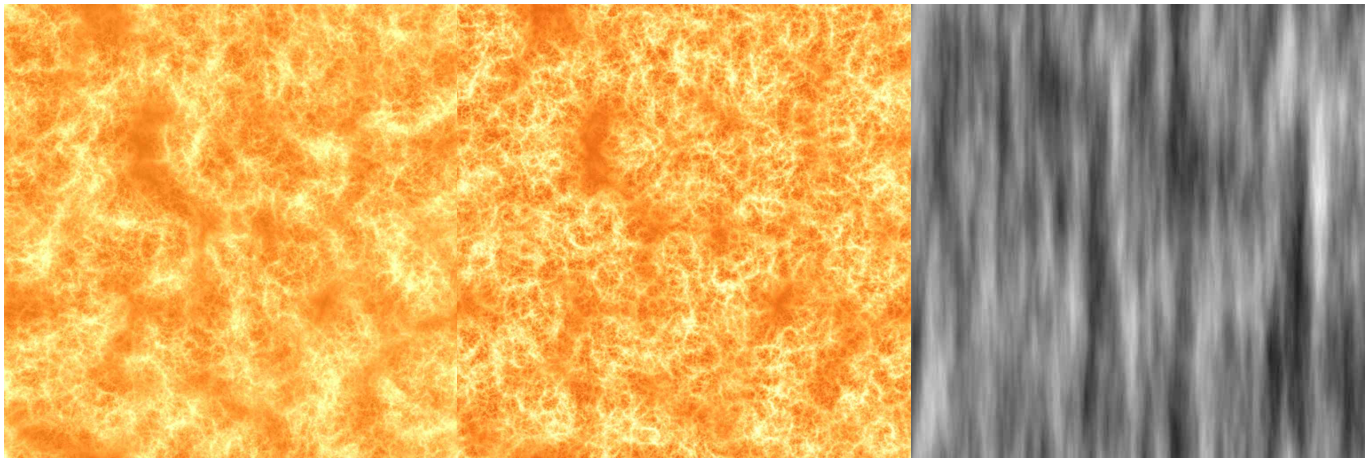
Fire Filter가 적용된 상태에서 방향키 ↑↓를 눌러 필터의 강도를 높이고 줄일 수 있습니다. Fire의 강도(밝기)는 0.8~1.7 범위로 제한됩니다(기본 1.3). 또한, 필터를 적용한 상태에서도 모델을 좌우로 움직일 수 있으며 필터 밖으로 벗어난 Cube 모델은 필터의 효과를 받지 않습니다.

(좌) fire bright = 1.7 / (중) fire bright = 1.3(기본) / (우) fire bright = 0.8



Fire Filter에 사용된 Texture들은 flame01, flame02, flame alpha입니다. Pixel Shader에서 각 Texture들을 샘플링할 때 y좌표를 감소시킨 후 상수 버퍼로 전달받은 texture translation 값과의 연산을 통해 용암이 흐르는 듯한 효과를 구현했습니다. 사용된 Texture들의 원본 이미지는 아래와 같습니다.

(좌) flame01.jpg / (중) flame02.jpg / (우) flameAlpha.png



[ 코드 – applicationclass.cpp ] - fire filter 초기화

```
// ----- Fire filter 초기화 ----- //
// 모델 생성
m_FireModel =new ModelClass;

result = m_FireModel->Initialize(m_Direct3D->GetDevice(), m_Direct3D->GetDeviceContext(),
                                 modelFilename, (WCHAR*)L"data/flame01.jpg",
                                 (WCHAR*)L"data/flame02.jpg", (WCHAR*)L"data/flameAlpha.png");

if (!result)
{
        MessageBox(hwnd, L"Could not initialize the window model object.", L"Error", MB_OK);
        return false;
}

// shader 생성
m_FireShader =new FireShaderClass;

result = m_FireShader->Initialize(m_Direct3D->GetDevice(), hwnd);
if (!result)
{
        MessageBox(hwnd, L"Could not initialize the fire shader object.", L"Error", MB_OK);
        return false;
}
```

[ 코드 – applicationclass.cpp ] - Render()에서 filter mode에 따른 렌더 변화

```
// Fire 모델 렌더
else if (m_filterMode == 2) {
        m_Direct3D->EnableAlphaBlending();      // Alpha Blending 사용

        // ----- Fire Model Render ----- //
        worldMatrix = XMMatrixScaling(2.0f, 2.0f, 2.0f);
        worldMatrix *= XMMatrixTranslation(0.0f, 0.0f, -1.1f);

        m_FireModel->Render(m_Direct3D->GetDeviceContext());

        // ----- Fire Filter의 Texture들 이동 값 ----- //
        static float trans =0.0f;
        trans +=0.001f;

        if (trans >=1.0f) {
                trans =0.0f;
        }
        // ------------------------------------------ //

        result = m_FireShader->Render(m_Direct3D->GetDeviceContext(), m_FireModel->GetIndexCount(),
                                      worldMatrix, viewMatrix, projectionMatrix,
                                      m_FireModel->GetTexture(0), m_FireModel->GetTexture(1),
                                      m_FireModel->GetTexture(2), trans, m_fireBright);

        if (!result)
        {
                return false;
        }

        m_Direct3D->DisableAlphaBlending();     // Alpha Blending 사용 끝
}
```

fire shader를 렌더할 때 변화하는 translation과 bright값을 상수 버퍼로 전달해 역동적인 장면을 그려냅니다.

[ 코드 – firePS.hlsl ] - 움직이는 장면을 위한 상수 버퍼 선언

```
cbuffer TranslationBuffer
{
    float textureTranslation;       // 텍스쳐 움직임
    float bright;                   // 텍스쳐 밝기

    float2 padding;
};
```

[ 코드 – firePS.hlsl ] - 움직이는 텍스쳐 구현

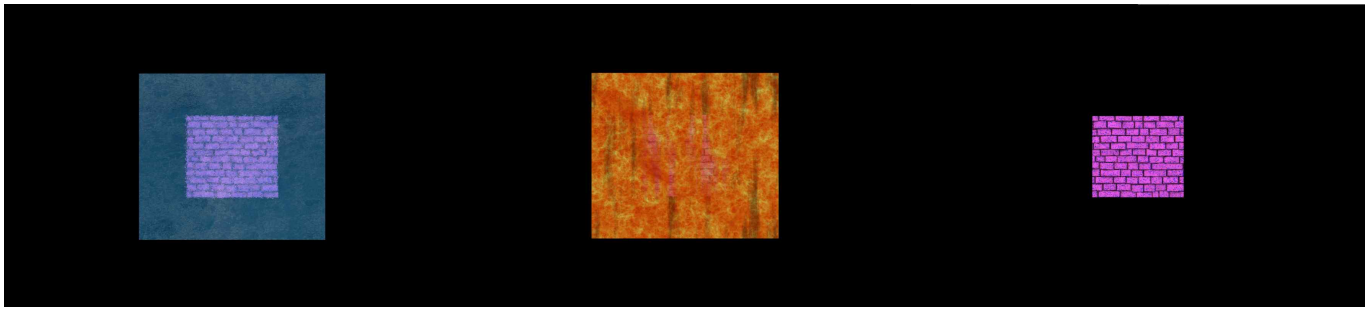
```
float4 FirePixelShader(PixelInputType input) : SV_TARGET
{
    float4 textureColor1;           float4 textureColor2;
    float4 alphaValue;              float4 color;

    input.tex.y -= textureTranslation;  // y좌표값 감소: 텍스쳐 아래로 움직임

    // Trans값을 사용한 Texture 샘플링
    alphaValue = alphaTexture.Sample(SampleType, input.tex - textureTranslation *0.1f);
    textureColor1 = shaderTexture1.Sample(SampleType, input.tex + textureTranslation *0.2f);
    textureColor2 = shaderTexture2.Sample(SampleType, input.tex - textureTranslation *0.2f);

    // bright 값이 커지면 어두워짐    // bright 값이 작아지면 밝아짐
    color = (textureColor1 * textureColor2) / bright;
    float points = alphaValue.x + alphaValue.y + alphaValue.z;

    if (points <=0.2)               color.a =0.7f;
    else if (points <=0.3f)         color.a =0.8f;
    else if (points <=0.5f)         color.a =0.9f;
    else                            color.a =1.0f;
    return color;
}
```

텍스쳐를 샘플링할 때 상수 버퍼로 가져온 translatioin 값을 좌표 연산에 사용하여 텍스쳐를 아래 움직임뿐만이 아닌 좌, 우로도 움직이게 하여 보다 자연스러운 용암 움직임을 작성했습니다. 상수 버퍼로 가져온 또 다른 값인 bright를 최종 컬러의 rgb 값에 나눠주어 밝기 조절이 가능하도록 연산을 추가했습니다.

마지막으로 alpha 값은 alpha texture 색상의 일정한 값에 따라 Pixel의 투명도를 조절하여 흐르는 용암 뒤로 Cube 모델이 자연스럽게 비치도록 분기를 설정해 구현했습니다.

### (07) - 사용자 입력을 통한 Filter 효과(3) - Filter 제거

숫자 0을 눌러 현재 적용된 필터를 제거합니다. Ice Filter 효과나 Fire Filter 효과가 적용된 상태에서 숫자 0을 누르면 현재 적용된 필터가 제거됩니다. 아래는 순서대로 숫자1, 숫자2, 숫자0을 누른 후의 모습입니다.

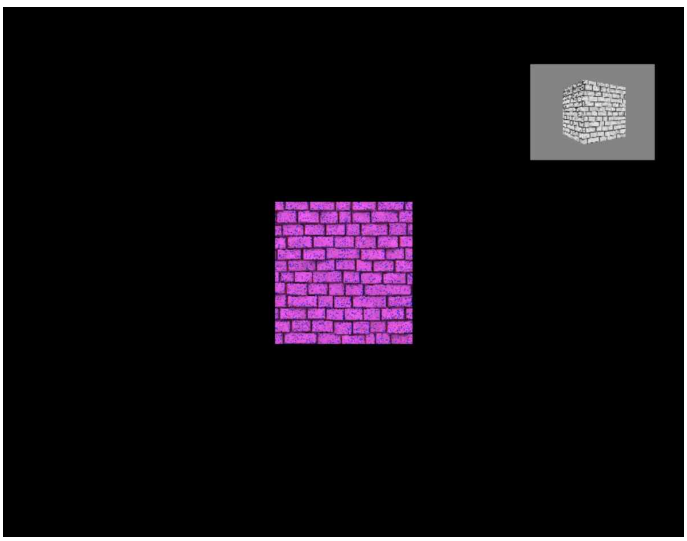


### (08) - Cube Model 관찰을 위한 Render To Texture

화면의 우측 상단에 Render To Texture가 나타나며, 이를 통해 각 필터를 적용함으로 인해 변화하는 Cube의 Texture 변화를 관찰할 수 있습니다. Filter를 적용한 상태에서는 Filter에 모델이 가려지기 때문에 변화를 관찰하기 어렵습니다. 따라서 필터 효과 없이 Render To Texture를 통해 Cube를 렌더합니다. Render To Texture를 통해 보이는 Cube는 일정한 속도로 회전하며 이를 통해 Cube의 4면 모두를 관찰할 수 있습니다.

Render To Texture에는 Light1(빨간색), Light2(파란색)이 아닌 LightPhong(하얀색) 조명이 사용됩니다. Shader에서는 Phong 조명 연산이 수행되며 이 조명은 Render To Texture 내의 모델에만 적용됩니다.

우상단 RTT를 통해 Phong 조명이 사용된 회전하는 Cube를 관찰



[ 코드 – applicationclass.cpp ] - Cube를 관찰하기 위한 RTT(Original RTT) 생성

```
// ---------- Original Renter Texture 생성 ---------- //
// 우상단에 필터 효과를 뺀 Original Model Render
// 렌더할 plane 생성
m_DisplayPlane =new DisplayPlaneClass;

result = m_DisplayPlane->Initialize(m_Direct3D->GetDevice(), 1.0f, 0.75f);
if (!result)
{
        MessageBox(hwnd, L"Could not initialize the window model object.", L"Error", MB_OK);
        return false;
}

// RTT 생성
m_RenderTextureOrigin =new RenderTextureClass;

result = m_RenderTextureOrigin->Initialize(m_Direct3D->GetDevice(), screenWidth, screenHeight,
                                           SCREEN_DEPTH, SCREEN_NEAR, 1);
if (!result)
{
        MessageBox(hwnd, L"Could not initialize the render texture object.", L"Error", MB_OK);
        return false;
}

// Plane의 Texture 생성
m_TextureShader =new TextureShaderClass;

result = m_TextureShader->Initialize(m_Direct3D->GetDevice(), hwnd);
if (!result)
{
        return false;
}

// Cube Model의 Texture 생성
m_OriginNormalShader =new OriginNormalShaderClass;

result = m_OriginNormalShader->Initialize(m_Direct3D->GetDevice(), hwnd);
if (!result)
{
        MessageBox(hwnd, L"Could not initialize the origin normal map shader object.", L"Error", MB_OK);
        return false;
}
```

[ 코드 – applicationclass.cpp ] - RenderToTextureOrigin() 함수 구현

```
bool ApplicationClass::RenderSceneToTextureOrigin(float rotation) {
	XMMATRIX worldMatrix, viewMatrix, projectionMatrix;
	bool result;

	m_RenderTextureOrigin->SetRenderTarget(m_Direct3D->GetDeviceContext());
	m_RenderTextureOrigin->ClearRenderTarget(m_Direct3D->GetDeviceContext(),
                                                0.5f, 0.5f, 0.5f, 1.0f); // 배경 색 지정
	m_Camera->SetPosition(0.0f, 0.0f, -5.0f);
	m_Camera->Render();

	m_Direct3D->GetWorldMatrix(worldMatrix);
	m_Camera->GetViewMatrix(viewMatrix);
	m_RenderTextureOrigin->GetProjectionMatrix(projectionMatrix);

	worldMatrix = XMMatrixRotationRollPitchYaw(0.0f, -1.3f + rotation, 0.0f);

	// Cube 렌더
	m_Model->Render(m_Direct3D->GetDeviceContext());

	// Shader 렌더: RTT를 위한 Origin Normal Shader와 Phong 조명 사용
	result = m_OriginNormalShader->Render(m_Direct3D->GetDeviceContext(),
                                                m_Model->GetIndexCount(),
                                                worldMatrix, viewMatrix, projectionMatrix,
                                                m_Model->GetTexture(0), m_Model->GetTexture(1),
                                                m_LightPhong->GetDirection(),
                                                m_LightPhong->GetDiffuseColor(), m_shiftColor);
	if (!result)
	{
		return false;
	}

	m_Direct3D->SetBackBufferRenderTarget();
	m_Direct3D->ResetViewport();

	return true;
}
```

기존의 Cube에 사용되던 Shader는 Light를 두 개 사용하기 때문에 단 하나의 Phong 조명을 사용하는 새로운 Shader 클래스를 작성했습니다. 이 Shader는 texture cube, texture normal, light, shift color 값을 인자로 받습니다. shift color는 filter mode에 따라 텍스쳐의 색상을 변하게 만드는 변수로 Shader의 상수 버퍼로 전달됩니다.

[ 코드 – originPS.hlsl ] - 필터에 따라 변하는 shift color 값을 상수 버퍼로 선언

```
cbuffer ShiftColorBuffer
{
    float4 shiftColor;
};
```

Cube Model에 Bump map이 사용됐기 때문에 normal을 구할 때와 Reflect Vector를 구할 때 모두 bump normal을 사용했습니다. Phong 조명 연산을 위해 Ambient, Diffuse, Specular를 계산했는데, Diffuse는 빛의 방향과 bump normal의 내적 연산으로 구현하고 Specular를 위한 reflect vector 또한 bump normal을 사용해 light direction과의 내적을 한 후 지수 연산(위의 코드에서는 64를 사용)을 통해 Specular 값을 도출해 만들었습니다.

[ 코드 – originPS.hlsl ] - Phong 조명 구현

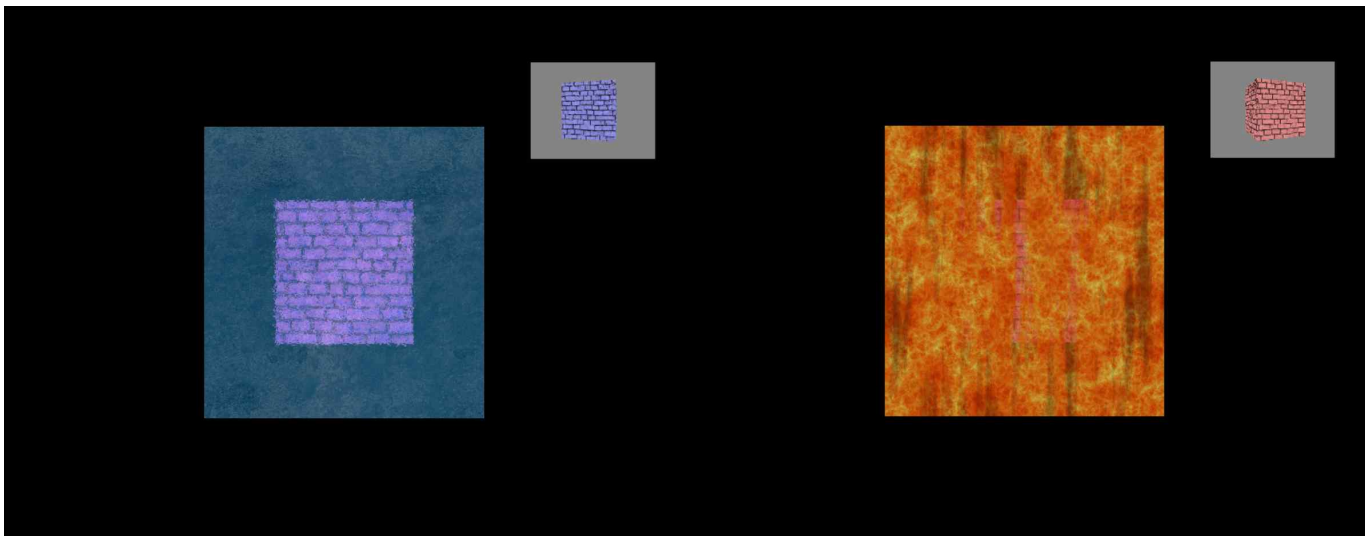
```
float4 OriginPixelShader(PixelInputType input) : SV_TARGET
{
    float4 textureColor;     float4 bumpMap;     float3 bumpNormal;

    float3 lightDir;     float lightIntensity;

    int ex;
    float3 viewingVec;     float3 reflectVec;     float sLightIntensity;

    float4 color;     float4 ambientColor;
    float ambientStrength =0.1f; // Ambient 강도 설정

    // ----- texture sampling ----- //
    textureColor = shaderTexture1.Sample(SampleType, input.tex);
    bumpMap = shaderTexture2.Sample(SampleType, input.tex);
    bumpMap = (bumpMap *2.0f) -1.0f;

    // ----- normal 구하기 ----- //
    bumpNormal = (bumpMap.x * input.tangent) + (bumpMap.y * input.binormal) +
                 (bumpMap.z * input.normal);
    bumpNormal = normalize(bumpNormal);



    // ----- 조명의 방향 ----- //
    lightDir =-lightDirection;
    lightDir = normalize(lightDir); // 방향 벡터 정규화

    // ----- Norm & Dir 벡터 내적 후 0~1값으로 치환 ----- //
    lightIntensity = saturate(dot(bumpNormal, lightDir));    // diffuse

    // ----- View 벡터 ----- //
    viewingVec = normalize(-input.position.xyz);     // View 벡터 == input의 반대 방향

    // ----- Reflect 벡터 ----- //
    reflectVec = reflect(lightDir, bumpNormal);

    // ----- Specular ----- //
    // ReflecVec & ViewVec 내적 후 0~1값 치환 후 지수 연산
    ex =64;
    sLightIntensity = saturate(pow(dot(reflectVec, viewingVec), ex));

    // ---- Ambient ----- //
    ambientColor = diffuseColor * ambientStrength;


    // ----- Phong Shading 계산 식 ----- //
    color = saturate((ambientColor) + (diffuseColor * lightIntensity) + (diffuseColor * sLightIntensity));


    // Combine the final light color with the texture color.
    color = saturate(color * textureColor * 1.3); // 1.3은 임시 값!

    // ---- filter에 따른 색상 연산 추가 ----- //
    color = saturate(color * shiftColor);

    return color;
}
```

## (09) - Render To Texture를 통한 Cube 모델의 변화 관찰

Ice Filter를 적용하면 shift color 변수의 r, g 값을 감소시켜 우측 상단 Render To Texture를 통해 보이는 Cube Model이 점차 파랗게 보이도록 합니다. Fire Filter 또한 비슷하게 g, b 값을 감소시켜 점차 빨갛게 변하게 됩니다. 아래는 각 필터를 적용한 후 일정 시간이 지나 Render To Texture의 Cube 모델이 변한 예시 이미지입니다.

Filter를 통해 Cube의 색상이 변하더라도 화면 중앙의 Cube는 색상이 변하지 않고 오직 Render Texture 내부 Cube의 Texture 색상만 변합니다. 또한, 필터를 변경할 땐 원래의 색상으로 돌아간 다음 서서히 색상히 변하도록 하고, 숫자 0을 눌러 Filter를 제거하면 원본 색상을 계속 유지합니다.

(좌) Ice 적용 후 파랗게 변함 / (우) Fire 적용 후 빨갛게 변함



[ 코드 – applicationclass.cpp ] - ChangeFilter()에서 filtermode에 따른 색상 변화 (Fire Filter 예시)

```
case 2:
        // 모델의 x좌표 값이 범위 안에 있다면 빨갛게
        if (m_cubePosX >=-1.0f && m_cubePosX <=1.0f) {
                // shift color 연산 수행 -> 빨갛게
                m_shiftColor.x =1.0f;
                m_shiftColor.y -= m_shiftValue;
                m_shiftColor.z -= m_shiftValue;
        }

        break;
```

각 Filter에서 방향키 ↑↓를 누름으로 필터의 강도를 변경함에 따라 색상 변화 속도 또한 증가하고 감소합니다. 강도에 따라 m_shiftValue 값이 증감하며 값의 범위는 0.001~0.0005로 제한됩니다. Input에 따라 변화하는 shiftValue는 m_shiftColor의 색상에 적용되어 Filter에 따라 색상이 변하도록 하고, m_shiftColor는 Render To Texture의 Cube 모델의 Shader에 상수 버퍼로 전달되어 Texture, Texture Normal, Phong 조명 연산 등을 수행한 최종 Pixel Color에 누적해주어 해당 색상으로 색상이 서서히 변하는 현상을 구현했습니다.

Filter Mode를 변경할 때 m_shiftColor와 m_shiftValue는 모두 초기 기본 값으로 초기화되어 이전 Filter의 결과가 다른 Filter에 누적되어 영향을 미치지 않도록 초기화 코드를 작성했습니다.

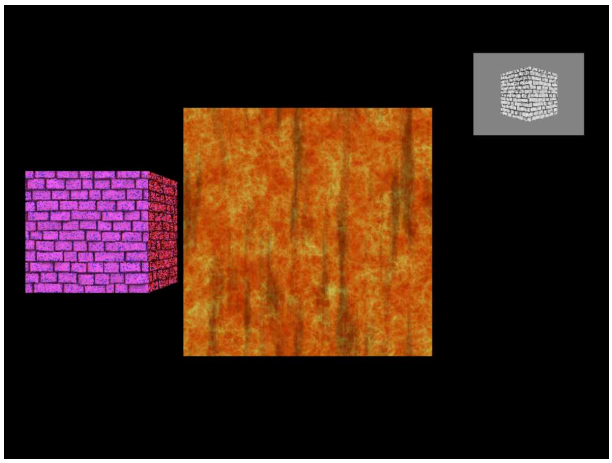
[ 코드 – originPS.hlsl ] - 사용자 Input에 따라 변하는 shiftColor를 상수 버퍼로 선언하고 최종 색상에 연산

```
cbuffer ShiftColorBuffer
{
    float4 shiftColor;
};
```

```
(Pixel Shader Main...)

// ----- Phong Shading 계산 식 ----- //
color = saturate((ambientColor) + (diffuseColor * lightIntensity) + (diffuseColor * sLightIntensity));

// Combine the final light color with the texture color.
color = saturate(color * textureColor *1.3); // 1.3은 임시 값!

// ---- filter에 따른 색상 연산 추가 ----- //
color = saturate(color * shiftColor);

return color;
```

### (10) - Cube의 위치에 따른 Render Texture 변화 제한

사용자가 Cube를 움직여 Filter 밖으로 벗어난 경우 Render To Texture에 가해지는 색상 변화가 일어나지 않습니다. Cube Position 값이 일정 범위 내에 있고 Filter가 켜진 상태에서만 색상 변화가 일어납니다.

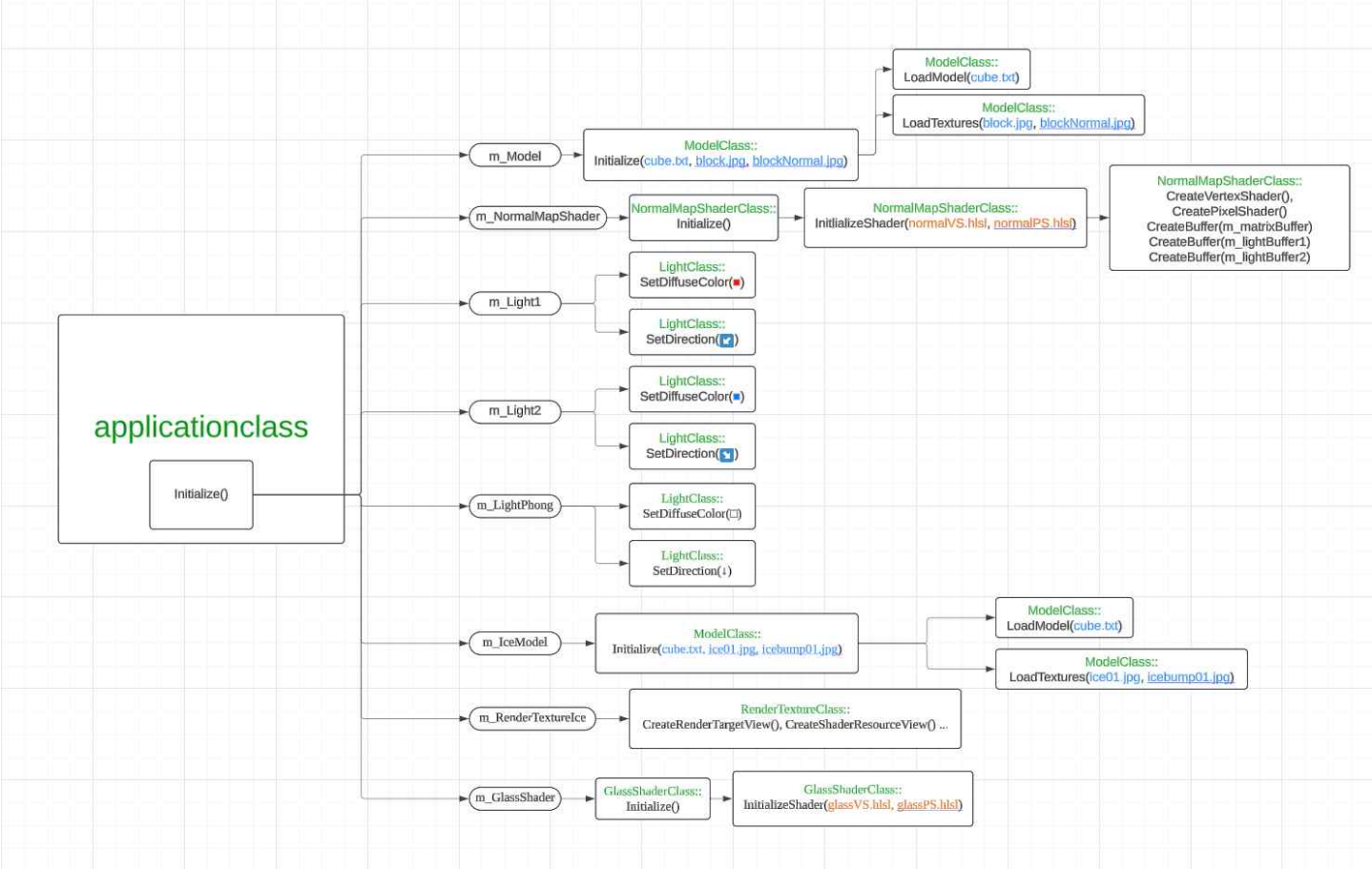


[ 코드 – applicationclass.cpp ] - ChangeFilter(): Cube가 Filter 범위를 벗어난 경우, 색상 변화 제한

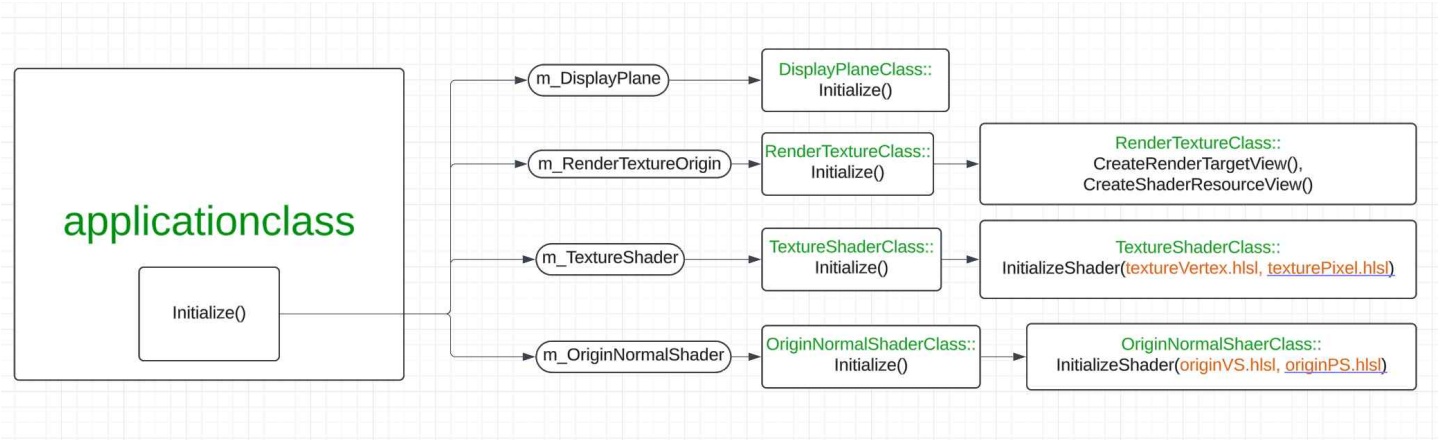
```
switch (m_filterMode) {
        case 1:
                result = RenderSceneToTextureIce(m_cubePosX);

                if (!result)
                {
                        return false;
                }

                // 모델의 x좌표 값이 범위 안에 있다면 파랗게
                if (m_cubePosX >=-1.0f && m_cubePosX <=1.0f) {
                        // shift color 연산 수행 -> 파랗게
                        m_shiftColor.x -= m_shiftValue;
                        m_shiftColor.y -= m_shiftValue;
                        m_shiftColor.z =1.0f;
                }

                break;
```

## [System Flow]

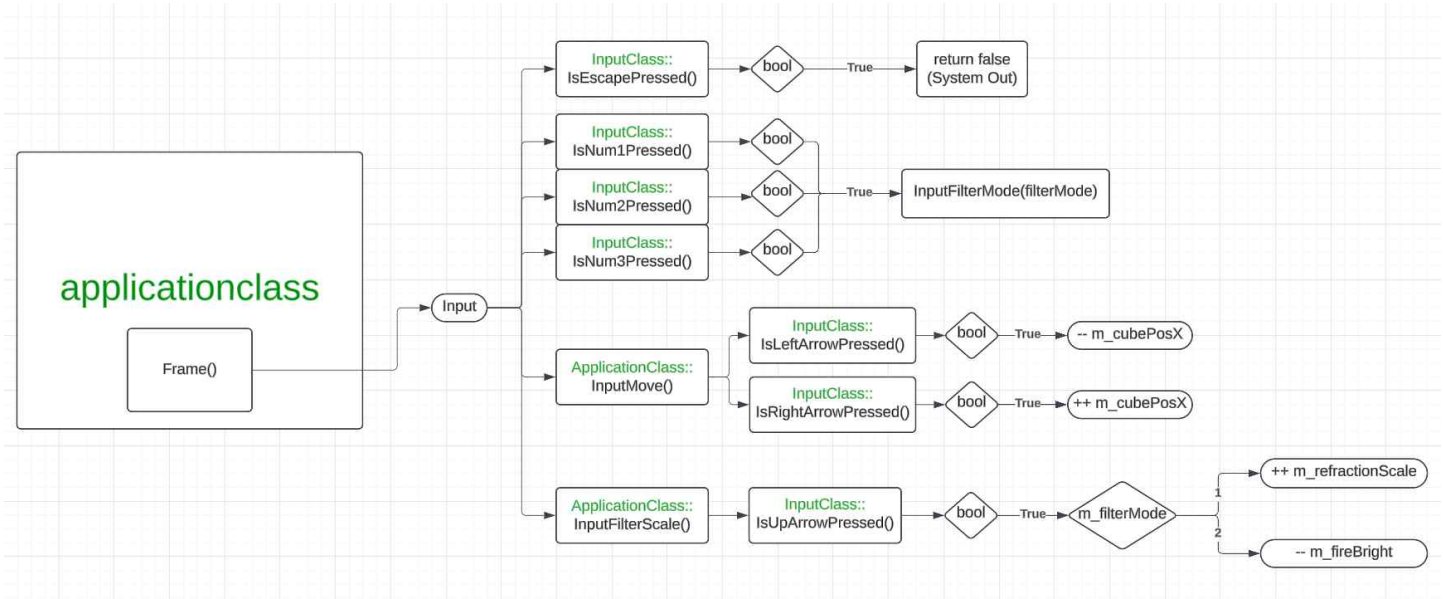
## A. 초기화 System Flow 1



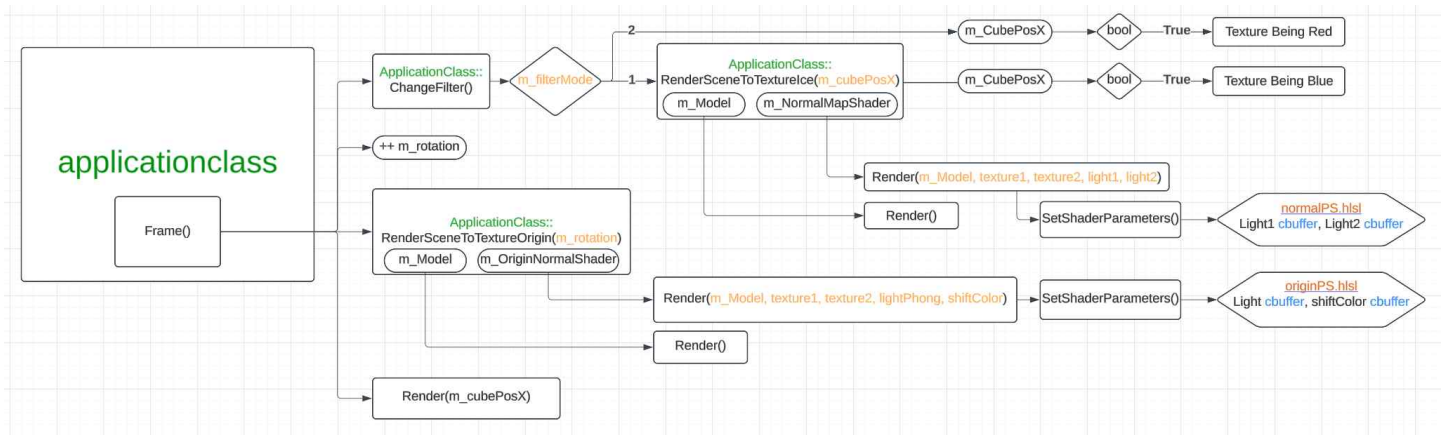

## B. 초기화 System Flow 2



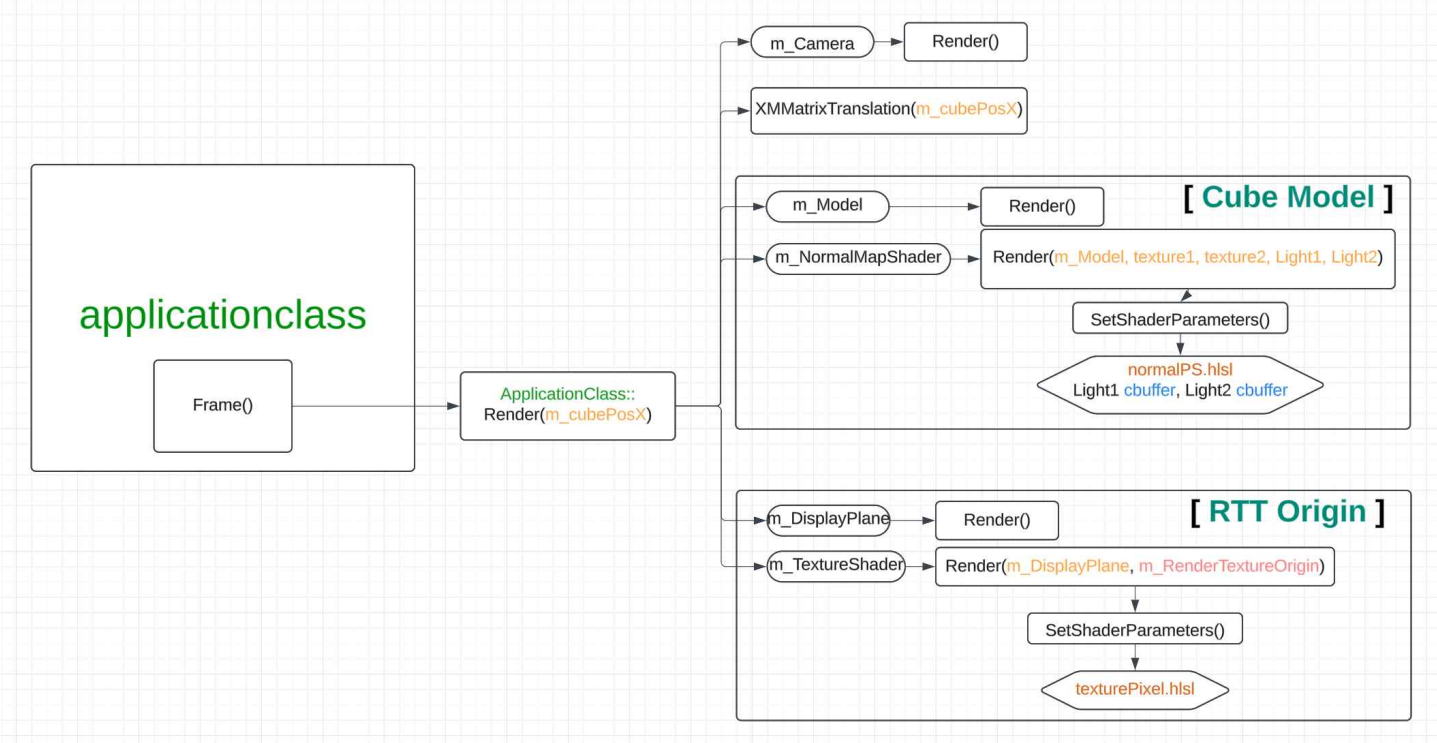
## C. Frame System Flow - Input




## D. Frame System Flow - Render



## E. Render System Flow (Cube, 우측 상단 Render Texture)




## F. Render System Flow (Filter Mode)