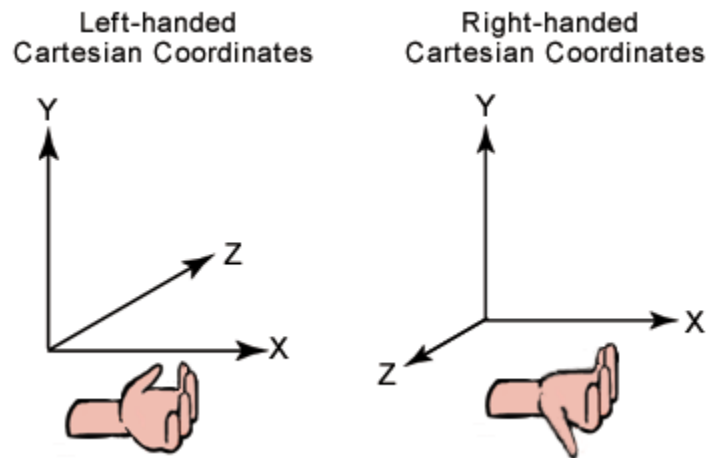


5.1. 렌더링 시스템

좌표계

Nebula의 기본 좌표계는 OpenGL과 동일한 오른손 좌표계입니다. 즉, y축이 위로 향하고 있는 오른손 좌표계를 사용하고 있습니다.



다른 플랫폼별 그래픽 API나 DCC(Digital Contents Creation) 툴의 좌표계 중에서 알아 둘만한 것들 중에는 다음과 같은 것들이 있습니다.

- OpenGL - 오른손 좌표계
- Direct3D - 왼손 좌표계
- 3ds Max - 오른손 좌표계 (단, z축이 위로 향한다)
- Maya - 오른손 좌표계

왼손 좌표계와 오른손 좌표계는 기본적으로 y축이 위로 향하고 있지만 3ds Max와 같이 z축이 위로 향하고 있는 경우도 많습니다.

메쉬

보통 메쉬(Mesh)라고 하면 삼각형으로 구성되어 있는 데이터로 DCC(Digital Contents Creation) 툴로부터 익스포트된 데이터를 의미합니다. 대부분 이들 데이터는 렌더링되어 화면에 나타나게 됩니다.

Nebula에서도 이러한 메쉬 데이터를 사용하고 있으며 자체 메쉬 데이터 파일 포맷을 가지고 있습니다.

Nebula에서 사용하는 메쉬 데이터 포맷은 두 가지가 있습니다. 첫 번째는 ASCII 파일 포맷으로 .n3d2 파일 확장자를 가지는 것이 있고, 두 번째는 바이너리 파일 포맷으로 이것은 파일 확장자로 .nvx2 확장자를 가집니다. 그러면 같은 타입의 데이터에 대해서 왜 이렇게 두 가지 종류의 파일로 구분되어 있는 것일까요?

ASCII 포맷은 눈으로 직접 확인이 가능하므로 개발할 때 경우에 따라 디버깅할 필요가 있을 때 유용합니다. 이와는 반대로 바이너리 포맷은 작은 크기의 용량을 가지므로 빠른 입출력이 가능하기 때문에 실제 릴리즈된 게임에서는 바이너리 포맷을 사용해야 합니다. 그래서 당연히 Nebula에서는 .n3d2 파일을 .nvx2로 변환하는 툴이 있습니다.

그런데 Nebula에서 기본적으로 제공하는 툴들을 보면 대부분 커맨드 라인(command-line) 툴들이 많습니다. 그 이유는 Nebula가 유닉스 운영 체제의 특징을 따르고 있기 때문입니다.

이렇게 여러 개의 작은 크기를 가지는 커맨드 라인 툴들을 만들어서 필요한 데이터 처리를 하는 이유는 무엇일까요?

크기가 작은 프로그램은 작성하기가 쉬울 뿐만 아니라 유지 보수하기도 쉽습니다. 당연히 버그가 발생할 소지도 적지만 발생하더라도 쉽게 수정할 수가 있습니다. 그래서 유닉스에서는 여러 개의 작업들을 처리하는 하나의 큰 프로그램을 작성하는 것이 아니라 하나의 프로그램은 하나의 일만 처리하도록 작성하는 것을 기본으로 하고 있으며 하나의 데이터에 다른 여러 종류의 처리가 필요한 경우에는 각각의 처리를 위한 프로그램들을 개별적으로 작성한 다음에 이를 파이프를 통해서 연결하여 처리할 수 있는 메카니즘을 제공하고 있습니다. 유닉스에서 제공하는 이 파이프(pipe)라는 메카니즘은 두 개의 프로세스 간의 통신을 설명하는 것으로 하나의 프로세스가 쓰고 다른 프로세스가 읽는 큐 형태로 순차적으로 프로세스를 거치면서 작업을 행할 수 있다는 특징을 가지고 있습니다. 쉽게 말하면 한 명령어(혹은 프로세스)의 출력을 다른 명령어의 입력으로 사용해서 결과를 얻는 것이라고 생각할 수 있습니다.

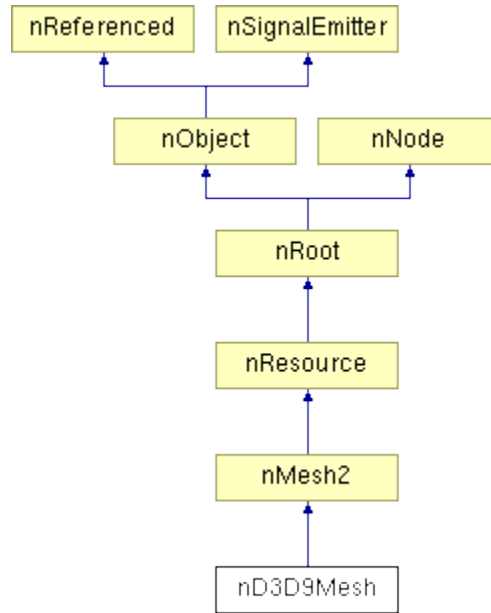
Nebula에서는 실제로 유닉스의 파이프를 사용하는 것은 아니며 다만 유닉스 응용 프로그램을 참고하여 비슷한 원리로 작업이 이루어질 수 있도록 흉내내고 있습니다.

파이프 작업에 대한 예 삽입할 것-Hyoun Woo Kim 4/14/08 11:10 PM

일반적인 커맨드 라인 툴의 경우 타이핑하는 수고를 제외하면 GUI 환경의 윈도우 프로그램에 비해서 크기도 작을뿐만 아니라 작성하기도 훨씬 쉬우며 필요에 따라서는 배치(batch) 작업을 통한 일괄 처리도 쉽게 할 수 있다는 장점이 있습니다. 하지만 때에 따라 GUI가 필요한 경우에는 Nebula에서는 Tcl/Tk나 wxPython을 사용하여 Win32나 MFC 등을 사용하여 C/C++ 기반으로 작성하는 것보다 훨씬 더 쉽게 작성할 수도 있습니다.

Nebula에서의 메쉬에 대한 처리를 하는 클래스는 nMesh2 클래스(code/nebula2/inc/gfx2)입니다. 이 클래스에서는 메쉬 데이터에 대한 인터페이스(멤버 함수)들을 제공하는 것 뿐만 아니라 정점 버퍼(Vertex Buffer)와 색인 버퍼(Index Buffer)의 생성과 관련한 인터페이스들도 제공하고 있습니다.

이러한 특징 때문에 nMesh2 클래스는 nD3D9Mesh 클래스 처럼 Direct3D 렌더링 API의 서브 클래스에 대한 정의가 필요 합니다.



다음 코드는 nD3D9Mesh 에서 정점 버퍼를 생성하는 코드입니다.

code/nebula2/src/gfx2/nd3d9mesh_main.cc

```

void nD3D9Mesh::CreateVertexBuffer()
{
    n_assert(this->vertexBufferSize > 0);
    n_assert(0 == this->privVertexBuffer);
    n_assert(0 == this->vertexBuffer);
    if (ReadOnly & this->vertexUsage)
    {
        // this is a read-only mesh which will never be rendered
        // and only read-accessed by the CPU, allocate private
        // vertex buffer
        this->privVertexBuffer = n_malloc(this->vertexBufferSize);
        n_assert(this->privVertexBuffer);
    }
    else
    {
        nD3D9Server* gfxServer = (nD3D9Server*)nGfxServer2::Instance();
        n_assert(gfxServer->d3d9Device);
        // this is either a WriteOnce or a WriteOnly vertex buffer,
        // in both cases we create a D3D vertex buffer object
        DWORD d3dUsage = D3DUSAGE_WRITEONLY;
        D3DPOOL d3dPool = D3DPOOL_MANAGED;
    }
}
  
```

```

this->d3dVBlockFlags = 0;
if (WriteOnly & this->vertexUsage)
{
    d3dUsage = (D3DUSAGE_DYNAMIC | D3DUSAGE_WRITEONLY);
    d3dPool = D3DPOOL_DEFAULT;
    this->d3dVBlockFlags = D3DLOCK_DISCARD;
}
if (ReadWrite & this->vertexUsage)
{
    d3dUsage = D3DUSAGE_DYNAMIC;
    &nbsp;nbsp;nbsp; d3dPool = D3DPOOL_SYSTEMMEM;
}
if (RTPatch & this->vertexUsage)
{
    d3dUsage |= D3DUSAGE_RTPATCHES;
}
if (PointSprite & this->vertexUsage)
{
    d3dUsage |= D3DUSAGE_POINTS;
}
// create buffer with software processing flag
// if the NeedsVertexShader hint is enabled, and the d3d device
// has been created with software or mixed vertex processing
if (gfxServer->AreVertexShadersEmulated() && (NeedsVertexShader & this-
>vertexUsage))
{
    d3dUsage |= D3DUSAGE_SOFTWAREPROCESSING;
    d3dUsage &= ~D3DUSAGE_WRITEONLY;
    d3dPool = D3DPOOL_SYSTEMMEM;
}
// create the vertex buffer
HRESULT hr = gfxServer->d3d9Device->CreateVertexBuffer(
    this->vertexBufferSize,
    d3dUsage,
    0,
    d3dPool,
    &(this->vertexBuffer),
    NULL);
n_dxtrace(hr, "CreateVertexBuffer() failed in nD3D9Mesh");
n_assert(this->vertexBuffer);
}
}

```

다음 코드는 nD3D9Mesh 에서 색인 버퍼(Index Buffer)를 생성하는 코드입니다.

code/nebula2/src/gfx2/nd3d9mesh_main.cc

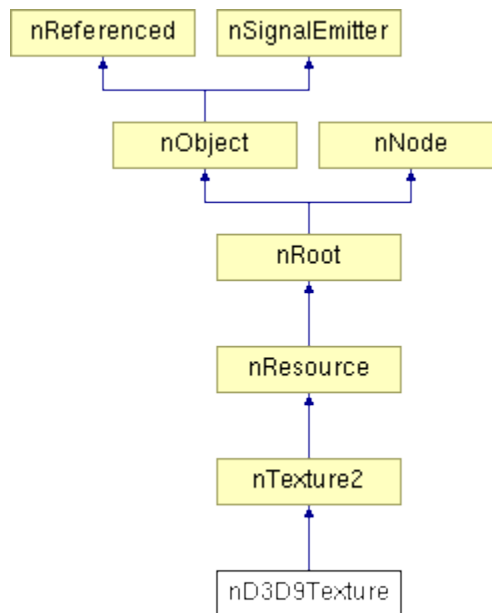
```
void nD3D9Mesh::CreateIndexBuffer()
{
    n_assert(this->indexBufferSize > 0);
    n_assert(0 == this->indexBuffer);
    n_assert(0 == this->privIndexBuffer);
    if (ReadOnly & this->indexUsage)
    {
        this->privIndexBuffer = n_malloc(this->indexBufferSize);
        n_assert(this->privIndexBuffer);
    }
    else
    {
        nD3D9Server* gfxServer = (nD3D9Server*)nGfxServer2::Instance();
        n_assert(gfxServer->d3d9Device);
        DWORD d3dUsage      = D3DUSAGE_WRITEONLY;
        D3DPOOL d3dPool     = D3DPOOL_MANAGED;
        this->d3dIBLockFlags = 0;
        if (WriteOnly & this->indexUsage)
        {
            d3dUsage = (D3DUSAGE_DYNAMIC | D3DUSAGE_WRITEONLY);
            d3dPool = D3DPOOL_DEFAULT;
            this->d3dIBLockFlags = D3DLOCK_DISCARD;
        }
        if (ReadWrite & this->indexUsage)
        {
            d3dUsage = D3DUSAGE_DYNAMIC;
            d3dPool = D3DPOOL_SYSTEMMEM;
        }
        if (RTPatch & this->indexUsage)
        {
            d3dUsage |= D3DUSAGE_RTPATCHES;
        }
        if (PointSprite & this->indexUsage)
        {
            d3dUsage |= D3DUSAGE_POINTS;
        }
        // create buffer with software processing flag
        // if the NeedsVertexShader hint is enabled, and the d3d device
```

```

// has been created with software or mixed vertex processing
if (gfxServer->AreVertexShadersEmulated() && (NeedsVertexShader & this-
>vertexUsage))
{
    d3dUsage |= D3DUSAGE_SOFTWAREPROCESSING;
    d3dUsage &= ~D3DUSAGE_WRITEONLY;
    d3dPool = D3DPOOL_SYSTEMMEM;
}
HRESULT hr = gfxServer->d3d9Device->CreateIndexBuffer(
    this->indexBufferSize,
    d3dUsage,
    D3DFMT_INDEX16,
    d3dPool,
    &(this->indexBuffer),
    NULL);
n_dxtrace(hr, "CreateIndexBuffer failed in nD3D9Mesh");
n_assert(this->indexBuffer);
}
}

```

텍스처



하드 디스크에서 텍스처를 읽는 것이 아니라 실행 중에 텍스처를 생성할 경우에는 다음의 방법으로 텍스처를 생성할 수 있습니다.

```
// 텍스처 객체 인스턴스는 nGfxServer2의 멤버함수를 통해서 생성한다.
tex = (nTexture2 *)refGfx2->NewTexture("mytexture");
if (!tex->IsUnloaded())
{
    int width, height;
    width = height = 128;
    // 16bit 텍스처 생성
    &nbsp; tex->SetUsage(nTexture2::CreateEmpty);
    tex->SetType(nTexture2::TEXTURE_2D);
    tex->SetWidth(width);
    tex->SetHeight(height);
    tex->SetFormat(nTexture2::A1R5G5B5);
    tex->Load();

    // 생성한 텍스처에 흰색으로 채운다.
    struct nTexture2::LockInfo surf;
    if (tex->Lock(nTexture2::WriteOnly, 0, surf))
    {
        unsigned short *surface = (unsigned short *)surf.surfPointer;
        for (unsigned int pixelByte=0; pixelByte < width*height; pixelByte++)
            surface[pixelByte] = 0xffff;
        tex->Unlock(0);
    }
}
```

Nebula의 기본 그래픽스 API는 Direct3D이므로 텍스처 객체를 생성하기 위해서는 Direct3D의 API를 사용하게 됩니다. 이 때문에 Nebula의 텍스처 객체 인스턴스를 생성하는 인터페이스인 `NewTexture()` 함수는 `nGfxServer2` 클래스의 멤버 함수입니다. 텍스처 객체를 생성한 다음에는 생성할 텍스처를 설정하고 `nTexture2::Load()` 함수를 호출하게 되면 텍스처가 생성이 됩니다. 생성한 텍스처는 픽셀들의 버퍼인데 이 버퍼에 임의의 값을 쓰기 위해서(예:특정 색상 채우기 등) 버퍼에 접근하려면 `nTexture2::Lock()` 함수를 호출해서 버퍼에 대한 포인터를 얻어 올 수 있습니다. 쓰기가 끝난 다음에는 `nTexture2::Unlock()` 함수를 호출해서 해제해 주어야 합니다.

반대로 버퍼의 값을 읽기 위해서는 다음의 방법으로 값을 읽을 수 있습니다.

```
struct nTexture2::LockInfo surf;
tex->Lock(nTexture2::ReadOnly, 0, surf)
ushort *surface = (ushort*)surf.surfPointer;
ushort color = surface[x + y*surf.surfPitch];
tex->Unlock(0);
```

텍스처에 쓰는 또 다른 방법으로 다음의 코드에서와 같이 메모리 상의 이미지 데이터를 텍스처로 카피하는 방법이 있습니다.

```
// 위에서 나온 예와 같이 임의의 텍스처를 생성.
...

// 텍스처의 버퍼(surface)에 대한 포인터를 얻는다.
nSurface *surface;

// 쓰기 버퍼(surface)를 레벨 0에 설정 .
tex->GetSurfaceLevel("/tmp/surface", 0, &surface);

// 원본 이미지 데이터를 텍스처로 복사.
surface->LoadFromMemory(imageData, dstFormat, width, height, imagePitch);
...
```

위의 코드 snip을 실제 게임에는 어떤 부분에 응용이 가능한지에 대한 설명이 추가되면 좋겠다. -
Hyoun Woo Kim 4/17/08 2:27 PM

쉐이더 (Shader)

쉐이더는 3차원 프로그램에서 GPU를 프로그래밍할 수 있도록 하기 위해서 등장한 것으로 지금은 렌더링에 있어서 쉐이더를 빼고는 이야기를 할 수 없을 정도로 쉐이더는 3D 엔진에 있어 이미 보편화되어 있는 기술일 뿐만 아니라 렌더링 시스템에서는 핵심이 되는 기술입니다. 쉐이더가 3D 오브젝트가 렌더링 될 때 렌더링 효과에 대한 것이므로 원하는 렌더링 효과를 위해서는 쉐이더 코드를 어떻게 작성하느냐도 중요하지만 3D 엔진에서는 이러한 수 많은 쉐이더 코드를 유연하고 효율적으로 처리할 수 있는 시스템을 구현하는 것이 중요합니다. 유연하고 효율적인 쉐이더 시스템을 위해서는 엔진에서 다음과 같은 항목들이 고려가 되어야 합니다.

- 폴백(Fallback) 시스템의 지원
- 렌더링 API에 따른 추상화
- 유연한 쉐이더 파라미터 시스템
- 배치(Batch) 처리

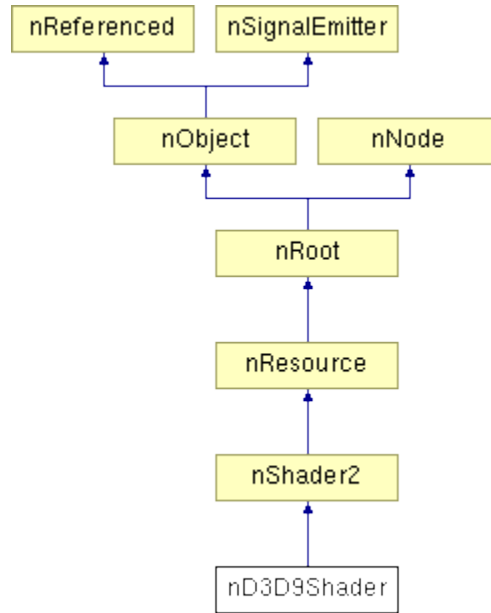
XBox 등과 같은 콘솔 시스템 처럼 모든 사용자의 하드웨어 환경이 동일한 경우에는 지원해야 하는 셰이더 모델과 렌더러도 하나의 하드웨어 환경만 고려하면 되지만 PC 플랫폼과 같이 사용자 하드웨어 시스템이 다양한 경우에는 여러 시스템에서도 잘 실행되도록 지원하는 것이 중요합니다. 그래서 DX9을 지원하는 최신의 그래픽 카드를 가진 사용자에게는 해당 하드웨어 스펙을 최대한 이용한 멋진 그래픽을 보여 주어야 하며 그렇지 못한 경우, 심지어는 DX7만을 지원하는 하드웨어 환경에서는 멋진 그래픽은 아니더라도 최소한 게임을 실행할 수 있도록은 해야 합니다. 이렇게 하위 버전에 대해서 최소한의 스펙으로도 구동 가능하도록 보장해 주는 것이 바로 **Fallback** 시스템의 지원입니다.

이것은 Nebula2를 작성할 때 DirectX의 버전의 DX9이 등장했을 때입니다. DX9은 이전 버전인 DX8와 비교할 때 셰이더 코드로 HLSL을 지원하는 것이 특징인데 Nebula2의 렌더러는 이것을 이용하여 DX9 계열의 카드의 성능을 최대한 이용할 수 있으면서도 이전의 고정 파이프라인(fixed function pipeline)인 DX7 계열의 카드도 지원하도록 (Fallback) 작성된 것이 특징입니다. Nebula2 엔진은 셰이더 중심의 엔진으로 Fallback 시스템의 지원도 셰이더 코드를 사용해서 지원하는 것이 특징입니다. Nebula2는 하나의 렌더러 모델로 DX9 계열의 하드웨어와 DX7 계열의 하드웨어를 모두 지원하고 있는데 양쪽 모두 셰이더 코드를 사용합니다. 고정 파이프 라인을 지원하기 위해서 엔진 내부에 하드코딩한 것이 아니라 고정 파이프 라인도 셰이더 코드를 사용해서 지원하고 있습니다. 셰이더 중심의 엔진이라고 하는 것은 이런 의미입니다. 폴백 시스템 지원을 위해서 두 개의 렌더러가 있는 엔진이거나 아니면 엔진 내부에 이전의 폴백 시스템 지원을 위한 코드가 하드 코딩되어 있는 경우에는 렌더러의 수정이나 새로운 기능의 추가가 매우 어렵고 또 유지 보수하기도 힘듭니다. Nebula2에서는 이런 것들을 모두 엔진 외부에서 셰이더 코드를 사용하여 변경에 유연한 것이 특징입니다.

렌더링 API에 대한 추상화는 엔진이 플랫폼에 따라 D3D나 OpenGL API를 적절하게 지원할 수 있는 것을 의미합니다. Nebula2에서 셰이더는 렌더링 API에 따른 추상화를 위해서 nShader2 클래스를 정의 하며 이 클래스가 있는 파일은 gfx2/nshader.h 입니다. 렌더링 API로 Direct3D를 사용하는 경우라면 셰이더 코드는 HLSL을 사용하는 이펙트 파일을 셰이더 파일로 사용해야 하며 이것은 nShader2 클래스를 상속하는 nD3D9Shader 클래스에서 처리합니다. Nebula2에서 렌더링 API는 D3D를 기본으로 사용하고 있으므로 셰이더 역시 HLSL 코드가 기본입니다. 만약에 셰이더 코드로 GLSL이나 CgFX를 사용하려면 nShader2 클래스를 상속하는 새로운 클래스를 정의하여 처리하면 됩니다.

Nebula2에서는 셰이더 코드도 하나의 리소스로 간주합니다. 그래서 셰이더도 다른 그래픽스 리소스와 마찬가지로 nGfxServer2의 멤버 함수를 사용해서 생성하도록 되어 있습니다. 셰이더 리소스 인스턴스를 생성하기 위한 C++코드는 아래와 같습니다.

```
nShader2* shader = gfxServer::NewShader("/resources/shader/metal0"/*NOH 셰이더 리소스 이름*/);
```



렌더링 상태(Render State)를 처리하기 위한 셰이더 파라미터(Shader Parameter)는 문자열과 쌍을 이루는 열거형(enum) 자료형으로 정의되어 있습니다. 셰이더 파라미터는 `gfx2/nshaderparams.h` 파일에 정의되어 있으며 렌더링 상태는 같은 폴더의 `nshaderstate.h` 파일에 정의되어 있습니다. 이것은 엔진 코드 내에서 렌더링 상태를 하드 코딩하지 않고 렌더링 상태를 모두 외부에서 설정할 수 있도록 하기 위해서입니다. 대신에 새로운 렌더링 상태를 추가하기 위해서는 이를 `nshaderstate.h` 파일에 추가하고 `nshaderstate.cc` 파일의 문자열 테이블도 함께 갱신해야 합니다.

gfx2/nshaderstate.h

```

enum Param
{
    Model = 0,           // matrix: the model matrix (aka World)
    InvModel,           // matrix: the inverse model matrix
    View,               // matrix: the view matrix
    InvView,            // matrix: the inverse view matrix
    Projection,         // matrix: the projection matrix
    ModelView,          // matrix: the model*view matrix
    InvModelView,       // matrix: the inverse of the model view matrix
    ModelViewProjection, // matrix: the model*view*projection matrix

    (생략)...

};
  
```

gfx2/nshaderstate.cc

```
static const char* StateTable[nShaderState::NumParameters] =
{
    "Model",
    "InvModel",
    "View",
    "InvView",
    "Projection",
    "ModelView",
    "InvModelView",
    "ModelViewProjection",

    (생략)...

};
```

쉐이더 파라미터에 대한 문자열이 필요한 경우 `StateTable[nShaderState::Model]`는 "Model"이라는 문자열을 얻게 됩니다. 또 문자열을 이용해서 원하는 쉐이더 파라미터로 치환하기 위해서는 이와 반대로 하면 됩니다.

배치(Batch)란 한 번의 렌더링 상태의 설정으로 가능한 많은 수의 다각형-일반적으로 삼각형-을 처리하는 것으로 렌더링의 최적화를 위해서 매우 중요한 개념입니다. 일반적으로 렌더링 상태의 설정은 재질(Material)의 변경에 가장 민감합니다. Nebula2에서는 쉐이더 중심의 특징 때문에 이 재질 역시 쉐이더를 통해서 지정하게 됩니다. 그래서 렌더링 속도를 빠르게 하기 위해서는 같은 재질(쉐이더)를 사용하는 오브젝트끼리 묶어서 처리해야 합니다. 그런데 게임마다 사용하는 쉐이더의 종류는 매우 다양합니다. Nebula2는 범용적인 게임을 지원하기 위해서 게임에 따라 달라지는 이러한 쉐이더 시스템의 특징을 엔진 내부가 아니라 외부에서 설정할 수 있도록 작성이 되어 있습니다. 이것이 바로 렌더링 경로(Render Path) 시스템입니다.

Shape 렌더링

이번에는 Nebula에서 오브젝트의 렌더링을 위해서는 어떠한 것들이 구성이 되어야 하는지에 대해서 살펴 보겠습니다.

Nebula에서는 하나의 메쉬는 `nShapeNode`(혹은 이 노드를 상속한 클래스) 클래스에 대응하는 것이 보통입니다. Nebula에서는 메쉬를 렌더링 하기 위해서는 기본적으로 메쉬 데이터와 텍스처 그리고 사용할 쉐이더가 반드시 지정이 되어야 합니다.

Nebula 스크립트에서는 다음과 같습니다.

```

new nshapenode shape0
  sel shape0
  ...
  .setmesh "meshes:mymesh.n3d2"
  .settexture "DiffMap0" "textures:tex01.dds"
  .setshader "static"
sel ..

```

주의

다른 엔진에서 보면 텍스처를 지정하지 않을 경우 엔진 내부에 설정된 색상이나 텍스처를 이용해서 렌더링 되도록 되어 있는 경우도 있지만 **Nebula**의 경우에는 텍스처를 설정하지 않는 경우 **shape** 노드를 생성할 때 에러가 발생하므로 주의해야 합니다. 틀에서 익스포트시 텍스처를 지정하지 않은 경우에는 **white.dds** 파일(**\$nebulax2/export/textures/system**)을 기본으로 사용하여 익스포트 되도록 설정되어 있습니다. 셰이더 역시 텍스처와 마찬가지로 반드시 설정이 되어야 하며, 익스포트시 설정되어 있지 않은 경우 **static** 셰이더가 설정되어 익스포트 됩니다.

Nebula의 메쉬와 관련한 다른 주의 사항으로는 하나의 메쉬 오브젝트가 두 개 이상의 텍스처를 가지는 경우입니다. 하나의 **nShapeNode**는 하나의 텍스처와 셰이더만 설정할 수 있습니다. 이렇게 하나의 오브젝트가 두 개 이상의 재질(**Material**)을 사용하는 경우에는 오브젝트가 하나더라도 재질의 개수만큼 **nShapeNode**를 가지게 됩니다. 다음은 하나의 오브젝트가 **tex01.dds**와 **tex02.dds**의 두 개의 텍스처 파일을 가지는 경우에 대해서 두 개의 **nShapeNode**를 생성한 **Nebula** 스크립트입니다.

```

new nshapenode shape0
  sel shape0
  ...
  .setmesh "meshes:mymesh.n3d2"
  # 첫 번째 그룹
  .setgroupindex 0
  .settexture "DiffMap0" "textures:tex01.dds"
sel ..
new nshapenode shape1
  sel shape1
  ...
  .setmesh "meshes:mymesh.n3d2"
  # 두 번째 그룹
  .setgroupindex 1
  .settexture "DiffMap0" "textures:tex02.dds"
sel ..

```

위에서처럼 두 개의 **nSapeNode**를 사용하는 경우라도 메쉬 데이터인 **mymesh.n3d2**는 하나의 메쉬 데이터 파일을 사용하고 있습니다. 이것은 메쉬 데이터와 관련한 정점 버퍼 및 색인 버퍼의 설정을 생각하면 쉽게 이해할 수 있습니다. 텍스처가 다른 경우에 렌더링할 때마다 텍스처 스테이지(**stage**)의 변경은 필요하지만 그 때마다 정점 버퍼와 색인 버퍼를 변경할 필요는 없습니다. 변경하는 경우 렌더링 상태의 변경이 필요하므로 그 만큼 렌더링 속도가 저하되게 됩니다. 그렇기 때문에 정점 버퍼 및 색인 버퍼는 하나만 생성한 다음에 설정을 해 두고 렌더링시 텍스처만 변경해서 렌더링하게 됩니다. 이 때 **tex01.dds** 텍스처를 렌더링할 때 이 텍스처를 사용하는 폴리곤만 렌더링해야 하는데 이것은 그룹 인덱스를 지정함으로써 설정할 수 있습니다. 메쉬의 그룹은 정점 버퍼에서 렌더링하는 **nShapdeNode**의 폴리곤의 색인에 대한 정보를 가지고 있습니다. 그래서 이 그룹을 이용해서 렌더링시 원하는 색인을 설정하여 렌더링하게 됩니다.

렌더링 경로(**Render Path**) 시스템

렌더링 경로 시스템을 이용하게 되면 만드는 게임에서 사용하는 셰이더의 종류와 렌더링시 이들 셰이더를 처리하는 순서를 엔진 코드는 전혀 수정하지 않고 **XML** 메타 파일을 사용해서 수정할 수 있으므로 매우 편리합니다. 여기서 사용하는 **XML** 파일을 렌더링 경로 파일이라고 하며 **\$nebula2/data/shader** 폴더 내에 위치하고 있습니다. 렌더링 경로 파일은 기본적으로 다음의 세 개가 존재합니다.

- dx7_renderpath.xml
- dx9_renderpath.xml
- dx9hdr_renderpath.xml

렌더링 경로 파일은 응용 프로그램을 시작할 때 스크립트 파일에서 설정할 수 있습니다. 다음은 **\$nebula2/data/scripts/startup.tcl** 파일에서 렌더링 경로를 설정하는 코드입니다. **DX9**을 지원하는 그래픽 카드의 경우 기본적으로 'dx9hdr_renderpath.xml' 렌더링 경로 파일을 사용하도록 설정되어 있습니다.

```
if {[exists /sys/servers/gfx]} {  
    # GFX 서버에서 시스템의 그래픽 카드에 대한 정보를 얻어 온다.  
    set featureSet [/sys/servers/gfx.getfeatureset]  
    if {($featureSet == "dx9") || ($featureSet == "dx9flt")} {  
        # DX9을 지원하는 그래픽 카드의 경우.  
        /sys/servers/scene.setrenderpathfilename "renderpath:dx9hdr_renderpath.xml"  
    } else {  
        # DX9을 지원하지 않는 그래픽 카드의 경우 고정파이프라인 사용.  
        /sys/servers/scene.setrenderpathfilename "renderpath:dx7_renderpath.xml"  
    }  
}
```

렌더링 경로 파일은 크게 **Shaders**, **RenderTarget**, **Variables**, **Section**의 네 부분으로 구성이 됩니다.

Shaders 부분은 게임에서 사용되는 모든 셰이더에 대한 선언이 이루어지는 곳입니다.

```
<Shaders>
<Shader name="passes" file="shaders:passes.fx" />
<Shader name="phases" file="shaders:phases.fx" />
<Shader name="compose" file="shaders:hdr.fx" />
<Shader name="static" file="shaders:shaders.fx" />
<Shader name="static_atest" file="shaders:shaders.fx" />
```

(생략...)

```
</Shaders>
```

셰이더는 .fx 이펙트 파일 이름 대신 *shader alias*라고 이야기하는 별칭(별명)을 사용하여 관리가 되는 데 렌더링 되는 오브젝트에 설정되는 셰이더의 별칭이 여기에서 정의가 됩니다. **"static"** 가장 기본적인 셰이더로 **diffuse map**과 **normal map**을 가지는 보통의 정적인 오브젝트를 렌더링할 때 사용되는 셰이더에 대한 별칭이며 이 셰이더는 'shaders:shaders.fx' 파일에 저장되어 있습니다. 이렇게 별칭을 사용하는 것은 별칭을 사용하는 것이 훨씬 더 직관적이며 하나의 파일에 여러 개의 코드 모듈들이 있는 경우에도 별칭을 사용하여 쉽게 구분할 수 있도록 하기 위함입니다. 위에서도 **static**과 **static_atest**(static 이면서 **alpha test**가 필요한 셰이더)는 같은 **shaders.fx** 파일에 저장되어 있지만 다른 별칭을 사용하여 구분하고 있습니다.

또한 렌더링 타겟(Render Target)도 이 렌더링 경로 파일에서 작성하게 됩니다.

```
<RenderTarget name="SkyScene" format="X8R8G8B8" relSize="1.0" />
<RenderTarget name="Scene" format="X8R8G8B8" relSize="1.0" />
<RenderTarget name="Shadow" format="A8R8G8B8" relSize="1.0" />
<RenderTarget name="SceneScaled" format="X8R8G8B8" relSize="0.5" />
<RenderTarget name="BrightPass" format="X8R8G8B8" relSize="0.5" />
```

```
<Float4 name="Luminance" value="0.299 0.587 0.114 0.0" />
<Float4 name="Balance" value="1.0 1.0 1.0 1.0" />
<Float name="Saturation" value="1.0" />
<Float name="HdrBrightPassThreshold" value="1.0" />
<Float name="HdrBrightPassOffset" value="1.0" />
<Float name="HdrBloomScale" value="1.0" />
```

렌더링시 복잡한 조명 계산식의 경우 여러 번의 패스에 걸쳐서 이전 패스에서의 계산 결과를 조금씩 바꿔 나가면서 렌더링하게 되는데 이것을 멀티 패스 텍스처 렌더링(Multipass Texture Rendering)이라고 합니다. 이것은 하드웨어 가속기가 소프트웨어 렌더링 시스템에 비해서 성능은 좋지만 유연성이 떨어지기 때문에 하드웨어 가속기는 단일 패스 렌더링에서 복잡한 조명 계산식을 계산하지 못합니다. 다르게 이야기하면 하드웨어가 단일 패스에서 한 표면에 적용할 수 있는 텍스처의 개수에는 한계가 있기 때문에 복잡한 조명식을 처리하기 위해서는 멀티 패스 렌더링이 꼭 필요합니다. 멀티 패스 렌더링은 각 패스에서 조명 처리식의 일부를 계산하고 중간 결과는 프레임 버퍼에 저장하는 방식으로 다음 패스에서는 이 프레임 버퍼에다 다시 해당 패스의 조명식을 계산하는 방식으로 진행이 됩니다. Nebula2의 렌더링 타겟이 바로 이 프레임 버퍼의 역할을 합니다.

멀티 패스에 사용되는 패스는 게임마다 사용하는 조명식이 다를 경우 게임 마다 다른 패스를 가지게 됩니다. 예를 들어 케이크는 다음과 같은 패스를 가지고 있습니다.

- 패스 1~4 : 범프맵 누적
- 패스 5 : 난반사 조명 효과
- 패스 6 : 정반사 성분을 포함한 기본 텍스처
- 패스 7 : 정반사 조명 처리
- 패스 8 : 방사 조명 처리
- 패스 9 : 블룸 효과
- 패스 10 : 화면 심광

이러한 패스는 게임에 따라 다른데 만약 패스가 엔진 내부에 하드 코딩되어 있어 새로운 게임을 제작할 때 마다 게임에 맞게 수정해야 한다면 범용성이 떨어지게 됩니다. 그러나 Nebula2에서는 이러한 패스를 렌더링 경로 파일에 지정할 수 있도록 하여 엔진 외부에서 손쉽게 수정할 수 있도록 되어 있습니다.

Nebula의 렌더링 패스는 Section, Pass, Phase 및 Sequence의 네 부분으로 구성이 되어 있습니다.

data/shaders/dx9hdr_renderpath.xml

```
<Section name="default" >
  <!-- depth pass -->
  <Pass name="depth" shader="passes" technique="tPassDepth"
renderTarget="Depth" clearColor="0.0 0.0 0.0 1.0" clearDepth="1.0" clearStencil="0">
    <Phase name="depth" shader="phases" technique="tPhaseDepth"
sort="FrontToBack" lightMode="Off">
      <Sequence shader="static"      technique="tStaticDepth"
shaderUpdates="Yes" mvpOnly="Yes" />
      <Sequence shader="environment" technique="tStaticDepth"
shaderUpdates="No" mvpOnly="Yes" />
    ...
  </Pass>
</Section>
```

```

<!-- render scene into hdr render target -->
<Pass name="SceneOpaque" shader="passes" technique="tPassEnvironment"
renderTarget="Scene">
  <Phase name="nolight" shader="phases" technique="tPhaseNoLight"
sort="FrontToBack" lightMode="Off">
    <Sequence shader="skybox"          technique="tSkyboxColorHDR"
firstLightAlpha="No".mvpOnly="Yes" />
    <Sequence shader="cubeskybox"      technique="tCubeSkyboxColorHDR"
firstLightAlpha="No".mvpOnly="Yes" />
  </Phase>
  <Phase name="opaque" shader="passes" technique="tPhaseOpaque"
sort="FrontToBack" lightMode="Shader">
    <Sequence shader="static" technique="tStaticColorHDRShadow"
firstLightAlpha="No".mvpOnly="Yes">
      <Texture name="AmbientMap1" value="Shadow" />
    </Sequence>
    <Sequence shader="environment"
technique="tEnvironmentColorHDRShadow" firstLightAlpha="No".mvpOnly="Yes"/>
  </Phase>
</Pass>
...

```

Pass는 몇 번의 장면 렌더링을 위해서 렌더링이 행해지는 횟수와 연관이 있습니다. **Phase**는 렌더링 타겟과 관련된 것이며 셰이더 별로 구분되는 것은 **Sequence**로 **Sequence** 차례대로 렌더링이 진행이 됩니다.

이렇게 외부에서 XML을 이용하여 렌더링 패스를 정의할 수 있으므로 새로운 렌더링 패스를 추가할 때에도 엔진의 코드 수정 없이 즉시 변경이 가능할 뿐만 아니라 하드웨어 사용에 따라 렌더링 패스를 조정하여 최적화된 상태로 구동할 수 있도록 다양한 옵션을 제공하는 작업도 매우 쉽게 처리할 수가 있습니다.

dx9hdr_renderpath.xml 파일에는 모두 다음의 세 개의 **Section**을 가지고 있습니다.

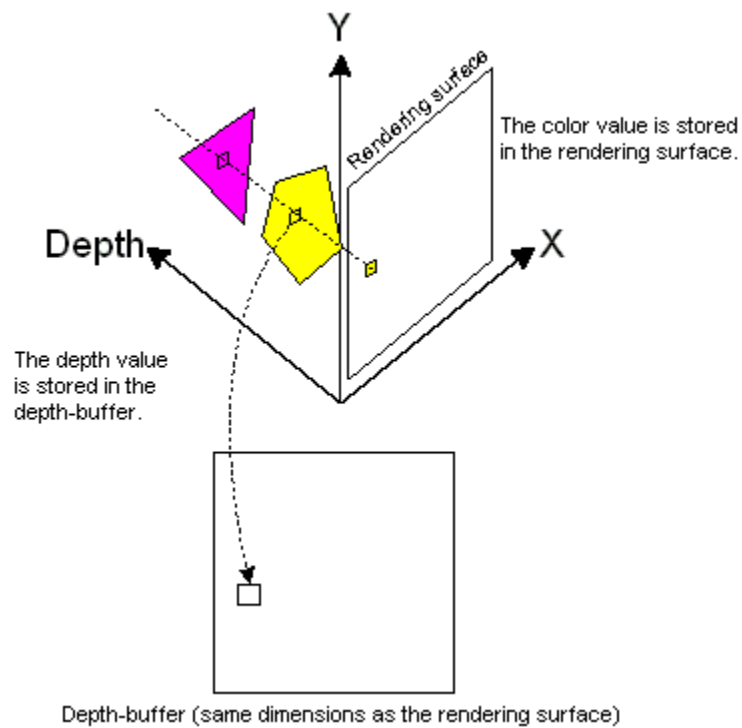
- 1) Default - 기본 렌더링 처리
- 2) Reflection - 반사 처리
- 3) Refraction - 굴절 처리

각각의 **Section** 마다 **Section** 고유의 **Pass**가 설정되며 **Default Section**에 설정된 렌더링 패스의 순서는 다음과 같습니다.

- 1) Depth
- 2) Occlusion Query
- 3) Shadow

- 4) Sky
- 5) Copy Sky
- 6) Opaque
- 7) Alpha
- 8) Dwnscale Scene
- 9) Bright pass filter
- 10) Bright pass bloom source
- 11) Bloom1
- 12) Bloom2
- 13) Bloom3
- 14) Compose
- 15) 3D GUI
- 16) Post draw

맨 처음에 깊이 버퍼에 렌더링을 하는데(Depth only rendering) 장면을 z축에 대해서 앞에서부터 뒤로 렌더링할 때 이 방법을 사용하면 렌더링 퍼포먼스를 향상 시킬 수 있습니다.





<깊이 버퍼에 렌더링한 그림>

- Lay out the contents of the depth buffer in an initial pass
- Ensures an overdraw of 1 for subsequent opaque pixels
- Helps with high-overdraw situations
- ...especially when longer shaders are involved
- Requires an extra geometry pass
- Can be optimized heavily though!
- Color writes disabled = faster depth-only throughput
- Use a reduced vertex structure (no need for binormal, tangent, color, texcoords)
- Use a reduced vertex shader (transform only – including skinning when needed)
- Alpha testing optimizations
- Use a reduced pixel shader for this: e.g. only fetch cut-out texture
- Disable alpha test after depth pass (change Z compare mode to EQUAL for those)

Reflection Section에 설정되어 있는 Pass는 다음과 같습니다.

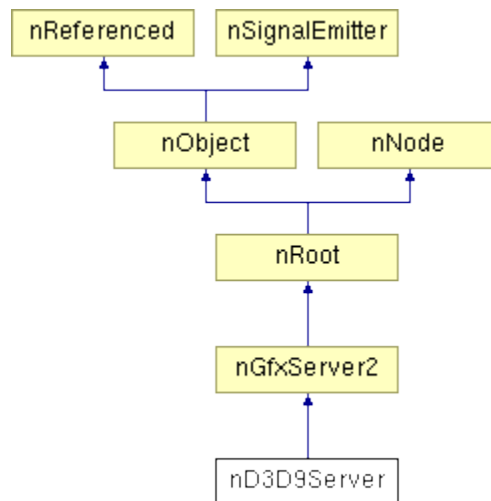
- 1) Depth
- 2) Opaque
- 3) Depth Compose

Refraction Section에 설정되어 있는 Pass는 다음과 같습니다.

- 1) Depth
- 2) Opaque
- 3) Depth compose

nGfxServer2 클래스

이 클래스는 그래픽스(Graphics) API의 Wrapper 클래스로 -Gfx-는 Graphics의 약어입니다. Nebula는 Direct3D를 기본 그래픽스 API로 사용하고 있습니다. 만약 특정 플랫폼으로 이식이 필요해서 OpenGL API를 사용해야 하는 경우에는 OpenGL API를 호출하는 nGfxServer2 클래스의 서브 클래스(sub class)를 만들어 주어야 합니다.



그래픽스 서버의 Nebula 객체 이름 공간의 위치는 '/sys/servers/gfx'입니다. win32 환경에서 이 클래스는 윈도우즈 응용프로그램의 생성(CreateWindow와 같은 win32 API의 호출)과 관련한 처리와 Direct3D 디바이스(Device)와 관련한 처리를 합니다. 윈도우즈와 관련된 처리를 하기 때문에 윈도우즈의 메시지 루프(WinProc)이나 Alt-Tab, Alt-F4에 대한 처리도 이 그래픽스 서버 클래스 및 이 클래스의 서브 클래스에서 처리합니다.

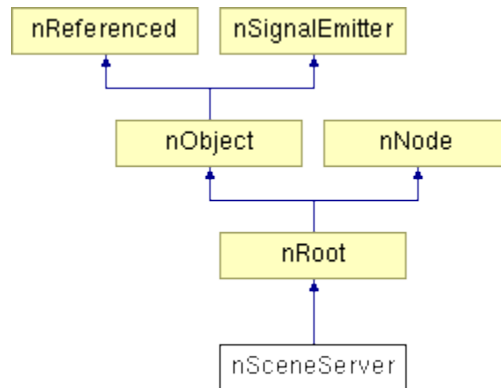
그리고, 개념적으로는 IDirect3DDevice와 유사하지만 IDirect3DDevice 인터페이스와 비교할 때 훨씬 간단한 인터페이스를 제공하는 것이 특징입니다. 때문에 디바이스의 Lost/Restore등의 상태도 이 그래픽스 서버에서 처리합니다. 그렇기 때문에 응용 프로그램에서는 이러한 상태 변경에 따른 처리에 신경 쓸 필요가 없으므로 프로그래밍이 훨씬 더 편리합니다.

또 이 그래픽스 서버에서는 메쉬(Mesh), 텍스처(Texture), 셰이더(Shader) 리소스 객체와 같은 그래픽 리소스의 생성도 함께 담당합니다.

장면서버 (nSceneServer)

장면 서버에서는 현재 장면에서 장면 노드(Scene Node)라고 이야기하는 렌더링 되는 노드들(혹은 비주얼 노드 - Visual Node)을 관리하고 이들을 렌더링하여 화면에 출력하는 역할을 하는 Nebula 엔진의

그래픽스 시스템 중 하나입니다.



이 장면 서버에서는 빠른 렌더링을 위해서 front-to-back 정렬이나 렌더링 상태(render state) 변화를 최소화하기 위한 작업들도 처리하게 됩니다.

front-to-back 정렬에 대한 박스 설명 삽입할 것.-Hyoun Woo Kim 4/21/08 10:19 PM

Nebula2 엔진이 셰이더 중심의 엔진이긴 하지만 일반적으로 렌더링의 모든 것을 셰이더의 변경으로만 처리할 수는 없습니다. 예를 들어 Nebula에서는 기본적으로 그림자 처리를 위해서 Stencil Shadow Volume을 지원하지만 사용자가 Shadow Map 방식을 사용하기를 원하는 경우에는 장면 서버를 상속하는 새로운 장면 서버 클래스를 생성하여 처리해야 합니다.

대부분의 다른 게임 엔진에서는 렌더러에서 렌더링 객체의 컬링(Culling)까지 함께 처리하는 것이 일반적이지만은 Nebula 엔진에서는 컬링은 응용 프로그램 쪽에서 처리하도록 하며 렌더러-장면 서버-에는 포함이 되어 있지 않습니다. 그러므로 장면 서버에 전달되는 객체들은 반드시 응용 프로그램에서 컬링되어 현재 프레임에서 렌더링 되는 객체들만 전달 되어야 합니다. Nebula 엔진의 렌더링 시스템은 이러한 특징을 가지고 있기 때문에 매 프레임마다 씬 그래프(Scene Graph)를 새로 구성하게 됩니다. 이렇게 렌더러에서 컬링을 하지 않는 이유는 게임에 따라 요구되는 적합한 컬링 시스템이 달라지기 때문입니다. 컬링에는 포탈(Portal), BSP, 차폐(Occlusion)과 같은 다양한 알고리즘이 있으며 이들은 게임의 특징에 따라 그 효율성이 각기 틀립니다. 다르게 이야기하면 어떤 게임에서는 BSP가 가장 효율적인 공간 분할 방식이지만 다른 게임에서는 포탈이 훨씬 더 게임의 목적에 적합한 방식이 될 수도 있다는 이야기입니다. 또한 게임이 인-도어(in-door) 게임이냐 아웃-도어(out-door) 게임이냐에 따라 필요한 컬링 시스템은 틀려지게 됩니다.

장면서버에서의 렌더링 루프는 다음과 같이 처리됩니다.

- 1) nSceneServer::BeginScene() 함수를 호출
- 2) Nebula 객체들 중에서 렌더링할 루트 오브젝트에 대해서 nSceneServer::Attach() 함수가 호출
- 3) nSceneServer::EndScene() 함수를 호출

- 4) nSceneServer::RenderScene() 함수를 호출
- 5) nSceneServer::PresentScen() 함수 호출

그러면 다음에서는 각 단계 별로 어떤 처리가 이루어지고 있는지에 대해서 살펴 보겠습니다.

1)단계

첫 번째 단계는 렌더링을 위한 준비 단계로 매 프레임마다 사용되는 변수들을 초기화하고 처음 렌더링 되는 노드의 경우 이 노드의 리소스를 읽어 들이는 처리를 합니다.

2)단계

렌더링시 주의해야 할 점은 렌더링할 노드는 매번 장면 서버로 전달해 주어야 한다는 것입니다. 이전에 장면 서버로 전달된 경우라도 새로 렌더링하기 위해서는 다시 장면 서버로 전달해 주어야 합니다.

3)단계

새로 렌더링 되는 노드들을 위해서 장면 서버의 그룹배열 초기화 등과 같이 초기화나 리셋 등의 처리를 합니다.

4)단계

네 번째 단계는 실제로 주어진 장면을 렌더링하는 단계입니다. 단, 실제로 화면에 장면을 출력하기 위해서는 nSceneServer::PresentScen() 함수를 호출해야 합니다. 그래서 장면에 다른 것들을 추가해서 그리고 싶은 경우에는 nSceneServer::RenderScene() 함수와 nSceneServer::PresentScen() 함수 사이에서 작업할 수도 있습니다.

```
void nSceneServer::RenderScene()
{
    nGfxServer2* gfxServer = nGfxServer2::Instance();

    // split nodes into shapes and lights
    this->SplitNodes();

    // NOTE: this must happen after make sure node resources are loaded
    // because the reflection/refraction camera stuff depends on it
    this->ValidateNodeResources();

    // compute light scissor rectangles and clip planes
    this->ComputeLightScissorsAndClipPlanes();

    // sort shape nodes for optimal rendering
    this->SortNodes();

    // render camera nodes in scene
    if (this->camerasEnabled)
```

```

{
    this->RenderCameraScene();
}

/// reset light passes in shape groups between renderpath
for (int i = 0; i < this->shapeBucket.Size(); i++)
{
    const nArray<ushort>& shapeArray = this->shapeBucket[i];
    for (int j = 0; j < shapeArray.Size(); j++)
    {
        this->groupArray[shapeArray[j]].lightPass = 0;
    }
}

// render final scene
PROFILER_START(this->profRenderPath);
int sectionIndex = this->renderPath.FindSectionIndex("default");
n_assert(-1 != sectionIndex);
this->DoRenderPath(this->renderPath.GetSection(sectionIndex));
PROFILER_STOP(this->profRenderPath);

// HACK...
this->gfxServerInBeginScene = gfxServer->BeginScene();
}

```

이 네 번째 단계는 다시 다음의 순서로 구분됩니다.

4-1) 리소스 유효성 점검

4-2) 같은 렌더링 패스(render pass)를 사용하는 비주얼 노드끼리 정렬

Nebula의 렌더링 패스는 셰이더 패스(Shader Pass), 광원 패스(Light Pass), 지오메트리 패스(Geometry Pass)로 구분되며 이 단계에서는 노드를 카테고리에 따라 따로 저장하고 다시 이들 노드의 셰이더에 따라 정렬합니다. 셰이더에 따라 정렬하는 것은 렌더링시 렌더링 상태의 변화를 최소화 시키기 위해서입니다. 이것은 빠르게 렌더링 하기 위한 배치(Batch) 작업과 관련된 작업입니다.

4-3) 렌더링 상태(Render Phase)에 따른 렌더링의 구분

장면서버에서는 광원에 따라 렌더링 상태(Render Phase)에 대한 처리는 시스템이 DX7 계열의 그래픽 카드를 사용하는 시스템인지 아니면 DX9 을 지원하는 그래픽 카드인지를 구분해서 처리하는데 다음의 세 가지로 구분되어 있습니다.(code/nebula2/src/scene/nsceneserver_main.cc 파일 참조)


```

{
    // don't render gui pass...
    continue;
}

if (curPass.GetDrawShadows() != nRpPass::NoShadows)
{
    PROFILER_START(this->profRenderShadow);
    this->GatherShadowLights();
    this->RenderShadow(curPass);
    PROFILER_STOP(this->profRenderShadow);
}
else if (curPass.GetOcclusionQuery())
{
    // perform light source occlusion query, this
    // marks the light sources in the scene as occluded or not
    this->DoOcclusionQuery();
}
else
{
    // default case: render phases and sequences
    int numPhases = curPass.Begin();
    int phaseIndex;
    for (phaseIndex = 0; phaseIndex < numPhases; phaseIndex++)
    {
        this->ffpLightingApplied = false;

        // for each sequence...
        nRpPhase& curPhase = curPass.GetPhase(phaseIndex);
        switch (curPhase.GetLightMode())
        {
            case nRpPhase::Off:
                this->RenderPhaseLightModeOff(curPhase);
                break;
            case nRpPhase::FFP:
                this->RenderPhaseLightModeFFP(curPhase);
                break;
            case nRpPhase::Shader:
                this->RenderPhaseLightModeShader(curPhase);
                break;
        }
    }
}
}

```

```

        curPass.End();
    }
}
rpSection.End();
}

```

깊이 버퍼에 렌더링하는 경우나, 스카이 박스를 그리는 경우 혹은 파티클(입자)의 렌더링에는 광원 처리가 필요하지 않습니다. 광원 처리에 대한 연산은 시간이 많이 걸리는 작업 중에 하나입니다. 그러므로 광원 처리가 필요 없는 작업의 경우 광원을 무시하고 렌더링하도록 해야 합니다. 이렇게 광원 처리의 유무에 따라 렌더링 함수를 구분해 놓은 이유는 광원 처리의 경우 광원의 개수 만큼 렌더링 처리를 하게 되므로 광원 처리가 필요 없는 경우는 따로 분리하여 처리하는 쪽이 훨씬 더 효율적이기 때문입니다.

장면 서버에서 광원 처리가 필요하지 않는 경우의 렌더링에는 `nSceneServer::RenderPhaseLightModeOff()` 함수를 호출하여 처리합니다.

nebula2/src/scene/nsceneserver_redner.cc

```

void nSceneServer::RenderPhaseLightModeOff(nRpPhase& curPhase)
{
    nGfxServer2* gfxServer = nGfxServer2::Instance();
    gfxServer->SetLightingType(nGfxServer2::Off);

    int numSeqs = curPhase.Begin();
    int seqIndex;
    for (seqIndex = 0; seqIndex < numSeqs; seqIndex++)
    {
        // check if there is anything to render for the next sequence shader at all
        nRpSequence& curSeq = curPhase.GetSequence(seqIndex);
        bool shaderUpdatesEnabled = curSeq.GetShaderUpdatesEnabled();
        int bucketIndex = curSeq.GetShaderBucketIndex();
        n_assert(bucketIndex >= 0);
        const nArray<ushort>& shapeArray = this->shapeBucket[bucketIndex];
        int numShapes = shapeArray.Size();
        if (numShapes > 0)
        {
            int seqNumPasses = curSeq.Begin();
            int seqPassIndex;
            for (seqPassIndex = 0; seqPassIndex < seqNumPasses; seqPassIndex++)
            {
                curSeq.BeginPass(seqPassIndex);

                // for each shape in bucket
                nMaterialNode* prevShapeNode = 0;
            }
        }
    }
}

```

```

int shapeIndex;
for (shapeIndex = 0; shapeIndex < numShapes; shapeIndex++)
{
    const Group& shapeGroup = this->groupArray[shapeArray[shapeIndex]];
    n_assert(shapeGroup.renderContext-
>GetFlag(nRenderContext::ShapeVisible));
    if (!shapeGroup.renderContext->GetFlag(nRenderContext::Occluded))
    {
        nMaterialNode* shapeNode = (nMaterialNode*)shapeGroup.sceneNode;
        if (shapeNode != prevShapeNode)
        {
            // start a new instance set
            shapeNode->ApplyShader(this);
            shapeNode->ApplyGeometry(this);
            WATCHER_ADD_INT(watchNumInstanceGroups, 1);
        }
        prevShapeNode = shapeNode;

        // set modelview matrix for the shape
        gfxServer->SetTransform(nGfxServer2::Model,
            shapeGroup.modelTransform);

        // set per-instance shader parameters
        if (shaderUpdatesEnabled)
        {
            shapeNode->RenderShader(this, shapeGroup.renderContext);
        }
        this->RenderShapeLightModeOff(shapeGroup);
    }
}
curSeq.EndPass();
}
curSeq.End();
}
}
curPhase.End();
}
}

```

렌더링시 셰이더, 광원, 지오메트리들은 다음과 같이 차례대로 갱신됩니다.

1. 셰이더(Shader) 의 갱신- 셰이더와 셰이더 파라미터의 설정에 대한 처리가 이루어 집니다. 렌더링 프레임마다 Diffuse 색상 값을 변경하는 등의 셰이더 파라미터 값의 애니메이션이 필요한 경우 이 단계에서 처리하게 됩니다.
2. 광원(Light) 의 갱신- 광원의 종류(point light, direction light 등)에 따라 광원의 방향등이 변경되는 경우 새롭게 설정한다거나 광원의 위치 변경에 대한 처리가 이루어 집니다. 광원 갱신에 대한 처리는 시스템이 DX9 카드를 지원하는 경우(셰이더를 사용하는 경우)에만 유효합니다.
3. 지오메트리(Geometry Pass) 의 갱신- nShaspeNode::RenderGeometyr()함수를 호출해서 메쉬 데이터를 갱신합니다.

그러면 nSceneServer::RenderPhaseLightModeFFP() 함수와 nSceneServer::RenderPhaseLightModeShader() 함수에서의 렌더링 처리에 대해서 좀 더 자세하게 살펴 보겠습니다.

RenderPhaseLightModeShader와 RenderPhaseLightModeFP 렌더링 루프에 대한 설명 - 코드 중심
으로 -Hyoun Woo Kim 6/23/08 6:40 PM

고정 파이프 라인에서의 렌더링

nebula2/src/scene/nsceneserver_redner.cc

```
void nSceneServer::RenderPhaseLightModeFFP(nRpPhase& curPhase)
{
    nGfxServer2* gfxServer = nGfxServer2::Instance();
    gfxServer->SetLightingType(nGfxServer2::FFP);
    int numSeqs = curPhase.Begin();
    int seqIndex;
    for (seqIndex = 0; seqIndex < numSeqs; seqIndex++)
    {
        // check if there is anything to render for the next sequence shader at all
        nRpSequence& curSeq = curPhase.GetSequence(seqIndex);
        bool shaderUpdatesEnabled = curSeq.GetShaderUpdatesEnabled();
        int bucketIndex = curSeq.GetShaderBucketIndex();
        n_assert(bucketIndex >= 0);
        const nArray<ushort>& shapeArray = this->shapeBucket[bucketIndex];
        int numShapes = shapeArray.Size();
        if (numShapes > 0)
        {
            int seqNumPasses = curSeq.Begin();
            int seqPassIndex;
            for (seqPassIndex = 0; seqPassIndex < seqNumPasses; seqPassIndex++)
            {
                curSeq.BeginPass(seqPassIndex);
            }
        }
    }
}
```

```

// for each shape in bucket
int shapeIndex;
nMaterialNode* prevShapeNode = 0;
for (shapeIndex = 0; shapeIndex < numShapes; shapeIndex++)
{
    const Group& shapeGroup = this->groupArray[shapeArray[shapeIndex]];
    n_assert(shapeGroup.renderContext-
>GetFlag(nRenderContext::ShapeVisible));
    if (!shapeGroup.renderContext->GetFlag(nRenderContext::Occluded))
    {
        nMaterialNode* shapeNode = (nMaterialNode*)shapeGroup.sceneNode;
        if (shapeNode != prevShapeNode)
        {
            // start a new instance set
            shapeNode->ApplyShader(this);
            shapeNode->ApplyGeometry(this);
            WATCHER_ADD_INT(watchNumInstanceGroups, 1);
        }
        prevShapeNode = shapeNode;
        // set modelview matrix for the shape
        gfxServer->SetTransform(nGfxServer2::Model,
            shapeGroup.modelTransform);
        // set per-instance shader parameters
        if (shaderUpdatesEnabled)
        {
            shapeNode->RenderShader(this, shapeGroup.renderContext);
        }
        this->RenderShapeLightModeFFP(shapeGroup);
    }
}
curSeq.EndPass();
}
curSeq.End();
}
}
curPhase.End();
}
}

```

```

void nSceneServer::RenderShapeLightModeFFP(const Group& shapeGroup)
{
    nGfxServer2* gfxServer = nGfxServer2::Instance();
}

```

```

vector4 dummyShadowLightMask;

// Render with vertex-lighting and multiple light sources
if (this->obeyLightLinks)
{
    // use light links, each shape render context is linked to
    // all light render context which illuminate this shape,
    // light links are provided by the application
    gfxServer->ClearLights();
    int numLights = shapeGroup.renderContext->GetNumLinks();
    int lightIndex;
    for (lightIndex = 0; lightIndex < n_min(numLights, nGfxServer2::MaxLights);
lightIndex++)
    {
        nRenderContext* lightRenderContext = shapeGroup.renderContext-
>GetLinkAt(lightIndex);
        const Group& lightGroup = this->groupArray[lightRenderContext-
>GetSceneGroupIndex()];
        n_assert(lightRenderContext == lightGroup.renderContext);
        n_assert(lightGroup.sceneNode->HasLight());
        lightGroup.sceneNode->RenderLight(this, lightGroup.renderContext,
lightGroup.modelTransform);
        lightGroup.sceneNode->ApplyLight(this, lightGroup.renderContext,
lightGroup.modelTransform, dummyShadowLightMask);
    }
}
else if (!this->ffpLightingApplied)
{
    // ignore light links, each shape is influenced by each light
    // Optimization: if lighting has been applied for this
    // frame already, we don't need to do it again. This will only
    // work if rendering doesn't go through light links though
    gfxServer->ClearLights();
    int numLights = this->lightArray.Size();
    int lightIndex;
    for (lightIndex = 0; lightIndex < n_min(numLights, nGfxServer2::MaxLights);
lightIndex++)
    {
        const Group& lightGroup = this->groupArray[this-
>lightArray[lightIndex].groupIndex];
        n_assert(lightGroup.sceneNode->HasLight());
        lightGroup.sceneNode->RenderLight(this, lightGroup.renderContext,

```

```

lightGroup.modelTransform);
    lightGroup.sceneNode->ApplyLight(this, lightGroup.renderContext,
lightGroup.modelTransform, dummyShadowLightMask);
    }
    this->ffpLightingApplied = true;
}
shapeGroup.sceneNode->RenderGeometry(this, shapeGroup.renderContext);
WATCHER_ADD_INT(watchNumInstances, 1);
}

```

쉐이더를 사용한 렌더링

nebula2/src/scene/nsceneserver_redner.cc

```

void nSceneServer::RenderPhaseLightModeShader(nRpPhase& curPhase)
{
    nGfxServer2* gfxServer = nGfxServer2::Instance();
    gfxServer->SetLightingType(nGfxServer2::Shader);
    this->ffpLightingApplied = false;
    // for each light...
    int numLights = this->lightArray.Size();
    int lightIndex;
    for (lightIndex = 0; lightIndex < numLights; lightIndex++)
    {
        gfxServer->ClearLights();
        const LightInfo& lightInfo = this->lightArray[lightIndex];
        const Group& lightGroup = this->groupArray[lightInfo.groupIndex];
        nRenderContext* lightRenderContext = lightGroup.renderContext;
        n_assert(lightGroup.sceneNode->HasLight());
        // do nothing if light is occluded
        if (!lightRenderContext->GetFlag(nRenderContext::Occluded))
        {
            // apply light state
            lightGroup.sceneNode->ApplyLight(this, lightGroup.renderContext,
lightGroup.modelTransform, lightInfo.shadowLightMask);
            // now iterate through sequences...
            int numSeqs = curPhase.Begin();
            // NOTE: nRpPhase::Begin resets the scissor rect, thus this must happen
            afterwards!
            this->ApplyLightScissors(lightInfo);
            this->ApplyLightClipPlanes(lightInfo);

```



```

        shapeNode->ApplyShader(this);
        shapeNode->ApplyGeometry(this);
        WATCHER_ADD_INT(watchNumInstanceGroups, 1);
    }
    prevShapeNode = shapeNode;
    // set modelview matrix for the shape
    gfxServer->SetTransform(nGfxServer2::Model,
        shapeGroup.modelTransform);
    // set per-instance shader parameters
    if (shaderUpdatesEnabled)
    {
        shapeNode->RenderShader(this, shapeGroup.renderContext);
    }
    lightGroup.sceneNode->RenderLight(this,
lightGroup.renderContext,
        lightGroup.modelTransform);
    this->RenderShapeLightModeShader(shapeGroup, curSeq);
    }
    }
    }
    curSeq.EndPass();
    }
    &nbsp; curSeq.End();
    }
    }
    curPhase.End();
    }
    }
    this->ResetLightScissorsAndClipPlanes();
}

```

5)단계

렌더링의 마지막 단계로 장면이 사용자에게 출력되는 단계입니다. (Direct3D 플랫폼인 경우에는 IDirect3DDevice9::Present() API가 호출됩니다)

```

void nSceneServer::PresentScene()
{
    nGfxServer2* gfxServer = nGfxServer2::Instance();
    if (this->gfxServerInBeginScene)
    {
        if (this->renderDebug)
        {

```

```
    this->DebugRenderLightScissors();
    this->DebugRenderShapes();
}
if (this->perfGuiEnabled)
{
    this->DebugRenderPerfGui();
}
gfxServer->DrawTextBuffer();
gfxServer->EndScene();
gfxServer->PresentScene();
}
gfxServer->EndFrame();
PROFILER_STOP(this->profFrame);
}
```

Scene 노드(Scene Node)와 렌더링

Nebula 엔진에서는 장면(Scene)에서 렌더링할 오브젝트는 nSceneNode 클래스를 상속 받은 클래스에서 처리하게 됩니다. 이는 Nebula의 장면 그래프(Scene Graph) 처리와 관련이 있는데 Nebula 엔진에서 장면 그래프처리와 관계한 클래스의 이름은 이름에 '-Node'가 붙습니다.


```

{
    n_assert(sceneServer);
    n_assert(renderContext);

    this->InvokeAnimators(nAnimator::Transform, renderContext);
    if (this->GetLockViewer())
    {
        // handle lock to viewer
        const matrix44& viewMatrix = nGfxServer2::Instance()-
>GetTransform(nGfxServer2::InvView);
        matrix44 m = this->tform.getmatrix();
        m = m * parentMatrix;
        m.M41 = viewMatrix.M41;
        m.M42 = viewMatrix.M42;
        m.M43 = viewMatrix.M43;
        sceneServer->SetModelTransform(m);
    }
    else
    {
        // default case
        sceneServer->SetModelTransform(this->tform.getmatrix() * parentMatrix);
    }
    return true;
}

```

nAbstractShaderNode 클래스

nAbstractShaderNode 클래스에서는 노드의 렌더링에 필요한 텍스처를 읽어 들이고 설정하는 처리를 합니다.

nebula2/src/scene/nabstractshadernode_main.cc

```

bool nAbstractShaderNode::LoadTexture(int index)
{
    TexNode& texNode = this->texNodeArray[index];
    if ((!texNode.refTexture.isValid()) && (!texNode.texName.IsEmpty()))
    {
        // load only if the texture is used in the shader
        if (this->IsTextureUsed(texNode.shaderParameter))
        {
            nTexture2* tex = nGfxServer2::Instance()->NewTexture(texNode.texName);
            n_assert(tex);
        }
    }
}

```

```

    if (!tex->IsLoaded())
    {
        tex->SetFilename(texNode.texName);
        if (!tex->Load())
        {
            n_printf("nAbstractShaderNode: Error loading texture '%s'\n",
texNode.texName.Get());
            return false;
        }
    }
    texNode.refTexture = tex;
    this->shaderParams.SetArg(texNode.shaderParameter, nShaderArg(tex));
}
}
return true;
}

```

nMaterialNode 클래스

nebula2/src/scene/nmaterialnode_main.cc

```

bool nMaterialNode::LoadShader()
{
    n_assert(!this->shaderName.IsEmpty());

    if (!this->refShader.isvalid())
    {
        const nRenderPath2* renderPath = nSceneServer::Instance()->GetRenderPath();
        n_assert(renderPath);
        int shaderIndex = renderPath->FindShaderIndex(this->shaderName);
        n_assert2(-1 != shaderIndex, nString("Shader \"" + this->shaderName + "\" not
found in \"" + renderPath->GetFilename() + "\"").Get());
        const nRpShader& rpShader = renderPath->GetShader(shaderIndex);
        this->shaderIndex = rpShader.GetBucketIndex();
        this->refShader = rpShader.GetShader();
        this->refShader->AddRef();
    }
    return true;
}

```

nebula2/src/scene/nmaterialnode_main.cc

```
bool nMaterialNode::RenderShader(nSceneServer* sceneServer, nRenderContext*
renderContext)
{
    nShader2* shader = this->refShader;
    nGfxServer2* gfxServer = nGfxServer2::Instance();

    n_assert(sceneServer);
    n_assert(renderContext);

    // invoke shader manipulators
    if (this->GetNumAnimators() > 0)
    {
        this->InvokeAnimators(nAnimator::Shader, renderContext);
        shader->SetParams(this->shaderParams);
    }

    // set shader override parameters from render context (set directly by application)
    shader->SetParams(renderContext->GetShaderOverrides());

    return true;
}
```

nSceneNode는 장면 그래프의 노드들의 최상위 클래스이고 nTransformNode, nAbstractShaderNode, nMaterialNode 클래스는 모든 노드의 렌더링에 필요한 기반 클래스들입니다. 바꾸어서 이야기하면 이들 클래스에서는 렌더링에 필요한 기능등을 제공하지만 실제 이들 클래스의 인스턴스를 직접 렌더링에 사용하지는 않습니다.

nShapeNode 클래스

nShapeNode 클래스는 정적인 오브젝트의 렌더링을 처리하는 클래스입니다.

nebula2/inc/scene/nshapenode.h

```
inline
bool
nShapeNode::HasGeometry() const
{
    return true;
}
```

nebula2/src/scene/nshapenode_main.cc

```
bool nShapeNode::LoadMesh()
{
    if ((!this->refMesh.isvalid()) && (!this->meshName.IsEmpty()))
    {
        // append mesh usage to mesh resource name
        nString resourceName;
        resourceName.Format("%s_%d", this->meshName.Get(), this->GetMeshUsage());

        // get a new or shared mesh
        nMesh2* mesh = nGfxServer2::Instance()->NewMesh(resourceName);
        n_assert(mesh);
        if (!mesh->IsLoaded())
        {
            mesh->SetFilename(this->meshName);
            mesh->SetUsage(this->GetMeshUsage());

            if (this->refMeshResourceLoader.isvalid())
            {
                mesh->SetResourceLoader(this->refMeshResourceLoader.getname());
            }

            if (!mesh->Load())
            {
                n_printf("nMeshNode: Error loading mesh '%s'\n", this->meshName.Get());
                mesh->Release();
                return false;
            }
        }
        this->refMesh = mesh;
        this->SetLocalBox(this->refMesh->Group(this->groupIndex).GetBoundingBox());
    }
    return true;
}
```

nebula2/src/scene/nshapenode_main.cc

```
bool
nShapeNode::ApplyGeometry(nSceneServer* /*sceneServer*/)
{

```

```

nGfxServer2* gfxServer = nGfxServer2::Instance();
n_assert(this->refMesh->IsValid());

// set mesh, vertex and index range
gfxServer->SetMesh(this->refMesh, this->refMesh);
const nMeshGroup& curGroup = this->refMesh->Group(this->groupIndex);
gfxServer->SetVertexRange(curGroup.GetFirstVertex(), curGroup.GetNumVertices());
gfxServer->SetIndexRange(curGroup.GetFirstIndex(), curGroup.GetNumIndices());
return true;
}

```

nebula2/src/scene/nshapenode_main.cc

```

bool nShapeNode::RenderGeometry(nSceneServer* /*sceneServer*/, nRenderContext*
/*renderContext*/)
{
    nGfxServer2::Instance()->DrawIndexedNS(nGfxServer2::TriangleList);
    return true;
}

```

nSkinShapeNode 클래스

스킨 애니메이션 모델을 렌더링하기 위한 클래스입니다. **nSkinShapeNode** 클래스는 스킨 애니메이션 모델의 렌더링에 대한 처리만 하며 모델의 애니메이션은 **nSkinAnimatior** 클래스에서 처리합니다. 그래서 **nSkinShapeNode** 클래스는 모델을 렌더링 하기 전에 **nSkinShapeNode::SetSkinAnimator()** 함수를 사용하여 모델의 애니메이션 처리를 위한 **nSkinAnimator** 클래스 인스턴스를 지정합니다.

nSkinShapeNode 클래스는 캐릭터 모델 처리에 주로 사용되는 클래스입니다. **Nebula** 엔진의 캐릭터 애니메이션 처리에 대해서 살펴 볼 때 더욱 자세하게 설명하도록 하겠습니다.

nShadowShapeNode/nShadowSkinShapeNode 클래스

nShadowShapeNode 클래스와 **nShadowSkinShapeNode** 클래스는 그림자 처리를 위한 클래스로 **nShadowShapeNode** 클래스는 정적인(static) 물체의 그림자를 **nShadowSkinShapeNode** 클래스는 캐릭터와 같은 스킨 애니메이션 오브젝트의 그림자 처리를 위한 클래스입니다.

이들 그림자 클래스들은 다른 노드 클래스들과는 달리 **nTransformNode** 클래스를 상속받습니다. 이것은 그림자 처리에 텍스처나 셰이더 처리를 필요로 하지 않기 때문입니다.

Nebula 엔진에서는 그림자 처리에 셰도우 볼륨(Shadow Volume)을 사용하는데 셰도우 볼륨에 대한 연산은 CPU에서 처리하기 때문에 텍스처나 셰이더와 같은 머티리얼 처리에 필요한 nAbstractShaderNode 클래스나 nMaterialNode 클래스의 상속을 필요로 하지 않습니다.

nBlendShapeNode 클래스

렌더링 컨텍스트(Render Context)

일반적으로 nSceneNode 클래스와 이 nSceneNode를 상속한 클래스를 비주얼 노드(Visual Node)라고 하며 이 클래스들은 렌더링이 중요한 목적입니다. 이 말은 게임 엔티티들의 고유의 속성은 비주얼 노드에서 따로 분리해서 처리해야 한다는 것을 의미합니다. Nebula에서는 렌더링과 게임 객체의 관리를 분리해서 처리하고 있으며, 비주얼 노드는 렌더링시 필요한 메쉬와 텍스처, 셰이더에 대한 포인터만을 가지고 있으며 이것은 매 프레임마다 필요에 따라 변경할 수 있습니다.

게임에서 쉽게 볼 수 있는 예로 한 화면에 여러 몬스터가 등장한 경우를 가정해 보겠습니다. 이들 몬스터들은 모두 모두 같은 동일한 리소스(메쉬 데이터, 텍스처 파일)를 사용하므로 렌더링 되어 화면에 출력되는 모습은 같습니다. 다만 화면에 렌더링 될 때 각 몬스터들은 다른 위치에 렌더링 되므로 이들의 위치(position)과 방향(orientation)은 모두 틀린 값을 가지게 되며 또한 애니메이션 상태도 경우에 따라서는 틀립니다. 가령 몬스터 A가 플레이어와 싸우고 있다면 몬스터 B는 플레이어를 향해서 뛰어 오고 있고 몬스터 C는 좀 더 떨어진 위치에서 배회하고 있는 경우를 생각해 볼 수 있습니다. 이 경우에 몬스터들을 렌더링하기 위해서 위치와 방향, 현재 플레이 되고 있는 애니메이션의 상태가 틀리기 때문에 다른 세 개의 리소스를 생성해야 한다면 메모리 낭비가 심할 것입니다.

Nebula에서는 이러한 처리를 위해서 렌더링 컨텍스트라는 것을 제공하고 있습니다. 렌더링 컨텍스트는 게임 엔티티를 렌더링할 때 게임 엔티티의 현재 상태를 비주얼 노드로 전달하는 기능을 합니다. 위의 예에서는 A,B,C의 세 몬스터가 있으므로 게임 엔티티는 세 가지 엔티티가 생성이 됩니다. 그리고 엔티티마다 렌더링을 위한 렌더링 컨텍스트를 생성합니다. 그러나 게임 리소스는 모든 몬스터가 동일한 메쉬와 텍스처를 사용하므로 한가지 리소스만 필요로 합니다. 게임 플레이시 게임의 상태에 따라 각각의 게임 엔티티들은 그 상태가 변하게 될 것이며 이 상태는 게임 엔티티들의 비주얼 노드를 렌더링할 때 렌더링 컨텍스트를 통해서 장면서버(Scene Server)에 전달이 됩니다. 바꾸어서 이야기하면 비주얼 노드는 상태를 가지지 않는데 렌더링시 이 상태에 대한 정보 - 게임 엔티티의 위치, 방향, 색상 등등 - 를 전달하는 역할을 하는 것이 바로 렌더링 컨텍스트입니다.

바로 게임 엔티티의 상태를 비주얼 노드로 전달하지 않고 렌더링 컨텍스트를 통해서 전달하는 것은 비주얼 노드로 전달해야 하는 상태가 게임 엔티티의 상태 뿐만 아니라 프로그램의 전역에 선언된 데이터를 전달해야 하는 경우도 있기 때문입니다. 이런 이유로 렌더링 컨텍스트라는 것을 두어 클래스의 처리에 대한 역할을 구분지어 보다 나은 설계를 하기 위해서입니다.

컨텍스트(Context)란 말의 의미에는 문맥, 정황, 흐름, 전후관계의 뜻을 담고 있습니다. Nebula2 엔진의 렌더링 컨텍스트도 게임 엔티티에 따라 같은 비주얼 노드라도 렌더링이 틀려지게 됩니다. 이러한 관점에서 보면 왜 이름이 렌더링 컨텍스트인지 쉽게 이해할 수가 있습니다.

렌더링 컨텍스트의 사용

기본적으로 화면에 렌더링 되는 모든 게임 엔티티는 고유의 렌더링 컨텍스트를 가집니다. 이 렌더링 컨텍스트는 `nSceneNode::RenderContextCreated()` 멤버 함수로 생성할 수 있습니다.

비주얼 노드의 렌더링 컨텍스트를 렌더링되는 최상위 비주얼 노드의 렌더링 컨텍스트만 생성하면 됩니다. 예를 들어 다음과 같이 `monsterA`는 `partA`와 `partB`의 두 노드로 구성이 되어 있는 경우 렌더링할 `nSceneServer`에 넘겨 주는 노드는 `monsterA`가 됩니다. 그러면 `partA`와 `partB`는 `monsterA` 노드의 자식 노드이므로 자동으로 렌더링 됩니다.

```
lib/characters/monsterA/partA
                             /partB
```

이렇게 렌더링 되는 노드가 여러 개의 자식 노드로 분리가 되어 있는 경우에는 렌더링 컨텍스트는 `monsterA`의 렌더링 컨텍스트만을 생성하면 됩니다. `partA`와 `partB`의 렌더링 컨텍스트를 모두 생성할 필요는 없다는 이야기입니다. 개별 게임 엔티티마다 하나의 렌더링 컨텍스트를 생성하다고 생각하면 이해가 쉬울 것입니다.

다음은 생성한 렌더링 컨텍스트를 초기화하고 업데이트는 방법에 대한 코드입니다.

```
// 렌더링 컨텍스트 초기화
nFloat4 wind = { 1.0f, 0.0f, 0.0f, 0.5f };
nVariable::Handle timeHandle = varServer->GetVariableHandleByName("time");
nVariable::Handle oneHandle = varServer->GetVariableHandleByName("one");
nVariable::Handle windHandle = varServer->GetVariableHandleByName("wind");
renderContext.AddVariable(nVariable(timeHandle, 0.5f));
renderContext.AddVariable(nVariable(oneHandle, 1.0f));
renderContext.AddVariable(nVariable(windHandle, wind));

// Specify the root node to render context.
// Note that render context can be only specified on the root node of
// scene nodes
renderContext.SetRootNode(rootNode.get());
rootNode->RenderContextCreated(&renderContext);
```

생성한 렌더링 컨텍스트는 비주얼 노드를 렌더링할 때 `nSceneServer::Attach()` 함수의 인자로 넘겨 주게 됩니다. 장면서버(Scene Server)로 전달된 렌더링 컨텍스트는 장면서버에서 비주얼 노드를 렌더링할 때 갱신됩니다.

```
// 렌더링 컨텍스트의 갱신.
renderContext.GetVariable(timeHandle)->SetFloat((float)time);
renderContext.SetFrameId(frameId++);
...
// 렌더링 컨텍스트를 장면서버로 전달.
if (sceneServer->BeginScene(viewMatrix))
{
    sceneServer->Attach(&renderContext);
    sceneServer->EndScene();
    ...
}
```

위의 코드에서 알 수 있듯이 비주얼 노드가 장면 서버로 바로 전달되어서 렌더링 되는 것이 아니라 렌더링 컨텍스트가 전달됩니다. 그런 다음 `nSceneServer::Attach()` 함수 내부에서 비주얼 노드의 `nSceneNode::Attach()` 함수를 호출하여 모든 자식 노드들을 포함해서 렌더링할 모든 비주얼 노드들을 장면서버의 그룹 배열(Group Array)에 추가합니다. 그룹 배열은 비주얼 노드들의 렌더링을 위한 내부 저장소로 비주얼 노드가 속하는 카테고리 별로 구분되어 다시 나누어지게 됩니다. 이들 비주얼 노드는 다시 렌더링 전에 비주얼 노드의 성격에 따라 분류가 되는데 비주얼 노드가 지오메트리라인 경우에는 지오메트리의 그룹 배열에, 광원인 경우에는 광원의 그룹배열에 추가 되게 됩니다.

렌더링 컨텍스트는 프레임 식별자(Frame ID), 변환 행렬(Transform Matrix)와 같은 게임 프로그램에서 사용하는 다양한 데이터를 장면서버로 전달하기 위해서 사용됩니다. 렌더링 컨텍스트를 사용하면 하나의 비주얼 노드를 여러 번 렌더링해야 할 때 편리합니다. 또한 렌더링 컨텍스트는 셰이더 변수들을 처리하는데에도 편리하게 사용됩니다.

렌더링 컨텍스트로 셰이더 변수들을 처리하는 방법을 살펴 보기 위해서는 `nSwingShapeNode::RenderShader()` 멤버 함수를 살펴 보기 바랍니다.

code/nebula2/src/nature/nswingshapenode_main.cc

```
bool nSwingShapeNode::RenderShader(nSceneServer* sceneServer, nRenderContext*
renderContext)
{
    if (nMaterialNode::RenderShader(sceneServer, renderContext))
    {
        // 렌더링 컨텍스트를 사용하여 바람의 방향과 세기에 대한 값을 얻는다.
        nVariable* timeVar = renderContext->GetVariable(this->timeVarHandle);
        nVariable* windVar = renderContext->GetVariable(this->windVarHandle);
        &nbsp;    n_assert(timeVar && windVar);
    }
}
```

```

nTime time = (nTime)timeVar->GetFloat();
const nFloat4& wind = windVar->GetFloat4();

// 바람 방향 벡터를 구성한다.
vector3 windVec(wind.x, wind.y, wind.z);

...
}

```

비주얼 노드에서는 `nVariableHandle` 인스턴스 값을 사용해서 렌더링 컨텍스트로부터 특정 변수 데이터 (`nVariable`)를 얻어 올 수 있으며 이 변수 데이터에서 다시 적절한 형으로 변환하여 필요한 값을 얻어 오게 됩니다.

위의 코드에서 `nSwingShapeNode` 클래스는 시간과 바람에 대한 핸들을 가지며 렌더링 시 이들 핸들을 통해서 응용 프로그램에 설정된 시간과 바람에 대한 데이터 값인 `timeVar`와 `winVar` 값을 얻어 오게 됩니다. 그런 다음 이들 데이터 값의 `GetFloat()`, `GetFloat4()` 멤버 함수를 호출해서 필요한 실수값을 얻어 와서 해당 `nSwingShapeNode` 인스턴스의 렌더링에 사용하게 됩니다.

핸들은 프로그램에서 전역적인 접근이 필요한 값의 경우에 사용하는 것으로 응용 프로그램 생성시 생성하는 것이 보통입니다. 하지만 해당 핸들을 얻어 올 때 그 핸들이 존재하지 않으면 새로운 핸들을 생성합니다. 다음은 `nVariableServer::GetVariableHandleByName()` 함수입니다.

code/nebula2/src/variable/nvariables_server_main.cc

```

nVariable::Handle nVariableServer::GetVariableHandleByName(const char* varName)
{
    n_assert(varName);

    // 주어진 이름의 핸들이 이미 존재하는지 먼저 찾는다.
    nVariable::Handle varHandle = this->FindVariableHandleByName(varName);
    if (nVariable::InvalidHandle != varHandle)
    {
        return varHandle;
    }
    else
    {
        // 새로운 변수 데이터를 생성후 등록
        VariableDeclaration newDecl(varName);
        this->registry.Append(newDecl);
        return this->registry.Size() - 1;
    }
}

```

그리고 이 핸들에 대한 접근이 필요한 클래스에서는 보통 클래스 헤더 파일에 다음과 같이 선언합니다.

```
code/nebula2/inc/nature/nswingshapenode.h
```

```
class nSwingShapeNode : public nShapeNode
{
    ...
private:
    // 시간 값에 대한 핸들.
    nVariable::Handle timeVarHandle;
    // 바람 세기에 대한 핸들.
    nVariable::Handle windVarHandle;
    ...
};
```

핸들은 일반적으로 리소스를 로딩할 때 얻어 옵니다.

```
code/nebula2/src/nature/nswingshapenode_main.cc
```

```
bool nSwingShapeNode::LoadResources()
{
    if (nShapeNode::LoadResources())
    {
        // 시간에 대한 핸들을 얻어 온다.
        this->timeVarHandle = nVariableServer::Instance()-
>GetVariableHandleByName("time");
        // 바람에 대한 핸들을 얻어 온다.
        this->windVarHandle = nVariableServer::Instance()-
>GetVariableHandleByName("wind");
        return true;
    }
    return false;
}
```

nSwingShapeNode는 나무와 같은 오브젝트 처리에 사용되는 클래스입니다. 이러한 오브젝트들은 게임 내에서 설정된 바람의 방향이나 세기가 변할 때 마다 나무 오브젝트 역시 흔들리는 방향이나 흔들리는 정도가 달라지도록 처리해야 하는데 바람의 경우 게임 내에서 다른 오브젝트들도 참조할 수 있는 변수입니다. 그래서 이러한 변수들은 프로그램에서 전역적으로 선언한 다음 필요한 클래스에 참조할 수 있는 방법을 제공하는 메카니즘이 바로 핸들입니다. 이렇게 핸들을 사용하는 이유는 전역 변수를 사용하지 않고 변수의 이름(문자열 값)으로 간편하게 찾을 수 있도록 하기 위함입니다.

차폐 질의(Occlusion Query)

Nebula2에서는 하드웨어를 이용한 차폐 질의(Occlusion Query) 기능을 제공하는데 이것을 이용하면 화면에 실제로 그려지는 텍셀(texel)의 개수를 알 수 있습니다.

Nebula2의 차폐 질의는 렌더링할 객체의 바운딩 박스를 먼저 Z 버퍼에 렌더링해서 Z test를 통해 실제로 그려지는 픽셀의 개수를 검사해서 차폐 여부를 결정하는 방법입니다. 만약 픽셀의 개수가 0인 경우에는 이 객체는 차폐된 객체이므로 렌더링할 필요가 없습니다. 이 방법을 사용하면 GPU/CPU의 병렬 처리를 최대화 할 수 있습니다.

Nebula2에서는 제공하는 차폐 질의는 다음의 과정으로 처리하게 됩니다.

- 1) 객체의 바운딩 볼륨을 렌더링한다.
- 2) 모든 오브젝트들에 대해서
 - 2-1) 질의 시작(Begin Query)
 - 2-2) 바운딩 볼륨을 다시 렌더링한다.
 - 2-3) 질의 종료(End Query)
 - 2-4) 차폐 질의 데이터에 대한 정보를 얻는다. 픽셀들의 가시성 여부가 0보다 큰 경우 이 객체는 차폐되지 않은 객체로 렌더링하게 된다.

Nebula2에서 차폐 질의에 대한 처리를 담당하는 클래스는 nOcclusionQuery 클래스입니다. 이 클래스는 추상 클래스로 실제 차폐 질의에 대한 구현은 이 클래스를 상속한 클래스에서 구현하게 됩니다.

Nebula2에서는 nOcclusionQuery 클래스를 상속하는 nD3D9OcclusionQuery 클래스에 D3D 하드웨어의 차폐 질의 기능을 이용하여 구현되어 있습니다.

D3D에서 하드웨어 질의 기능을 위한 API는 IDirect3DQuery9 으로 차폐 질의는 D3DQUERYTYPE_OCCLUSION 질의 타입을 사용합니다. 이 질의 타입은 Z-test를 통과한 픽셀의 개수를 알려주는데 이를 이용하면 차폐 여부를 판단할 수가 있습니다.

차폐 처리는 nSceneServer::DoRenderPath() 함수에서 렌더링시 렌더링 패스(Pass)가 차폐질의를 위한 패스인 경우 nSceneServer::DoOcclusionQuery() 함수를 호출해서 처리하게 됩니다.

차폐 질의에 대한 렌더링 패스는 'data/shaders/d3d9hdr_renderpath.xml' 파일에 정의되어 있습니다.

```
<!-- perform occlusion culling -->  
<Pass name="occlusionQuery" occlusionQuery="Yes" />
```

렌더링 패스가 '차폐질의'인 경우 차폐질의를 위한 nSceneServer::DoOcclusionQuery() 함수가 호출됩니다.

```
nebula2/src/scene/nsceneserver_render.cc
```

```
void nSceneServer::DoRenderPath(nRpSection& rpSection)  
{
```

```

nGfxServer2* gfxServer = nGfxServer2::Instance();
int numPasses = rpSection.Begin();
int passIndex;
for (passIndex = 0; passIndex < numPasses; passIndex++)
{
    // for each phase...
    nRpPass& curPass = rpSection.GetPass(passIndex);

    // check if this is the GUI pass, and gui is disabled...
    if (curPass.GetDrawGui() && !this->GetGuiEnabled())
    {
        // don't render gui pass...
        continue;
    }

    if (curPass.GetDrawShadows() != nRpPass::NoShadows)
    {
        PROFILER_START(this->profRenderShadow);
        this->GatherShadowLights();
        this->RenderShadow(curPass);
        PROFILER_STOP(this->profRenderShadow);
    }
    else if (curPass.GetOcclusionQuery())
    {
        // perform light source occlusion query, this
        // marks the light sources in the scene as occluded or not
        this->DoOcclusionQuery();
    }
    else
    {
        ...
    }
}
rpSection.End();
}

```

nebula2/src/scene/nsceneserver_occlusion.cc

```

void nSceneServer::DoOcclusionQuery()
{
    PROFILER_START(this->profOcclusion);

```

```

n_assert(this->occlusionQuery);

if (this->occlusionQueryEnabled)
{
    nGfxServer2* gfxServer = nGfxServer2::Instance();

    // get current viewer position, NOTE: move the viewer position
    // into the screen onto the near plane since the occlusion check
    // needs to check whether the viewer is inside the occlusion bounding box
    // to check, if we don't account for the near plane then the front plane
    // of the occlusion plane might be clipped which would return 0 drawn pixels
    // when the object really isn't occluded (simply turning off back face
    // culling won't help either in some cases!)

    // FIXME: hmm, this method sucks... better to check viewer position against
    // a slightly scaled bounding box in IssueOcclusionQuery()!
    const vector3& viewerPos = gfxServer-
>GetTransform(nGfxServer2::InvView).pos_component();
    if (gfxServer->BeginScene())
    {
        // only update ModelViewProjection matrix in shaders...
        gfxServer->SetHint(nGfxServer2::MvpOnly, true);

        // issue queries...
        this->occlusionQuery->Begin();
        int num = this->rootArray.Size();
        for (int i = 0; i < num; i++)
        {
            Group& group = this->groupArray[this->rootArray[i]];
            if (group.renderContext->GetFlag(nRenderContext::DoOcclusionQuery))
            {
                this->IssueOcclusionQuery(group, viewerPos);
            }
        }
    }
    this->occlusionQuery->End();

    // get query results...
    // (NOTE: we could split this out and query the results
    // later, since the query will run in parallel to the CPU...
    // if only we had something useful todo in the meantime)
    int numQueries = this->occlusionQuery->GetNumQueries();
    for (int queryIndex = 0; queryIndex < numQueries; queryIndex++)

```

```

{
    bool occluded = this->occlusionQuery->GetOcclusionStatus(queryIndex);
    if (occluded)
    {
        Group* group = (Group*)this->occlusionQuery->GetUserData(queryIndex);
        group->renderContext->SetFlag(nRenderContext::Occluded, true);
        WATCHER_ADD_INT(watchNumOccluded, 1);
    }
    else
    {
        WATCHER_ADD_INT(watchNumNotOccluded, 1);
    }
}
this->occlusionQuery->Clear();

gfxServer->SetHint(nGfxServer2::MvpOnly, false);
gfxServer->EndScene();
}
}
PROFILER_STOP(this->profOcclusion);
}

```

nebula2/src/scene/nsceneserver_occlusion.cc

```

void nSceneServer::IssueOcclusionQuery(Group& group, const vector3& viewerPos)
{
    nSceneNode* sceneNode = group.sceneNode;
    n_assert(sceneNode);

    // initialize occlusion flags
    group.renderContext->SetFlag(nRenderContext::Occluded, false);

    // special case light:
    if (sceneNode->HasLight())
    {
        // don't do occlusion check for directional light
        nLightNode* lightNode = (nLightNode*)sceneNode;
        if (nLight::Directional == lightNode->GetType())
        {
            return;
        }
    }
}
}

```

```

// get conservative bounding boxes in global space, the first for
// the occlusion check is grown a little to prevent that the object
// occludes itself, the second checks if the viewer is in the bounding
// box, and if yes, no occlusion check is done
const bbox3& globalBox = group.renderContext->GetGlobalBox();

// check whether the bounding box is very small in one or more dimensions
// which may lead to z-buffer artifacts during the occlusion check, in that
// case, don't do an occlusion query for this object
// FIXME: could also be done once at load time in Mangalore...
vector3 extents = globalBox.extents();
if (extents.x < 0.001f || extents.y < 0.001f || extents.z < 0.001f)
{
    return;
}

bbox3 viewerCheckBox(globalBox.center(), globalBox.extents() * 1.2f);
// check if viewer position is inside current bounding box,
// if yes, don't perform occlusion check
if (!viewerCheckBox.contains(viewerPos))
{
    bbox3 occlusionBox(globalBox.center(), globalBox.extents() * 1.1f);
    // convert back to a matrix for shape rendering
    matrix44 occlusionShapeMatrix = occlusionBox.to_matrix44();
    this->occlusionQuery->AddShapeQuery(nGfxServer2::Box, occlusionShapeMatrix,
&group);
}
}

```

하드웨어 질의를 통해 차폐 여부를 질의하는 것은 `nOcclusionQuery::AddShapeQuery()` 함수를 사용하게 됩니다.

nebula2/src/gfx2/nd3d9occlusionquery.cc

```

void nD3D9OcclusionQuery::AddShapeQuery(nGfxServer2::ShapeType type, const
matrix44& modelMatrix, const void* userData)
{
    n_assert(this->inBegin);
    HRESULT hr;

    // create a new query
    nD3D9Server* d3d9Server = (nD3D9Server*)nGfxServer2::Instance();

```

```

IDirect3DDevice9* d3d9Dev = d3d9Server->d3d9Device;
n_assert(d3d9Dev);

IDirect3DQuery9* d3dQuery = 0;
hr = d3d9Dev->CreateQuery(D3DQUERYTYPE_OCCLUSION, &d3dQuery);
if (SUCCEEDED(hr))
{
    // store query so we can check its status later
    Query newQuery;
    newQuery.d3dQuery = d3dQuery;
    newQuery.userData = userData;
    this->queryArray.Append(newQuery);

    // start the query
    hr = d3dQuery->Issue(D3DISSUE_BEGIN);
    n_assert(SUCCEEDED(hr));

    // render the shape to check for
    d3d9Server->DrawShapeNS(type, modelMatrix);

    // tell the query that we're done, note that this is an asynchronous
    // query, so we'll get the result later in GetOcclusionStatus()
    hr = d3dQuery->Issue(D3DISSUE_END);
    n_assert(SUCCEEDED(hr));
}
else if (D3DERR_NOTAVAILABLE)
{
    // hmm, maybe no occlusion queries on this device...
    // just store a null pointer, we'll just simulate a failed query later on
    Query newQuery;
    newQuery.d3dQuery = 0;
    newQuery.userData = userData;
    this->queryArray.Append(newQuery);
}
else
{
    // an error...
    n_dxtrace(hr, "nD3D9OcclusionQuery::AddShapeQuery(): CreateQuery failed!");
}
}

```

nD3D9Server::DrawShapeNS() 함수는 큐브, 구체 등과 같은 기본 프리미티브 도형을 셰이더 없이

렌더링할 때 사용하는 함수입니다.

nebula2/src/gfx2/nd3d9server_shapes.cc

```
void nD3D9Server::DrawShapeNS(ShapeType type, const matrix44& model)
{
    n_assert(0 != this->shapeMeshes[type]);
    this->PushTransform(nGfxServer2::Model, model);
    this->refShader->CommitChanges();
    HRESULT hr = this->shapeMeshes[type]->DrawSubset(0);
    n_dxtrace(hr, "DrawSubset() failed in nD3D9Server::DrawShape()");
    this->PopTransform(nGfxServer2::Model);
}
```

게임 엔티티마다 고유의 질의 인덱스를 가지게 되며 이 질의 인덱스를 사용하여 해당 엔티티의 차폐 여부를 알 수가 있습니다. 차폐 여부에 대한 정보는 `nOcclusionQuery::GetOcclusionStatus()` 함수를 사용하여 알 수 있습니다. 만약 이 함수의 리턴값이 참인 경우 픽셀이 전혀 보이지 않는다는 것을 의미하므로 이 객체는 실제 렌더링시 그릴 필요가 없습니다.

nebula2/src/gfx2/nd3d9occlusionquery.cc

```
bool nD3D9OcclusionQuery::GetOcclusionStatus(int queryIndex)
{
    n_assert(!this->inBegin);
    Query& query = this->queryArray[queryIndex];
    IDirect3DQuery9* d3dQuery = query.d3dQuery;
    if (0 == d3dQuery)
    {
        // special case: no occlusion query available on this device,
        // always return that we're not occluded
        return false;
    }
    // an occlusion query returns the number of pixels which have passed
    // the z test in a DWORD
    DWORD numVisiblePixels = 0;
    n_assert(d3dQuery->GetDataSize() == sizeof(DWORD));

    // note: this method may return S_OK, S_FALSE or D3DERR_DEVICELOST.
    // S_FALSE is not considered an error!
    HRESULT hr;
    do
    {
        hr = d3dQuery->GetData(&numVisiblePixels, sizeof(DWORD), D3DGETDATA_FLUSH);
```

```
}  
while (hr == S_FALSE);  
  
// return true if we're fully occluded  
return (numVisiblePixels == 0);  
}
```