

3.1. Nebula Device 엔진 개요

Nebula2 엔진의 소스 코드는 \$nebulacode/nebulacode2 폴더 내에 위치해 있습니다. 'code/nebulacode2'의 소스 파일은 다시 헤더 파일과 소스 파일이 다른 디렉토리로 나누어서 위치하고 있는데 헤더 파일은 'code/nebulacode2/inc' 디렉토리에 위치하며 소스 파일은 'code/nebulacode2/src' 디렉토리에 위치합니다.

커널(Kernel)

여기에서는 Nebula엔진의 커널(kernel)에 대해서 살펴 보겠습니다.

커널은 흔히 운영체제(Operating System)을 이야기할 때 사용하는 개념으로 다음과 같이 정의하고 있습니다.

"????? ????? ????? ????? ? ??? ?? ??? ????? ?????? ?? . ??? ????? ??? ? (RAM)?? ?????, ?????
???? ? ??? ??? ?? ??? ?????."

Nebula 엔진에서 커널을 담당하는 모듈은 nKernelServer 모듈이며 운영체제에서의 커널과 마찬가지로 프로그램 실행시에 가장 먼저 생성 되는 부분으로 다음과 같은 일을 처리하고 있습니다.

- 파일 시스템
- 로그(Log) 메시지 처리 시스템
- 메모리 관리
- 객체의 생성과 해제
- 스크립팅 시스템
- 스레드(Thread) 관련 처리
- 타이머(Timer) 처리

Nebula 응용 프로그램에서 커널 객체는 제일 먼저 생성해야 하는 객체입니다. 커널 객체를 생성하기 위해서는 커널 서버를 생성해야 합니다.

```
// 커널 서버 객체 인스턴스의 생성  
nKernelServer kernelServer;
```

커널 서버 객체 인스턴스를 생성한 다음에는 커널 서버에 사용할 패키지를 선언한 다음 실제 사용하기 위해서는 커널 서버에 패키지를 추가해야 합니다. 여기에서 패키지란 앞에서 빌드 시스템을 살펴 볼 때 언급한 패키지로 모듈들의 집합이라고 생각할 수 있습니다.

```
//nebula 패키지 사용을 선언
nNebulaUsePackage(nnebula);
...
```

```
// nebula 패키지 추가. nebula의 기본적인 모듈들이 이 패키지 내에 위치한다.
kernelServer.AddPackage(nnebula);
```

패키지를 추가하면 추가된 패키지의 모듈들을 생성하고 이용할 수 있게 됩니다. 앞서 **Nebula**의 가장 중요한 특징 중 하나로 모듈러 프로그래밍을 이야기 했었는데 이렇게 패키지를 추가해서 원하는 서비스를 사용하도록 되어 있는 것도 **Nebula**의 모듈러 프로그래밍적인 특징 때문입니다.

그러면 이제 커널에서 처리하는 각각의 일들에 대해서 하나씩 살펴 보도록 하겠습니다.

파일 입출력 및 파일 시스템

Nebula 엔진은 기본적으로 멀티 플랫폼 엔진으로 플랫폼에 독립적인 파일 입출력이 가능한 파일 시스템을 제공하고 있으며 파일 시스템의 특징은 다음과 같습니다.

- 파일 서버 클래스에서는 **Windows, Linux, Mac OS X** 등의 호스트 OS의 파일 시스템에 대한 추상화를 통해 호스트 OS에 관계 없이 동일한 인터페이스를 제공한다.
- **completely removes differences between Windows-, Linux-or Xbox filesystems**
- 파일(**nFile**)과 디렉토리(**nDirectory**)에 대해서 각각의 클래스를 사용, 객체지향적으로 처리하고 있다.
- Win32 플랫폼에서는 ANSI- C 파일 시스템 처리 함수 대신 **CreateFile(), WriteFile(), ReadFile()** 과 같은 Win32 의 파일 시스템 함수를 사용하여 처리하고 있다.
- 디렉토리에 대한 별칭인 'Mangale Path' 컨셉을 제공하고 있다.
 - 긴 경로명 대신 별칭을 사용하는 것.
 - "**C:\Program Files\...\textures\image.dds**"와 같은 긴 경로명을 사용하는 대신 "**textures:image.dds**"와 같이 짧고 직관적인 경로명을 사용한다.
 - 설정된 경로가 필요할 때 **Nebula** 엔진에서 원래 경로로 치환한다.
 - **AmigaOS**에서 영향을 받음
- **NPK** 파일 시스템으로 파일들을 패킹할 수 있는 기능을 제공한다.
- **RAM** 파일 시스템 기능을 제공한다.

NPK 파일 시스템은 파일 아카이브(**File Archives**)에 대한 것으로 하나의 파일처럼 파일과 디렉토리를 묶은 다음 파일 시스템에서 처리할 수 있도록 하는 것을 말합니다. 이것을 이용하면 다음과 같은 장점과 특징이 있습니다.

- similar to Quake´s PAK files
- packs directory hierarchies into a single file

- no wasted disk space with many small files
- drastically smaller loading time
- cleaner application directory layout
- integrated with the Nebula file subsystem

RAM 파일 시스템은 메모리(RAM)의 일부분 하드 디스크 파일처럼 사용하는 것으로 서로 다른 프로세스에서 같은 메모리 영역을 파일처럼 접근할 수 있도록 하여 사용할 때 유용합니다. 예를 들어 3ds MAX에 모델을 익스포트할 때 특정 RAM 파일 영역으로 익스포트한 다음 모델 뷰어에서는 이 영역의 데이터를 읽어 들이도록 하는 것입니다. 이렇게 하면 뷰어를 3ds MAX 플러그인으로 만들지 않아도 되므로 프로그램의 유지보수가 쉬워지며 또한 두 프로세스 간에 필요한 데이터들을 메모리 상에서 접근하게 되므로 속도도 빠르다는 장점이 있습니다. 이렇게 공유 메모리를 사용하여 작업하는 것은 두 프로세스가 같은 하드웨어에서 실행되어야 하는 제약이 따르지만 같은 물리적 메모리에 접근할 수 있다면 정보 전달에 있어 가장 빠른 방법입니다.

RAM 파일은 nRamFileServer 클래스에서 처리되며 이 때 파일 객체는 nRamFile 클래스 인스턴스입니다.

nebula2/src/file/nsharedmemory.cc

```
void nSharedMemory::Open()
{
    n_assert(!this->IsOpen());

#ifdef __WIN32__
    // Open existing file mapping.
    this->mapHandle = OpenFileMapping(FILE_MAP_ALL_ACCESS, // Read+write access.
                                     FALSE, // Don't inherit handle.
                                     this->GetName());

    if (this->mapHandle == NULL)
    {
        n_printf("Failed to open file mapping (Error: %d).\n", GetLastError());
        return;
    }

    // Map view of file.
    this->mapHeader = MapViewOfFile(this->mapHandle, // Map handle.
                                     FILE_MAP_ALL_ACCESS, // Read+write access.
                                     0,
                                     0,
                                     0);

    if (this->mapHeader == NULL)
```

```

    {
        CloseHandle(this->mapHandle);
        n_printf("Failed to map view of file (Error: %d).\n", GetLastError());
        return;
    }
#elif defined(__LINUX__) || defined(__MACOSX__)
#else
#error "nSharedMemory::Open() is not implemented yet!"
#endif

    this->mapBody = (char*)this->mapHeader;
    this->mapBody += HeaderSize;
    this->isOpen = true;
    this->writable = false;
    this->LockMemory();
    this->ReadHeader();
}

```

공유 메모리에 접근할 때는 일반적인 read와 write의 호출과 같은 직렬화(serialization)가 이루어지지 않기 때문에 공유 메모리에 접근하는 프로세스 간에는 반드시 Race Condition 및 DeadLock 문제가 해결되어야 합니다. nSharedMemory::Open() 함수를 호출하면 이 함수에는 공유 메모리에 대한 정보를 가져 오기 전에 LockMemory() 함수를 먼저 호출하여 다른 프로세스의 메모리 접근을 막습니다.

```

void nSharedMemory::LockMemory()
{
    n_assert(this->IsOpen());
    n_assert(this->mapHeader != 0);

    char* p = (char*)this->mapHeader;
    while (*p != 0); // Wait for unlock.
    *p = 0;
}

```

nRamFile 객체의 메모리를 해제할 때에는 nSharedMemory 객체이맵핑된 메모리 영역에 대한 해제가 필요한데 이 작업은 UnmapViewOfFile()과 CloseHandle() Win32 함수를 사용합니다.

nebula2/src/file/nsharedmemory.cc

```

void
nSharedMemory::DestroyMapping()
{

```

```

#ifdef __WIN32__
    // Unmap view of file.
    if (!UnmapViewOfFile(this->mapHeader))
    {
        n_printf("Failed to unmap view of file (Error: %d).\n", GetLastError());
        return;
    }

    // Close map handle.
    if (!CloseHandle(this->mapHandle))
    {
        n_printf("Failed to close map handle. (Error: %d)\n", GetLastError());
        return;
    }
#elif defined(__LINUX__) || defined(__MACOSX__)
    if (-1 == munmap(this->mapHeader, this->capacity))
    {
        n_printf("Failed to unmap view of file (Error: %d).\n", errno);
        return;
    }
    if (-1 == close(this->mapHandle))
    {
        n_printf("Failed to close mapped file. (Error: %d).\n", errno);
        return;
    }
    if (-1 == remove(this->mapFileName.Get()))
    {
        n_printf("Failed to delete mapped file. (Error: %d).\n", errno);
        return;
    }
    if (-1 == remove(this->mapFileName.Get()))
        this->mapFileName.Clear();
#else
#error "nSharedMemory::DestroyMapping() not implemented yet!"
#endif

    this->capacity = 0;
    this->count = 0;
    this->mapBody = 0;
    this->mapHeader = 0;
    this->mapHandle = 0;
}

```

IPC(Inter-Process Communication)

Nebula 2에서는 콘솔에서 스크립트 명령어를 통해서 두 프로세스간에 통신하는 것이 가능합니다. Nebula2에서의 IPC 서버는 nRemoteServer 클래스를 사용하여 처리하고 있으며 이것을 이용하면 원격 디버깅 등에 유용하게 사용할 수가 있습니다.

Nebula2에서 IPC 처리를 위한 클래스들은 다음과 같습니다.

- nIpcServer
- nIpcClient
- nIpcBuffer
- nIpcPeer
- nIpcAddress

다음은 IPC 클라이언트 처리에 관련한 소스 코드입니다.

```
...
// 접속할 주소 설정
nIpcAddress ipcAddress(hostArg.Get(), portArg.Get());

// IPC 클라이언트 객체 인스턴스 생성
nIpcClient ipcClient;
ipcClient.SetBlocking(true);

// 접속
ipcClient.Connect(ipcAddress)

...

nIpcBuffer msg(4096);
while (running && lineOk)
{
    char line[2048];

    // 메시지 전송
    msg.SetString("psel");
    if (ipcClient.Send(msg))
    {
        // 메시지 받기
```

```

    if (!ipcClient.Receive(msg))
    {
        printf("Error receiving prompt string!\n");
        running = false;
    }
    ...
}

// 접속 해제
ipcClient.Disconnect();

```

IPC 클라이언트에 대한 소스 코드는 'nebula2/src/tools/nremoteshell.cc' 파일을 참고하기 바랍니다.

로그 시스템

게임 엔진에서의 각 모듈에서는 모듈의 특성이나 모듈이 사용되는 응용 프로그램의 서비스 형태에 따라 각기 알맞은 메시지를 출력해야 할 필요가 있습니다. 예를 들어 네트워크를 통해서 수시로 발생하는 로그의 경우 파일에 기록하는 것이 좋지만 응용 프로그램이 시작할 때 초기화 등의 실패를 알리는 메시지는 다이얼로그나 메시지 박스를 사용해서 출력하는 것이 훨씬 더 직관적입니다. **Nebula**에서는 이런 로그 메시지 출력을 위해서 포맷된 형식의 메시지를 파일이나 해당 플랫폼의 다이얼로그로 출력할 수 있는 로그 시스템을 제공하고 있습니다.

Nebula2 엔진에서 제공하는 로그 메시지 함수는 다음과 같은 것들이 있습니다.

- n_printf()
- n_message()
- n_error()
- n_dbgout() : win32 only

Nebula 엔진의 로그 시스템은 다음의 방법으로 사용할 수 있습니다.

- 1) 로그 핸들러를 생성한 다음 생성한 핸들러를 커널 서버에 설정한다.
- 2) 로그 메시지를 출력한다.

```

// 로그 핸들러를 생성
#ifdef __WIN32__
    nWin32LogHandler logHandler("nviewer");
    kernelServer.SetLogHandler(&logHandler);

```

```
#endif

...
// 로그 메시지 출력
n_message("Foo: %d", numfoo);
```

메모리 관리

Nebula2에서는 디버그 모드에서 메모리 누수 등을 발견할 수 있도록 메모리 할당을 위한 커스텀 코드를 제공합니다. 엔진에서 이렇게 커스텀 메모리 할당 함수를 제공하지 않고 플랫폼의 네이티브 함수를 그대로 사용하는 경우 나중에 프로젝트가 커진 후에 변경하려면 매우 힘듭니다. 그러므로 엔진에서 이러한 메모리 할당과 관련한 함수를 제공해서 개발 시작부터 엔진의 API로 개발할 수 있도록 하는 것이 좋습니다.

Nebula의 메모리 할당 함수는 'code/nebula2/inc/kernel/ntypes.h'에 다음과 같이 정의되어 있습니다.

```
#if defined(_DEBUG) && defined(__WIN32__)
#define n_new(type) new(__FILE__, __LINE__) type
#define n_placement_new(place, type) new(place, __FILE__, __LINE__) type
#define n_new_array(type, size) new(__FILE__, __LINE__) type[size]
#define n_delete(ptr) delete ptr
#define n_delete_array(ptr) delete[] ptr
#define n_malloc(size) n_malloc_dbg(size, __FILE__,
__LINE__)
#define n_calloc(num, size) n_calloc_dbg(num, size, __FILE__, __LINE__)
#define n_realloc(memblock, size) n_realloc_dbg(memblock, size, __FILE__, __LINE__)
#define n_free(memblock) n_free_dbg(memblock, __FILE__, __LINE__)
#else
```

디버그용 메모리 할당 함수들의 실제 구현은 `ndbgalloc.cc` 파일에 정의되어 있습니다. 다음은 `ndbgalloc.cc` 파일에 정의되어 있는 `n_malloc_dbg()` 함수의 코드입니다. 이렇게 메모리 관리 함수를 따로 만들어서 사용하면 메모리 할당에 대한 추적이나 메모리 누수에 대한 처리를 할 수 있으므로 개발 시 유용합니다.

nebula2/src/kernel/ndbgalloc.cc

```
void* n_malloc_dbg(size_t size, const char* filename, int line)
{
```

```

void* res = _malloc_dbg(size, _NORMAL_BLOCK, filename, line);
if (nMemoryLoggingEnabled)
{
    n_printf("%lx = n_malloc(size=%d, file=%s, line=%d)\n", res, size, filename, line);
}
return res;
}

```

메모리 관리와 관련한 코드를 직접 작성하는 방법도 있지만 [Dmalloc](http://dmalloc.com/)(<http://dmalloc.com/>)과 같은 디버깅용 외부 라이브러리로 교체하여 사용하는 방법도 있습니다.

쓰레드

Nebula2에서는 멀티 쓰레딩을 손쉽게 하기 위해서 nThread, nMutex, nEvent와 같은 래퍼(wrapper) 클래스를 제공하고 있습니다.

멀티 쓰레드는 플랫폼마다 틀린데 예서는 사용하는 플랫폼별 쓰레드 서비스는 다음과 같습니다.

- Windows: Win32 threading routines
- Linux: Posix threads

Nebula2 엔진에서는 다음의 경우에 쓰레드를 사용하여 처리하고 있습니다.

- blocking Socket 루틴
- 리소스의 백그라운드 로딩

(※ 멀티 쓰레드와 관련한 또 다른 중요한 이슈로는 멀티 코어와 관련한 이슈들이 있습니다만 현재의 Nebula2에서는 멀티 코어를 지원하지 않습니다.)

타이머

타이머에 대한 처리는 타이머 서버(nTimeServer)에서 처리하고 있으며 Nebula 엔진에서는 1/1000초 단위의 정확도를 가진 타이머를 제공하고 있습니다. 이 타이머에 대한 처리는 해당 플랫폼에 따라 틀린데 win32에서는 QueryPerformanceCounter() API를 사용하며 Linux 플랫폼에서는 gettimeofday() API를 사용하고 있습니다.

게임에서의 타이머는 게임 플레이와 관련한 시뮬레이션 속도와 게임 렌더링과 관련한 게임 프레임 속도를 구분하여 별도로 처리하는 것이 중요합니다. 그 이유는 멀티 플레이와 같은 게임에서는 사용자마다 시스템 사양이 각기 틀릴 수 있는데 이렇게 하드웨어 시스템이 다른 경우에는 게임의 렌더링 속도도 시스템에 따라 달라질 수 있지만 게임 플레이와 관련한 게임 시뮬레이션의 속도는 사용자의 시스템과는 무관하게 모든 플레이어에게 동일하게 적용이 되어야지 하기 때문입니다. Nebula 엔진에서는 이렇게 게임

시뮬레이션을 위한 타이머와 게임 프레임 처리를 위한 타이머를 구분하여 별개로 처리할 수 있게 되어 있습니다.

폴딩 객체(Folding Object) 시스템

폴딩 객체(Folding Object)라는 것은 Nebula 객체를 생성할 때 가장 상위 수준의 오브젝트로 폴딩 객체 다음에 생성되는 Nebula 객체들을 모두 폴딩 객체의 자식 노드가 됩니다.

폴딩 객체는 Nebula 스크립트에서 다음과 같이 정의되어 있습니다.

```
# ---
# $parser:ntclserver$ $class:ntransformnode$
# ---

new ntransformanimator rot
  &nbsp; sel rot
  ...
  sel ..

new nshapenode shape0
  sel shape0
  sel ..
  sel ..
```

폴딩 객체는 Nebula 스크립트에서 제일 첫 부분에 정의되며 '\$parser' 와 '\$class'로 정의됩니다.

\$parser는 이 스크립트 파일을 파싱할 스크립트 서버를 설정하는 것으로 위에서는 Tcl 서버 (ntclserver)로 설정이 되어 있습니다. 다음에 \$class 로 정의하는 Nebula 객체가 바로 폴딩 객체입니다. 위의 예에서는 nTransformNode가 폴딩 객체가 됩니다. 그리고 폴딩 객체를 정의한 다음에 생성되는 Nebula 객체들은 모두 이 폴딩 객체의 자식 노드가 됩니다. 그런데 Nebula 객체를 생성하기 위해서는 디렉토리 이름과 같은 형식글 가지는 객체의 이름이 필요한데 위에 정의에서는 객체의 종류만 정의되어 있고 이름은 정의되어 있지 않습니다. 폴딩 객체의 객체 이름은 스크립트 파일 이름을 객체의 이름으로 사용하게 됩니다. 만약에 스크립트 파일 이름이 'torus.n2'라고 하면 위의 스크립트에서 생성되는 객체는 다음과 같이 동일하게 됩니다.

```
new ntransformnode torus
  sel torus
  ...
  new ntransformanimator rot
    sel rot
  ...
  sel ..
```

```
new nshapenode shape0
  sel shape0
  sel ..
  sel ..
```

폴딩 객체와 디렉토리를 이용하면 리소스 관리를 편리하게 하는 방법이 있는데 그러면 이번에는 이것에 대한 예를 살펴 보도록 하겠습니다.

임의의 캐릭터와 관련한 메쉬, 텍스처 등의 리소스들을 디렉토리 구조를 나누어서 관리한다고 가정해 보겠습니다. '*mycharacter.n2*'라는 디렉토리에 해당 캐릭터를 렌더링하는 필요한 모든 리소스 데이터가 위치해 있다고 하면 캐릭터는 보통 텍스처, 애니메이션, 메쉬 데이터 등을 가지게 되므로 해당 디렉토리는 다음과 같이 구성이 됩니다.

```
/mycharacter.n2 - 리소스 파일을 포함하는 디렉토리.
  /_main.n2 - Nebula 스크립트 파일 (반드시 이름이 '_main.n2' 여야 한다)
  /body.dds - 텍스처 파일
  /walk.nanim2 - 애니메이션 파일
  /body.n3d2 - 메쉬 데이터 파일
```

위의 경우에 폴딩 객체는 다음의 방법으로 읽어 들일 수가 있습니다.

```
...
kernelServer->PushCwd(this->refRootNode.get());
kernelServer->Load("mycharacter.n2"); // 인자로 디렉토리 이름을 넘겨 준다.
kernelServer->PopCwd();
...
```

`nKernelServer::Load()` 함수에 디렉토리 이름을 설정하게 되면 `Load()` 함수에서는 해당 디렉토리의 '*_main.n2*' Nebula 스크립트 파일을 찾아서 읽어 들입니다. 이 때 디렉토리 이름이 반드시 '*.n2*'로 끝나야 하는 것은 아니지만 해당 디렉토리는 반드시 '*_main.n2*' Nebula 스크립트 파일이 존재해야 합니다.

앞에서 살펴 본 바와 같이 폴딩 객체와 디렉토리를 이용하면 연관성이 있는 리소스들을 하나의 디렉토리에 위치시켜서 리소스를 관리하는 방법 그대로 엔진에서 읽어 들일 수 있으므로 관리뿐만 아니라 사용이 편리하다는 이점이 있습니다.

※ 주의) 위의 예는 사용자의 편의에 따라 사용하는 것이므로 리소스를 관리할 때 반드시 이렇게 폴딩 객체와 디렉토리를 사용해야 하는 것은 아닙니다.

시그널(Signal) 시스템

Nebula 시그널 시스템은 이벤트 처리를 위한 신호(signal) 처리 시스템입니다. 키보드, 마우스와 같은 입력 장치의 이벤트나 기타 메시지 이벤트 등이 발생했을 때 함께 호출되어야 하는 함수를 등록해서 이벤트 처리를 손쉽게 할 수 있도록 하는 시스템입니다.

시그널 시스템 모듈은 '`code/nebula2/inc/signals/`' 디렉토리에 위치해 있습니다.

(시그널 시스템 모듈은 Bruce Mitchener의 제안으로 Mateu Batle이 작성한 모듈로 RadonLabs의 패키지에는 없는 모듈입니다.)

Nebula 서버(Server)

Nebula 엔진의 서버 객체들은 **Nebula** 객체의 이름공간에서 `/sys/servers/` 경로에 위치합니다.

Nebula 엔진을 이해하기 위해서 주의 깊게 살펴 보아야 할 **Nebula Server** 군에는 다음의 것들이 있습니다.

- Script Server
- GFX Server
- Audio Server
- Input Server
- SceneServer
- Persistence Server
- Resource Server

그 외 중요한 서버로는 다음의 것들이 있습니다.

- nConServer
- nPrefServer

그러면 다음에는 앞에서 언급한 **Nebula**의 서버군들에 대해서 차례대로 살펴 보도록 하겠습니다.

스크립트(Script) 서버

스크립트 시스템은 많은 게임 엔진에서 기본적으로 제공하는 기능 중의 하나입니다. 이들 엔진 중에서 Quake(Quake-C)나 Unreal(Unreal Script)과 같이 직접 제작한 스크립트 언어를 제공하는 엔진들도 있고, 다른 스크립트 언어를 내장하는 엔진들도 있습니다. **Nebula**는 후자에 속하는 엔진으로 현재 Tcl, Python, Lua, Ruby의 네 가지 스크립트 언어를 지원하고 있습니다. 이것은

Tcl은 Radon Labs사에서 **Nebula** 엔진의 기본으로 내장 시킨 스크립트 언어이며, Radon Labs사에서 제작되는 게임에는 Tcl을 스크립트 언어로 사용하고 있습니다. 그런데 **Nebula** 엔진의 Tcl은 원래 Tcl의 명령어 중에서 게임에 필요하다고 중요하다고 생각되는 36개의 명령어만 따로 모아서 기존의 Tcl보다 훨씬 더 가볍게 만든 버전을 사용하고 있습니다. **Nebula** 엔진에서는 이것을 MicroTcl이라고 부르고 있

는데 이 **MicroTcl**은 기존의 **Tcl**이 **660KB**인 것에 비해 그 크기가 **160KB** 밖에 되지 않아 **XBox** 등과 같은 제한된 메모리 시스템을 가지고 있는 콘솔 플랫폼 등의 개발에도 적합한 것이 특징입니다. 그리고 **Tcl** 외의 앞에서 언급한 스크립트 언어는 **Nebula** 엔진의 오픈 커뮤니티에서 기부한 모듈들입니다.

Nebula 스크립트 시스템의 매력은 **C++** 함수와 스크립트를 쉽게 바인딩할 수 있는 직관적인 방법을 제공하며, 또한 다양한 스크립트 언어를 확장하여 추가할 수 있도록 추상화가 잘 되어 있다는 것입니다. 만약에 원한다면 **JavaScript**나 혹은 여러분이 직접 제작한 스크립트 언어도 **Script Server** 클래스를 상속하여 작성하는 것만으로 **Nebula** 엔진과 결합하여 확장시킬 수가 있습니다.

플랫폼 추상화(**Platform Abstraction**)

최근의 게임 엔진들은 **PC** 플랫폼 뿐만 아니라 **XBox**, **PlayStation**과 같은 다양한 플랫폼을 지원하는 것이 특징입니다. 그런데 이러한 다양한 플랫폼을 지원하기 위해서는 플랫폼에 종속되어 있는 특정 플랫폼 **API**에 대한 추상화 레이어를 지원해야 할 필요가 있습니다. 대표적인 것으로 **DirectX**나 **OpenGL**과 같은 그래픽과 관련한 플랫폼 **API**를 들 수 있습니다. 그래픽 **API**의 경우 **PC** 플랫폼이나 **XBox** 플랫폼에서는 **DirectX API**의 사용이 일반적이지만 **PlayStation**과 같은 플랫폼에서는 **OpenGL** 표준에 따르는 **API**를 사용하고 있습니다. 그러므로 게임 엔진에서는 플랫폼마다 틀린 이러한 **API**들에 대한 추상화 레이어 (**Abstraction Layer**)를 제공하여 개발자가 각 플랫폼마다 플랫폼 종속적인 **API**를 모두 알아야 할 필요 없이 하나의 플랫폼에서만 작동하도록 제작하면 다른 플랫폼으로도 쉽게 이식이 가능하도록 해야 합니다. 이것을 보통 플랫폼 추상화라고 하며 이렇게 플랫폼 추상화가 필요한 모듈로는 설명한 그래픽 **API** 외에도 사운드와 사용자 입력 처리와 관련한 모듈을 예로 들 수 있습니다.

Nebula에서는 다음과 같이 그래픽, 사운드, 사용자 입력과 관련한 플랫폼 추상화 레이어를 제공합니다.

- 그래픽스(**GFX**) 서버 - **nGfxServer2**
- 사운드(**Sound**) 서버 - **nAudioServer3**
- 사용자 입력(**Input**) 서버 - **nInputServer**

이상의 서버 모듈들은 나중에 따로 지면을 할애해서 자세하게 살펴 보겠습니다.

콘솔(**Console**) 서버

대부분의 게임 엔진들이 게임 내에서 스크립트 명령어나 설정을 위한 명령어를 입력할 수 있는 콘솔 시스템을 제공하고 있지만 **Nebula**의 콘솔 시스템이 일반적인 게임 엔진에서 제공하는 콘솔과의 차이점이라고 한다면 **Nebula**의 콘솔은 셸(**Shell**)이라는 점입니다. 기본적인 스크립트 명령어의 입력 외에도 현재 시스템에 로딩된 모든 **Nebula** 객체를 살펴 볼 수 있을 뿐만 아니라 이들의 값을 변경하거나 새로운 **Nebula** 객체를 생성하고 삭제하는 것도 가능합니다. 이러한 작업은 유닉스나 도스의 셸에서 파일을 다루는 일과 동일합니다. 심지어는 콘솔에서 **Nebula** 객체가 출력되는 형태나 접근하는 방법 역시 매우 흡사합니다.

콘솔 스크린샷 하나 넣을 것 -Hyoun Woo Kim 4/4/08 9:13 AM

이러한 관점에서 보게 되면 왜 Nebula 객체가 일반적인 OS의 '디렉토리 + 파일' 이름의 형식을 취하고 있는지를 쉽게 알 수 있습니다.

Nebula 콘솔의 또 다른 강력한 기능 중 하나로 감시자(watcher)라는 것이 있습니다. 이 감시자는 Nebula 객체의 상태를 콘솔에 표시할 수 있도록 하는 기능입니다.

```
/>sel /sys/servers/console  
/sys/servers/console> .addwatch('/usr/scene/myobject.getfoo')
```

우선 sel 명령어로 콘솔 객체(console)로 이동한 다음 'addwatch' 명령어를 사용하여 '/usr/scene/myobject'에 대한 getfoo 함수의 리턴값을 추가했습니다. 이렇게 추가하고 나면 매번 게임 루프를 돌 때마다 getfoo를 호출하여 getfoo가 리턴하는 값을 확인할 수 있습니다. 이러한 기능은 컴파일러에서 제공하는 중단점(breakpoint)등을 이용하지 않고 게임 내에서 객체의 상태를 디버깅할 수 있는 매우 유용한 기능 중의 하나입니다.

영속성(Persistence) 서버

Nebula에서는 Nebula 객체를 Nebula 스크립트(.n2)로 저장할 수 있는 기능을 제공합니다. 객체의 저장은 Nebula 객체의 이름 공간에서 저장하고 싶은 객체로 이동한 다음 콘솔에서 save 명령어로 저장하게 되면 바로 저장이 됩니다.

콘솔에서 아래와 같이 입력하게 되면 tiger.n2라는 파일이 생성이 됩니다.

```
/usr/scene/tiger>.save tiger.n2
```

생성된 파일은 ASCII 포맷이므로 일반적인 텍스트 편집기로 열어 볼 수 있으며 이 파일에는 myobject의 정보가 저장이 되어 있습니다.

```
# ---  
# $parser:ntclserver$ $class:ntransformnode$  
# ---  
new ntransformnode tiger  
  sel tiger  
  .setlocalbox -0.006938 1.773186 1.350278 1.862026 1.826938 4.253572  
  new ntransformnode model  
    sel model  
    .setlocalbox -0.006938 0.870908 1.350278 1.862026 1.826938 4.253572  
    .setposition 0.000000 0.902278 0.000000  
  new nshapenode static_0_0  
    sel static_0_0
```

```

.setlocalbox -0.006938 0.870908 1.350278 1.862026 1.826938 4.253572
.settexture "DiffMap0" "textures:examples/tiger.dds"
.settexture "BumpMap0" "textures:examples/tiger_bump.dds"
.setvector "MatDiffuse" 1.000000 1.000000 1.000000 1.000000
.setvector "MatEmissive" 0.000000 0.000000 0.000000 0.000000
.setfloat "MatEmissiveIntensity" 1.000000
&nbsp; .setvector "MatSpecular" 0.500000 0.500000 0.500000 1.000000
.setfloat "MatSpecularPower" 32.000000
.setint "CullMode" 2
.setfloat "BumpScale" 0.000000
.setshader "static"
.setmesh "meshes:examples/tiger_s_0.n3d2"
.setgroupindex 0
.setneedsvertexshader false
sel ..
sel ..
sel ..
# ---
# Eof

```

첫 줄의 \$parser:ntclserver는 이 스크립트 파일을 읽어 들일 때 tcl server를 이용하여 스크립트 파일을 읽어야 한다는 것을 의미합니다. 스크립트 파일에는 세 개의 Nebula 객체가 있습니다. 그리고 위의 스크립트를 읽어 들여서 Nebula 객체를 생성하게 되면 생성된 Nebula 객체의 이름은 Nebula 이름 공간에서 '/tiger/model/static_0_0'로 보이게 됩니다.