HDRI 포맷 분석: OpenEXR

Analysis of High Dynamic Range Image Format: OpenEXR

김성예(S.Y. Kim) CG기반기술연구팀 연구원
표순형(S.H. Pyo) CG기반기술연구팀 연구원
최병태(B.T. Choi) CG기반기술연구팀 책임연구원, 팀장

본 고는 가장 최근에 개발된 HDRI 포맷인 OpenEXR 파일 포맷에 대하여 살펴본다. 지금까지 HDRI 포맷으로서 사용되었던 Log Luv TIFF나 Radiance RGBE와 같은 FP 타입의 HDRI 포맷은 압축 방식에 있어서 모두 손실 압축(loss compression)이거나 또는 너무 많은 메모리(32bits FP)를 소비하는 단점이 있었다. 그러나 ILM에 의해서 개발된 EXR 영상 포맷은 16bits FP 데이터 타입에 무손실 압축(lossless compression)을 제공할 뿐만 아니라 유연한 확장성을 제공하고 있기 때문에 앞으로 영상 산업에서의 비주얼 이펙트 표현에 중요한 역할을 할 것으로 기대된다.

I. 서 론

OpenEXR[1]은 ILM(Industrial Light & Magic)[2]에 의해서 개발된 HDRI(High Dynamic Range Image) 포맷이다. HDRI가 무엇이며, 왜 필요한지에 대해서는 "HDRI 기술 동향[3]"을 참조한다. OpenEXR은 이미 ILM에 의해서 해리포터와 마법사의 돌(2001), 맨인블랙II(2002), 갱스오브뉴욕(2002), 싸인(2002) 등의 영화에서 사용되어 왔고 현재도 계속 사용되고 있다. ILM은 OpenEXR을 2003년 1월 무료 소프트웨어로 릴리즈 하였으며 Pixar의 RenderMan 렌더링 패키지를 위한 디스플레이 드라이버와 포토샵 용 플러그인도 오픈되어 있다[4]. OpenEXR의 구성은 다음과 같다.

- IlmImf: OpenEXR 영상을 읽고 저장할 수 있는 라이브러리
- Half: half 타입의 값을 다루기 위한 C++ 클래스
- Imath: Matrix, 2차원 및 3차원 변환, linear/quadratic/cubic 방정식 등을 지원하는 수학 라이브러리
- exrdisplay: OpenEXR 영상을 디스플레이 해주는 간단한 애플리케이션

현재 OpenEXR의 릴리즈 버전은 1.0.5이며, Windows와 Mac의 OS X를 비롯하여 Linux, Unix 등 모든 운영 체제에서 사용이 가능하다. 또한 NVIDIA's GeForce FX와 Quadro FX 그래픽 카드에서 하드웨어 렌더링이 지원된다.

본 고에서는 OpenEXR 파일 포맷이 개발된 배경에서부터 특징을 설명하고 파일 포맷에 대하여 자세히 분석해 보도록 한다. 또한, 관련 소프트웨어의 개발 상황에 대해서도 설명한다.

II. OpenEXR의 역사

ILM은 비주얼 이펙트 산업에서의 더 높은 컬러 신뢰도(color fidelity)에 대한 요구에 부응하기 위하여 EXR 포맷을 개발하였다. 2000년에 프로젝트가 시작되었을 때, ILM은 현존하는 여러 파일 포맷을 검토하였지만 기존의 파일 포맷에서는 다음과 같은 문제점이 제기되었다.

- 8bits 또는 10bits 포맷은 HDR 장비로부터 획득된 높은 콘트라스트(contrast)를 가지는 영상을 저장하기 위해 요구되는 dynamic range에 한계가 있다.
- 16bits 정수형 포맷은 전형적으로 색상 값을

0(검정색)에서 1(흰색) 사이로 표현한다. 그러나 HDR 장비나 film negative에 의해서 획득될 수 있는 그 이상의 값(예: chrome highlight)은 표현하지 못한다.

- 32bits FP(Floating-Point) TIFF는 비주얼 이펙트 작업에 너무 큰 용량을 제공한다. 32bits TIFF는 VFX 영상을 위한 dynamic range와 충분한 정밀도(precision) 이상을 제공하지만, 이것은 저장공간이 너무 많이 요구된다.

이러한 이유로 ILM은 자체 HDR 영상 포맷을 개발하여 EXR이라 하고 이를 공개함으로써 OpenEXR이라 하였다.

III. OpenEXR의 특징

OpenEXR은 16bits FP 컬러 컴포넌트를 가지는 HDR 파일 포맷이다. IEEE-754 FP 명세가 16bits 포맷을 정의하지 않기 때문에 ILM은 'half'라는 포맷을 새롭게 생성하였다. Half 값은 부호(sign) 1bit, 지수(exponent) 5bits, 가수(mantissa) 10bits를 가진다. Linear 영상들을 위해서 이 포맷은 30 f-stop을 지원하며, 각 f-stop 당 컬러 컴포넌트 당 1024(2^{10}) 스텝을 지원한다. 또한, 추가적인 10 f-stop을 가짐으로써 폭넓은 f-stop을 지원하도록 하고 있다. Half 포맷은 NVIDIA의 Cg[5]에서의 half 데이터 타입과 동일하기 때문에 개발자가 GeForce FX와 같은 NVIDIA의 GPUs(Graphic Processing Units) 상에서 직접 OpenEXR 영상을 처리할 수 있도록 한다. OpenEXR은 half 데이터 타입뿐 아니라 16bits UI(Unsigned Integer)와 32bits FP 데이터 타입도 지원한다. 또한 OpenEXR 영상은 임의의 수의 채널을 가질 수 있고, 각 채널마다 서로 다른 데이터 타입을 가질 수 있다. 현재 릴리즈 버전은 여러 가지 무손실 압축 기법을 지원하며 그 중에는 film grain을 가지는 영상에 대해서 2:1 정도의 높은 압축률을 달성할 수 있다.

OpenEXR의 주요한 특징을 정리하면 다음과 같다.

- High dynamic range와 색상 정밀도가 기존의 8bits 또는 10bits 파일 포맷보다 크다.
- 16bits FP 픽셀을 지원하며, half라고 불리는 픽셀 포맷은 NVIDA의 Cg에서 사용되는 half 데이터 타입과 일치하기 때문에 GeForce FX와 Quadro FX 3D 그래픽 솔루션에서 지원될 수 있다.
- 여러 가지 무손실 영상 압축 알고리듬을 가진다. 제공되는 코덱 중 일부는 2:1의 무손실 압축이 가능하다.
- C++ 클래스를 확장함으로써 쉽게 새로운 압축 코덱이나 영상 타입이 추가될 수 있다. 스트링(string), 벡터(vector), 정수형(integer)과 같은 새로운 속성(attribute)도 헤더에 추가시킬 수 있다.

(그림 1)은 OpenEXR 영상에 노출을 달리하여 나타나는 효과를 보여준다.

(a) 오른쪽 방향으로 노출 증가

(b) 왼쪽 방향으로 노출 증가

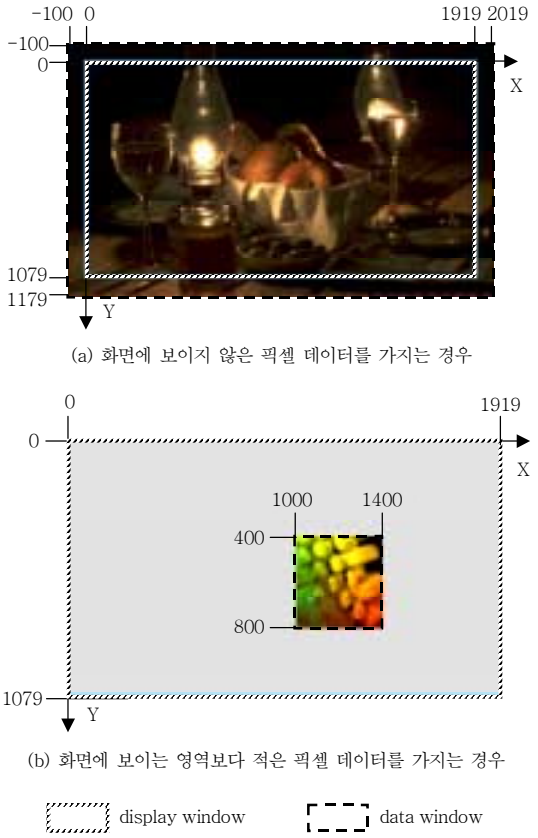
(그림 1) OpenEXR 영상

IV. OpenEXR 파일 포맷

OpenEXR의 픽셀 데이터를 표현하는 픽셀 공간(pixel space)은 2차원 (x, y)좌표로 표현되고 x는 왼쪽에서 오른쪽으로 y는 위에서 아래로 증가하는 방향을 가진다. 'pixels'는 픽셀 공간에 위치한 데이터를 나타낸다. OpenEXR 영상의 경계는 디스플레이 윈도(display window)라 하고 픽셀 공간의 축에 평행한 직사각형 영역으로 (x_{min}, y_{min}), (x_{max}, y_{max})에 의해서

정의된다. OpenEXR 파일은 모든 픽셀에 대한 픽셀 데이터를 디스플레이 윈도 내에 가지고 있는 것은 아니다. 즉, 디스플레이 윈도 외부의 픽셀에 대한 정보도 모두 파일에 가지고 있다. 유효한 픽셀 데이터에 대한 영역은 픽셀 공간의 데이터 윈도(data window)에 의해서 정의된다. 이 또한 축에 평행한 직사각형 영역이다. (그림 2)는 OpenEXR 파일의 픽셀 공간, 디스플레이 윈도와 데이터 윈도 구조를 나타낸다.

모든 OpenEXR 영상은 하나 이상의 채널을 가지고 있으며 각 채널은 이름(name), 데이터 타입(type), x와 y 샘플링률(sampling rate)을 가진다. 채널의 이름은 텍스트 스트링으로 표현되며 영상 파일을 읽는 프로그램에게 채널 내의 데이터를 어떻게 해석할지에 대하여 알려주는 역할을 한다. 몇 가지 미리 정의된 채널 이름에 대한 해석방식은 <표 1>과 같다.



(그림 2) OpenEXR 파일의 공간 구조

<표 1> 정의된 채널

채널	설명
R	Red intensity
G	Green intensity
B	Blue intensity
A	Alpha/opacity: 0은 투명한 픽셀, 1은 opaque 픽셀

OpenEXR은 <표 2>와 같이 3가지 채널 데이터 타입을 지원한다.

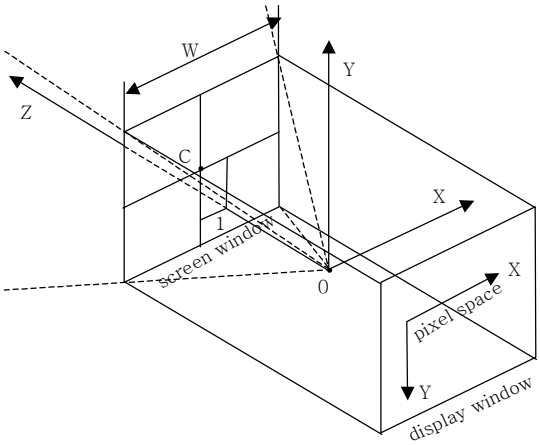
<표 2> 채널 데이터 타입

타입	설명
HALF	16bits FP, 일반적인 영상 데이터
FLOAT	32bits IEEE-754 FP, 16bits로 부족한 데이터에 대해서 사용(예: 깊이 채널)
UNIT	Blue intensity, 32bits UI, Object ID와 같은 픽셀 당 이산적인 데이터에 대해 사용

채널의 x, y 샘플링률인 s_x, s_y는 영상의 데이터 윈도 내에 존재하는 픽셀을 결정한다. RGBA 영상에 대해서 s_x, s_y는 모든 채널에 대해서 각각 1이다. 그리고 각 채널은 모든 픽셀에 대한 데이터를 포함한다. 다른 타입의 영상에 대해서, 일부 채널은 서브 샘플링(sub-sampling) 되기도 하는데 예를 들면, 4:2:2 YUV 영상에서 s_x, s_y는 Y 채널에 대해서는 각각 1, U, V 채널에 대해서는 s_x가 2가 된다.

x, y의 픽셀 공간에서의 영상은 (그림 3)과 같은 형태의 카메라 좌표계에 의해 원근 투영(perspective projection)되어 생성된다. 카메라는 3차원 카메라 좌표계의 원점 O에 위치하며 z의 양의 방향을 향한다. 카메라에 의해서 기록된 사각형 영역을 스크린 윈도(screen window)라 하며, 픽셀 공간에서 영상 파일의 디스플레이 윈도에 해당된다. 스크린 윈도의 크기와 위치는 윈도의 중심 C의 x, y 좌표와 윈도의 너비 W에 의해서 정해진다. 높이는 C와 W, 디스플레이 윈도 그리고 픽셀의 aspect ratio에 의해서 계산될 수 있다.

OpenEXR 파일은 크게 헤더(header)와 픽셀 데이터(pixels)의 두 부분으로 나뉘어지며 헤더는 픽셀 데이터를 설명하는 속성들의 리스트이다. <표 3>



(그림 3) OpenEXR의 픽셀 공간과 카메라 좌표계와의 관계

<표 3> OpenEXR 헤더의 속성 리스트

이름	설명
displayWindow, dataWindow	디스플레이 윈도, 데이터 윈도
pixelAspectRatio	픽셀의 높이로 폭을 나눈 값
Channels	영상 채널
Compression	영상 내 모든 채널의 픽셀 데이터에 적용되는 압축 방식
lineOrder	스캔라인 순서 (increasing Y, decreasing X)
screenWindowWidth, screenWindowCenter	영상을 생성하는 원근투영을 표현

은 헤더에 포함된 몇 가지 정의된 속성을 나타낸다. 또한 헤더에 임의의 이름과 타입을 가지는 속성을 추가할 수도 있다.

V. 압축

OpenEXR은 3가지의 다른 데이터 압축 방식을 제공한다. 3가지 방식은 모두 무손실 압축 기법이다. 즉, 압축 후에도 픽셀 데이터 정보는 변경되지 않는다(<표 4> 참조).

VI. OpenEXR API: IlmImf

OpenEXR의 IlmImf 라이브러리를 이용한 Open-

<표 4> OpenEXR의 지원 압축 방식

방식	설명
PIZ	픽셀 데이터에 Wavelet Transform이 적용되고 결과는 Huffman 인코딩 된다. 이 기법은 ILM에서 전형적으로 사용되는 영상 타입에 대해서 가장 좋은 압축률을 제공한다. Film grain을 가지는 포토그래픽 영상에 대해서 파일은 비압축시 보다 35%에서 55% 정도 감소된다.
ZIP	수평적으로 인접한 픽셀들간에 차이가 zlib 압축 라이브러리를 사용하여 압축된다. ZIP 복원(decompression)은 PIZ 복원보다 더 빠르지만 ZIP 압축은 상당히 느리다. 포토그래픽 영상에 대해서 45%에서 55%가 감소하는 경향이 있다.
RLE	수평적으로 인접한 픽셀들간의 차이를 Run-length 인코딩 한다. 이 방식은 빠르고 large flat 영역을 가지는 영상에 대해서 잘 동작한다. 하지만 포토그래픽 영상에 대해서는 비압축에 비해서 25%에서 40%만 감소된다.

EXR 파일을 읽고 저장하는 방법에 대해서 설명한다. IlmImf 라이브러리는 2가지 방식의 읽기/저장 방식을 지원한다. 한 가지 방식은 InputFile, OutputFile 클래스에서 구현된 일반적인 인터페이스를 이용한 것으로 임의 채널에 대한 접근을 지원한다. 두번째 방식은 Rgba-only 인터페이스를 이용한 것으로 RgbaInputFile과 RgbaOutputFile 클래스에서 구현되고 사용하기에 더 쉽다. 하지만 16bits(half) RGBA 채널에 대한 접근에 한계를 가진다.

1. 일반적 인터페이스 사용

가. 파일 저장

<표 5>는 각각 HALF, FLOAT 데이터 타입의 G, Z 이름을 가지는 2 채널과 width, height 크기의 픽셀 데이터를 가지는 OpenEXR 파일을 저장하는 예를 나타낸다.

라인 7에서 OpenEXR 헤더를 생성한다. 이때 헤더의 디스플레이 윈도와 데이터 윈도는 모두 (0, 0), (width-1, height-1)로 설정된다. 라인 8, 9는 영상 채널의 데이터 타입을 명시한다. 라인 10에서 명시된 이름과 헤더를 가지는 OutputFile 객체를 생성하고 라인 11에서 22까지는 OutputFile 객체가 영상 채널을 위한 픽셀 데이터를 어떻게 메모리에 배치하는지를 보여준다. 라인 11에서 FrameBuffer 객체

<표 5> 일반적 인터페이스를 이용한 파일 저장

```
1  void writeGZ1 (const char fileName[],
2                 const half *gPixels,
3                 const float *zPixels,
4                 int width,
5                 int height)
6  {
7  Header header (width, height);
8  header.channels().insert ("G", Channel (HALF));
9  header.channels().insert ("Z", Channel (FLOAT));
10 OutputFile file (fileName, header);
11 FrameBuffer frameBuffer;
12 frameBuffer.insert ("G",                  // name
13     Slice (HALF,                         // type
14         (char*) gPixels,                 // base
15         sizeof (*gPixels) * 1,           // xStride
16         sizeof (*gPixels) * width));     // yStride
17 frameBuffer.insert ("Z",                  // name
18     Slice (FLOAT,                        // type
19         (char*) zPixels,                 // base
20         sizeof (*zPixels) * 1,           // xStride
21         sizeof (*zPixels) * width));     // yStride
22 file.setFrameBuffer (frameBuffer);
23 file.writePixels (height);
   }
```

가 생성된 후, 각 영상 파일의 채널을 위해 Slice가 추가된다. Slice는 한 채널의 메모리 레이아웃을 나타내며 Slice 객체의 생성자는 type, base, xStride, yStride의 4개의 인자를 가진다. type은 HALF, FLOAT, UNIT와 같은 픽셀 데이터 타입을 나타내고 다른 3개의 인자는 (1)과 같이 픽셀 (x, y)의 메모리 주소를 정의하는 데 사용된다.

$$(x, y) = base + x \times xStride + y \times yStride \quad (1)$$

<표 5>에서 (x, y)의 G 채널에 대한 주소는 IlmImf 라이브러리에 의해서 (2)와 같이 계산될 수 있다.

$$\begin{aligned}(x, y) &= (half*)((char*)gPixels \\ &\quad + x \times sizeof(half) \times 1 \\ &\quad + y \times sizeof(half) \times width) \\ &= (half*)((char*)gPixels \\ &\quad + x \times 2 + y \times 2 \times width)\end{aligned} \quad (2)$$

(x, y)의 Z 채널에 대한 주소는 (3)과 같이 계산될 것이다.

$$\begin{aligned}(x, y) &= (float*)((char*)zPixels \\ &\quad + x \times sizeof(float) \times 1 \\ &\quad + y \times sizeof(float) \times width) \\ &= (float*)((char*)zPixels \\ &\quad + x \times 4 + y \times 4 \times width)\end{aligned} \quad (3)$$

라인 23의 writePixels()는 메모리로부터 파일로 영상의 픽셀을 복사한다. writePixels()의 인자는 얼마나 많은 스캔라인이 파일로 복사될 것인지를 알려주며 만일 영상 파일이 FrameBuffer 객체가 대응하는 Slice를 가지지 않는 채널을 포함한다면, 파일 내 그 채널의 픽셀은 0의 값으로 채워진다. 만일, FrameBuffer 객체가 채널을 가지지 않는 파일에 대한 Slice를 포함하면, 그 Slice는 무시된다. OutputFile을 명시적으로 닫는 함수는 없으며, 함수 writeGZ1()에서 빠져나가면 로컬 OutputFile 객체는 사라지고 파일은 닫히게 된다.

나. 파일 읽기

<표 6>은 HALF 타입의 R, G 채널과 FLOAT 타입의 Z 채널을 가지는 OpenEXR 파일 데이터를 읽기 위한 예를 보여준다. 채널 중 하나가 파일에 존재하지 않으면, 대응하는 메모리 버퍼가 적절한 기본 값으로 채워진다.

라인 7에서 우선 InputFile 객체를 생성함으로써 명시된 이름을 가지는 파일을 연다. 라인 8부터 13에서 Array2D 클래스를 사용하여 영상의 R, G, Z 채널을 위한 메모리 버퍼를 할당한다. 이 버퍼는 파일의 데이터 윈도 안의 모든 픽셀을 가질 수 있는 크기로 할당한다. 이제 라인 14에서 FrameBuffer 객체를 생성한다. 이 객체는 IlmImf 라이브러리에 앞서 생성된 버퍼를 알려준다. 각 영상 채널에 대해서 FrameBuffer에 Slice를 추가한다. R 채널에 대해서 픽셀 (dw.min.x, dw.min.y)는 &rPixels[0][0]의 주소를 가진다. 각 Slice는 fillValue를 가지는데 만약 영상 파일이 Slice를 위한 채널을 포함하지 않으면 대응하는 메모리 버퍼가 fillValue로 채워진다. Slice의 다른 두 인자인 xSampling과 ySampling은 YUV 비디오

데이터와 같이 채널의 일부가 서브 샘플링되는 영상을 위해서 사용된다. 만일 영상이 서브 샘플링 채널을 포함하지 않으면 xSampling과 ySampling은 1로 설정된다. 자세한 사항은 ImfFrameBuffer.h와 ImfChannelList.h[4]를 참조한다. 메모리 버퍼의 레이아웃을 표현한 후, readPixels()를 호출하면 파일로부터 버퍼로 픽셀 데이터가 복사된다. readPixels()는 파일 내의 스캔라인에 대한 랜덤 접근을 가능하게 한다.

<표 6> 일반적 인터페이스를 이용한 파일 읽기

```
1  void readGZ1 (const char fileName[],
2                Array2D<half>  &rPixels,
3                Array2D<half>  &gPixels,
4                Array2D<float> &zPixels,
5                int &width, int &height)
6  {
7  InputFile file (fileName);
8  Box2i dw = file.header().dataWindow();
9  width = dw.max.x - dw.min.x + 1;
10 height = dw.max.y - dw.min.y + 1;
11 rPixels.resizeErase (height, width);
12 gPixels.resizeErase (height, width);
13 zPixels.resizeErase (height, width);
14 FrameBuffer frameBuffer;
15 frameBuffer.insert ("R",                // name
16    Slice (HALF,                        // type
17    (char*)(&rPixels[0][0] - dw.min.x -
         dw.min.y * width),               // base
18    sizeof(rPixels[0][0]) * 1,          // xStride
19    sizeof (rPixels[0][0]) * width,     // yStride
20    1, 1,                               // x, y sampling
21    0.0));                              // fillValue
22 frameBuffer.insert ("G",                // name
23  Slice (HALF,                          // type
24    (char*)(&gPixels[0][0]- dw.min.x -
         dw.min.y * width),               // base
25  sizeof (gPixels[0][0]) * 1,           // xStride
26  sizeof (gPixels[0][0]) * width,       // yStride
27     1, 1,                              // x, y sampling
28     0.0));                             // fillValue
29 frameBuffer.insert ("Z",                // name
30  Slice (FLOAT,                         // type
31    (char*)(&zPixels[0][0]- dw.min.x -
         dw.min.y * width),               // base
32  sizeof (zPixels[0][0]) * 1,           // xStride
33  sizeof (zPixels[0][0]) * width,       // yStride
34    1, 1,                               // x, y sampling
35    FLT_MAX));                          // fillValue
36 file.setFrameBuffer (frameBuffer);
37 file.readPixels (dw.min.y, dw.max.y);
   }
```

다. 프레임 버퍼에 채널 인터리빙

이것은 위에서 설명한 나.의 다른 방법이 될 수 있다. 영상 파일을 읽을 때 각 채널을 분리된 메모리 버퍼에 저장하는 대신에 하나의 버퍼에 인터리빙 할 수 있다. 이 버퍼는 구조체 배열이며, 다음과 같이 정의된다.

```
typedef struct GZ {
    half g;
    float z;
}
```

<표 7>은 3개의 채널 대신에 단지 2개의 채널만을 읽고 있다는 것을 제외하고는 <표 6>의 코드와 거의 동일하다. 유일한 차이는 FrameBuffer 객체의 Slice를 위한 base, xStride, yStride를 계산하는 방법이 다르다는 것이다.

<표 7> 채널 인터리빙을 이용한 파일 읽기

```
1  void readGZ2 (const char fileName[],
2                Array2D<GZ> &gzPixels,
3                 int &width, int &height)
4  {
5  InputFile file (fileName);
6  Box2i dw = file.header().dataWindow();
7  width  = dw.max.x - dw.min.x + 1;
8  height = dw.max.y - dw.min.y + 1;
9  gzPixels.resizeErase (height, width);
10 FrameBuffer frameBuffer;
11 char *base = (char*) (&gzPixels[0][0] -
     dw.min.x - dw.min.y * width);
12 int xStride = sizeof (gzPixels[0][0]) * 1;
13 int yStride = sizeof (gzPixels[0][0]) * width;
   frameBuffer.insert ("G",
14                Slice (HALF,
15                       base +  offsetof (GZ, g),
16                       xStride,
17                       yStride));
18 frameBuffer.insert ("Z",
19                Slice (FLOAT,
20                        base +  offsetof (GZ, z),
21                        xStride,
22                        yStride));
23 file.setFrameBuffer (frameBuffer);
24 file.readPixels (dw.min.y, dw.max.y);
25 }
```

2. RGB-only 인터페이스 사용

가. RGBA 파일 저장

RGBA 영상 파일을 저장하는 방법은 매우 간단하다. <표 8>의 라인 6에서 RgbaOutputFile 객체의 생성은 OpenEXR 헤더를 생성하여 명시된 이름을

가지고 파일을 열고 헤더의 속성을 설정한다. 그리고 파일 안에 생성된 헤더를 저장한다. 헤더의 디스플레이 윈도와 데이터 윈도는 모두 (0, 0), (width-1, height-1)로 설정된다. 채널 리스트는 HALF 타입의 R, G, B, A의 4개 채널을 포함한다. 라인 7은 어떻게 픽셀 데이터가 메모리에 위치될 것인지를 나타낸다. <표 8>에서 포인터 pixels는 width*height 크기 픽셀 배열의 시작 위치를 나타내며 다음과 같은 Rgba 구조로 표현된다.

```
struct Rgba {
   half r;  // red
   half g;  // green
   half b;  // blue
   half a;  // alpha (1-opacity)
}
```

라인 7의 setFrameBuffer()는 3가지 인자를 가지고 있다. 픽셀 (x, y)의 주소를 결정하기 위해서 RgbaOutputFile 객체는 (1)과 같은 수식을 사용한다. 이 경우에 base, xStride, yStride는 각각 pixels, 1, width로 설정된다. 따라서 (x, y)의 주소는 pixels + 1×x + width×y가 된다. 라인 8의 writePixels()는 메모리로부터 파일로 영상의 픽셀을 복사한다. writePixels()의 인자로 사용된 height는 얼마나 많은 스캔라인을 복사할 것인지 나타낸다. 마지막으로, writeRgba1() 함수를 빠져나갈 때 RgbaOuputFile 객체는 파괴되고 파일은 닫혀진다. 그렇다면 왜 writePixles()에 얼마나 많은 스캔라인이 복사될 것인지를 넘겨 주어야 할까? RgbaOutputFile 객체는 데이터 윈도로부터 스캔라인의 수를 알아낼 수 없기 때문일까? 그 이유는 IlmImf 라이브러리가 한 번의 writePixels() 호출로 모든 스캔라인을 파일에 복사하도록 하지 않기 때문이다. 많은 프로그램들이 영상 파일 내의 각 스캔라인을 개별적으로 또는 작은 블록 단위로 쓸 수 있도록 요구하고 있다. 이러한 방식은 렌더링 영상에서도 상당한 크기를 가지는 영상의 일부를 렌더링이 끝난 후 부분별로 즉각적으로 화면에 보여줄 수 있도록 하는 장점이 있다. IlmImf 라이브러리는 위에서 아래 또는 아래에서 위 방향으로 스캔라인을 저장할 수 있도록 하고 있다. 방향은 파일 헤더의 lineOrder 속성 INCREASING_Y, DECREASING_Y에 의해서 정해진다. 기본값으로 스캔라인은 위에서 아래 방향을 향하는 INCREASING_Y 값이다. 위와 같은 예에서 주목할 것은 파일을 작성하는 작업이 성공했는지에 대해서 확인하는 명시적인 검사가 수행되지 않는다는 것이다. 만일, IlmImf 라이브러리가 에러를 발견하면 C-스타일의 에러 코드를 되돌리는 대신에 C++ 예외 처리를 수행한다. 예를 들면, writeRgba1()을 호출하는 프로그램은 다음과 같은 하나의 try/catch 블록을 이용하 여 모든 가능한 에러 상황을 다룰 수 있다.

```
try {
writeRgba1 (fileName, pixels, width, height);
} catch (const std::exception &exc) {
std::cerr << exc.what() << std::endl;
}
```

<표 8> RGBA-only 인터페이스를 이용한 파일 저장

```
1  void writeRgba1 (const char fileName[],
2                   const Rgba *pixels,
3                   int width,
4                   int height)
5  {
6  RgbaOutputFile file (fileName, width,
     height, WRITE_RGBA);
7  file.setFrameBuffer (pixels, 1, width);
8  file.writePixels (height);
   }
```

나. 사용자 정의(Custom) 속성 저장

OpenEXR의 특징에서 설명한 바와 같이 OpenEXR은 파일 헤더에 속성을 추가할 수 있다. <표 9>는 comment라는 이름의 스트링 속성과 cameraTransform이라는 이름의 4×4 매트릭스 속성을 추가한 경우의 예를 보여준다.

라인 11의 RgbaOutputFile 객체는 <표 8>과 다르다. <표 8>에서는 암시적으로 헤더가 생성되어 파일에 저장되었다. <표 9>에서는 명시적으로 헤더를 생성하고 새로운 속성을 추가한다. 라인 11에서 RgbaOutputFile 객체는 자신의 헤더를 생성하는 대신에 주어진 헤더 인자를 사용하게 된다.

<표 9> RGBA-only 인터페이스를 이용한 헤더 속성 추가

```
1  void writeRgba3 (const char fileName[],
2          const Rgba *pixels,
3          int width,
4          int height,
5          const char comment[],
6          const M44f &cameraTransform)
7  {
8  Header header (width, height);
9  header.insert ("comment", StringAttribute
     (comment));
10 header.insert ("cameraTransform",
     M44fAttribute (cameraTransform));
11 RgbaOutputFile file (fileName, header,
     WRITE_RGBA);
12 file.setFrameBuffer (pixels, 1, width);
13 file.writePixels (height);
   }
```

다. RGBA 파일 읽기

RGBA 영상 파일을 읽는 것은 저장하는 것만큼이나 쉽다. 주어진 파일 이름을 가지고 RgbaInputFile 객체를 생성하면 파일을 열고 헤더를 읽게 된다. 라인 7에서 RgbaInputFile 객체에 파일의 데이터 윈도를 요청한 후, 라인 10에서와 같이 픽셀 데이터를 위한 버퍼를 할당한다. 버퍼 안의 스캔라인 수는 데이터 윈도의 height와 동일하고 스캔라인 당 픽셀의 수는 width와 동일하다(<표 10> 참조).

RgbaOutputFile의 writePixels()와 달리 readPixels()는 readPixles(y1, y2)의 형태로 2개의 인자를 가진다. y1으로부터 y2까지의 y좌표를 가지는 모든 스캔라인에 대한 픽셀을 프레임 버퍼로 복사하게 된다. 이러한 방식은 어떤 순서로의 스캔라인에 대한 접근도 허용한다. 영상은 한 번에 모두 읽혀질 수 있고 또는 한 번에 한 스캔라인씩 또는 몇 개의 스캔라인으로 구성된 블록 단위로 읽혀질 수 있다. 하지만 이러한 랜덤 액세스가 가능할 지라도 저장될 때와 동일한 순서로 읽혀지는 것이 좀더 효과적이다. 랜덤 액세스는 seek 기능을 요구하기 때문에 느려질 수 밖에 없다.

<표 10> RGBA-only 인터페이스를 이용한 파일 읽기

```
1  void readRgba1 (const char fileName[],
2                  Array2D<Rgba> &pixels,
3                  int &width,
4                  int &height)
5  {
6  RgbaInputFile file (fileName);
7  Box2i dw = file.dataWindow();
8  width  = dw.max.x - dw.min.x + 1;
9  height = dw.max.y - dw.min.y + 1;
10 pixels.resizeErase (height, width);
11 file.setFrameBuffer (&pixels[0][0] -
     dw.min.x - dw.min.y * width, 1, width);
12 file.readPixels (dw.min.y, dw.max.y);
   }
```

라. 사용자 정의 속성 읽기

일반적으로 RgbaInputFile 객체를 생성함으로써 파일을 연다. findTypedAttribute <T>(n)의 호출은 타입 T와 이름 n을 가지는 속성에 대한 헤더를 찾아내어 포인터를 반환한다. 이때 헤더가 이름 n을 가지는 속성을 포함하고 있지 않거나 또는 n의 속성이 T가 아니면 0를 되돌린다. 일단 원하는 속성의 포인터를 찾으면 라인 9와 라인 11과 같이 n->value()를 이용하여 접근할 수 있다(<표 11> 참조).

여기서 명시적으로 0 포인터에 대한 검사를 수행함으로써 존재하지 않는 속성에 대한 가능성을 다루어 볼 수 있다. fineTypedAttribute()의 다른 형태인 typedAttribute()는 역시 이름과 타입을 가지고 속성을 검색한다. 그러나 만약 요청된 속성이 존재하지 않으면 0을 반환하는 것이 아니라 예외 처리를 수행하게 된다. 즉 아래와 같은 사용이 가능하다.

<표 11> RGBA-only 인터페이스를 이용한 OpenEXR 헤더에 추가된 속성 읽기

```
1  void readHeader (const char fileName[])
2  {
3  RgbaInputFile file (fileName);
4  const StringAttribute *comment =
     file.header().findTypedAttribute <String -
     Attribute> ("comment");
5  const M44fAttribute *cameraTransform =
     file.header().findTypedAttribute
     <M44fAttribute> ("cameraTransform");
6  if (comment)
7    cout << "commentWn"   << comment
      ->value() << endl;
8  if (cameraTransform)
9    cout << "cameraTransformWn" << camera-
     Transform->value() << flush;
   }
```

```
void readComment (const char fileName[],
                  string &comment)
{
RgbaInputFile file (fileName);
comment =
   file.header().typedAttribute<StringAttribute>
   ("comment").value();
}
```

VII. OpenEXR 관련 소프트웨어

1. Viewer: exrdisplay

exrdisplay는 OpenEXR 공식 사이트[1]에서 제공하는 간단한 OpenEXR 파일 뷰어이다. 이 프로그램을 이용하면, 영상 내의 16bits FP 픽셀 데이터를 프레임 버퍼에 8bits 정수형 데이터로 변환하는 방식으로 영상 데이터에 대한 변경 없이 노출을 변경시킬 수 있다. 이때, 16bits FP 픽셀을 "raw" 데이터라 하고 8bits 픽셀 데이터를 "cooked" 데이터라p고 한다. (그림 4)는 exrdisplay를 이용한 Open-EXR 파일의 뷰잉과 노출 변화를 보여준다. 오른쪽의 그림은 왼쪽보다 노출을 증가시킨 결과이다.

(그림 4) exrdisplay를 이용한 뷰잉과 노출 변화

2. Bitmap I/O Plugin for 3ds Max

Splutterfish는 3ds Max를 위한 HDRI용 I/O plugin을 제작하였으며, 또한 OpenEXR용 I/O plugin[6]을 개발하여 제공하고 있다((그림 5) 참조). Splutterfish에서 제공하는 plugin은 OpenEXR이 제공하는 유연성과 확장성을 이용하여 표준 포맷의 범위를 약간 확장한 것이다. 예를 들면, OpenEXR API 자체가 full-latitude 32bits FP RGBA 파일을 지원하고 있지만 대부분의 OpenEXR 소프트웨어가 단지 16bits의 half FP RGBA 파일만을 읽을 수 있도록 지원하고 있다. OpenEXR 패키지에 포함되어 있는 exrdisplay 조차도 half FP 타입을 사용하여 인코딩되지 않은 RGBA 데이터는 인식하지 못하기도 한다. 하지만 Splutterfish에서 제공하는 plugin은 full-latitude 32bits FP RGBA 파일을 지원한다.

(a) OpenEXR plugin의 파일 열기 인터페이스

(b) OpenEXR plugin의 파일 저장 인터페이스

(그림 5) Splutterfish의 OpenEXR I/O Plugin

3. Plugin for PhotoShop

OpenEXR 공식 사이트에서는 OpenEXR 파일을 읽고 저장할 수 있는 Max OS 9.2.2, Mac OS X 10.2.4, Windows 2000용 Adobe PhotoShop7.0 plugin을 함께 제공한다. 이 plugin은 비공식적으로 Windows 98, Mac OS 8.1, PhotoShop 5.5버전 이상에서 잘 작동하는 것으로 알려져 있다.

4. Plugin from eyeon Software

eyeon Software[7]는 2003년 1월 2D 데스크톱 영상 합성 툴인 Digital Fusion 4에서 OpenEXR 포맷의 지원을 공식 발표하였으며 이는 ILM 외부에서 개발된 애플리케이션 중에서는 OpenEXR 파일 포맷을 지원하는 첫번째 시도였다((그림 6) 참조).

(그림 6) Digital Fusion 4에서의 OpenEXR 사용

5. Applications: exrtools

exrtools[8]는 OpenEXR 파일 포맷의 영상을 다루는 데 필요한 여러 가지 커맨드 라인 유틸리티들의 모음이다. exrtools에는 톤 매핑(tone mapping)[3]에서부터 블러링(blurring), 영상 포맷 변환 등과 같은 유틸리티가 포함되어 있으며 소스 코드도 공개되어 있다. <표 12>는 exrtools에 포함된 각각의 애플리케이션과 그 기능을 나타낸다.

<표 12> exrtools의 Applications

Applications	기능
exrblur	Gaussian blur를 적용
exrchr	Spatially-varying chromatic adaptation을 적용
exrcamtm	iCAM 기법을 이용한 톤 매핑
exrnlm	Non-linear masking correction 수행
exrnormalize	Normalize
exrpptm	Photoreceptor physiology 기법을 이용한 톤 매핑
exrstats	Statistics 디스플레이
exrtopng	PNG 영상으로 포맷 변환
jpegtoexr	JPEG 영상으로부터 포맷 변환
pngtoexr	PNG 영상으로부터 포맷 변환
ppmtoexr	PPM 영상으로부터 포맷 변환

VIII. 결론 및 전망

지금까지 HDRI 포맷인 OpenEXR 파일 포맷에 대하여 살펴보았다. 앞서 설명한 바와 같이 ILM은 그들의 VFX 작업 과정에서 이미 OpenEXR 파일 포맷을 사용해 왔으며 그 결과에서 이 포맷의 사용이 얼마나 효과적이었는지 보여 주었다. OpenEXR 포맷이 가지는 가장 큰 장점은 무손실 압축방식에 있다고 할 수 있다. 또한 C++ 클래스 기반의 유연한 확장성의 제공과 소스 공개는 개발자들로 하여금 손쉽게 이 포맷에 대해서 접근하고 이용할 수 있도록 하였다. 더불어 비주얼 이펙트 분야의 선두주자인 ILM에서 개발되었고 지금도 계속 사용되고 있다는 점과 NVIDIA의 Cg에서 사용되는 half 데이터 타입으로 표현될 수 있도록 한 전략도 장점이라고 볼 수 있겠다.

앞으로는 Splutterfish의 3ds Max 용 OpenEXR I/O 지원 plugin 개발에서 예상할 수 있듯이 기존의 HDRI 포맷을 지원하던 다른 CG 소프트웨어에도 이 새로운 포맷을 지원하는 기능들이 추가될 것으로 예상된다.

참 고 문 헌

[1] http://www.openEXR.com
[2] http://www.ilm.com/
[3] 김성예, 최병태, "High Dynamic Range Image 기술동향," ETRI 주간기술동향, 통권 1065호, 2002, pp.1-15.
[4] http://savannah.nongnu.org/projects/openEXR/
[5] http://developer.nvidia.com/page/cg_main.html
[6] http://www.splutterfish.com/sf/sf_gen_page.php3?page=plugins
[7] http://www.eyeonline.com
[8] http://scanline.ca/exrtools