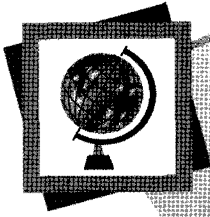
| 특집 03 |



3차원 렌더링 : 기술적 이슈를 중심으로

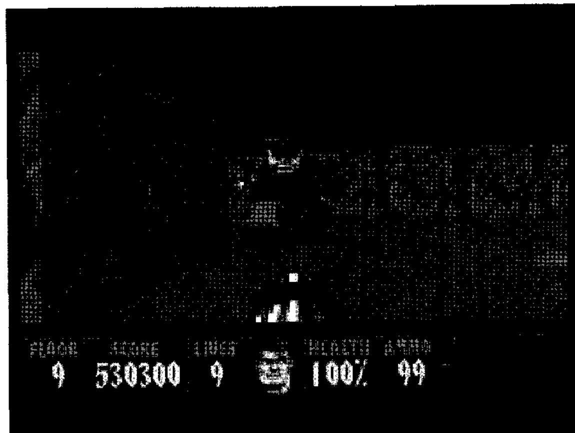
김성용 (제이씨엔터테인먼트)
한정현 (고려대학교)

목 차

1. 렌더링 엔진
2. 그래픽스 API 및 하드웨어
3. 기술 사례 : 픽셀별 라이팅
4. 렌더링 : 기능적 요소
5. 결 론

1. 렌더링 엔진

1992년 id Software가 최초의 3차원 게임인 Wolfenstein 3D(그림 1)를 선보인 이래, 3차원 게임은 게임 시장에서 주류의 자리를 차지하게 되었다. Playstation과 Xbox 시리즈로 대표되는 콘솔, 그리고 PC 및 오락실용 아케이드 게임기 등 고사양 게임하드웨어 플랫폼은 물론이고, 최근에는 상대적으로 저사양인 휴대폰과 같은 모바일 하드웨어 플랫폼에서도 3차원 게임이 선보이고 있다. 게임 장르 별로 보아도 유아용 게임을 제외하고는 현재 대부분의 게임이 3차원으로 제작된다고 보아도 무방할 것이다. 이 같은 시장 수요에 부응하여 다수의 3차원 렌더링 엔진이 개발되었는데, 대표적인 콘솔 및 PC 게임엔진으로 id Software의 Doom III[1], Criterion의



(그림 1) Wolfenstein 3D : 스크린 샷

(그림 2) Unreal Engine III : 스크린 샷

Renderware[2], Epic Games의 Unreal II 및 III[3], Intrinsic의 Alchemy[4], NDL의 GameBryo[5] 등이 있다. (그림 2)는 2004년 봄에 발표된 Unreal Engine III의 스크린샷으로, 최신의 컴퓨터 그래픽스 기술을 총동원하여 매우 인상적인 렌더링 결과를 보여주었다.

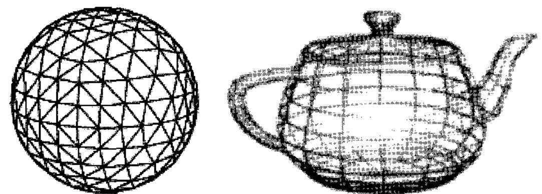
2. 그래픽스 API 및 하드웨어

렌더링 엔진의 임무는 실시간에 3차원 환경 및 물체를 그려주는 것으로, 이는 OpenGL[6] 또는 DirectX[7]와 같은 API에 기반해 있다. Silicon Graphics 사 주도의 OpenGL Architecture Review Board (ARB)에 의해 1992년 version 1.0이 발표된 OpenGL은, 1998년의 version 1.2 등을 거쳐, 2004년 9월에 version 2.0이 발표되었다. OpenGL은 게임 전용이라기보다는 범용적인 그래픽스 API이다. 하지만, id Software의 Quake 3 등 호평을 받는 다수의 게임이 OpenGL을 이용해 개발되었다.

윈도우 플랫폼 전용으로 개발된 마이크로소프트의 DirectX는 렌더링 기능을 담당하는 DirectX Graphics 이외에 DirectInput, DirectSound 등 게임 제작에 필요한 다종다양한 API를 제공하여, 이른바 멀티미디어 API로 불리며 실제 게임 개발에 있어 OpenGL보다 많은 사용자 층을 확보하고 있는 사실상의 표준이다. 1995년 처음 발표된 이래 빠른 속도로 후속 버전이 개발되어, 2004년 8월에 version 9.0c가 발표되었다.

OpenGL과 DirectX는 모두 그래픽스 가속 하드웨어의 기능을 응용 프로그램에게 제공하는 API이다. 현재 3차원 게임이 대중화된 것은 그래픽스 하드웨어 가속기 또는 GPU(Graphic Processor Unit)의 발달에 전적으로 기인한다. 1996년에 발매된 3dfx사의 Voodoo 칩을 효시로 열리게 된 PC용 3차원 그래픽스 카드 시장은 NVIDIA[8]와 ATI[9]사가 경쟁적으로 출시한 저가 고성능 칩에 의해 그 규모가 폭발적으로 증가하여 왔다 (<표 1> 참조).

GPU의 기본적인 임무는 렌더링 파이프라인을 담당하는 것이다. 렌더링 파이프라인은 (그림 3)과 같은 다면체 (폴리곤 메시)를 입력 받아, 폴리곤의 버텍스를 처리하는 단계(vertex processing)와 최종 화면을 구성하는 화소를 생성하는 픽셀 처리 (pixel processing) 단계로 구



(그림 3) 폴리곤(다각형) 메시 예

분된다. 초기의 PC 그래픽스 카드는 픽셀 처리 기능만을 가지고 있었으나, 점차 버텍스를 기하변환하고 여기에 라이팅을 계산하는(transformation & lighting : T&L) 버텍스 처리 기능까지 포괄하게 된다. 그러나, 이렇게 진화된 렌더링 파이프라인은 고정된 (fixed) 기능을 수행하므로, 게임 프로그래머가 표현할 수 있는 범위는 하드웨어적으로 고정된 방식에 제약될 수밖에 없었다. 예를 들어, 광원(light source)은 point, directional, spot 세 가지 중에서 하나를 선택해야 하고, 정반사(specular reflection) 계산에 있어서는 이른바 Blinn-Phong 모델을 사용할 수밖에 없었으며, 이러한 방식으로 폴리곤 버텍스마다 계산된 색상 혹은 라이팅 값을 보간하여 폴리곤 내부를 채우는 이른바 버텍스별 라이팅 기법을 받아들일 수밖에 없었다. 게임 프로그래머가 Blinn-Phong 모델 대신 다른 기법을 사용하고자 할 경우는, 그래픽스 카드가 제공하는 고성능의 파이프라인을 포기하고, 이를 소프트웨어적으로 구현할 수밖에 없었다.

〈표 1〉 PC 그래픽스 카드

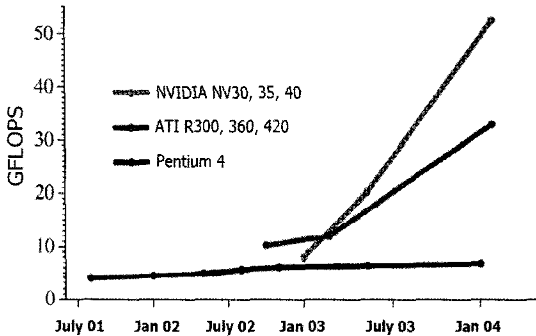
	주요 그래픽스 카드	특징
1996	3dfx Voodoo I	Triangle setup & clipping
1997	NVIDIA Riva128	Competing with Voodoo I (relatively high performance and low cost)
1998	NVIDIA RivaTNT 3dfx Voodoo II ATi Rage128	Hot year for gaming hardware, and also for game development (Quake2, Starcraft, Unreal, etc.)
1999	NVIDIA GeForce 256	Hardware transformation & lighting
2000	ATi Radeon NVIDIA GeForce2 GTS	Competition of the two major companies, NVIDIA and ATi
2001	NVIDIA GeForce3 (NVIDIA GeForce2 MX) ATi Radeon 8500	Vertex and pixel shader support of DirectX 8.1
2002	Radeon 9700 (Radeon 9000 NVIDIA GeForce4 Ti, MX)	DirectX HLSL, occlusion query, etc.
2003	NVIDIA GeForce FX 5950U ATi Radeon 9800 XT	DirectX 9.0 OpenGL Shading Language
2004	NVDIA GeForce 6800 ATi Radeon X800 XT	DirectX Shader 3.0 (NVIDIA Geforce 6800 only) OpenGL 2.0
2005 ~	NVDIA GeForce 7800 ATi Radeon X1800 XL	64bit texture flitering and blending

이 같은 파이프라인의 고정된 기능이 가지는 문제를 극복하고자 제안된 것이 프로그래밍 가능한 (programmable) 파이프라인이고, 이를 구현한 것이 바로 쉐이더(shader)이다. 쉐이더 프로그램은 GPU에 대한 명령의 집합이다. 즉, GPU 명령어를 이용해 본인이 원하는 렌더링 방식을 직접 코딩하는 것이다. 전술한 바와 같이 렌더링 파이프라인은 버텍스 처리와 픽셀 처리로 구분되므로, 쉐이더 프로그램도 버텍스 처리를 담당하는 버텍스 쉐이더와, 픽셀 처리를 담당하는 픽셀 쉐이더로 구분된다(이는 DirectX 용어이고, OpenGL에서는 vertex program과 fragment program이라는 용어를 사용한다).

DirectX 8.0의 경우, 쉐이더 프로그램은 어셈블리어로 코딩해야 했다. 그러나, DirectX 9.0에서 C언어 스타일의 High Level Shading Language (HLSL)[7]가 소개되었다. OpenGL 역시, 2003년 발표된 version 1.5 및 2004년 발표된 version 2.0에서 'OpenGL Shading Language (GLSL)'에서 로우레벨 및 하이레벨 언어를 모두 지원한다. 한편, NVIDIA에서도 고급언어인 Cg(C for graphics)[10]를 발표했다. 이같은 고급 언어로 작성된 쉐이더 프로그램은 기계어로 컴파일되어 해당 GPU에서 실행된다.

쉐이더 프로그램은 사실감 있는 영상의 실시간 렌더링에 커다란 기여를 하고 있다. Doom III에서의 범프 매핑, Dead or Alive Xtreme (DOAX) Beach Volleyball 게임의 스키닝 애니메이션, 최근 관심을 끌고 있는 High Dynamic Range (HDR) 렌더링 및 카툰 렌더링 기법들은 모두 쉐이더 프로그램의 결과물이다. 또한, Unreal Engine III 게임엔진도 DirectX 9.0과 Shader 3.0을 이용하여 매우 사실적인 영상을 실시간에 렌더링할 수 있음을 보여주었다.

프로그래밍이 가능하다는 점과 더불어 현대의 GPU를 각광받게 하는 또 하나의 요소는 그 뛰어난 성능에 있다. 최신의 자료는 아니지만, (그림 4)는 CPU와 GPU의 성능이 얼마만한 차이를 유지하면서 발전해 왔는지를 극명하게 보여준다. 현대 데스크탑에서의 현란한 3차원 세계는 바로 이런 GPU 성능향상의 결과물이다.



(그림 4) CPU 와 GPU 성능 비교

한편, 3차원 그래픽스 영역을 넘어, GPU를 이용하여 영상처리는 물론 공학 제 분야의 문제를 해결하고자 하는 시도들도 있고, 이를 일컬어 GPGPU (General Purpose GPU) 연구라 한다.

본 원고의 3절에서는 그래픽스 하드웨어 발전에 따라 게임에서 어떠한 렌더링 기술이 채택되어 왔는지를 보여주기 위하여, 렌더링 품질을 높이는 데에 핵심적인 역할을 하는 픽셀별 라이팅 기술에 대해서 소개한다. 한편 4절에서는 일반적인 렌더링 엔진이 제공하는 기능적인 요소에 대해 몇 가지 예를 중심으로 기술한다. 3절 및 4절의 내용을 이해하기 위해서는 3차원 그래픽스에 대해 약간의 지식이 요구될 것이다.

3. 기술 사례 : 픽셀별 라이팅

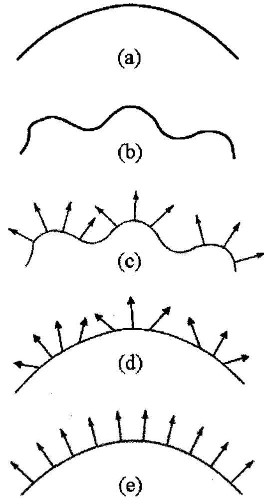
그래픽스 하드웨어의 성능이 발전함에 따라서 게임에 사용되는 기술도 변화 발전하고 렌더링 품질을 높이기 위한 많은 새로운 방법들이 제안되고 있다. 라이팅(lighting)은 광원과 모델의 상호작용에 의해 결정되는 픽셀의 색상을 결정하는 작업이다. 게임에서 어떤 라이팅 방법을 사용하느냐에 따라서 렌더링의 품질이 좌우된다고 볼 수 있을 만큼, 라이팅은 3D 게임에서 매우 중요한 이슈이다. DOOM3나 Half-Life2, FarCry와 같은 최신의 게임들을 보면, 기존의 게임들에 비해서 배경이나 모델의 렌더링 품질이 매우 정교하고 사실적임을 알 수 있다. 이 절에서는 그러한 고품질 렌더링에 사용된 기술을 알아 볼 것인데, 이를 위해 실시간 라이팅에 널리 사용되는 Blinn-Phong 라이팅 식을 살펴보자.

$$I = I_a + k_d(N \cdot L) + k_s(N \cdot H)^\alpha$$

위 식은 폴리곤 메시 표면의 한 점에서의 라이팅 값을 계산하는 식인데, 이 식이 복잡하다고 느껴진다면, 라이팅 계산에 있어 주어진 점의 노멀 (법선) 벡터 N이 사용된다는 사실만 주목하면 된다. 실제로 라이팅 계산에 있어 가장 중요한 벡터가 이 노멀벡터 N과 광원의 방향을 나타내는 벡터 L인데, N과 L이 가까울수록 해당 점의 라이팅 값이 밝아지게 된다.

고정 파이프라인에서는 Blinn-Phong 라이팅 식을 버텍스마다 계산하여 라이팅 값을 얻은 뒤에, 이들을 보간하여 폴리곤을 구성하는 각 픽셀들의 라이팅 값을 결정한다. 이것이 바로 버텍스별 라이팅이다. 이 경우, 버텍스의 개수가 라이팅 품질에 직접적인 영향을 끼치게 된다. 물체 전체에 걸쳐 보다 사실적인 라이팅 결과를 얻기 위해서는 매우 세밀한 고해상도 모델을 사용해야 한다. 그러나, 게임의 성능을 위해서는 어느 정도 이상으로 폴리곤 수를 늘리는 것이 불가능하다.

최근 게임 기술 분야에서는 최신 그래픽스 하드웨어를 활용하는 픽셀별 라이팅 (per-pixel lighting) 기법이 널리 채택되고 있다. 즉, 쉐이더를 이용하여 "각 픽셀마다" 위의 Blinn-Phong 라이팅 식을 혹은 프로그래머가 원하는 특별한 라이팅 식을 직접 코딩해서 라이팅 값을 계산한다. 이러한 픽셀별 라이팅 기법을 사용하면 (그림 5) (a)와 같은 단순한 (적은 수의 폴리곤을 가진) 메시를 사용하면서도 마치 (그림 5) (b)와 같은 고해상도 모델을 렌더링한 것과 같은 정교하고 사실적인 렌더링 결과를 얻을 수 있다. 픽



(그림 5) 노멀 매핑

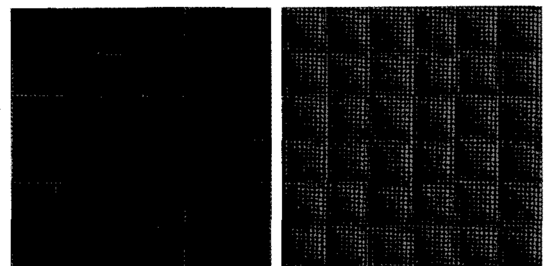
셀별 라이팅을 활용하는 가장 대표적인 기법인 범프 매핑(bump mapping) 혹은 노멀 매핑(normal mapping)을 통해 이를 알아보자.

(그림 5) (b)와 같은 물체의 표면에 빛을 비추어 보면 울퉁불퉁하게 (bumpy하게) 보일 것이다. 그 이유는 (그림 5) (c)에 보인 바와 같이 물체 표면 상에서의 노멀 벡터의 변화가 심하기 때문이다. Blinn-Phong 라이팅 식을 이용해 색상 값을 계산하는데, 노멀벡터가 결정적인 역할을 함을 상기하자. 노멀 매핑 기법이란, 렌더링의 효율을 위해 (그림 5) (a)와 같은 단순한 모델을 사용하되, (그림 5) (c)의 노멀벡터를 텍스처로 저장해 놓고 이를 렌더링 과정에서 꺼내어 사용하는 것이다. 즉, (그림 5) (d)에 도시된 바와 같은 모델 및 노멀벡터들을 사용하는 것이다. (그림 5) (a) 메시의 노멀벡터가 (그림 5) (e)에 도시되었는데, 노멀벡터의 변화가 완만함에 주목하라. 이들 벡터를 그대로 사용한다면 절대로 울퉁불퉁한 질감을 표현할 수 없다.

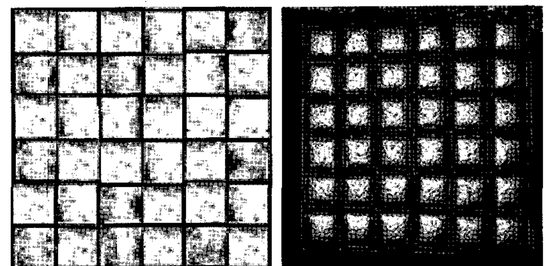
노멀 값들을 텍스처로 저장한 것이 바로 노멀맵(normal map)인데, 대개 노멀 벡터를 RGB 값으로 변환하여 이미지로 저장한다. 한편, 노멀맵을 생성하는 방법은 크게 두 가지로 나뉜다. 고해상도 모델을 일단 만들고 이로부터 노멀을 추출한 뒤, 실제 렌더링을 위해서는 고해상도 모델을 저해상도로 낮추어 사용한다. 또 하나의 방법으로는, 그래픽 아티스트가 모델에 입혀질 색상맵을 제공하면 이로부터 노멀벡터를 추출하기도 한다. (그림 6) (a)가 색상맵의 예이고, (그림 6) (b)는 이로부터 추출된 노멀맵을 도시화한 것이다(노멀백터의 xyz요소를 RGB 요소로 변환해 저장한다고 했음을 상기하자. 그러한 RGB 값을 그대로 읽어 들여 그린 것이 (그림 6) (b)이다).

한편, Blinn-Phong 라이팅 식의 마지막 항은 정반사 효과(specular lighting)를 위한 것인데, 이를 물체의 특정 부위로 한정시키기 위해 정반사 효과를 픽셀별로 마스킹하는 글로스맵(gloss map)이 사용되기도 한다(그림 6 (c) 참조). 노멀맵과 글로스맵을 사용하여 렌더링한 결과가 (그림 6) (d)에 도시되어 있다.

이러한 노멀매핑 기법은 컴퓨터 그래픽스 분야에서 오래 전에 소개되었지만, 이들이 게임에



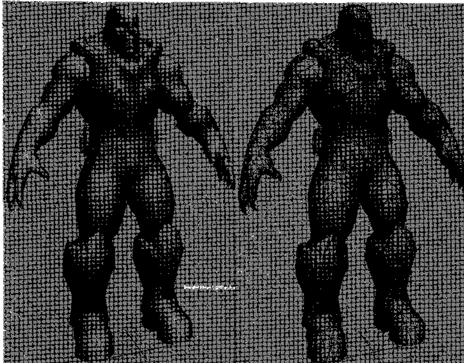
(a) 색상맵 b) 노말맵



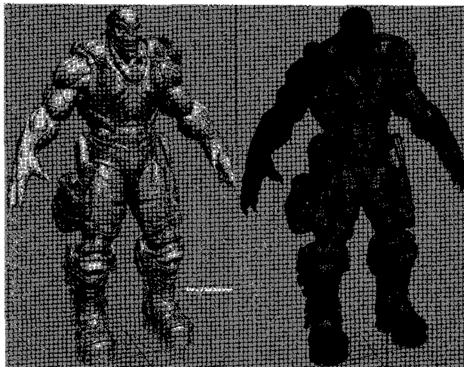
(c) 글로스맵 (d) 최종 이미지

(그림 6) 노멀 매핑 구현

구현되기 시작한 것은 바로 쉐이더가 보급되면서부터이다. 픽셀 쉐이더의 가장 중요한 임무가 텍스처 처리인데, 노멀매핑 역시 픽셀 쉐이더에서 수행된다. 노멀매핑 구현에 있어 버텍스 쉐이더는 픽셀 쉐이더를 위해 광원 벡터 (L) 제공 등의 기능을 수행한다.



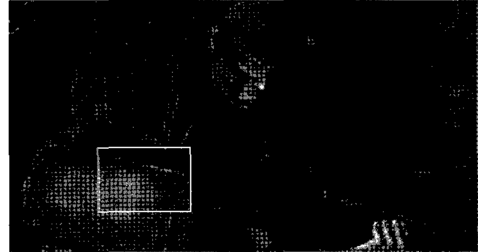
(a) 저해상도 모델

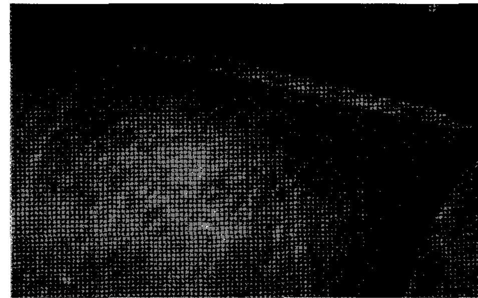


(b) 고해상도 모델

(c) 렌더링 결과

(그림 7) Unreal III 엔진 : 모델링 및 렌더링

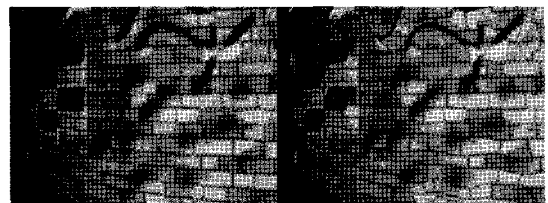




(그림 8) 노멀매핑 사례 (PS3 War Hawk)

Unreal III 엔진에서는 (그림 7) (a)와 같이 5,200개 다각형으로 구성된 저해상도 모델을 사용하여 노멀매핑 등을 비롯한 몇 가지 텍스처링 기법을 통해 마치 (그림 7) (b)에 보인 바와 같이 2백만개 이상의 폴리곤으로 구성된 고해상도 모델을 렌더링한 것과 같은 효과를 나타내었다. (그림 7) (c)의 렌더링 결과를 보라. 최신 게임들에서 이러한 픽셀별 라이팅이 광범위하게 채택되고 있다. (그림 8)은 PlayStation3 게임인 War Hawk에서의 노멀매핑 예를 보여준다(물론, 메이저 개발사의 AAA급 게임의 경우, 이미지 텍스처와 메시 데이타, 레벨 데이타와 같은 그래픽 리소스 자체의 훌륭함이 고품질 렌더링에 커다란 기여를 한다).

노멀매핑은 픽셀별 라이팅 기법의 하나의 예



(그림 9) Parallax 맵 (좌측 : 적용 전, 우측 : 적용 후)

에 불과하다. 프로그래밍이 가능한 쉐이더 기능을 이용하여 렌더링 품질을 높이기 위한 각종 기법들이 활발하게 제안되어 사용되고 있다. 그러한 기법중에 Parallax 매핑기법[19]이란 것이 있다. 이는 각 픽셀에서의 원래의 텍스처 좌표를 그대로 사용하는 대신 텍스처 좌표를 수정하여 울퉁불퉁한 표면을 보다 사실적이고 원근감 있게 표현하는 방법이다. 이에 대한 설명은 본 원고의 범위를 넘어가므로, 관심있는 독자는 참고문헌을 보라.

4. 렌더링 : 기능적 요소

상용 게임을 개발하기 위해서는 게임에 등장하는 캐릭터를 비롯하여 지형과 각종 배경 등의 요소들을 효과적으로 표현할 수 있어야 한다. 이 절에서는 게임의 렌더링 엔진이 제공해야 하는 대표적인 기능적 요소들을 나열하고 그 내용을 간략하게 고찰하고자 한다.

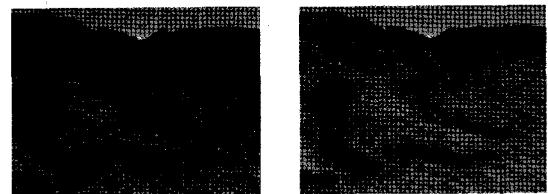
4.1 캐릭터 렌더링과 애니메이션

캐릭터는 대부분의 게임 장르에서 상호 작용의 핵심이며 게임 플레이의 주체가 되는 중요한 객체이다. RPG(Role Playing Game)나 FPS(First Person Shooter) 게임을 비롯한 대부분의 3D 게임 장르에서는 사람과 같은 관절체 캐릭터가 등장하는데, 이를 위해서는 폴리곤 메시(polygon mesh)를 관절의 위치 및 방향에 따라서 행렬을 이용하여 적절하게 변형하는 스키닝(skinning) 기능이 제공되어야 한다. 스키닝을 사용함으로써 폴리곤 메시와 관절체의 동작을 서로 독립적으로 만들 수 있으므로 매번 새로운 메시를 만들지 않아도 다양한 애니메이션을 쉽게 추가할 수 있는 장점이 있다. 실제로, 초창기의 FPS에서는 스키닝 대신 키프레임별로 만들어진 메시를 버텍스 단위로 보간하여 캐릭터 애니메이션을 표현하기도 하였다. Quake가 좋은 예이다.

스키닝을 구현하는 방법은 다양하다. 가장 기본적인 방식으로 모든 버텍스의 변형 연산을 CPU에서 계산하는 소프트웨어 스키닝 방식이 있다. 그 외에 고정 파이프라인에서 제공하는 vertex blend 및 indexed vertex blend 기능을 이용하는 방법이 있고, 최근에 나온 방법으로 버텍스 쉐이더를 이용하여 쉐이더 상수에 각 관절의 행렬 값들을 전달하는 쉐이더 스키닝 기법도 있다. 하드웨어에서 쉐이더 가속 기능을 지원하는 경우 쉐이더 스키닝 방식을 사용하는 것이 일반적이며, 상용 게임에서는 해당 하드웨어의 명세에 따라서 이들 기법들을 적절히 혼합하여 사용한다.

최근에는 게임에 등장하는 인체형 캐릭터를 보다 사실적으로 묘사하기 위해서, 머리카락이나 옷자락, 치마와 같은 부드러운 물체들의 움직임을 물리 시뮬레이션으로 만들어내기도 한다. 이 경우 연산 부하는 많지만, 미리 키프레임 애니메이션으로 만들어낸 것에 비해서 동적이고 상황에 맞는 사실적인 움직임을 표현할 수 있다는 장점이 있다. 이러한 연산을 수행하기 위해서 하복[11]이나 노보덱스[12]같은 별도의 물리 엔진을 사용하기도 한다.

4.2 지형 렌더링

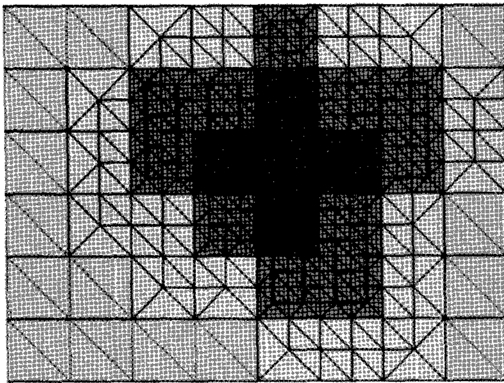


(그림 10) 지형 원형과 간략화된 지형

옥외 지형의 walk-through 또는 fly-through를 지원하는 게임의 경우 지형 렌더링은 필수요소 중 하나이다. 국내 게임 장르의 주류를 형성하고 있는 MMORPG의 경우가 지형의 walk-through를 필요로 하는 대표적인 예이다. 화면에 보여지는

지형 범위가 광범위할 경우, 실시간 렌더링이 어려우므로, 이를 해결하기 위해서 간략화된 지형 데이터를 사용하는 LOD(level of detail) 기법이 많이 채용되어 왔다.

지형의 실시간 렌더링을 위해서 제안된 LOD 기법들은 여러 가지가 있는데 크게 연속적인 LOD(continuous LOD) 기법과 정적인 LOD(static LOD) 기법으로 나눌 수 있다. 연속적인 LOD 기법은 지형의 기하 정보와 카메라와의 위치를 고려해서, 전체 지형에 대한 적절한 세부 수준의 지형 폴리곤을 매 프레임마다 연속적으로 생성해내는 방법이다. 4진트리 구조를 바탕으로 점진적으로 LOD 폴리곤을 생성하는 방법[13,14]이나 binary triangle tree를 이용하는 ROAM[15]이 이에 해당한다. 정적인 LOD 기법은 전체 지형을 보다 작은 블록으로 구획화하여, 각 블록에 대해서 몇 단계의 LOD 폴리곤을 미리 생성한 뒤에, 렌더링 시에 각 블록 별로 적절한 LOD 수준을 선택하는 방식이다.



(그림 11) 인접한 블록 사이의 삼각형화

연속적인 LOD 기법들은 쉐이더가 활발하게 보급되기 이전에 일반적으로 쓰이던 기법들이다. 이들은 모든 버텍스 및 폴리곤 정보를 시스템 메모리에 유지하고, 프레임마다 전체 지형에 대한 정확한 LOD를 CPU에서 계산하는 방식이므로 CPU를 과도하게 사용한다는 단점이 있다.

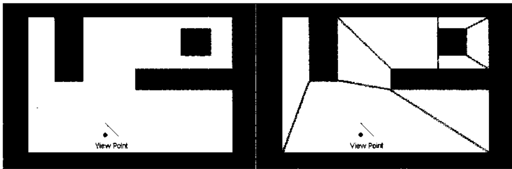
최근에는 GPU가 범용적으로 사용되고 있으므로 게임에 사용되는 지형 렌더링도 GPU를 활용하여 이룰 수 있게 되었다. CPU와 GPU의 병렬성을 높이기 위해서는, 기존에 CPU가 하던 일의 부담을 줄이고, 적절한 개수의 폴리곤에 대한 렌더링을 GPU가 처리하도록 배분하되, 시스템 메모리에서 비디오 메모리로의 자료 전송을 줄여야 한다[16]. 정적인 LOD 방식은 각 블록의 모든 LOD 레벨에 대한 기하 정보를 미리 비디오 메모리에 저장한 뒤에, CPU에서는 현재 블록에 적합한 LOD 레벨을 선택하고, 각 블록 LOD 폴리곤은 GPU에서 처리하므로 CPU의 부담을 줄이고 높은 성능을 얻을 수 있다(그림 11 참조).

(그림 12) 인접한 블록 사이의 틈(crack)

정적 LOD 기법을 사용할 경우 발생하는 문제점 중 하나는 (그림 12)와 같이 인접한 블록 사이의 LOD 수준의 차이에 따라서 T-접합점(T-junction) 혹은 틈새(crack)가 발생할 수 있다는 것이다. 이를 해결하기 위해서 인접한 두 블록 사이의 LOD 수준이 동일하거나 최대한 한 수준만의 차이가 나도록 제한하고, 서로 인접한 블록에서 발생할 수 있는 모든 경우에 대해서 미리 LOD 폴리곤들을 생성한 뒤 렌더링 시에 적절한 것을 선택하는 방법이 있다[17]. 한편, 지형에 LOD를 적용하면 카메라가 이동함에 따라 LOD 수준이 변하면서 폴리곤이 튀는(popping) 현상이 발생한다. 이러한 튐 현상을 해결하기 위해서 두 LOD 수준의 버텍스를 프레임 변화에 따라 부드럽게 보간하는 Geomorphing 방식이라는 방법이 제안되었는데, 이를 버텍스 쉐이더를 사용하여 구현하기도 한다[18].

4.3 정적 장면 렌더링

게임을 만드는 기반기술과 제작역량이 발전함에 따라서, 게임의 배경이 되는 장면들도 점점 사실적이면서 복잡해지고 있다. 최근에 나온 Half-Life2나 Doom3의 경우 복잡한 도시와 건물이 매우 세부적으로 묘사되어 사용자들로부터 호평을 받은 바 있다. 이러한 복잡한 건물을 효과적으로 렌더링하기 위해서는 각종 실시간 렌더링 기법들을 활용하여 화면에 그려질 폴리곤의 수를 줄이는 것이 관건이다. 3D 게임의 대중화를 연 FPS 장르 게임의 경우 전통적으로 방과 복도로 구성된 실내 장면을 주로 사용한다. 이런 실내 장면을 표현하는 데에 많이 사용되는 기법으로 포탈(portal) 렌더링과 PVS(Potentially Visible Set) 기법이 있다.



(그림 13) 포탈에 의한 공간 구획화 (단면)

포털 렌더링에서는 3차원 공간을 볼록한 블록들로 분할한 뒤, 이들을 포탈(portal)이라고 불리는 폴리곤을 통해서 연결되어 있는 것으로 생각한다(그림 13 참조). 만약 카메라가 특정 블록 안에 있다면, 카메라에서 보여지는 부분은 해당 블럭과, 그 블록이 소유한 포털들에 의해 이어진 블럭들로 제한된다. 이러한 원리에 의해서 전체 장면이 아무리 복잡하더라도 카메라에 보이는 영역만을 찾아서 빠르게 선택적으로 렌더링할 수 있다. 한편 PVS는 3차원 공간에서 여러 영역들로 구획화해서 각 영역들에서 보여질 가능성이 있는 영역들을 미리 계산해서 저장하는 방식이다. 이렇게 저장된 값을 이용해서, 렌더링 시에 현재 카메라가 속한 영역에서 보일 가능성이 있는 영역들을 빠르게 찾음으로써 렌더링 될 폴리곤 수를 매우 효과적으로 줄일 수 있다.

정적 장면 렌더링과 관련된 중요한 기술 중에 공간분할 기법이 있는데, 이것은 복잡한 장면에서 렌더링 성능을 올리기 위해서 공간을 계층적 구조로 구획화하는 것을 말한다. 이는 FPS 게임에서 BSP(Binary Space Partitioning) 트리가 도입된 이래 게임에서 성능을 높이기 위해 널리 사용되어 왔다. BSP는 평면을 이용해 전체 공간을 재귀적으로 절단하고 구획화하는 기법이다. 이를 이용해 시야 바깥으로 벗어나는 영역을 빠르게 계산할 수 있고(view frustum culling), 차폐물에 의해 가려지는 부분을 걸러낼 수 있으며 (occlusion culling), PVS 계산에도, 때로는 충돌 검사에도 효율적으로 활용할 수 있다. Doom 이후로 FPS 게임 엔진에서는 BSP 트리 구조가 많이 사용되어 왔으며, 이외에도 4진트리나 8진트리 등의 구조가 활용되기도 한다.

5. 결 론

본 원고에서는 게임의 렌더링 엔진에 대해서 저수준 렌더링 파이프라인 요소, 쉐이더에 관한 기술적 이슈, 그리고 기능적 요소들에 대해서 간략하게 고찰해 보았다. 실시간 렌더링 기술 및 게임과 관련된 이슈는 매우 방대하여 다양한 연구 분야가 존재한다. 본 원고에서 계속 강조한 바처럼, 최근 쉐이더를 지원하는 그래픽스 하드웨어가 널리 보급되고 있으므로, 이제 쉐이더는 게임 개발에서 필수라고 해도 과언이 아니다. 렌더링 품질과 성능을 향상시키기 위해서는 저수준의 렌더링 파이프라인을 철저히 이해하고 쉐이더를 활용하여 이를 구현하려는 노력이 필요하다. 또한 렌더링 엔진의 여러 기능들 중에서 어느 특정 모듈이 전체적인 파이프라인에서 병목현상(bottleneck)을 유발할 수도 있다. 따라서, 게임상의 모든 모듈이 적절하게 조화를

이루도록 하고, CPU와 GPU에 작업을 적절하게 분배하여 병렬성을 높이면서, CPU와 GPU 간 데이터 전송량을 최소화하는 것은 게임을 개발함에 있어서 항상 염두에 두어야 하는 사항이다.

렌더링 파이프라인과 쉐이더 프로그래밍은 PC 게임에만 국한된 것은 아니다. 지금까지 언급한 하드웨어의 기능들과 여러 기술적인 이슈들은 Xbox360이나 PlayStation3와 같은 콘솔 게임기에 그대로 적용되는 사항이며, 심지어는 모바일 게임도 예외가 아니다. ATI 등의 주도 하에 모바일 3차원 가속기 개발 및 상용화가 매우 빠른 속도로 진행되고 있으며, 이에 대한 API로 OpenGL ES(OpenGL for Embedded Systems)[20]가 이미 2003년 여름에 발표되었고, 쉐이더를 지원하는 version 2.0이 2005년 8월에 발표되었다. 따라서, 3차원 모바일 게임 개발 과정은 수년 전 PC에서 3차원 게임 개발할 때 거쳤던 경로를 밟아가게 될 것이다. 상대적으로 저사양인 모바일 기기에서의 실시간 렌더링을 보장하기 위해서는 최적화된 파이프라인 코딩이 필수적이다.

참고문헌

[1] id Software, http://www.idsoftware.com

[2] Render Ware, http://www.renderware.com

[3] Epic Games, http://www.unrealtechnology.com

[4] Vicarious Vision, http://www.vvision.com

[5] Emergent Game Technologies, http://www.gamebryo.com

[6] OpenGL, http://www.opengl.org

[7] Microsoft DirectX, http://www.microsoft.com/directx

[8] nVidia Developers page, http://developer.nvidia.com/page/home

[9] ATi Developer page, http://www.ati.com/developer

[10] Randima Fernando and Mark J. Kilgard, The Cg Tutorial, Addison Wesley, 2003.

[11] Havok, http://www.havok.com

[12] Ageia, http://www.ageia.com

[13] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. ACM SIGGRAPH 96, August 1996, pp.109~118, 1996.

[14] Stefan Röttger, Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Real-Time Generation of Continuous Levels of Detail for Height Fields. In Proc. WSCG '98, pp. 315~322, 1998.

[15] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, Mark B. Mineev-Weinstein, ROAMing terrain : real-time optimally adapting meshes, Proceedings of the 8th conference on Visualization '97, p.81-88, October 18-24, 1997.

[16] Optimizing the graphics pipeline, Proceedings of the GDC 2003.

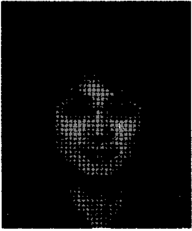
[17] Willem H. de Boer, Fast Terrain Rendering Using Geometrical MipMapping, 2000, http://www.flipcode.com/tutorials/geomipmips.pdf

[18] Glenn Corpes, Procedural Landscapes, Proceedings of the GDC 2001, http://www.cix.co.uk/glennc/gdcetalk files/frame.htm

[19] T. Welsh, Parallax Mapping with offset limiting : A per-pixel approximation of uneven surfaces, Infiscape Corporation, 2004.

[20] Khronos Group, http://www.khronos.org/opengles/

저자약력



김 성 용

1998년 KAIST 전산학과 학사
2000년 KAIST 전산학과 석사
2000년~2001년 나래디지탈엔터테인먼트 연구원
2001년~2002년 디지털매직엔터테인먼트 개발팀장
2003년~현재 제이씨엔터테인먼트 프로그래밍 팀장
e-mail : kimsungyong@gmail.com

한 정 현

1982년~1988년 서울대학교 컴퓨터공학과 학사
1989년~1991년 University of Cincinnati Computer Science 석사
1991년~1996년 USC Computer Science 박사
1996년~1997년 미국 상무성 National Institute of Standards and Technology(NIST) 연구원
1997년~2004년 성균관대학교 정보통신공학부 전임강사, 조교수, 부교수
2004년~현재 고려대학교 컴퓨터학과 부교수
2001년~2004년 산업자원부 지정 성균관대학교 게임기술개발지원센터 센터장
2004년~현재 문화관광부 지정 고려대학교 게임기술연구센터 센터장
e-mail : jhan@korea.ac.kr