게임엔진: 렌더링

한 정 현
(고려대학교 컴퓨터학과)

1. Introduction

1992년 id Software가 최초의 3차원 게임인 Wolfenstein 3D(그림 1)를 선보인 이래, 3차원 게임은 게임 시장에서 주류의 자리를 차지하게 되었다. 저사양의 하드웨어에서 구동되는 모바일 게임을 제외하고는 현재 대부분의 게임이 3차원으로 제작

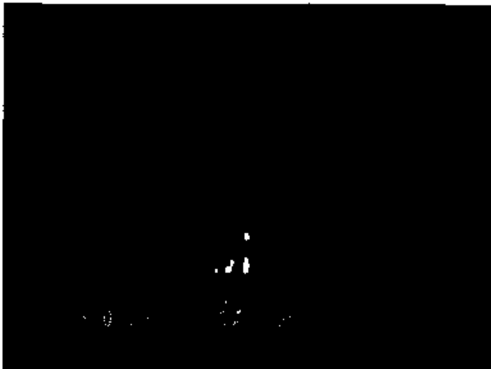


그림 1. Wolfenstein 3D: 스크린 샷

그림 2. Unreal III 엔진: 스크린 샷

*E-mail: jhan@korea.ac.kr

된다고 보아도 무방할 것이다. 이 같은 시장 수요에 부응하여 다수의 3차원 렌더링 엔진이 개발되었는데, 대표적인 것으로는 id Software의 Doom III[1], Criterion의 Renderware[2], Epic Games의 Unreal II 및 III[3], Intrinsic의 Alchemy[4], NDL의 GameBryo[5] 등이 있다. 이 중 가장 최근의 것으로는 2004년 봄에 발표된 Unreal III 엔진이 있는데, 이는 최신의 컴퓨터 그래픽스 기술을 총동원하여 매우 인상적인 렌더링 결과를 보여주었다(그림 2 참조).

2. Low-level Rendering

게임 엔진 중 렌더링 모듈의 임무는 실시간에 3차원 환경 및 물체를 그려주는 것으로, 이는 OpenGL[6] 또는 DirectX[7]와 같은 이른바 low-level 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 여름에 version 2.0이 발표될 예정이다. OpenGL은 게임 전용이라기 보다는 범용적인 graphics library이다. 하지만, id Software의 Quake 3 등 호평을 받는 다수의 게임이 OpenGL을 이용해 개발되었다.

Microsoft에 의해 개발된 DirectX는 (Windows, MacOS, Linux, Unix 등에서 모두 구동되는 cross-platform standard로 제안된) OpenGL과 달리 Windows 플랫폼 전용이다. 또한, 렌더링 기능을 담당하는 DirectX Graphics 이외에 DirectInput, DirectSound 등 게임 제작에 필요한 다종다양한 API를 제공하여, 이른바 multimedia API로 불리며 실제 게임 개발에 있어 OpenGL보다 많은 사용자 층을 확보하고 있는 *de facto* standard이다.

표 1. PC 그래픽스 카드

	Milestone graphics cards	Features
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

1995년 처음 발표된 이래 빠른 속도로 후속 버전이 개발되어, 2003년에 version 9.0b가 발표되었다.

OpenGL과 DirectX는 모두 그래픽스 가속 하드웨어의 API이고, 이 가속기의 역할은 이른바 rendering pipeline을 구현하는 것이다. 현재 3차원 게임이 대중화된 것은 가속기의 발달, 특히 PC 그래픽스 카드 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 rendering pipeline을 담당하는 것이다. Rendering pipeline은 3차원 꼭지점을 처리하는 vertex processing과, 화면에 최종적으로 화소를 생성하는 rasterization 단계로 구분된다. 초기의 PC 그래픽스 카드는 rasterization 기능만을 가지고 있었으나, 점차 transformation and lighting (T&L) 등 vertex processing 기능까지 포괄하게 된다. 그러나, 이렇게 진화된 rendering pipeline은 '고정된 (fixed)' 기능을 수행하므로, 게임 프로그래머가 표현할 수 있는 범위는 하드웨어적으로 고정된 방식에 제약될 수 밖에 없었다. 예를 들어, 광원(light source)은 point, directional, spot 세 가지 중에서 하나를 선택해야 하고, 정반사(specular reflection) 계산에 있어서는 half-way vector를 이용하는 Blinn-Phong 모델을 사용할 수 밖에 없었다. 제작 중인 게임에 Blinn-Phong 모델 대신 다른 기법을 사용하고자 하는 개발자는, 그래픽스 카드가 제공하는 고성능의 pipeline을 포기하고, 이를 소프트웨어적으로 구현할 수 밖에 없었다.

이 같은 fixed function pipeline의 유연성 문제를 극복하고자 제안된 것이 *programmable* pipeline이고, 이를 구현한 것이 바로 shader이다. Shader program은 GPU에 대한 명령의 집합이다. 즉, GPU 명령어를 이용해 본인이 원하는 렌더링 방식을 직접 코딩하는 것이다. 전술한 바와 같이 rendering pipeline은 vertex processing과 rasterization으로 구분되므로, shader program도 vertex processing을 담당하는 vertex shader와, rasterization을 담당하는 pixel shader로 구분된다(이는 DirectX 용어이고, OpenGL에서는 vertex program과 fragment program이라는 용어를 사용한다).

아래 예가 보여주는 것처럼, DirectX 8.0의 경우, shader program은 어셈블리어로 코딩해야 했다.

```
//c0~c3: view_proj_matrix
//c4~c6: texture_matrix0

dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3

dp4 oT0.x, v0, c4
dp4 oT0.y, v0, c5
dp4 oT0.z, v0, c6
```

그러나, DirectX 9.0에서 C언어 스타일의 High Level Shading Language(HLSL)[7]가 소개되었다. OpenGL 역시, 2003년 7월 발표된 version 1.5 및 2004년 여름 발표 예정인 version 2.0에서 'OpenGL Shading Language (GLSL)'에서 로우레벨 및 하이레벨 언어를 모두 지원한다. 한편, NVIDIA에서는 low-level API에 독립적인 Cg(C for graphics)[10]를 발표했다. 이같은 하이레벨 언어로 작성된 shader program은 기계어로 컴파일되어 해당 GPU에서 실행된다. 위에 예시한 DirectX 8.0 어셈블리어 코드를 DirectX 9.0 HLSL로 다시 코딩하면 아래와 같다.

```
float4x4 view_proj_matrix;
float4x4 texture_matrix0;

struct VS_OUTPUT {
  float4 Pos : POSITION;
  float3 Pshade : TEXCOORD0;
};

VS_OUTPUT main (float4 vPosition : POSITION) {
  VS_OUTPUT Out = (VS_OUTPUT) 0;
  Out.Pos = mul (view_proj_matrix, vPosition);
  Out.Pshade = mul (texture_matrix0, vPosition);
  return Out;
}
```

Shader program은 사실감 있는 영상의 실시간 렌더링에 커다란 기여를 하고 있다. Doom III에서의 bump mapping, 유명한 Dead or Alive Xtreme (DOAX) Beach Volleyball 게임의 스키닝 애니메이션, 최근 관심을 끌고 있는 High Dynamic Range 렌더링 및 카툰 렌더링 기법들은 모두 shader program의 결과물이다. 또한, 최근 발표된

그림 3. Unreal III 엔진: 모델링 및 렌더링

Unreal III 엔진도 DirectX 9.0과 Shader 3.0을 이용하여 매우 사실적인 영상을 실시간에 렌더링할 수 있음을 보여줬는데, 5,200개 다각형으로 구성된 모델(그림 3: 중앙)을 normal map과 displacement mapping을 사용하여 렌더링하여 마치 2백만개의 다각형으로 구성된 모델(그림 3: 왼쪽)을 렌더링한 것과 같은 효과를 나타내었다(그림 3: 오른쪽).

이 절에서 우리는 OpenGL과 DirectX를 기준으로 (shader 코딩을 포함한) low-level API에 대해 살펴보았다. 하나의 게임 엔진이 OpenGL과 DirectX 중 하나의 API만을 지원한다면, 이식성에서의 문제가 대두된다. 또한, 하나의 API에서도 후속 버전에 대한 호환성 문제가 발생한다. 이에 대한 해결 방식으로 대부분의 상용 게임엔진은 low-level API (OpenGL과 DirectX)를 감싸는 (wrapping하는!!) mid-level API를 제공한다. 이러한 mid-level API는 추상(abstract) API 또는 wrapper라고도 불린다.

3. Realtime Rendering Issues: An Example

게임엔진은 실시간 렌더링에 필요한 최적화된 라이브러리 및 제작 툴을 제공해야 한다. 사실적인 영상의 실시간 렌더링에 필수적인 모듈은 그림자 생성, 물/불/연기 등 자연형상을 위한 파티클 시스템, 포털 및 BSP 컬링, 충돌탐지, levels of detail (LOD) 생성 및 처리, 지형 렌더링 등이 있다. 이 같은 실시간 렌더링 관련 이슈들은 몇몇 교재에 잘 정리되어 있다[11,12]. 상용 게임 엔진은 이들 알고리즘의 최적화된 구현물을 제공해야 한다. 본 원고에서는 방대한 분량의 실시간 렌더링 내용을 정리하는 대신, 'LOD 지형 렌더링'을 예로 들어 실시간 이슈를 설명하고자 한다.

옥외 지형의 walk-through 또는 fly-through를 지원해야 하는 3차원 게임 개발에 있어 지형 렌더링은 필수요소 중 하나이다. 국내 게임 장르의 주

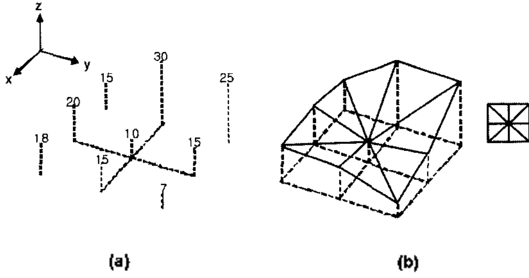


그림 4. Height field

유를 형성하고 있는 multi-player online game (MOG)은 지형의 walk-through를 필요로 하는 대표적인 예이다.

지형 표현 기법으로 가장 많이 쓰이는 것은, 그림 4(a)에 도시된 바와 같이 지형 높이를 정규적으로 샘플링한 height field이다. 렌더링은 그림 4(b)에서 처럼 삼각형 메쉬(triangular mesh) 생성을 통해 이루어진다. 이러한 지형 데이터는 실제 지형으로부터 얻어올 수도 있고[13], 지형 에디터 등을 이용해 편집 생성할 수도 있다. 그런데, height field 데이터는 종종 그 크기가 매우 방대해, 실시간에 렌더링되기 어려운 경우가 있다(비행 시뮬레이팅 게임에서 대평원을 fly-through하는 경우를 상상하라!). 이를 위해서 LOD 기법이 필요하다.

그림 5(a)는 3X3 크기의 height field 데이터에서 5개의 꼭지점을 삭제해 2X2 크기로 간략화한 예를 보여준다. 3X3 배열 별로 이 같은 간략화를 되풀이하면, 그림 5(b)와 같은 LOD를 얻을 수 있다. 이 같은 방식으로 생성된 지형 LOD의 문제점

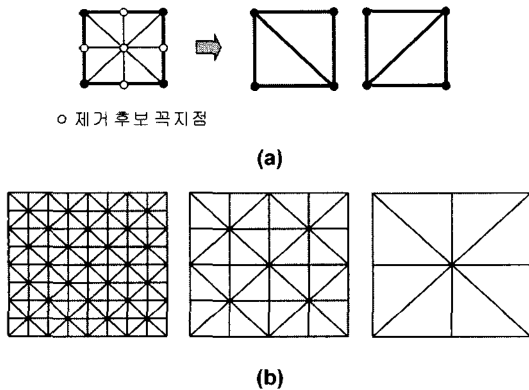


그림 5. 메쉬 간략화

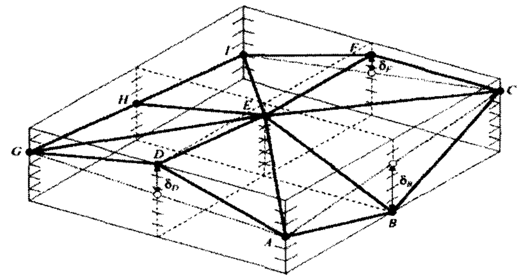


그림 6. 꼭지점 제거 에러 측정

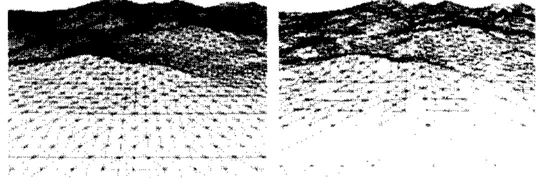


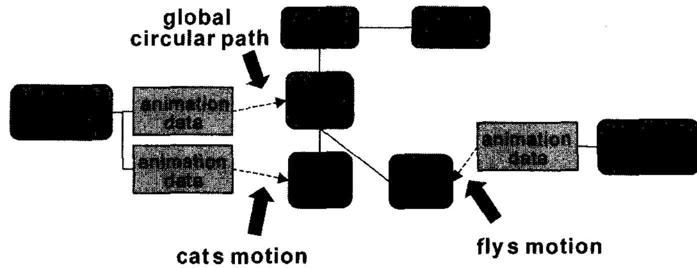
그림 7. 지형 렌더링 결과

은, 지형 표면의 실제 곡률을 전혀 고려하지 않았다는 데 있다. 예를 들어, 평원 지형에서는 그림 5(b) 오른쪽과 같은 거친(coarse) LOD를 이용해 렌더링해도 문제가 없겠지만, 산악 지형의 경우는 상세한(fine) LOD를 사용해야 하는데, 그림 5(b)와 같은 기계적인 방식으로는 이 같은 선별적(selective) LOD를 달성할 수 없다는 것이다.

선별적 LOD 생성을 위해서는, 하나의 꼭지점이 제거 되었을 때 야기되는 화면 상의 에러를 측정하는 방법이 일반적이다. 그림 6에 도시된 바와 같이, 꼭지점 B가 제거되므로 해서 발생하는 에러는 δ_B로 계산될 수 있는데, 시점(viewpoint) 등의 뷰잉 조건이 결정되면, δ_B가 화면상에 투영 되었을 때 점유하는 화소의 개수를 계산할 수 있다. 사용자가 정한 허용 기준치보다 이 화소 수가 작으면 해당 꼭지점은 삭제된다. 이 방식을 이용하면, 그림 7에 도시된 바와 같이, 평원 지형에서는 거친 LOD를, 산악 지형에서는 상세한 LOD를 얻게 된다.

위에서 기술한 기법은 Lindstrom[14]이 제안한 것으로, 실제 구현에 있어서는 에러 계산을 근사화시켜 프레임율을 극대화하였고, 동시에 crack 방지 등을 위한 복잡한 삼각화 기법을 제안하였다. 한편, 실제 게임 개발에 있어서는 Lindstrom 기법보다 더욱 최적화된 다른 기법[15,16]들이 사용되지만, 선

(a)



(b)

그림 8. Scene Graph 예

별적 LOD를 생성하는 근본적인 원리는 동일하다고 볼 수 있다.

4. High-level Rendering

렌더링 파이프라인 전체를 이해하는 것은 매우 어려운 일일뿐더러, 게임 개발자가 low-level 혹은 mid-level API를 일일이 호출하면서 게임 을 개발하는 것은 비효율적일 수 있다. 이를 해결하기 위하여 제공되는 것이 high-level API 이며, 대개 scene graph API가 이 역할을 한다.

Scene graph는 렌더링될 장면 혹은 세계 전체에 대한 기하 모델 및 변환(transformation), 텍스처 정보, LOD, 광원 정보, 렌더링 상태 정보 등을 계층적으로 구조화한 것이다.

실제 세계 및 그에 속하는 물체들은 종종 계층 구조를 가지고 있다. 사람의 팔 애니메이션을 예로 들어 보자. 사람의 손 동작(위치와 방향)은 손목의 동작에, 손목의 동작은 다시 팔꿈치의 동작에, 팔꿈치의 동작은 다시 어깨의 동작에 종속적이다. 이러한 종속적 계층 구조가 설정 되었다면, 움직이는 손을 렌더링하기 위해 계층 구조를 따라 변환 행렬을 곱해가면 될 것이다. 또한, 이러한 계층 구조는 충돌 탐지 등에도 효율적으로 이용될 수 있다.

이제 게임에서의 scene graph 활용 예를 살펴보도록 하자. 그림 8(a)는 고양이(cat)와 나비(fly)의 간단한 애니메이션 장면을 보여준다. 고양이는 일정 거리를 유지하면서 나비를 따라 움직이는데, 나비는 원 궤도(global circular path)를 따라 일정한 속도로 이동한다. 따라서 고양이와 나비는 원 궤도를 따라 동일한 운동을 한다. 한편, 나비는 날개짓(fly's motion)을, 고양이는 눈을 깜빡이는 동작(cat's motion)을 한다. 이를 scene graph로 표현한 것이 그림 8(b)이다. 이러한 scene graph는 다음과 같은 간단한 코딩을 통해 생성된다.

```
Scene *scene = new Scene; // **
Camera *camera = new Camera;
camera->SetMatrix( ... );
scene->SetCamera(camera);

Group *whole_group = new Group; // **
whole_group->SetMatrix( ... );
scene->AddChild(whole_group); // **

Mesh* cat_mesh = new Mesh( ... ); // **
whole_group->AddChild(cat_mesh); // **

Mesh* fly_mesh = new Mesh( ... ); // **
whole_group->AddChild(fly_mesh); // **

AnimCtrl* ctrl_1 = new AnimCtrl;
ctrl_1->AddAnimData( ... , whole_group);
ctrl_1->AddAnimData( ... , cat_mesh);

AnimCtrl* ctrl_2 = new AnimCtrl;
ctrl_2->AddAnimData( ... , fly_mesh);
```

위 프로그램의 디테일을 이해하기 보다는 고양이와 나비 계층구조의 개략을 알고 싶은 독자는 ** 코멘트된 문장만을 보면 된다. 위의 예에서 볼 수 있는 바와 같이, 노드 삽입 및 삭제 연산만으로

scene graph가 완성됨을 주목하라. 이렇게 해서 scene graph가 생성되면, 아래 코드와 같이, 매 프레임마다 애니메이션 데이터를 수정한 뒤, graph를 따라가며 렌더링을 수행하면 된다.

```
main_loop(){
  ctrl_1->Update(current_time);
  ctrl_2->Update(current_time);
  scene->Render();
}
```

상용 게임엔진 매뉴얼에는 해당 엔진에서 제공되는 노드 종류와 그 상속 관계 등이 명시되어 있으며, 또한 개발자 편의에 맞춰 새로운 노드를 추가할 수 있는 기능을 제공하고 있다. 물론, 실제 게임엔진에서 사용되는 scene graph는 위의 예보다 훨씬 많은 종류의 노드를 포함하는 등 복잡도를 보이지만, scene graph를 생성하고 렌더링하는 기본 원리는 동일하다. 3절에서 기술한 지형 LOD 생성 루틴도, 위 예에서의 애니메이션 컨트롤러와 같은 역할을 하여 적절한 지형 LOD를 생성할 것이고, 이렇게 해서 구성된 scene graph를 렌더링하면 하나의 프레임이 완성되는 것이다.

5. Conclusion

본 원고에서는 렌더링 엔진을 low-level API, high-level API, 실시간 렌더링 이슈의 세 부분으로 나누어 간략히 고찰해 보았다. 실시간 렌더링 이슈는 매우 방대하여, 다양한 연구 분야가 존재한다. High-level API인 scene graph를 이용하여 효율적으로 scene을 관리하는 기술은 연구로서의 가치가 있는 분야는 아니지만, 실제 게임 개발에 있어서는 매우 중요한 기능을 한다. 마지막으로 low-level API를 자유자재로 다루기 위해서는 렌더링 파이프라인을 철저히 이해하는 것이 중요하다. 이는 programmable GPU의 존재 때문이기도 하고, PC 영역을 벗어나, Xbox와 PayStation2 등의 콘솔 게임, 오락실 용 아케이드 게임, 그리고 최근 각광을 받고 있는 모바일 게임 등 분야에서의 게임 개발에도 렌더링 파이프라인 개념이 그대로 이용되기 때문이다.

모바일 게임의 경우를 살펴보면, ATi 등의 주도하에 모바일 3차원 가속기 개발 및 상용화가 매우 빠른 속도로 진행되고 있으며, 이에 대한 API로 OpenGL ES(OpenGL for Embedded Systems)[17]가 이미 2003년 여름에 발표된 바 있음을 발견할 수 있다. 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.

Acknowledgement

이 논문은 2004년도 고려대학교 특별연구비에 의하여 연구되었음.

References

1. http://www.idsoftware.com
2. http://www.renderware.com
3. http://www.unrealtechnology.com
4. http://www.intrinsic.com
5. http://www.gamebryo.com
6. http://www.opengl.org
7. http://www.microsoft.com/directx
8. http://developer.nvidia.com/page/home
9. http://www.ati.com/developer
10. R. Fernando and M. Kilgard, *The Cg Tutorial*, Addison Wesley, 2003.
11. T. Moller and E. Haines, *Real-time Rendering* (2nd edition), A K Peters, 2002.
12. D. Eberly, *3D Game Engine Design*, Morgan Kaufmann Publishers Inc., 2001.
13. http://www.usgs.gov
14. P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner, "Real-Time, Continuous Level of Detail Rendering of Height Fields," *SIGGRAPH96*.
15. M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, and M. Mineev-Weinstein, "ROAMing Terrain: Real-time Optimally Adapting Meshes," *IEEE Visualization97*
16. D. Wagner, "Terain Geomorphing in the Vertex Shader," Shader X^2: Shader Programming Tips & Tricks with DirectX 9, WOrdware Publishing, 2004.
17. http://www.khronos.org/opengles/