

[**3D 엔진 제작의 실제-1**](http://webdizen.new21.net/blog/entry/3D-엔진-제작의-실제-1)

*장언일 (uijang@ccr.co.kr)*

처음 시작하는 독자라면 간단하고 보기 쉬운 것부터 직접 구현해 보는 것이 중요하다. 필요에 따라 다른 사람이 이미 구현한 소스가 상당한 도움이 되기도 한다. 복잡한 구조의 C++나 템플릿을 마구 남용해서 짜나가는 것보다 간단한 구조부터 복잡한 구조로 잘 짜여진 소스를 보는 것은 프로그래밍 습관에 많은 영향을 미치기 때문이다.

필자가 많은 사람들에게 질문을 받는 것 중 하나가 ‘어떻게 3D를 공부하는가’이다. 3D 학습법은 특별히 정형화된 순서를 가지고 있지 않기 때문에 이것저것 가리지 말고 공부해야 하며 처음부터 난이도 높은 것에 많은 시간을 보내지 말아야 한다. 또한 3D 용어부터 친숙해져야 하며 그러한 용어들이 무엇을 뜻하는지에 대해 계속 지식을 쌓아 나가야 한다. 처음 프로그래밍할 때 ASE 뷰어기(3DS-MAX의 SCENE이 텍스트로 출력된 파일이다)를 만들어 볼 것을 권한다. ASE 뷰어기의 자료는 인터넷이나 3D 관련 책자에 많이 있으니 참고하면 된다. 이것을 만들어 보면 3D에서 자료구조를 어떻게 잡아야 할 것인지에 대해 도움을 받을 것이다.  
  
**3D 학습 방법**  
필자가 3D를 처음 접하던 시절에는 그 흔한 텍스처 맵핑을 구현하는 서적조차 구하기가 어려웠을 정도로 자료가 거의 없었다. 그나마 PC 통신을 통해 해외 FTP에서 자료를 구하면 어셈블리로 구현된 소스(어셈블리를 하는 독자라면 잘 알다시피 분석하는 데에 있어서 상당히 난해하다)를 며칠씩 분석했던 걸로 기억한다. 물론 덕분에 어떻게 하면 텍스처 맵핑을 빨리 하는가에 대해서는 내공을 쌓을 수 있었다.  
  
**일단 구현해 봐라?**  
처음 시작하는 독자라면 간단하고 보기 쉬운 것부터 직접 구현해 보는 것이 중요하다. 필요에 따라 다른 사람이 이미 구현한 소스가 상당한 도움이 되기도 한다. 복잡한 구조의 C++나 템플릿을 마구 남용해서 짜나가는 것보다 간단한 구조부터 복잡한 구조로 잘 짜여진 소스를 보는 것은 프로그래밍 습관에 많은 영향을 미치기 때문이다. 어느 정도 구현 능력이 생기면 구현보다는 문제에 봉착했을 때 어떤 방법으로 해결할 것인가가 매우 중요하다는 것을 깨닫게 될 것이다. 때로는 원하는 알고리즘을 위해 몇백 몇천 라인의 소스를 뒤져서 알아내는 것도 도움이 된다. 존 카멕이 공개한 퀘이크 I 소스 분석은 최고 난이도가 아닐까 싶다.  
  
**전체 이론을 파악하라**  
다른 사람의 소스를 분석하고 자기 것으로 만들어서 구현하는 능력이 생기면 ‘어떻게 만들어야 하는가’에 대한 이론을 신경써야 한다. 물론 하나를 얻기 위해 많은 소스를 본다는 것은 많은 인내심과 노력이 필요하다. 하물며 필자는 누군가 한 마디를 해주면 해결될 일을 며칠씩 고생해서 알아내는 경우도 많았다. 이런 경우 수단과 방법을 가리지 말고 사방에 도움을 요청해 보자.   
  
**모르는 것에 대해 두려워 말라**  
만약 여러분이 상용 게임 엔진 제작자와 만나는 일이 생긴다면 게임 엔진에 관해 이야기하고 싶은 것이 무척 많을 것이다. 그러나 필자 경험상 공부하는 학생들을 만나보면 주눅이 들어서인지, 모른다는 것에 대해 자존심이 상한 것인지 막상 질문하는 경우가 흔치 않다. 배움의 입장에서 자존심을 세우거나 낯가림을 하는 것은 매우 어리석은 일임을 기억해야 한다.  
예전에 필자는 DEMO GROUP에서 유명했던 Jmagic이라는 사람에게 잘 쓰지도 못하는 영어로 “당신이 사용한 모션블로 알고리즘을 다오!”라고 영문 메일을 보내서 친절하게 답변을 받았던 적이 있었다. 이처럼 대부분 실력 있는 사람은 지식 공유에 관해 상당히 관대하기 때문에 주저없이 물어보길 바란다(그렇다고 존 카멕에게 BSP 알고리즘을 가르쳐 달라는 것은 무식한 행동일 듯 싶다). 어찌됐건 가끔 그렇게 물어봤던 것들이 그 당시 필자에겐 많은 도움이 됐다.  
  
**C++, OOP로 엔진 설계**  
엔진을 처음 만드는 사람에게는 C++를 함부로 쓰지 말라고 이야기하고 싶다. 잘못된 구조로 프로그램을 계속 짜다보면 나중에는 원점부터 다시 시작해야 하는 상황이 생길 수 있기 때문이다. 물론 처음부터 아예 다시 짤 생각을 하는 고집센 C++ 프로그래머라면 시도해 봐도 무방하다. 하지만 가능하면 직관적인 구조를 지닌 C 스타일을 기반으로 필요한 부분에 C++를 넣어서 프로그램하길 바란다.  
배경 엔진  
배경 엔진은 가장 기본이 되는 부분으로 실제 눈에 보이는 부분에 대한 처리부터 충돌을 위한 정보와 AI를 구현하기 위한 패스를 얻을 수 있는 방법까지도 포함한다. 이러한 일련의 과정을 제대로 돌아가게 하기 위해서는 배경 엔진 구조가 매우 중요하다. 배경 구조는 엔진이 나아가야 할 방향이 결정되는 기반이라고 할 수 있다.   
온라인 게임이든 다른 게임이든 넓은 월드에서 대략 어디에 위치해 있고, 어느 부분을 렌더링 걸어야 하며, 주변에 보이는 캐릭터나 동적인 물체들 또는 총에 의해 깨지는 드럼통을 관리하기 위해서는 월드 구역을 나눠서 관리해야 한다. 요즘은 3D 그래픽카드의 능력이 매우 우수하므로 구조 없이도 가능하지만 많은 제약들이 생길 것이다. 그러한 배경 구조 없이 MMORPG를 만들기 시작한 평범한(?) 프로그래머의 시각에서 본 가상 일기를 살펴보자.  
  
**배경 구조 없이 만들어 본 가상 일기**  
이번 프로젝트는 MMORPG를 만드는 것이다. 그래픽하는 사람들이 폴리곤을 어느 정도 써야 하는지를 물어보기에 여기저기 알아 본 결과 다음과 같은 1차 결론을 얻었다.

◆ 월드의 폴리곤 전체를 4∼5만 개로 만들어야 적당한 속도가 나지 않을까  
◆ 최근 게임 추세는 캐릭터에 폴리곤을 많이 쓰던데 나중에 어떻게 될지 모르니까 1500개 정도가 어떨까

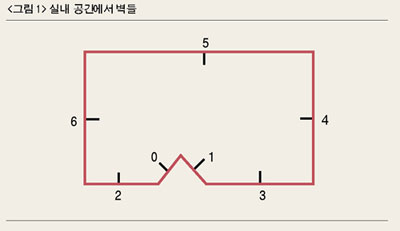
이와 같이 그래픽하는 사람들에게 이야기를 하고 나서 화면에 캐릭터를 뿌려보게 되었다. 기획자가 월드는 어느 정도 넓어야 게임이 된다고 해서 4∼5만 개로 월드를 크게 만들고 보니 눈앞에 보이는 배경 폴리곤의 각이 많아져서 시각적으로 부자연스러워 보였다. 이렇게 만들다 보면 배경에 그 흔한 풀조차 세우기가 힘들어질 것 같았다. 이런 상황에서 충돌 체크를 하기 위해 폴리곤 4∼5만 개를 모두 검색해 캐릭터가 서 있는 것을 계산하다 보니 캐릭터를 띄울수록 속도가 많이 떨어지는 것을 느꼈다. 일단 배경 폴리곤 숫자를 2만 개 정도로 줄이고 캐릭터는 1000개 안으로 줄이도록 그래픽 팀에 전했다. 게임 그래픽의 질이 현저히 떨어졌지만 그나마 실행 속도가 빨라진 것 같아 의지대로 작업을 계속 진행해 나갔다. 그러나 게임에 필요한 요소를 넣을수록 실행 속도가 떨어지는 데다 제한 조건이 너무 많아 팀 분위기도 쳐지고 어떻게 해야 할지 모르겠다.  
  
이처럼 가상 일기는 다소 과장된 부분이 있지만 배경 구조는 필수 요소이며 구조 구성에 따라 전체 엔진 성능까지 좌우할 정도로 매우 중요하다. 그럼 배경 구조를 위해 어떤 구조가 있는지 알아보자.

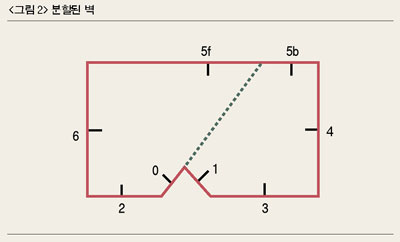
[**3D 엔진 제작의 실제-2**](http://webdizen.new21.net/blog/entry/3D-엔진-제작의-실제-2)

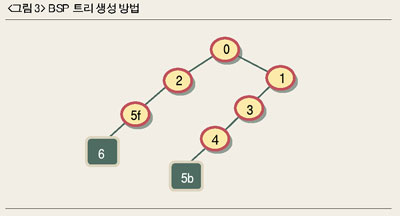
*장언일 (uijang@ccr.co.kr)*

처음 시작하는 독자라면 간단하고 보기 쉬운 것부터 직접 구현해 보는 것이 중요하다. 필요에 따라 다른 사람이 이미 구현한 소스가 상당한 도움이 되기도 한다. 복잡한 구조의 C++나 템플릿을 마구 남용해서 짜나가는 것보다 간단한 구조부터 복잡한 구조로 잘 짜여진 소스를 보는 것은 프로그래밍 습관에 많은 영향을 미치기 때문이다.

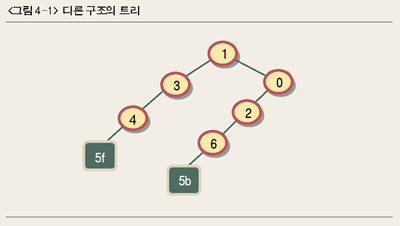
**배경 구조**  
배경 구조는 기본적으로 트리 구조를 가지고 있다. 트리 구조는 트리를 운행하다가 특정 노드를 렌더링할 필요가 없을 경우 하위 노드는 검색하지 않는다는 장점이 있다. 이 특성을 이용하면 월드를 크게 늘리더라도 렌더링하는 영역이 일정하기 때문에 속도에 지장을 주지 않을 수 있다.  
  
**BSP 트리**  
BSP(Binary Space Partitioning) 트리를 설명하는 데 있어서 대부분 벽을 세워 놓고 이야기를 하게 된다. 가장 이해하기 쉬우므로 필자 역시 벽을 세워서 이야기를 하겠다.

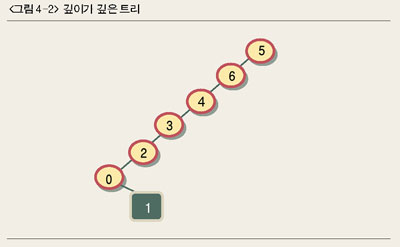




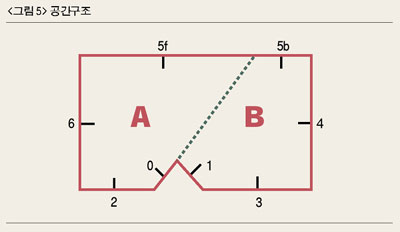


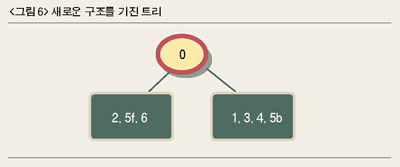
<그림 1>에서 긴 선분은 벽이라고 가정하자. 그리고 벽 앞에 짧은 선분은 벽이 향하고 있는 방향이다. 트리는 기준이 되는 벽을 중심으로 점차 만들어 가되 기준 벽에 의해 나눠지는 벽의 경우 분할해야 한다. <그림 2>에서 5번 벽의 경우 0번 벽에 의해 5f, 5b 벽으로 나눠진 경우이다.  
처음 선택된 0번벽(평면)을 중심으로 앞에 있는 폴리곤들은 front쪽(벽 방향을 중심으로 앞일 경우 front, 뒷일 경우 back, 트리 그림에서는 왼쪽이 front 오른쪽이 back이다)으로 몰면 0번의 front 쪽은 2, 5f, 6번이 존재하고 back 쪽은 1, 3, 4, 5b가 존재한다. 다음으로 2, 1번 벽을 각각 기준 평면으로 잡고 front, back을 설정한다. 어떤 벽들을 선택하느냐에 따라 <그림 4-1>, <그림 4-2>처럼 구조가 다른 트리가 생성된다.



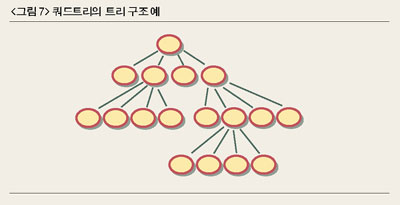


여기서 용어를 정리하면 각각의 동그라미를 나타내는 기준 평면들을 노드(node), 그리고 트리의 제일 마지막에 있는 공간을 나타내고 있는 것을 리프(leaf)라고 한다. 참고로 BSP 트리에서 노드의 front나 back에 아무 것도 존재하지 않는 경우를 솔리드 리프(solid leaf)라고 한다.   
이렇게 구성하게 되면 BSP의 또 다른 위력 중 하나가 철저한 이진 공간분할로 인해 플레이어가 어디에 있던 소팅(sorting)이 필요없다는 것이다. 트리 자체가 소팅돼 있다고 보면 된다. 따라서 트리를 타면서 제일 깊숙한 곳부터 폴리곤을 렌더링할 경우 어디에 있든 먼저 그려야 할 것과 나중에 그려야 할 폴리곤들이 정확히 구분되어지는 능력도 있다.  
<그림 4-1>의 경우 5f와 5b는 1번 폴리곤에 의해서 잘려진 평면이므로 <그림 2>에 있는 5f, 5b와는 다르다는 것을 참고하길 바란다. <그림 4-2>의 경우 트리가 한쪽으로 치우쳐 있어서 상당히 깊다. 이럴 경우 비대칭적 구조로 인해 트리 검색이 비효율적이기 때문에 BSP 트리 컴파일러를 만들 경우 최적화하는 작업을 해줘야 한다. 똑같은 구조로는 트리 깊이(루트부터 들어가는 횟수)와 front, back 자체가 균일하게 퍼진 트리가 검색 속도도 더 빠르다.  
  
**트리 구조 공간의 다른 접근법**  
사실 지금껏 봐온 그림들의 경우 트리에는 2가지 공간밖에 없다. <그림 5>에서 그려진 A, B의 공간은 트리의 제일 마지막 끝인 5b와 6의 공간이라고 생각하면 된다. 만약 5만 개의 폴리곤들을 이와 같이 트리 구조로 만든다면 트리에 의해서 분할되는 폴리곤도 있을 테니 트리 구조의 깊이만 몇천 개가 존재하고 분할된 10여만 개(추측)의 폴리곤들이 생성될 것이다. 따라서 여러분은 예전 둠(Doom)을 만들던 시절과 달리 폴리곤 하나하나에 신경써야 되는 것이 아니라 많은 폴리곤을 사용하기도 하고 이제 BSP 트리 중에 소팅이 굳이 필요없기 때문에 트리 구성을 할 필요가 없다.  
이처럼 구성하게 되면 솔리드 리프가 존재하지 않지만 0번 평면을 가진 노드를 중심으로 front, back 리프가 생성되므로 공간의 효율성은 훨씬 강력해진다.



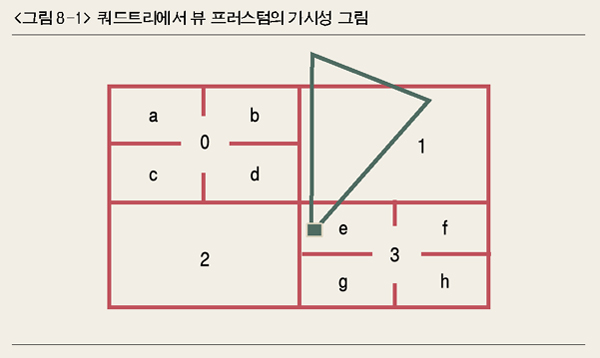


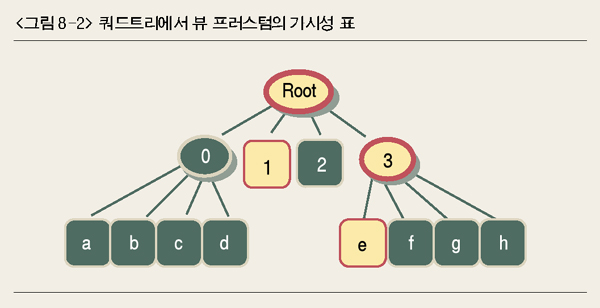
**옥트리와 쿼드트리**  
Oc(8)이라는 접두사와 Quad(4)라는 접두사를 보면 알다시피 노드가 갈라지는 개수에 의한 트리 구조라고 보면 간단하다. 즉 옥트리(Octree)는 하위 노드가 8개이고, 쿼드트리(Quadtree)는 하위 노드가 4개이다. 보통 실제 3D의 층 구조가 아닌 경우 실제로 2D 구조의 경우 굳이 복잡하게 옥트리로 하지 않고 높이를 제외해서 쿼드트리를 사용하는 것이 좋다.  
<화면 1>을 보면 폴리곤이 많이 모여있는 곳에 사각형들의 노드 바운딩 박스가 밀집되어 있는 것을 볼 수 있다. BSP의 경우 BSP 컴파일러를 만드는 일이 무척이나 어렵고 복잡하지만 옥트리와 쿼드트리는 그리 어렵지 않게 만들 수 있는 장점이 있다. 그렇지만 컴파일이 이미 되어 있는 BSP 파일을 사용하는 것은 오히려 더 직관적이며 사용하기 편하다. 인터넷에서 퀘이크 III 맵 뷰어기는 셀 수 없이 많아도 퀘이크 III 맵 에디터기가 거의 없는 것을 보면 BSP 트리를 만드는 게 어려워서 그렇지 만들어진 트리를 사용하기는 쉽다는 것을 알 수 있다. 기타 트리 구조에 잘 맞는 포털(portal) 렌더링이라는 것에 대해서도 알아 볼 필요는 있는데, 이에 관한 내용은 지난 2002년 4월 본지의 김성완씨가 쓴 ‘실전! 강의실’ 원고를 참고하기 바란다.



[**3D 엔진 제작의 실제-3**](http://webdizen.new21.net/blog/entry/3D-엔진-제작의-실제-3)

*장언일 (uijang@ccr.co.kr)*   
  
처음 시작하는 독자라면 간단하고 보기 쉬운 것부터 직접 구현해 보는 것이 중요하다. 필요에 따라 다른 사람이 이미 구현한 소스가 상당한 도움이 되기도 한다. 복잡한 구조의 C++나 템플릿을 마구 남용해서 짜나가는 것보다 간단한 구조부터 복잡한 구조로 잘 짜여진 소스를 보는 것은 프로그래밍 습관에 많은 영향을 미치기 때문이다.   
  
**뷰 프러스텀**   
뷰 프러스텀(view frustum)이라는 것을 간단히 정의하자면 렌더링할 때 보이는 부분에 관해 도형 형태로 나타내지는 영역이라고 보면 되겠다. 트리 방식의 구조를 가지게 되면 노드의 가시성 여부에 따라 더 이상 그 노드를 검색할 필요가 없다는 장점이 있다.





<그림 8-1>을 보면 쿼드트리에서 뷰 프러스텀 영역에서 벗어난 노드에 소속된 리프들은 렌더링을 할 필요가 없다.  
각각의 노드와 리프들은 x, y, z 평면에 정렬된 바운딩 박스 정보를 가지고 뷰 프러스텀 컬링을 통해 렌더링할 리프들을 얻어낸다. 트리구조로 되어 있어서 먼저 0, 1, 2, 3을 가지고 있는 커다란 루트 노드로 가시여부를 판단한다. 루트 노드가 가시영역에 들어와 있기 때문에 다시 각각의 0, 1, 2, 3의 바운딩 박스로 가시여부를 판단한다. 0번의 경우 뷰 영역에 들어오지 않았기 때문에 더 이상의 자식 노드나 리프의 가시여부를 판단할 필요가 없다. 반면 3번의 경우 뷰 영역에 들어오기 때문에 자식 노드나 리프를 검색해야 한다.  
<그림 8-2>에서 타원은 노드를, 네모는 리프를 뜻하며 보이지 않는 리프와 노드는 회색, 보이는 리프와 노드는 흰색으로 표시되어 있는 쿼드트리이다. 이 뷰 프러스텀을 코딩에 적용하는 몇 가지 방식들이 있는데 그 중에서 상당히 빠르고 간단해서 필자가 사용하고 있는 방법을 소개하겠다(지면관계상 정확한 원리와 방법은 생략한다).  
우선 뷰 매트릭스(view matrix)와 프로젝트 매트릭스(project matrix)를 인자로 넣어줘야 한다. MakeFrustumPlane() 함수는 뷰 매트릭스와 프로젝트 매트릭스가 결정되고 프러스텀 평면을 미리 생성해 놓는 함수이다. 여기서 생성된 6개의 프러스텀 평면(위, 아래, 좌우, near, far)으로 IsBBoxInFrustumPlane() 함수를 이용해 각각의 노드나 리프의 바운딩 박스와 체크를 해서 벗어났는지 안 벗어났는지를 판정하면 된다. 이것은 배경말고도 캐릭터나 동적인 물체들의 바운딩 박스를 만들어 이 함수로 가시여부를 판별해서 사용하면 유용하다.  
다음은 프러스텀 플레인 생성 소스 코드와 생성된 프러스텀 플레인으로 바운딩 박스와 충돌여부를 점검하는 코드이다.

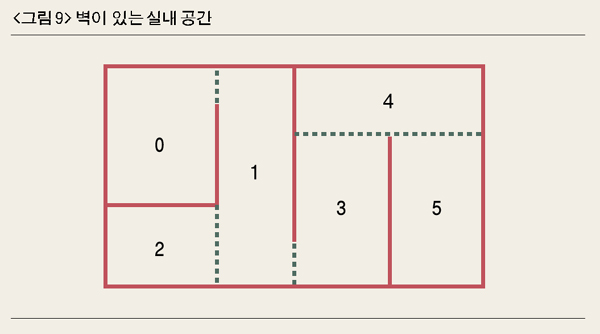
#define Vector3fDist(v) (sqrtf(v[0]\*v[0]+v[1]\*v[1]+v[2]\*v[2]))  
#define Vector3fDot(a,b) (a[0]\*b[0]+a[1]\*b[1]+a[2]\*b[2])  
  
static float st\_fFrustumPlane[6][4];  
  
// 프러스텀 평면을 만든다. 뷰와 프로젝트 매트릭스가 결정되면 한번만 생성한다.  
void MakeFrustumPlane(D3DXMATRIX \*pMatView,D3DXMATRIX \*pMatProj)  
{  
D3DXMATRIX view\_proj;  
  
D3DXMatrixMultiply(&view\_proj,pMatView,pMatProj);  
// left clip plane  
st\_fFrustumPlane[0][0] = -(view\_proj.\_14 + view\_proj.\_11);  
st\_fFrustumPlane[0][1] = -(view\_proj.\_24 + view\_proj.\_21);  
st\_fFrustumPlane[0][2] = -(view\_proj.\_34 + view\_proj.\_31);  
st\_fFrustumPlane[0][3] = -(view\_proj.\_44 + view\_proj.\_41);  
  
// right clip plane  
st\_fFrustumPlane[1][0] = -(view\_proj.\_14 - view\_proj.\_11);  
st\_fFrustumPlane[1][1] = -(view\_proj.\_24 - view\_proj.\_21);  
st\_fFrustumPlane[1][2] = -(view\_proj.\_34 - view\_proj.\_31);  
st\_fFrustumPlane[1][3] = -(view\_proj.\_44 - view\_proj.\_41);  
  
// top clip plane  
st\_fFrustumPlane[2][0] = -(view\_proj.\_14 - view\_proj.\_12);  
st\_fFrustumPlane[2][1] = -(view\_proj.\_24 - view\_proj.\_22);  
st\_fFrustumPlane[2][2] = -(view\_proj.\_34 - view\_proj.\_32);  
st\_fFrustumPlane[2][3] = -(view\_proj.\_44 - view\_proj.\_42);  
  
// bottom clip plane  
st\_fFrustumPlane[3][0] = -(view\_proj.\_14 + view\_proj.\_12);  
st\_fFrustumPlane[3][1] = -(view\_proj.\_24 + view\_proj.\_22);  
st\_fFrustumPlane[3][2] = -(view\_proj.\_34 + view\_proj.\_32);  
st\_fFrustumPlane[3][3] = -(view\_proj.\_44 + view\_proj.\_42);  
  
// near clip plane  
st\_fFrustumPlane[4][0] = -(view\_proj.\_14 + view\_proj.\_13);  
st\_fFrustumPlane[4][1] = -(view\_proj.\_24 + view\_proj.\_23);  
st\_fFrustumPlane[4][2] = -(view\_proj.\_34 + view\_proj.\_33);  
st\_fFrustumPlane[4][3] = -(view\_proj.\_44 + view\_proj.\_43);  
  
// far clip plane  
st\_fFrustumPlane[5][0] = -(view\_proj.\_14 - view\_proj.\_13);  
st\_fFrustumPlane[5][1] = -(view\_proj.\_24 - view\_proj.\_23);  
st\_fFrustumPlane[5][2] = -(view\_proj.\_34 - view\_proj.\_33);  
st\_fFrustumPlane[5][3] = -(view\_proj.\_44 - view\_proj.\_43);  
  
for(int i=0; i<6; i++)  
{  
float denom = 1.0f / Vector3fDist(st\_fFrustumPlane[i]);  
st\_fFrustumPlane[i][0] \*=denom;  
st\_fFrustumPlane[i][1] \*=denom;  
st\_fFrustumPlane[i][2] \*=denom;  
st\_fFrustumPlane[i][3] \*=denom;  
}  
}  
  
// 바운딩 박스가 프러스텀 평면 안에 존재하면 TRUE를, 리턴 바깥에 있다면 FALSE를 리턴  
BOOL IsBBoxInFrustumPlane( float bb\_min[3],float bb\_max[3] )  
{  
float near\_p[3];  
  
for( int i=0; i<6; i++ )  
{  
if( st\_fFrustumPlane[i][0] > 0 )  
{  
if( st\_fFrustumPlane[i][1] > 0 )  
{  
if( st\_fFrustumPlane[i][2] > 0 )  
{  
near\_p[0] = bb\_min[0];  
near\_p[1] = bb\_min[1];  
near\_p[2] = bb\_min[2];  
}  
else  
{  
near\_p[0] = bb\_min[0];  
near\_p[1] = bb\_min[1];  
near\_p[2] = bb\_max[2];  
}  
}  
else  
{  
if( st\_fFrustumPlane[i][2] > 0 )  
{  
near\_p[0] = bb\_min[0];  
near\_p[1] = bb\_max[1];  
near\_p[2] = bb\_min[2];  
}  
else  
{  
near\_p[0] = bb\_min[0];  
near\_p[1] = bb\_max[1];  
near\_p[2] = bb\_max[2];  
}  
}  
}  
else  
{  
if( st\_fFrustumPlane[i][1] > 0 )  
{  
if( st\_fFrustumPlane[i][2] > 0 )  
{  
near\_p[0] = bb\_max[0];  
near\_p[1] = bb\_min[1];  
near\_p[2] = bb\_min[2];  
}  
else  
{  
near\_p[0] = bb\_max[0];  
near\_p[1] = bb\_min[1];  
near\_p[2] = bb\_max[2];  
}  
}  
else  
{  
if( st\_fFrustumPlane[i][2] > 0 )  
{  
near\_p[0] = bb\_max[0];  
near\_p[1] = bb\_max[1];  
near\_p[2] = bb\_min[2];  
}  
else  
{  
near\_p[0] = bb\_max[0];  
near\_p[1] = bb\_max[1];  
near\_p[2] = bb\_max[2];  
}  
}  
}  
if( Vector3fDot(st\_fFrustumPlane[i],near\_p) + st\_fFrustumPlane[i][3] > 0 )  
return FALSE;  
}  
return TRUE;  
}

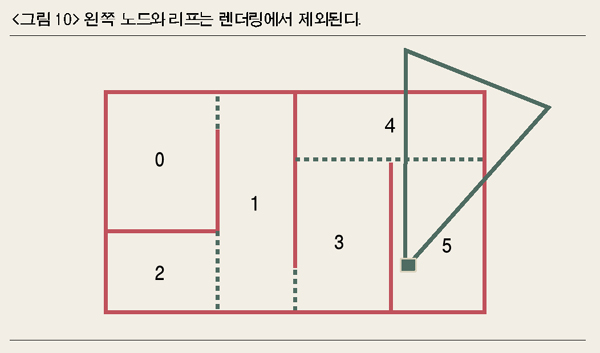
[**3D 엔진 제작의 실제-4**](http://webdizen.new21.net/blog/entry/3D-엔진-제작의-실제-4)

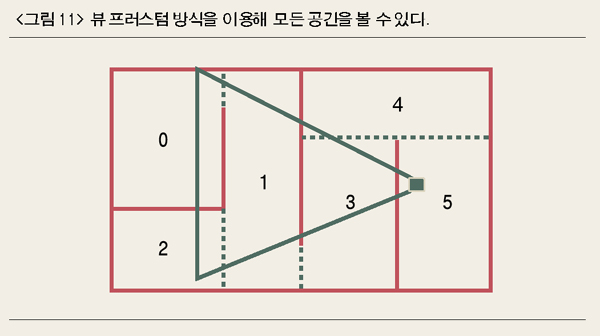
*장언일 (uijang@ccr.co.kr)*

처음 시작하는 독자라면 간단하고 보기 쉬운 것부터 직접 구현해 보는 것이 중요하다. 필요에 따라 다른 사람이 이미 구현한 소스가 상당한 도움이 되기도 한다. 복잡한 구조의 C++나 템플릿을 마구 남용해서 짜나가는 것보다 간단한 구조부터 복잡한 구조로 잘 짜여진 소스를 보는 것은 프로그래밍 습관에 많은 영향을 미치기 때문이다.

**PVS**  
퀘이크 I을 만들 당시 존 카멕이 좀더 렌더링 속도를 올리려고 고민을 하던 중 갑자기 “유레카”라고 소리지르고 이것을 만들었다는 일화가 있는 방법이다. 실제로 PVS(Potentially Visablity Set)가 원래 있던 것인지는 필자도 잘 모르겠지만 속도를 올리기 위해 군살을 없애는 방법이기 때문에 자신의 엔진에 이 PVS를 사용한다면 ‘온 세상이 아름다워’ 보일거라고 생각한다.  
BSP에서 원하는 영역을 트리를 타고 다니면서 렌더링하는 것은 트리가 깊어질수록 찾아내는 속도가 다소 걸리는 편이다. 물론 트리와 뷰 프러스텀만으로 렌더링은 가능하지만 속도에 관해서는 약간 매끄럽지 못하는 부분이 있기 때문에 그러한 해결책을 찾아보려고 했던 것으로 추측된다. PVS는 리프(플레이어가 있는 곳의 공간)마다 보여지고 있는 리프들의 look-up 테이블이라고 보면 간단하다.







<그림 9>처럼 0, 1, 2, 3, 4, 5 인덱스의 총 6개 리프가 있는 하나의 BSP 맵이 있다고 가정하면 플레이어는 0∼5번의 공간에 위치할 수 있기 때문에 그 공간에서 렌더링이 가능한 리프들을 테이블로 만드는 것이다. 즉 5번 리프에 플레이어가 위치했을 때는 0, 1, 2, 3은 벽에 의해 가려지기 때문에 렌더링할 필요가 없고 단지 4, 5번 리프만 렌더링하면 된다. 따라서 각각의 리프 테이블들은 0번 리프 테이블의 경우 0, 1로 1번은 0, 1, 2, 3으로 2번은 1, 2, 3으로 3번은 1, 2, 3, 4로 4번은 3, 4, 5로 5번은 4, 5의 테이블을 가지고 있는 것이 바로 PVS이다.  
<그림 10>을 보면 5번 리프에 플레이어가 위치해 있고 아래에서 오른쪽 위로 쳐다보고 있는 상황이라 뷰 프러스텀으로 노드와 리프의 바운딩 박스 체크를 해 나가면 왼쪽 노드와 리프는 렌더링에서 제외되므로 뷰 프러스텀 방식과 PVS 방식의 속도 차이는 별로 없다. 그렇지만 <그림 11>의 경우는 결국 뷰 프러스텀 방식만으로는 안 보이는 영역까지 렌더링을 하게 된다.   
이 PVS 테이블로 되어 있는 리프 중 보이는 방향에 따라 렌더링할 필요가 없는 리프도 있기 때문에 뷰 프러스텀으로 한번 더 걸러주고 렌더링하는 방법이 가장 깔끔한 최종 렌더링 방법이다. 보통 BSP의 리프 수는 맵 크기와 폴리곤 수에 따라 다르지만 비교적 큰 맵들은 수천 개 정도 되므로 PVS 테이블 크기가 엄청나게 된다. 4000개의 리프가 있고 각 리프당 평균적으로 500개의 리프를 가진다면, 그리고 2바이트의 리프 인덱스라고 해도 테이블의 크기는 4000×500×2 = 4MB이다. 그렇다면 렌더링을 해야 되는 리프들을 비트(bit)로 켜놓는 비트 플래그 방식으로 접근하면 (4000/8)×4000 = 2MB이다.  
그러나 이 비트 플래그는 평균적으로 약 400여 바이트(3500/8)가 0번이라는 값을 가지고 있기 때문에(보이는 것보다 안 보이는 리프가 더 많다) 이것을 예전의 스프라이트 0번 압축 방식과 비슷한 RLE 압축 방식(pcx 그림 파일 압축방식)으로 데이터를 압축해서 사용하면 용량이 상당히 줄어들게 된다. 이 RLE 방식으로 추가 적용하면 몇십 만 바이트로 그 큰 월드의 렌더링 데이터 테이블을 가질 수가 있다(퀘이크 I부터 사용한 방식이다).  
  
**R3 엔진에서 BSP 트리 변형**  
필자는 R3 엔진을 처음 설계할 당시 구조에 관해서 한참 생각을 했었다. BSP 트리를 쓰자니 퀘이크 엔진이나 언리얼 엔진처럼 맵 에디터를 만들어야 하고 야외처리 부분(terrain)도 따로 만들어서 접목시켜야 할 생각을 하니 할 일이 태산같았다. 더군더나 에디터를 만든다고 해도 그 에디터를 통해서 맵을 제작할 그래픽 아티스트나 게임 디자이너들이 걱정되었다.  
실제로 퀘이크 에디터나 언리얼 에디터는 프로그램적으로 처리되는 것을 신경써서 만들어야 똑같은 화면을 보여주는데 있어서도 제대로 속도가 나기 때문에 구조 설계에 있어서 그래픽적인 감각만으로는 배경을 만들 수가 없다. 그래픽을 하는 사람이 프로그램적으로 어떻게 처리되는지를 알아야 제대로 최적화된 배경을 만들 수 있기 때문이다. 이는 개발시 상당히 심각한 문제를 야기하기 때문에 어떻게 해야 할지 고민을 한창 했을 때였다. 평소 필자와 안면이 있는 개발자를 만나서 이런저런 얘기를 하던 중에 한 가지 좋은 힌트를 얻어 냈다. 그 개발자는 BSP에 대해서 잘 모르지만 BSP가 트리 구조로서 두 단계로 나누어지니까 x, y, z 평면을 순서대로 자르면 BSP가 되지 않겠느냐고 말했다(이러한 방법을 kd-tree라고 한다는 것을 나중에서야 알게 되었다).  
참 재미있는 발상이라고 생각하고 집에 와서 곰곰히 생각해보니 실질적으로 x, y, z를 순서대로 자른다면 옥트리와 같은 구성이지만 실질적으로 트리 자체는 BSP가 되기 때문에 접근하기가 무척 쉽고 공간구조 관리를 좀더 직관적으로 할 수 있다는 생각이 들었고, 더군더나 BSP를 사용하면서 게임 툴을 직접 안 만들고 그래픽 디자이너가 잘 사용하는 3DS-MAX를 최대한 활용할 수 있는 게 가능하다는 생각으로 중요한 고민들이 풀리기 시작했다.  
3DS-MAX를 툴로 사용하는 데 있어 장단점은 그래픽하는 사람들이라면 일단 누구나 쉽게 접근할 수 있다는 점이다. 3DS-MAX 자체를 일반 FPS(First Person Shooter) 에디터처럼 BSP를 뽑아내려면 여러 가지 작업조건(퀘이크 에디터나 언리얼 에디터를 이용해서 직접 만들어보라)들이 너무나 까다롭게 붙을 수밖에 없다. 이를 그래픽하는 사람들에게 일일이 가르쳐야 하며, 이렇게 하면 그래픽하는 사람들은 제한조건으로 인해 작업을 원할하게 할 수가 없어 작업 속도와 퀄리티에 상당한 영향을 받을 수밖에 없다.  
그렇다면 직접 에디터를 만들어야 하는데 에디터는 하루아침에 만들어지는 것도 아니고 에디터를 만드는데 아주 유능한 전담 프로그래머가 있어야 한다. 왜냐하면 전문 그래픽 툴로서 수년간 버전업을 하며 만들어진 3DS-MAX에 익숙하던 그래픽 작업자들이 직접 만든 툴로 작업하기엔 너무나 불편하기 때문에 그에 못지 않은 편한 작업 툴을 만드는 것은 너무나 많은 시간이 소요되기 때문이다(엔진보다 훨씬 더 많은 노력과 시간을 투자해야 한다). 어찌됐건 그 친구의 재미있는 발상을 계기로 필자가 중요하게 나아가야 할 부분에 대해 그렇게 결정했던 것이 3DS-MAX를 써서 BSP 구조를 가지며 배경 작업을 할 수 있는 R3 엔진의 가장 강력한 장점이 되었다.  
그러나 3DS-MAX는 전용 그래픽용 툴이기 때문에 게임상에서 필요한 내용들을 추가하기 위해서 해야 할 일들도 만만치가 않았다. 실제 R3 엔진은 월드를 만드는 것을 제외하고 기본적으로 게임에 필요한 요소들을 집어넣는 종합 툴을 비롯해서 기타 컨버터 등 유틸리티가 20개가 넘는다. 그렇지만 전용 툴을 만드는 것보다도 훨씬 프로그래밍 작업 시간과 그래픽 작업 시간들을 단축시킬 수 있는 개발 환경을 구축했기 때문에 상당한 이점이 있다.  
실제 필자는 BSP 장점 중 구조적인 관리 부분만을 사용하기 위해서 x, y, z 평면으로 자르는 순서를 edge가 긴 것부터 기준 삼아 분할해 나갔다. 따라서 길쭉한 월드의 경우 같은 평면을 더 자르는 방식으로 트리를 구성했다(가능하면 최종 리프는 정방형 비슷하게 하기 위해서…).  
그리고 한 가지 덧붙이자면 옥트리든 BSP 트리든 폴리곤을 자르는 게 정석이다. 그렇지만 필자의 예전 엔진에서 일정 격자로 구성해서 폴리곤을 잘랐을 때는 원본 폴리곤 개수에 비해 폴리곤을 분할함으로 인한 용량이 너무나도 비효율적으로 늘어났기 때문에 이번에는 자르지 않고 인덱싱 방식을 취하기로 했다. 왜냐하면 x, y, z 평면으로 자르는 방식은 공간을 효율적으로 분할하는 방법이 아니기 때문에 폴리곤을 분할했다면 경우에 따라 원래 폴리곤의 2∼3배 불어난 상태로 관리해야 된다는 엄청난 단점이 있기 때문이다.  
  
**다음에는 실전 코드를 중심으로**  
다른 BSP 문서에서 설명하던 패턴과는 다른 방식으로 몇 가지를 짚어서 이야기를 해봤는데 과연 이해하는 데 도움이 됐을까 싶다. 다음 호에는 머터리얼 블럭화와 BSP의 실전 코드를 중심으로 이야기해 보겠다.   
  
*정리 | 박은정 | whoami@korea.cnet.com*