Gamasutra.com

# 멀티스레드 지형 평활화 입문

Holger Gruen
2006년 5월 31일

http://gamasutra.com/features/20060531/gruen_01.shtml

## 서론

멀티코어 프로세서는 주류가 되고 있다. 현재 판매되는 PC 와 신형 게임 콘솔은 대부분 여러 하드웨어 스레드의 병렬 실행을 허용한다. 게임 프로그래머들로서는 이렇게 추가적인 전산 능력이 제공되는데도 사용하지 않는다면 정말 수치스러운 일이 될 것이다. 신형 콘솔은 순차처리를 실행하는(out-of-order) 코어를 보유하고 있는 것으로 보이지 않는다([Stokes05] 참조). 따라서 스레딩은 가용 실행 단위를 실제로 병행하여 사용할 수 있는 유일한 방법이다.

멀티스레드 게임 엔진은 그 작성이 까다로운 과제이다(예, [Gabb05] 참조). 렌더링이나 인공지능(AI) 조작, 물리학이나 충돌 탐지와 같은 다양한 게임 태스크는 다른 게임 태스크에서 얻는 결과의 순차적 가용성에 의존하는 경우가 많다.

게임 엔진 개발업자가 물리학이나 렌더링과 같이 다양한 게임 태스크를 병행하여(태스크 레벨 병행) 실행하기로 할 경우, 다른 태스크로부터 필요한 결과를 적시에 이용하는 것이 복잡해질 수 있다. 요컨대 동기화 비용이 크게 늘어나는 것은 바람직하지 않다. 또 다른 가능성은 데이터 병행 방식으로(예, 병행하는 4 개 문자에 대한 경로 확인) 동일한 유형의 여러 문제를 해결하는 것이다. 게임 엔진의 효율성과 확장성을 위해서는 태스크 레벨과

데이터 레벨 병행의 결합을 이용하여 엔진을 구현하는 것이 이상적이다. 이는 보다 많은 코어에 대한 확장성의 이용을 보장한다.

멀티스레드 소프트웨어 개발과 관련하여, 관련 개념 전체와 코딩상의 함정을 대부분 설명하고 있는 문서를 다운로드할 수 있다([DevMTApps] 참조). 또한 VTune*(핫스팟 확인용)과 ThreadCecker*(레이스 조건 및 기타 스레딩 버그 확인용), ThreadProfiler*(로드 균형 및 기타 성능 문제 확인용)와 같은 툴은 멀티 스레드 개발 노력을 돕는 귀중한 툴이라는 점에 유의해야 한다([IntelTools] 참조). 이러한 툴의 도움으로 PC 에 대해 해결하는 스레딩 문제는 다른 멀티 코어 플랫폼에 대해서도 도움이 될 수 있다.

[West06]은 게임 엔진의 다양한 코어를 활용하는 방법에 대한 아이디어를 제시하는 입문을 작성하였다. 본 논문은 그의 접근방식에서 한 걸음 나아가 실시간 지형 높이 필드 평활화(그림 1 참조)를 실현하기 위한 멀티스레딩의 활용을 설명하고 있다. 지형의 가시 부분은 실시간 프레임에서 평활화되어 가시적인 지오모핑(geo-morphing)이나 파핑(popping) 없이 부드러운 LOD 이행을 달성하고 있다. 또한 최적의 방식에 가깝게 지형에 대한 최고 예산을 지출할 수 있다는 점을 확인할 수 있다.

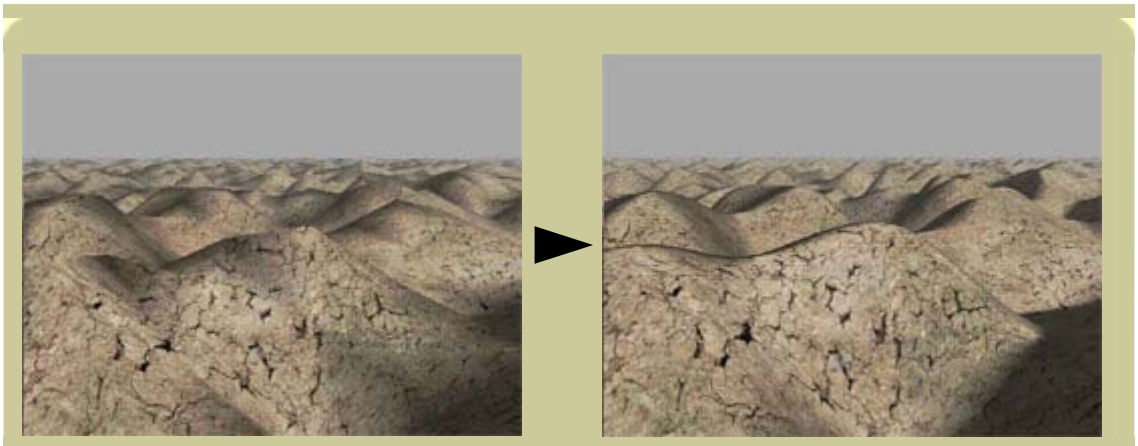

**Figure 1**

여기서 설명하고 있는 지형 평활화는 플로팅 포인터 렌더 타깃([Bunnell05] 참조)이나 'Tessellation through Instancing' ([Gruen05] 참조)을 이용하여 그래픽 카드에서도 실행할 수 있다. 또한 DirectX10*과 GeometryShaders 도 조만간 사용할 수 있을 것이다. 그래도, 대다수 게임은 해상도가 낮거나 중간 정도로 제한된 프로세서이기 때문에, 프로세서 태스크를 다른 코어에 오프로드하는 것이 도움이 될 수 있다. 또한 대다수 게임은 그래픽 카드가 높은 디스플레이 해상도로 제한된다. 따라서 디스플레이 해상도가 높을 경우, 다른 작업을 위하여 그래픽 카드 파워가 절실하기 필요할 것이기 때문에 그래픽 카드에서 로드를 옮기는 것이 합리적이다. 물론 이는 모두 프로세서와 그래픽 카드 작업 부하의

균형을 유지하는 방식에 따라 달라진다. 프리 DirectX10* 하드웨어에서 멀티플 패스로 렌더링을 할 경우, 프로세서에 대한 테셀레이션은 그래픽 카드에 여러 차례 렌더링하는 것보다 속도가 빨라질 수 있다.

데이터 병행 평활화를 실현하기 위한 OpenMP* ( [OpenMP] 참조)의 이용을 첫 번째 접근방식으로 연구하였다. OpenMP*를 이용하여 비동기식 스레딩을 실현하는 것은 더욱 어렵기 때문에, 비동기식 테셀레이션을 실현하기 위하여 렌더링을 수행할 때 새 지형 메시만 픽업하는 윈도우* 스레딩 API 를 이용하는 방식도 표시하고 있다. 두 번째 접근방식은 프레임 타임에 아무런 영향도 주지 않는다. 설명된 알고리듬은 다량의 코어와 스케일에 대해서도 균형잡힌 작업부하를 근사하게 생성할 수 있다. 이를 이용하여 한 개의 스레드나 자유 코어가 무거운 평활화 리프팅을 수행하거나 충분히 활용하지 않았다고 알고 있는 코어에 소규모 작업 부하를 분배하게끔 할 수 있다.

다음으로 본 논문은 Bezier 패치를 이용하여 희박한 지형 높이 필드를 전체적으로 부드러운 표면으로 바꾸는 방법을 논의하고 있다. 다음으로, 컬링과 LOD, 스레드 작업 부하 선택을 실현하는 방법과 대량의 방수 메시를 생성하는 방법을 설명하고 있다. 마지막으로 평활화 작업을 멀티스레드하는 방법에 대한 논의로 논문을 마무리한다.

## 지형 평활화

단순성을 이유로, 본 논문은 일차원 사각형 바이큐빅 Bezier 패치를 선정하여 희박한 높이 필드를 평활화하고 있다. 본 패치는 높이 필드 전반에 걸쳐 높이 값을 평활화하는데만 사용되며, 다른 좌표는 선형 내삽법에 의하여 생성한다. 아래 텍스트는 간결함을 위하여 독자가 Bezier 패치의 작동 방식을 숙지하고 있다고 추정한다. 제대로 작성된 입문을 확인하려면 [Farin96]을 참조한다.

그림 1 은 대형 높이 필드에 속하는 4x4 지역을 나타내고 있다. 이 지역의 좌측 및 하단 코너는 높이 필드 내 기억위치(x,y)(둘 다 정수값이다)에 있다. 가능 수직선으로 표시되는 높이값은 다르다는 것을 뜻하지만, 도면의 가독성을 높일 수 있는 것으로 나타난다. 높이 필드 중앙에 표시된 패치 p 는 높이 필드에 의해 정의되는 그리드 셀을 대표한다. P 의 그리드 셀은 *h(x+1,y+1)*, *h(x+2,y+1)*, *h(x+1,y+2)* and *h(x+2,y+2)*로 정의한다.

바이 큐빅 Bezier 로 p 를 교체하기 위하여, patchone 은 16 개 제어 높이를 정의해야 한다. 이러한 제어 높이는 *b00, b10, b20, b30, b01, b11, b21, b31, b02, b12, b22, b32, b03, b13, b23*, *b33* 으로 표시한다.

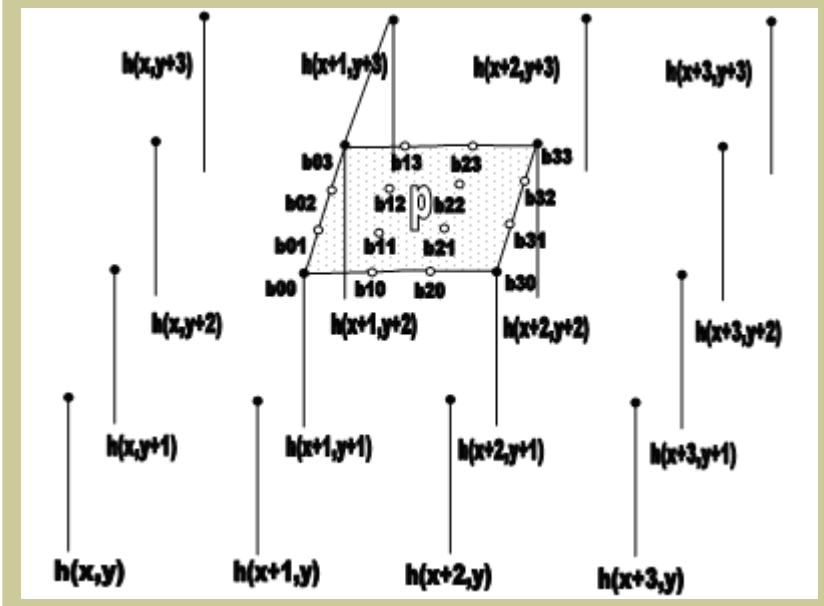


**Figure 2**

간결성을 위하여, 즉 본 논문은 멀티스레딩을 중점적으로 다루기 때문에, Formula 1 은 전체 C1 평탄 지형 표면을 달성하기 위하여 제어 높이를 정하는 알고리듬 하나만 제시한다.

$$b00 = h(x+1, y+1)$$
$$b30 = h(x+2, y+1)$$
$$b03 = h(x+1, y+2)$$
$$b33 = h(x+2, y+2)$$
$$b01 = b00 + \frac{1}{6}(h(x+1, y+2) - h(x+1, y)), b02 = b03 - \frac{1}{6}(h(x+1, y+3) - h(x+1, y+1))$$
$$b10 = b00 + \frac{1}{6}(h(x+2, y+1) - h(x, y+1)), b20 = b30 - \frac{1}{6}(h(x+3, y+1) - h(x+1, y+1))$$
$$b31 = b30 + \frac{1}{6}(h(x+2, y+2) - h(x+2, y)), b32 = b33 - \frac{1}{6}(h(x+2, y+3) - h(x+2, y+1))$$
$$b13 = b03 + \frac{1}{6}(h(x+2, y+2) - h(x, y+2)), b23 = b33 - \frac{1}{6}(h(x+3, y+2) - h(x+1, y+2))$$
$$b11 = b01 + b10 - b00$$
$$b21 = b31 + b20 - b30$$
$$b12 = b13 + b02 - b03$$
$$b22 = b23 + b32 - b33$$

**Formula 1**

표면을 평가하는 방법은 무엇인가? 각 패치는 각각 4 개의 제어 높이에 의해 정의되는 수직 Bezier 곡선 4 개를 정의한다는 것이 핵심 통찰력이다. 곡선들은 다음과 같다.

1. Curve 0 : *b00, b01, b02, b03 로 정의함*
2. Curve 1 : *b10, b11, b12, b13* 로 정의함
3. Curve 2 : *b20, b21, b22, b23 로 정의함*
4. Curve 3 : *b30, b31, b32, b33 로 정의함*

각 곡선은 제어 높이에서 유도할 수 있는 계수가 있는 3 차 다항식으로 작성할 수 있다([Foley90] 참조). 일정한 보폭으로 각 곡선을 통과하기 위해 전향 차분([Foley90] 참조) 을 이용할 수 있다. 본 논문을 위한 데모 구현은 SSE(예, [Klimovitski01] 참조) 컴파일러 내장함수를 이용하여 이를 수행하고 있다. 각 단계별로, 전향 차분은 4 개의 높이값을 제공한다. 이러한 4 개의 높이값은 다시 수평 방향으로 패치를 가로지르는 곡선을 정의하는 4 개의 제어 높이로 이용된다. 이번에도 전향 차분은 이러한 수평 곡선을 스텝시키는데 사용된다. 이번에는 SSE 를 이용하여 FED 단계의 3 개 애드(add)를 4 차원 애드와 셔플 연산으로 변형시킨다. 방향 미분계수는 이와 유사한 방식으로 스텝된다. 하지만 코드는 수직 방향으로 근접한 파생물만 생성한다는 점에 유의해야 한다.

## 컬링 및 LOD 선택

이번에는 현재 시각 절두체 내부에 있는 높이 필드의 셀을 신속하게 식별하는 효율적인 컬링 알고리듬이 있는 것으로 가정해 보자. 이러한 컬링 알고리듬의 결과는 가시성 있는 그리드 셀의 목록이다.

뷰어 근처에 있는 패치는 매우 상세하게, 뷰어에서 떨어져 있는 패치는 상세하지 않게 테셀레이트하는 것이 목표이다. 데모에서의 구현은 모든 그리드 셀에 대해 디테일 수준(LOD)을 선택한다. 이러한 LOD 값은 그리드셀의 중앙과 뷰어와의 거리를 토대로 하고 있다. LOD 는 FFD 알고리듬이 각 패치의 가장자리를 따라 생성하는 포인트 수를 정의하는 부동소수점으로 선택된다. LOD 의 비정수 부분은 D3D 가 곡선 표면 패치를 위한 세그먼트 수를 정의하는 것과 동일한 방식으로 정의된다([D3D05] 참조). 이는 개별 LOD 의 파핑이 없기 때문에 이동 카메라가 대단히 평탄한 시점 종속적 LOD 이행을 생성하도록 보장한다.

데모는 테셀레이트된 지형 전부에 대해 대형 삼각형 스트립 한 개를 생성한다. 또한 인접한 패치 사이의 간격을 스티치하는 별도의 삼각형들을 생성한다. 에지 별 LOD 를 [D3D05]로 이용하면 삼각형 스티칭이 필요없기 때문에 더욱 고급 프로그램이 될 수 있다. 전향 차분을 위한 설정은 이러한 방식을 손쉽게 코드화할 수 있기 때문에 퍼셀(per-cell) LOD 접근방식을 이용하였다.

평탄 수학과 컬링, 메시 생성이 포함되었기 때문에, 텍스트는 계속해서 실제 테셀레이션 작업을 수행하기 위한 다중 스레드를 위하여 균형 잡힌 작업부하를 생성하는 방법을 설명하고 있다.

## 멀티스레딩을 위한 균형잡힌 작업 부하 조성

컬링 알고리듬에 의하여 수집한 테셀레이션 계산결과를 하드웨어 스레드 조합에 골고루 분배하는 것이 목표이다. 고른 작업 부하를 분배하기 위해서는 특정 테셀레이션 태스크의 복잡성을 명시하는 특정 값이 필요하다. 다행히, 각 그리드 셀의 LOD 값은 해당 패치의 테셀레이션 비용을 설명하고 있다. 또한 LOD 값은 주로 대단히 제한된 범위에 속한다. 데모 구현에서 LOD 범위는 2.0 에서 30.0 에 달한다.

따라서 알고리듬은 동일한 정수 LOD 와 함께 모든 셀을 별도의 목록에 수집한 다음 테셀레이트할 패치를 버킷으로 분류하여, 동일한 테셀레이션 작업을 대략적으로 수행한다. 작업자 스레드 수를 감안하면, 코드는 모든 스레드에 대해 처음에는 비어 있는 작업부하 목록을 생성한다. 이제 라운드 로빈 형태로 작업 부하가 가장 복잡한 버킷에서 시작하여, 작업 부하를 스레드에 분배한다. 이러한 프로세스는 아래와 같은 의사 코드 Code Fragment 1 에 의하여 약술한다.

```
int maxworkload = 0;
int threadindex = 0;

set current bucket to buckets with highest LOD
patches;

while( current bucket not empty )
{
Remove patch from current bucket;
Add patch to workload of thread[ threadindex ];
if( workload of thread[ threadindex ] >
maxworkload )
{
   // update max workload
   maxworkload = workload of
```

```
thread[ threadindex ];

  // next thread in a round robin fashion
  threadindex = ( threadindex +  1 ) %
NumThreads;
}

if( current bucket empty )
{
    Choose next non empty bucket with smaller
workloads;
    If not found break;
}
}
```

**Code Fragment 1**

현재의 최대 작업 부하와 현재 스레드 작업 부하를 비교할 때 현재의 스레드 작업 부하를 평가하는데 사용되는 퍼 스레드(per thread) 값을 추가하여 이러한 알고리듬을 더욱 강화할 수 있다. 이를 통해 동일한 알고리듬으로 제어된 불균일 작업 부하의 생성이 가능하다.

모든 셀과 관련된 LOD 값을 통해, 특정 테셀레이션 작업 부하에 의하여 생성할 버텍스의 수를 산정하는 것이 대단히 쉬울 것이다. 다수의 스레드에 작업 부하를 오프로드할 때는 애플리케이션의 메인 스레드에 의하여 고정되었던 버텍스 버퍼의 시작 주소와 함께 각 스레드에 오프셋을 제공한다. 이를 통하여 스레드는 테셀레이션 결과를 비중복 메모리 영역에 작성할 수 있다.

지금까지 다수의 스레드에 대한 작업 부하 분배를 생성하는 방법을 예시하였다. 다음으로 이러한 워크로드를 픽업하는 멀티스레드 코드를 구성하는 방법에 대해 논하고자 한다. 먼저 OpenMP*의 사용을 논한 다음 Windows* 스레딩 API 의 사용을 설명하겠다.

## OpenMP*를 이용한 멀티스레드 테셀레이션

OpenMP*를 이용하여 특수 OpenMP* 컴파일러 지시어를 소스 코드에 추가함으로써 애플리케이션에 스레드 레벨 병행을 추가할 수 있다. 이러한 지시어는 초기 코드의 의미론을 변경하지 않는 프라그마 형태를 띠기 때문에 비침해적이다. 곧 확인하게 되겠지만, 이러한 지시어는 추가하기가 쉬우며, 이를 이용하여 코드의 병행을 증가시킬 수 있다. 물론 OpenMP*를 지원하는 컴파일러가 필요하지만, 다행히 VS2005*뿐만 아니라 Intel C/C++ 컴파일러도 이를 지원한다. OpenMP* 프라그마를 이용한 소스 코드는

마이크로소프트사*의 컴파일러도 이를 지원하기 때문에 Xbox360 포트할 수 있다. 하지만, 본 논문은 OpenMP*에 대한 입문서가 아니다. 이에 대한 심층적인 소개에 대해서는 [OpenMP]를 참조하기 바란다.

OpenMP*를 위한 주요 사용 모델은 포크 앤 조인(fork and join) 멀티스레딩으로, 이는 스레드 조합이 주요 실행 흐름에서 분기하여 공유된 태스크 조합에서 협력한다는 것을 뜻한다. 작업을 끝내면 다시 합류하는 것이다. OpenMP*는 내부 스레드 풀을 사용하기 때문에 스레드 제작이나 정리 비용이 없다.

멀티스레드 테셀레이션의 목적을 위하여 데이터 병행 OpenMP* 프라그마를 이용하여 *for()*-loop 를 병행한다. Code Fragment 2 에 의해 표시된 바와 같이, OpenMP*는 N 테셀레이션 작업 부하를 병행하여 작업하도록 지시된다. OpenMP* 실행시간은 이러한 작업을 수행하는데 사용할 스레드 수를 결정한다. 기본값은 기계가 지원하는 하드웨어 스레드 수이다. 하지만 OpenMP* 라이브러리 콜을 통하여 이를 변경할 수 있다.

```
void parTessellate( workload* wl )
{
  #pragma omp parallel for
  for (int i = 0; i < N; i++)
   {
    Tessellate( wl[ i ] );
   }
}
```

**Code Fragment 2**

그림 2 는 4 개의 하드웨어 스레드가 가능한 기계에서 일어날 수 있는 일을 보여주고 있다. MainThread 역시 다른 스레드와 협력한다는 점에 유의해야 한다. 또한, 메인 스레드는 컬링과 작업 부하 분배, 드로잉을 수행한다는 점도 유념한다.

데모 애플리케이션을 시작할 경우, 방금 설명한 바와 같이 테셀레이션 작업 부하의 데이터 병행 테셀레이션을 수행하기 위하여 OpenMP*를 이용하는 방식으로 시작한다. 데모에 포함된 슬라이더는 기계가 동시에 실행할 수 있는 최대 하드웨어 스레드를 이용하기 위하여 OpenMP*를 표현하는데 사용할 수 있다.

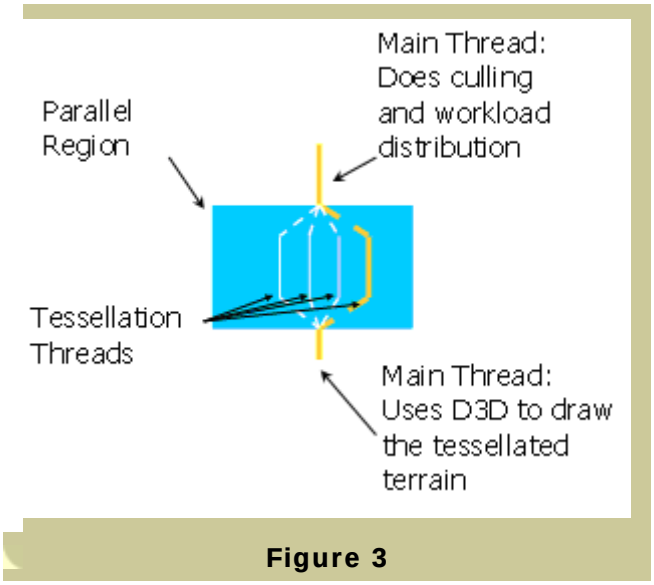


**Figure 3**

불행히도 모든 스레드에 대한 SSE 의 집중적인 사용은 하이퍼스레딩 시스템의 논리 프로세서를 모두 사용하기 때문에 큰 효과가 없으며, 감속을 초래할 수도 있다. 메인 스레드에 실행되는 D3D*와 그래픽 드라이버 역시 SSE 장치를 이용한다. 논리 프로세서를 모두 사용하여 속도를 늘리고 싶다면 SSE 장치를 전혀 사용하지 않는 테셀레이션 코드를 추가로 작성해야 한다. 데모는 선호도 마스크를 이용하여 HT 코어 논리 프로세서 두 대 가운데 한 대만 테셀레이션에 사용된다는 것을 확인할 수 있다(하기 내용 참조). 그래도 리얼 4 코어 기계를 입수한다면, 데모는 이의 사용을 허용한다.

데모가 실제로 듀얼 코어 머신에서 속도를 증가시킬 수 있다는 것을 입증하기 위해서는 아래 작업을 수행해야 한다.

1. 장치 설정이 수직 동기화가 오프 되었음을 나타내는지 확인한다.
2. 사용할 스레드 수를 1 개로 선택한다.
3. 'OpenMP 사용하기'를 표시한다.
4. 60FPS 로 내려갈 때까지 시청 거리를 늘린다. 그래픽 카드는 60 이상의 FPS 에서 초기 설정을 충분히 실행시킬 수 있을 만큼 빠르다고 가정한다.
5. 사용할 스레드 수를 2 개로 늘린다.
6. 투 코어 머신을 보유할 경우에 한하여 프레임 속도가 다시 올라가는 것을 확인해야 한다. 가속이 없거나 거의 없을 경우, 테셀레이션 작업 부하는 제한 요인이 되지 않는다. 이 경우 그래픽 카드가 변형되었거나 메모리(전달)이 제한적일 가능성이 높으며, 이는 렌더링에 상대적으로 비용이 많이 소요된다는 것을 뜻한다. 이를 점검하기 위해서는 "테셀레이션 실행"의 표시를 해제할 수 있다. 해제 이후, 카드가 테셀레이션으로 생성되는 버텍스 부하를 그리는 속도가 얼마나 빠른지 확인해야 한다.

증가 속도가 반드시 높은 것은 아니라는 것을 확인할 수 있을 것이다. 시스템과 그래픽 카드에 따라 프레임 속도는 60 에서 75 FPS 로 증가할 수 있으며, 속도는 25% 가량 증가하는 것이다. 이번에도 속도의 증가는 시스템이 테셀레이트된 화면을 표현하는 속도에 따라 결정된다. 테셀레이션 비용에 비해 표현 비용이 소액일 경우, OpenMP*를 이용하여 확보한 속도 증가는 높을 수 있다. 필자가 사용한 시험용 시스템은 속도가 50% 증가했다.

윈도우* 태스크 관리자를 도입할 경우, OpenMP*는 특정 코어나 프로세서에 대한 스레드를 고정시키기 위하여 선호도 마스크를 이용하지 않는 것이 분명해진다. 윈도우*는 코어 활용을 최소화하려고 노력하며 테셀레이션 스레드 스케줄을 재조정하는 것을 알 수 있다. 본 논문의 목적에 크게 나쁘지 않지만, 스레드를 특정 코어에 결합하는 것이 합리적일 수 있다.

특정 시스템에서 속도 증가가 높지 않은 이유(렌더링이 상대적으로 고비용일 경우)는 테셀레이션 작업에 소요되는 시간이 이러한 시스템에서 전체 프레임 프로세서 부하 가운데 상대적으로 적은 비율을 차지하기 때문이다. 컬링과 작업 부하 분배, 렌더링은 시간을 활용한다. 이것을 반드시 문제라고 할 수는 없으며, 실제로 Amdahl 의 법칙으로 예측할 수 있다([DevMTApps] 참조). 요컨대 이 법칙은 최대 병행 가속은 코드의 직렬 부분에 의해 제한을 받는다고 설명하고 있다. 렌더링은 한 개의 스레드에서만 수행되기 때문에 가속을 제한한다. 그래도 렌더링과 테셀레이션 작업을 분리하여 프레임 속도를 높일 수 있다. 이를 수행하는 방법은 아래에서 계속 살펴보기로 한다.

## 비동기식 멀티스레드 테셀레이션

테셀레이션에 비해 렌더링 비용이 높은 시스템에서 최대 프레임 속도에 도달하려면, 컬링과 테셀레이션에서 렌더링을 완전히 분리하는 것이 바람직하다. 기본 개념은 이를 수행할 때 신형 지형 테셀레이션만 픽업하는 것이다. 카메라 이동을 극복하기 위하여 확대 시각 절두체를 위한 삼각형 스트립을 생성할 수 있다. 데모는 이를 수행하지 않는다. 따라서 신속한 회전을 위해서는 단시간 동안 이용할 수 있는 지형이 없다는 사실을 알아야 한다.

데모에서 실현되는 비동기식 스레딩 아키텍처('OpenMP 사용하기' 표시를 해제할 경우 활성화됨)는 그림 3 에 표시하였다. 이러한 아키텍처를 위해서는 번갈아 사용되는 버텍스 버퍼가 2 개 필요하다. 버텍스 버퍼 한 대는 메인 스레드로 표현한다. 또 한 대의 버텍스 버퍼는 테셀레이션 스레드에 의하여 비동기식으로 충전된다. 새 테셀레이션이 제공될 경우 메인 스레드는 모든 프레임을 점검한다. 신형 테셀레이션이 제공될 경우, 그때부터 신형 버텍스 버퍼의 작성을 위하여 이를 사용한다. 그런 다음 다른(구형) 버텍스 버퍼를 고정시키고 이를 테셀레이션 스레드에 전달하여 충전시킨다. 이는 라운드 로빈 형태로 수행한다.

스레드의 동기화는 윈도우* 이벤트를 이용하여 처리한다. 하나의 이벤트는 테셀레이션을 새로 시작해야 하는 메인 테셀레이션 스레드를 알리는데 사용된다. 메인 테셀레이션 스레드는 다른 이벤트를 이용하여 신형 테셀레이션이 제공된다는 점을 메인 스레드에 알린다. 메인 테셀레이션 스레드 자체는 먼저 컬링과 작업 부하 분배를 수행한다. 그 이후, 스레드는 추가 테셀레이션 스레드를 개시할 이벤트 조합을 알린다. 즉, 시스템에 코어가 2 대 이상 있을 경우, 추가 테셀레이션 스레드는 메인 테셀레이션 스레드와 협력하여 테셀레이션을 마무리한다. 각 추가 테셀레이션 스레드는 메인 테셀레이션 스레드가 자체 이벤트를 설정하여 업무를 완료할 경우 이를 메인 테셀레이션 스레드에 알린다.

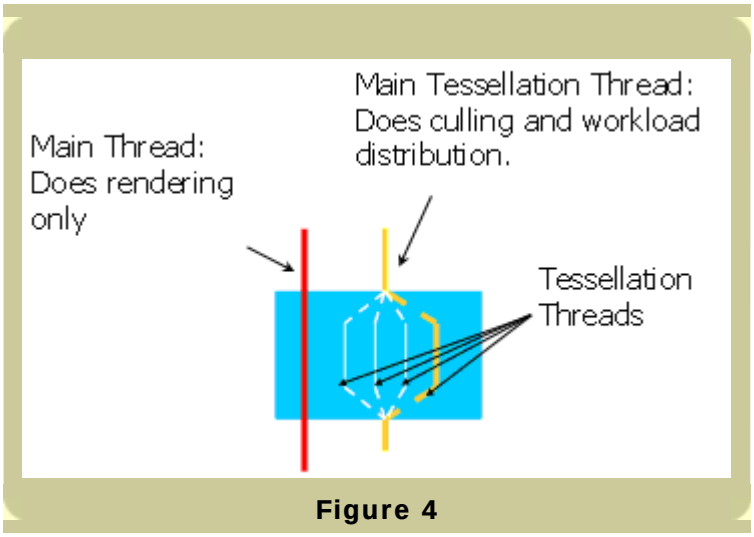


**Figure 4**

메인 테셀레이션 스레드는 메인 스레드에 신호를 보내기 전에 시블링들이 전부 마무리될 때까지 기다리기 위하여 *WaitForMultipleObjects()*를 수행한다.

데모 애플리케이션은 실제로 완전히 비동기식으로 실행되는 것이 아니라, 최종 프레임이 개시한 테실레이션 스레드에 의하여 최종 테실레이션이 수행될 때까지 메인 스레드가 대기한다. 흥미로운 것은 그래도 불완전한 지형의 프레임이 보인다는 점이다. 이는 메인 테실레이션 스레드가 실제 프레임을 그릴 때 메인 스레드가 이용하는 것과 동일하지 않은 컬링용 뷰콘을 픽업했기 때문이다. 한 프레임 래그를 인정하면 이를 수정할 수 있다.

이제 '테실레이션 대기' 표시를 해제하면 완전히 비동기식 모드로 이동할 수 있다. 이 경우 메인 스레드를 수행할 때는 신형 테실레이션만 사용한다.

테실레이션에 의해 사용되는 모든 스레드는 데모를 시작할 때 제작하기 때문에, 데모를 실행할 때는 스레드 제작이나 정리는 진행되지 않는다. 이에 덧붙여, 메인 스레드를 비롯하여 모든 스레드는 자신에게 적합한 선호도 마스크를 정하여 코어 가운데 정확하게 하나의 논리 프로세스를 선호할 수 있다. 이는 각 코어의 테실레이션에 소요되는 프로세스

시간을 정확하게 파악하기 위하여, 즉 윈도우*가 정말 선호도 마스크를 존경하는지 확인하기 위하여 윈도우* 태스크 관리자의 이용을 구현하기 위해 수행된 것이다.

'OpenMP 사용하기' 표시를 해제하고 비동기식으로 실행할 때 스레드 수에 맞는 슬라이더는 다양하게 사용된다. 슬라이더는 메인 테실레이션 스레드를 비롯하여 비동기식 테실레이션에 사용될 스레드 수를 지정한다. 투 코어 머신에서는 코어 한 대의 값만 남아야 한다.

OpenMP* 모드에 비해, 테실레이션 비용을 비교했을 때 렌더 비용이 높을 경우 동일한 시청 거리와 테실레이션 설정을 이용하면 프레임 속도가 높아지는 것을 확인할 수 있어야 한다. 테실레이션과 비교했을 때 렌더링이 저렴할 경우 OpenMP*에 비해 속도 증가가 작을 것이다. 사용자의 기계에 따라 이는 플레이어가 더 먼 곳을 보도록 허용하거나 테실레이션의 품질을 증가시킬 수 있다는 것을 뜻한다. '테실레이션 대기' 표시를 해제하면, 사용자가 확인하는 프레임 속도는 테실레이션 작업 부하의 복잡성과 무관해진다. '테실레이션 실행' 표시를 해제할 때 확인하는 속도와 동일해야 한다.

## 데모

데모의 소스 코드(그림 5 참조)는 다운로드할 수 있기 때문에 누구나 검토할 수 있다. 실행된 컬링 코드는 결코 최적이라고 할 수 없지만, 사용자 본인의 컬링 코드를 샘플에 붙여 넣을 수 있다.

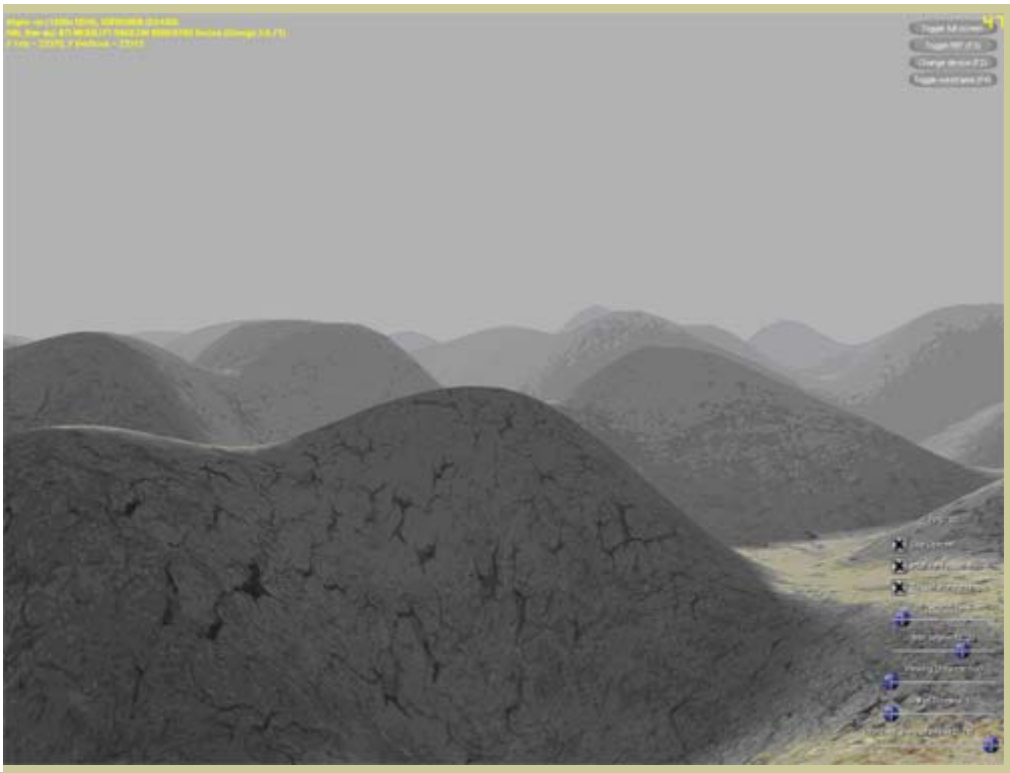

**그림 5 (download source)**

데모는 메모리의 높이 필드에서 패치 그리드를 사전에 산정한다. 작동 중에 합성되어 메모리에 전혀 어울리지 않는 높이 필드를 해결하기 위하여 코드를 변경하는 것은 쉽다. 또한 신형 콘솔의 벡터 단위에 적합한 코드로 SSE 내장함수를 포트하는 것도 쉬울 것이다.

테셀레이션 코드 외에, CPU 탐지를 구현하는 라이브러리용 소스 코드(필자의 동료 Leigh Davies 가 작성하였음)도 확인할 수 있다. CPU 탐지 라이브러리는 HT 코어인 논리 프로세서를 탐지할 수 있도록 하는 코어와 논리 프로세서를 열거한다. CPU 탐지 코드는 모든 IA32* 프로세서뿐만 아니라 인텔 프로세서에서도 작동하는 것으로 알고 있다.

## 결론

본 논문은 확장 방식으로 멀티스레드 지형 평활화를 수행하는 법에 대하여 기술하였다. 사용자가 코드 사용을 허용할 경우 테셀레이션 속도는 빨라질 것이다. 초기 성능 시험은 OpenMP* 코드 경로는 듀얼 코어 프로세서 시스템에서 초당 약 2000-4000 만 버텍스로 지형을 테셀레이트 및 표시할 수 있음을 나타내고 있다. 아울러 사용된 그래픽 카드는 동적 버텍스 버퍼에서 초당 약 7000 만 버텍스로 테셀레이트된 지형을 그릴 수 있다. 이를 통하여 역동적 지형 테셀레이션을 수행하고 성장하는 식물과 같이 기타 역동적인 구조 생성을 구현하기 위하여 추가 코어를 성공적으로 사용할 수 있다는 것을 알 수 있다. 모양새가 서로 다른 나무들이 우거진 숲을 상상해 보자. 아울러, 그래픽 카드에서 태스크를

오프로드하기 위하여 추가 코어를 사용할 수 있다는 점도 입증되었다. 그렇지 않을 경우, 그래픽 카드는 본연의 작업 이외에도 지형 테셀레이션을 수행해야 하기 때문이다. 신형 콘솔은 그래픽 카드에 보다 효율적으로 동적 기하학을 추가할 수 있는 방식을 보유하고 있는 것으로 보이기 때문에([Stokes05] 참조), 본 논문에 기술된 접근방식을 응용하면 대단히 성공적일 수 있을 것이다.

## 참고문헌


[DevMTApps] 'Developing Multithreaded Applications: A Platform Consistent Approach' online at http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/mrte/53797.htm?page=1

[D3D05] DirectX9c December SDK – Available online from www.microsoft.com

[Farin96] Farin, Gerald E. "Curves and Surfaces for Computer-Aided Geometric Design"Academic Press Inc. (London) Ltd (8. Oktober 1996)

[Foley90] Foley James D., van Dam Andries, Feiner. Steven K., Hughes John F. ,"Computer Graphics", Addison Wesley 1990

[Gruen05] – Gruen Holger, "Efficient Tessellation on the GPU through Instancing", Journal Of Game Development Volume 1, Issue 3, Thomson Delmar Learning, December 2005

[Bunnell05] Bunnell Michael, "Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping", GPU Gems II, Addison Wesley 2005

[IntelTools] http://www.intel.com/cd/software/products/asmo-na/eng/threading/219783.htm

[Gabb05] Gabb Henry, "Threading 3D Game Engine Basics" available online at http://www.gamasutra.com/features/20051117/gabb_01.shtml

[OpenMP] www.openmp.org

[Klimovitski01] Klimovitski Alex, "SSE/SSE2 Toolbox Solutions for Real-Life SIMD Problems", Game Developer Conference 2001, available online at http://www.gamasutra.com/features/gdcarchive/2001E/Alex_Klimovitski3.pdf

[Stokes05] Stokes Jon
‘Inside the Xbox 360, part I: procedural synthesis and dynamic worlds’ available online at http://arstechnica.com/articles/paedia/cpu/xbox360-1.ars
‘Inside the Xbox 360, Part II: the Xenon CPU’ available online at http://arstechnica.com/articles/paedia/cpu/xbox360-2.ars
‘Introducing the IBM/Sony/Toshiba Cell Processor — Part I: the SIMD processing units’ available online at http://arstechnica.com/articles/paedia/cpu/cell-1.ars
‘Introducing the IBM/Sony/Toshiba Cell Processor -- Part II: The Cell Architecture’ available online at http://arstechnica.com/articles/paedia/cpu/cell-2.ars

[West06] West Nick, “The Inner Product: Multi-core Processors” Game Developer Magazine Volume 13, Number2, February 2006