특집: 게임 태스크 스케줄링을 스스로 하라

Jérôme Muffat-Méridol

필자는 2008 년 생애 처음으로 데모 파티 Evoke in Germany 에 참석하게 되었다. 필자는 다음과 같은 질문이 제기되었을 때, 게임에서의 멀티코어 최적화 및 스레드에 작업을 효율적으로 전개하기 위해 Intel Threading Building Blocks(Intel TBB)를 사용하는 방법 대해 얘기했었다. 64K 에서 이것을 사용할 수 있나?

64K 데모에 대한 규칙은 "최대 65536 바이트, 자체 실행가능" 이렇게 간단하지만, 그 결과는 종종 신뢰할 수 없는 경우가 있다. Intel TBB 는 정말로 우아하고 슬림한 라이브러이지만 200KB 에서는 그렇지 않다. 그러나 필자는 이에 대해 '노(no)'라고 말하기가 싫다.

어쩔 수 없이, 필자는 Intel Threading Building Blocks 의 워킹 스케일 모델 종류에 대한 개념을 보완해야 했다. 이것은 최소 태스크 스케줄러라고 할 수 있는데 연구나 해체, 플레이를 쉽게 하는 것이라고 보면 된다. 필자에 임무가 주어졌다!

Nulstein 은 이 요구를 해결하기 위해 필자가 만든 데모이다. 이 데모는 대부분의 게임 플랫폼에 채택할 수 있는 태스크 스케줄링을 구현하기 위한 간단하지만 효과적인 방법을 보여준다. Nulstein 코드를 다운로드하려면 이 링크를 클릭.

태스크 스케줄링

태스크 스케줄러에 대해 잘 모르거나 태스크 스케줄러가 게임에서 왜 유용한지에 대해 잘 모른다면, 스레드와 태스크 간의 차이점에 그 실마리가 있다. 스레드는 가상의 무한 작업 스트림으로, 다른 스레드와 동기화해야 하는 경우 차단한다. 다른 한편으로 태스크는 다른 태스크와 독립적으로 수행되는 소량의 작업을 실행하는 짧은 작업 스트림으로, 차단하지는 않는다.

이러한 속성은 프로세서가 물리적 스레드를 실행할 수 있을 때 이와 동시에 많은 태스크를 실행할 수 있게 하며, 태스크 스케줄러의 작업은 주로 한 태스크가 끝나면 새 태스크를 찾는 것으로 귀착된다. 이것은 어느 한 태스크가 스스로 그 태스크의 일부로 또는 연속체로써 새 태스크를 생성할 수 있도록 추가할 때 상당히 강력해진다.

다소 작은 태스크들의 모음에서 작업을 분할한다는 것은 직접적인 개념으로, 스레드가 정상적으로 차단할 수 있지만 좀 더 어려울 수 있는 상황을 다룬다. 대부분의 경우 태스크는 예상 조건이 발생할 때까지 다른 태스크를 소비할 수 있으며, 그렇지 않은 경우 보통 대기점 주위의 2 개의 태스크에서 작업을 분할하고 *간접적으로* 동기화가 일어날 수 있게 하는 것은 간단한 문제이다. 그러나 우리는 나중에 다시 여기로 돌아와야 한다.

작업을 태스크로 분해하고 태스크 스틸링에 스케줄러를 사용하는 것은 멀티코어 프로세서를 사용하기 위한 편리하고 강력하며 효율적인 방법이다.

프로그래밍 관점에서 보면, *n* 로지컬 코어를 탑재한 시스템에서 Nulstein 은 *n-1* 워커 스레드를 만들어 태스크를 실행하는 게임의 메인 스레드를 지원한다. 각 워커는 실행할 준비가 된 태스크 목록인 자신의 "작업 더미"를 관리한다. 하나의 태스크가 끝날 때마다 워커는 해당 더미의 맨 위에서 다음 태스크를 고른다. 이와 유사하게 태스크가 만들어지면 그 만들어진 태스크는 더미의 맨 위에 바로 떨어진다.

이것은 하나의 전체 작업 대기열을 갖는 것 보다 훨씬 더 효율적이다. 각 스레드는 어떠한 분쟁도 없이 독립적으로 작업할 수 있기 때문이다. 그러나 캐시가 있다. 일부 더미는 다른 것보다 훨씬 더 빨리 비워진다. 이런 경우, 스케줄러는 바쁜(busy) 스레드 더미의 맨 아래 1/2 정도를 스틸하여 굶주린(starving) 스레드에 제공한다. 이것은 분쟁을 상당 수준 제한하는 것으로 드러났는데, 단 2 개의 스레드만이 이 작업을 수행하는 데 필요한 상호 배제에 의해 영향을 받기 때문이다.

태스크와 태스크 풀 개요

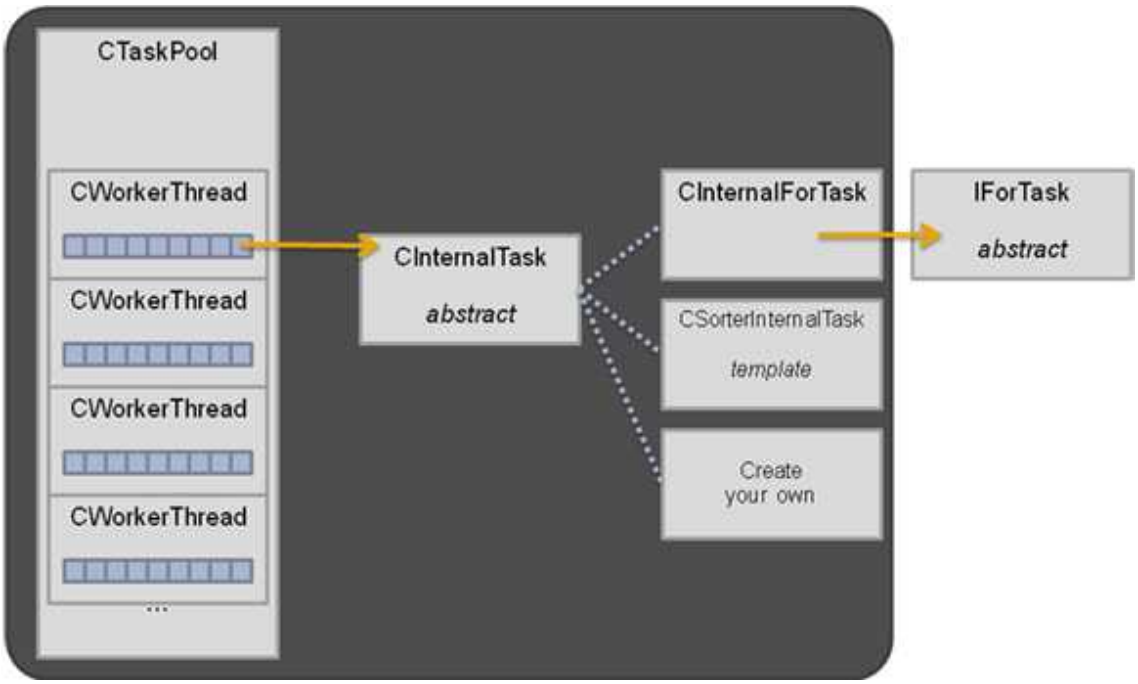

그림 1

태스크 엔진 코드는 TaskScheduler.h/.inl/.cpp(헤더, 인라인 및 코드)에 있다. CTaskPool 은 본래 CWorkerThread 의 모음으로, 여기에서 대량의 로직이 상주한다. CInternalTask 는 모든 태스크에 대해 추상 수퍼클래스인데, ParallelFor 및 CSorter 의 구현 시 이 클래스의 수퍼클래스를 사용하게 된다.

ParallelFor 는 가장 단순한 형태의 병렬 코드이다. 상호 독립적으로 반복을 실행할 수 있는 루프. 한 섹션을 처리하기 위한 범위와 방법이 주어지면, 사용가능한 스레드에 대해 작업이 전개되고 이 작업이 전체 범위를 커버하게 되면 ParallelFor 가 다시 돌아간다.

CSorter 는 단순한 병렬 병합 소트를 구현하여 새 태스크를 기존의 임계치 보다 더 큰 블록에 대해 생성한다. 목표는 비록 코드 크기를 줄이는 것이지만, 2 개의 아이템을 비교해야 할 때마다 항상 오버헤드가 함수를 호출하는 것을 막기 위해 이것은 C++ 템플리트로 실행된다.

아래에는 매우 편리한 효과가 나와 있다. 이러한 함수를 사용하는 코드는 여전히 시리얼 코드로 이해될 수 있다. ParallelFor 주위의 코드는 이 전과 후에, 그리고 이것이 읽히는 순간에 실행한다.

단순한 루핑 및 소팅 이상으로 사용하려면 자체적인 태스크를 생성해야 한다. 이것 역시 다음과 같이 매우 간단하다.

```
{
CTaskCompletion Flag;
CMyTask* pTask;

pTask = new CMyTask(&Flag,...);
pThread->PushTask(pTask);
...
pThread->WorkUntilDone(&Flag);
}
```

특정 태스크는 CMyTask 에 의해 구현되고 이것이 수행될 때 추적을 위해 Flag 를 사용한다. (pThread 가 현재 스레드여야 함을 기억할 것.) PushTask 가 호출되고 나면, 태스크는 스케줄러에 의해 실행되기 적합해지거나 혹은 다른 스레드에 의해 스틸될 수 있다. 현재 스레드는 WorkUntilDone 이 호출될 때까지 더 많은 태스크 푸싱을 포함한 다른 작업을 계속해서 수행할 수 있다. 이 마지막 호출은 완료 플래그가 설정될 때까지 스레드의 더미에서 태스크를 실행하거나, 다른 스레드로부터의 스틸을 시도할 것이다. 다시 한 번, 이것은 태스크가 호출의 일부분으로 직렬로 실행된 것처럼 보이게 된다.

```
{
CMyTask* pTask;

pTask = new CMyTask(pThread->m_pCurrentCompletion,...);
pThread->PushTask(pTask);
}
```

이 대체 형식에서, 태스크는 연속체로 만들어지며 기다리는 것이 무엇이건 간에 이 새 태스크를 기다리게 될 것이므로 완료되기를 기다리지 않아도 된다. 가능한 경우, 이것은 동기화 작업이 비교적 덜 필요하기 때문에 더 나은 접근방식이 될 수 있다.

이러한 기본 블록은 직렬로 읽히는 게임, 패션에서 사용되는 모든 종류의 병렬 알고리즘을 구현하기에 충분하다. 여전히 공유 데이터에 액세스하는 것을 걱정해야 하지만, 읽기 쉬운 상태로 남아 있는 일련의 단계로 작동하는 코드는 계속해서 쓸 수가 있다.

스케줄러의 내부

내부에서 어떤 일이 일어나고 있는지를 살펴보면, CTaskPool 이 중심 오브젝트임을 알 수 있는데, 이것이 워커 스레드를 만들고 보유한다. 처음에 이것들은 세마포어 상에서 차단된 대기(blocked waiting) 상태이며 스케줄러는 유휴상태로 CPU 를 소비하지 않는다. 첫 번째 태스크가 제출되고 나면 이 태스크는 모든 스레드 사이에서 분할되고(가능한 경우) 세마포어는 한 단계에서 *worker_count* 에 의해 상승되어 모든 스레드를 가능한 한 매우 가깝게 호출한다.

풀은 이 루트 태스크에 대한 완료 플래그를 계속 추적하고 워커들은 태스크가 설정될 때까지 계속 실행한다. 완료되고 나면, 모든 스레드는 다시 유효상태가 되고 개별 세마포어를 사용하여 새 태스크를 수락하기 전 모든 스레드가 유효상태로 돌아가는지 확인한다. 개념적으로 보면, 모든 워커들은 다음과 같이 항상 동일한 상태에 있다. 모두 유휴 또는 모두 실행.

CWorkerThread 의 역할은 태스크를 처리하는 것인데, 태스크를 처리, 대기열에 포함 및 스틸링하기 위해 분해될 수 있다.

처리 – threadproc 는 앞에서 언급된 세마포어를 처리하며 활성 상태일 때 DoWork(NULL)를 반복적으로 호출한다. 이 방법은 태스크가 더 이상 없을 때까지 더미로부터 태스크를 불러들인 다음, 다른 워커에서 스틸링을 시도하고 스틸할 것이 없는 경우 다시 돌아간다. DoWork 는 또한 태스크가 계속하기 전 다른 태스크를 기다려야 하는 경우 WorkUntilDone 에 의해 호출될 수 있는데, 이러한 경우 예상되는 완료 플래그는 매개변수로 통과되고 DoWork 는 이것이 설정된 것으로 확인되는 즉시 다시 돌아갑니다.

대기열 – 스틸링은 분쟁을 야기할 위험이 있다. 대기열에서의 작업은 락이 필요하므로, 몇 개의 명령만 보호하려면 스피닝 뮤텍스를 사용해야 한다. PushTask 는 태스크의 완료 플래그를 증가시키고 대기열의 맨 위에 태스크를 놓는다. 대기열이 꽉 찼을 때와 같은 특수한 경우, 올바른 결과가 생성되는 즉시 태스크를 실행해야 한다.

또한 대기열이 꽉 찬 경우 다른 워커들이 너무 비지하거나 스틸링 중임을 기억하는 것도 도움이 된다. 태스크는 또한 스틸할 것이 아무것도 없을 때 작업을 대기열에 포함시키는 지점이 없기 때문에 싱글코어 시스템의 특수한 경우에서 직접 실행된다. 그러면 전체 스케줄러는 우회되고 라이브러리의 오버헤드는 사라지게 된다.

스틸링 – StealTasks 는 스틸링을 처리하는데, GiveUpSomeWork 를 원하는 대상이 있는지 확인하는 다른 모든 워커에서 순환한다. 워커에 대기열에 있는 태스크가 단 1 개인 경우, 워커는 이 태스크를 2 개로 분할하여 "1/2 태스크"를 유휴상태의 스레드로 전송하려 할 것이다. 태스크를 분할하지 않았거나 2 개 이상 있는 경우 워커는 1/2 태스크를 반환한다(끌어모음). 워커들이 대기열이 빈 상태에서는 StealTasks 에서 회전하지 않는다는

사실은 워커가 반환되게 하여 태스크가 사용가능하게 되는 즉시 작업할 수 있게 한다. 이것은 대기 시간이 작업 처리량 보다 더 중요한 경향이 있는 게임의 상황에서 중요하다

스케줄러에 이보다 더 중요한 것은 없다. 나머지는 소스 코드에서 발견하기 위해 남겨지는 최고의 구현 디테일이다. 그러나 이 작업을 하기 전에, 여러분은 Nulstein 데모가 간접적인 동기화를 최대한 이용하고 대부분의 프레임을 병렬로 획득하기 위해 스케줄러를 어떻게 사용해야 하는가를 알아야 한다.

병렬 게임 루프

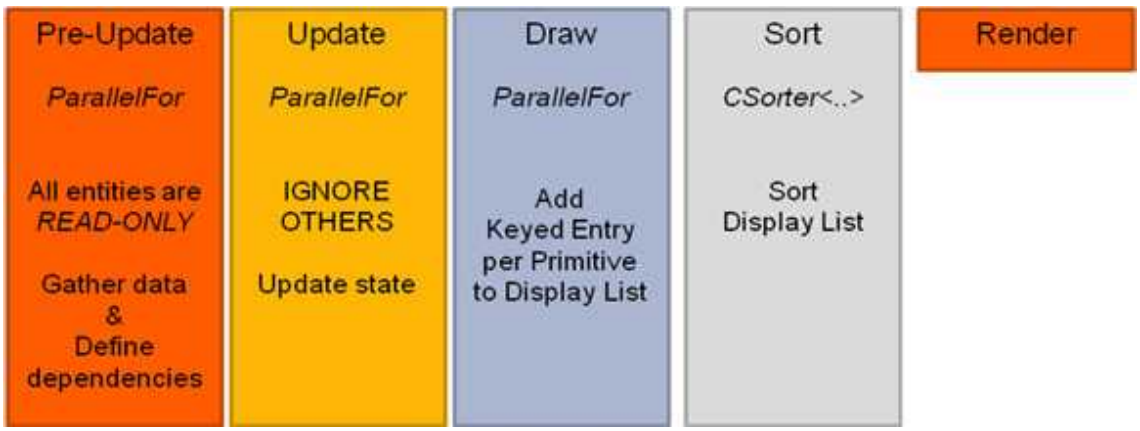

그림 2

프레임에는 전통적으로 2 개의 기본 단계가 있다. 바로 시간을 앞당기는 *업데이트*와 이미지를 만드는 *드로우*이다. Nulstein 에서, 이러한 단계는 병렬구조 획득을 위해 더 세분화되었다.

업데이트는 다시 2 개의 단계로 나뉘어 진다. 첫 번째는 *사전 업데이트* 단계로, 여기서는 모든 엔티티가 다른 것을 모두 읽을 수는 있지만 공개 상태를 수정할 수 없다. 이로 인해 모든 엔티티는 "이전 프레임"에서의 상태를 기준으로 의사결정을 내릴 수 있다. 그런 다음 엔티티는 두 번째 단계에서 변경사항을 적용하는데, 이것이 바로 실제 *업데이트*이다. 두 번째 단계의 규칙은 바로 엔티티가 자신의 상태를 쓸 수는 있지만 다른 엔티티에 액세스해서는 안 된다는 것이다.

이러한 규칙을 통해 이 두 단계는 모두 단순한 ParallelFor's 로 실행될 수 있으며 엔티티 간에 강한 의존성이 없는 한 쉽게 구현될 수 있으므로 이전 프레임의 상태를 사용할 수 없게 된다. 전통적인 예로는 다음과 같이 자동차에 부착된 카메라가 있다. 자동차 내부에서 뷰포트가 이동할 필요는 없으며 카메라를 업데이트하기 전에 정확한 위치와 방향을 알아야만 한다.

이러한 경우 엔티티가 다른 엔티티(또는 여러 엔티티)에 따라 자체적으로 선언할 수 있으며 다른 엔티티가 업데이트되어야지만 업데이트될 수 있다. 그리고 모두 알다시피 업데이트가 완료되면, 독립 오브젝트가 업데이트된 상태를 읽을 수 있게 된다. 데모에서, 이것은 소형 큐브가 더 큰 큐브의 코너에 단단히 부착된 채로 남아 있을 수 있도록 관리하는 방법이다.

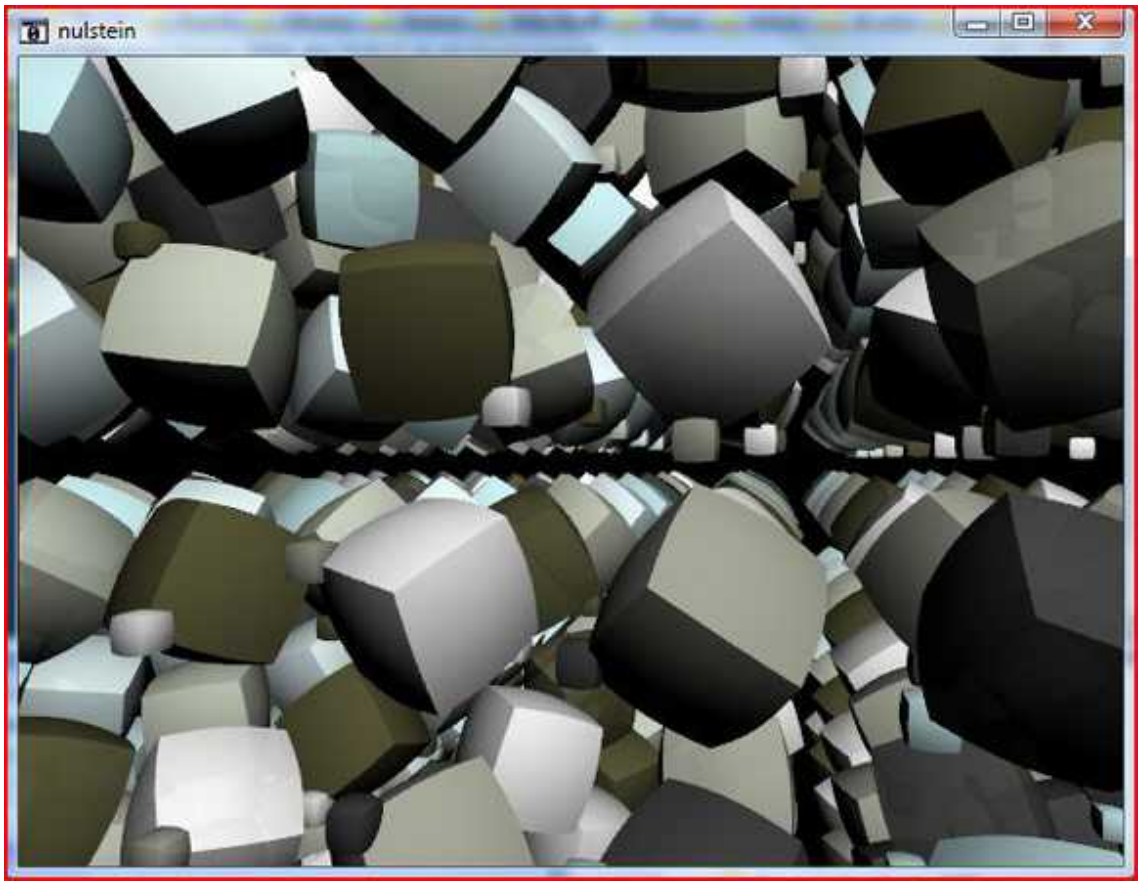

그림 3

드로우 단계는 세 단계로 나누어 진다. *드로우* 중에는, 모든 엔티티가 호출되어 렌더링해야 하는 항목을 나열하고 해당 항목을 표시 목록에 추가한다. 목록에는 ID 및 z-순서, 알파 블렌딩, 재료 등 기타 데이터를 암호화하는 64 비트 키가 포함됩니다.

이것은 ParallelFor 를 통해 수행되는데, 이때 각 스레드는 어레이의 독립형 섹션에 추가됩니다. 이 단계 중, 가시성 컬링 및 동적 버퍼 채움과 같은 작업은 병렬로 수행될 수 있습니다.

모든 엔티티가 드로우하고자 하는 것을 선언하고 나면, 어레이가 병렬로 수행될 수 있는 *소트*를 검토한다(대략 천 개 정도의 엔티티가 있다 하더라도 차이는 없음). 마지막으로,

씬이 렌더링되면 소팅된 목록의 각 항목이 실제 드로우가 호출하는 부모 엔티티를 다시 불러들인다.

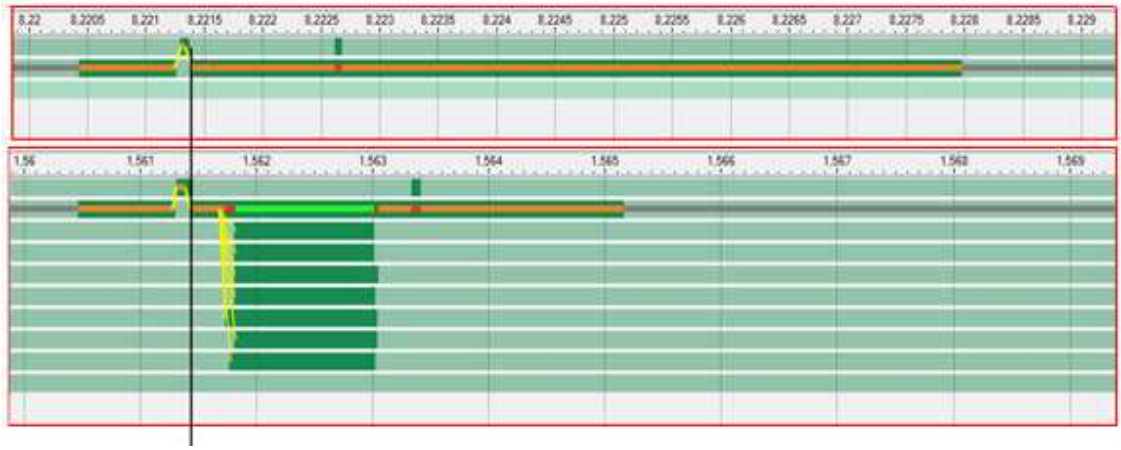

그림 3

그림 3 을 보면, 태스크 스케줄러가 on 및 off 인 상태에서 Intel Core i7 프로세서 3.2GHz 에서 동일한 스케일로 실행하는 릴리즈 빌드에 대한 2 개의 Intel Thread Profiler 캡처가 있다. 이것은 릴리즈 빌드이기 때문에, 주석은 없지만 태스크 스케줄러를 사용하는 이점이 명백하게 드러나며, 작업은 녹색줄로 표시된다(위쪽이 직렬 케이스, 아래쪽이 병렬 케이스). 직렬로 남아 있는 단계는 *렌더* 단계로 이는 주로 DirectX 와 그래픽 드라이버에서 소비되는데, 이 때 회색 라인은 vblank 를 위해 대기하는 데 소요되는 시간을 나타낸다.

아래 그림 4 는 프로필 모드의 데모를 보여주는 데, 이 모드는 실제 작업을 솔리드 섹션으로 보여주기 위해 인스트루멘트된다. 이를 통해 태스크가 모든 스레드에 대해 어떻게 전개되는가를 알 수 있다(타이밍은 정확하지 않음: 인스트루멘테이션은 회전 뮤텍스의 성능에 엄청난 영향을 미친다).

그림 4

결과적으로 이 프로젝트는 40K 이하에서 실행가능하며, kkrunchy by Farbrausch 와 같은 exe 패커를 사용하는 경우 실제로는 16K 까지 떨어진다. 따라서, 지금 필자에게 64K 에서 스틸링하는 데 태스크 스케줄러를 사용할 수 있는가를 질문한다면, 확실하게 예!라고 답할 수 있다.

필자는 우리가 무언가를 진정으로 이해하기 위해서는 실험이 필요하다고 생각하기 때문에 이 프로젝트가 병렬 프로그래밍에 관심있는 사람들이 잘 가지고 놀 수 있는 멋진 장난감을 제공하기를 희망한다.

(극심한 크기 제한이 다소 덜한 프로젝트라면, Intel Threading Building Blocks 가 훨씬 더 많은 최적화와 기능들을 제공해 주므로 필자는 이것으로 바꾸어 볼 것을 추천한다.)

참고 문헌

Reinders, James. Intel Threading Building Blocks. USA: O'Reilly Media, Inc., 2007.

Pietrek, Matt. Remove Fatty Deposits From Your Applications Using Our 32-Bit Liposuction Tools. Microsoft Systems Journal, October 1996 issue.

Ericson Christer. Order your graphics draw calls around! http://realtimecollisiondetection.net/blog/?p=86