※ 본 아티클은 CMP MEDIA LLC와의 라이선스 계약에 의해 국문으로 제공됩니다

Gamasutra.com

# 스킨 메쉬(Skinned Mesh) 추출 최적화

Ferns Paanakker
2007년 4월 18일

http://www.gamasutra.com/features/20070418/paanakker_01.shtml

현재의 게임 타이틀들은 모두 애니메이션이 적용된 캐릭터를 제작할 때 스킨 메쉬(Skinned mesh)를 사용한다. 스킨 메쉬는 캐릭터 애니메이션을 작성하고 렌더하는데 직관적인 방법을 제공하기 때문이다. 많은 하드웨어 플랫폼이 스킨 메쉬를 지원하지만(PC, Xbox, PSP 등), 동시에 사용 가능한 조인트(joint)의 개수가 차이가 나는 등, 플랫폼의 하드웨어 성능은 서로 다르다.

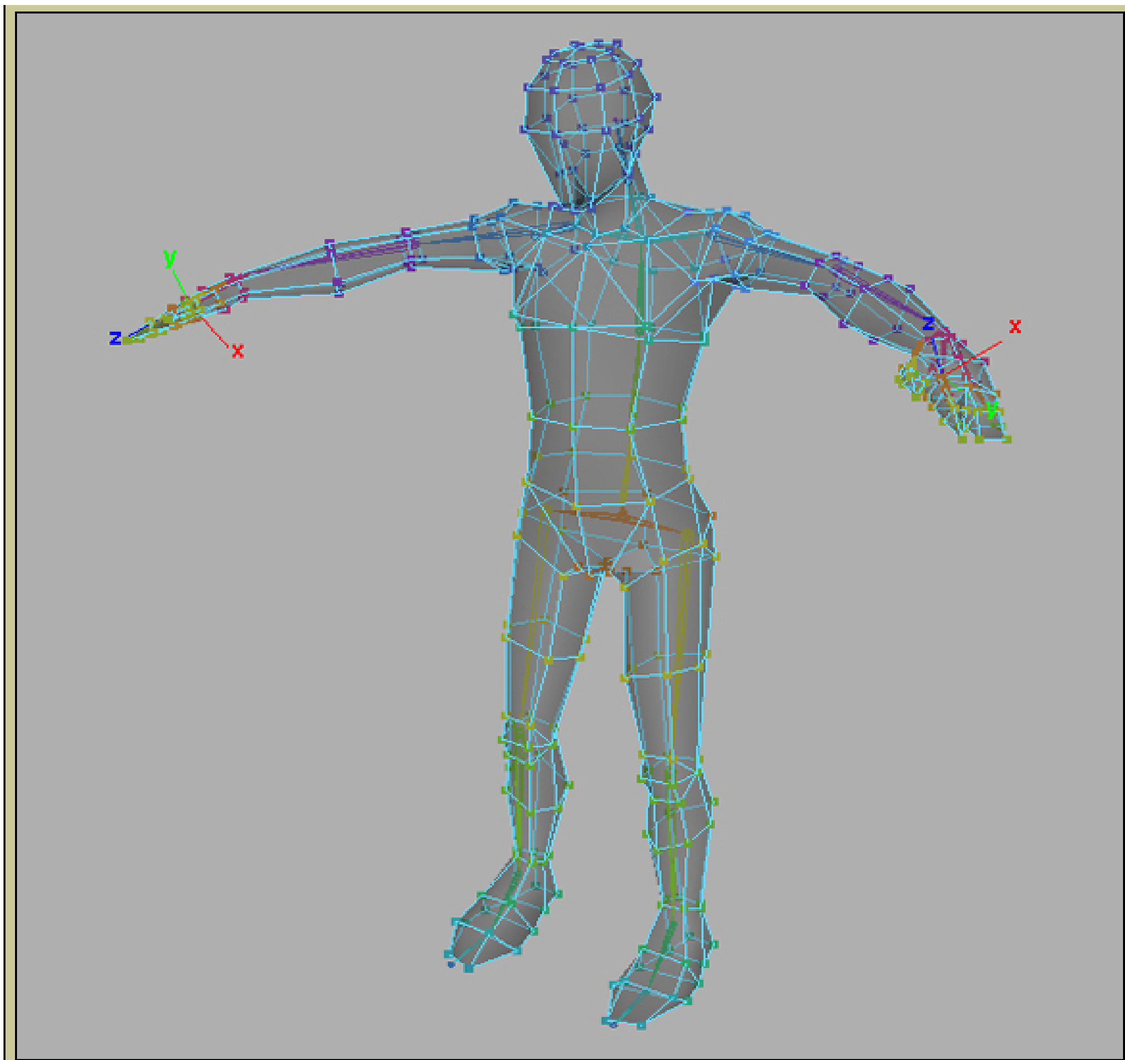
이러한 한계를 극복하기 위해서는 하드웨어가 직접적으로 지원하는 조인트 개수보다 많은 조인트를 사용하는 모델들을 여러 부위로 나눠서 모든 부위를 그래픽카드가 한 번에 처리할 수 있도록 하면 된다. 만약 이러한 알고리즘이 메쉬를 나누기 위해 사용된다면 비효율적인 모델 분할로 이어져 데이터 처리에 큰 영향을 끼칠 수 있으며 아티스트로 하여금 어쩔 수 없이 적은 조인트를 사용하거나 더 단순한 모델을 만들 수 밖에 없게 할 수 있다.(즉, 아티스트들이 작업의 대부분을 다시 해야 한다는 것이며 애니메이션의 질이 제한될 수도 있다는 것이다.)

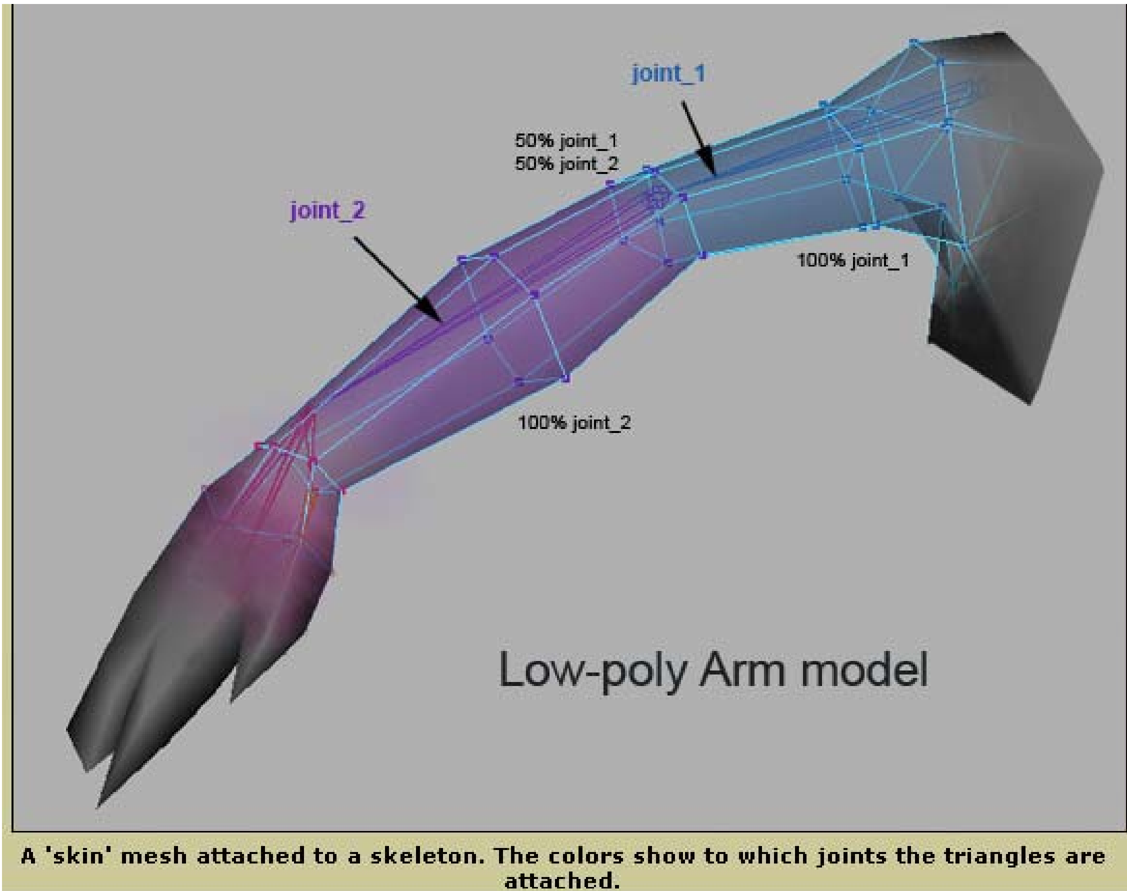
이 글에서는 무제한의 조인트를 다룰 수 있으며 동시에 파티션을 허용하여 최대한의 성능을 낼 수 있는 전처리(preprocessing) 소프트웨어의 구현에 대해 설명한다. 다수의 목표 플랫폼은 클래스를 만들어 목표 하드웨어의 구조를 시뮬레이트하며, 최적의 솔루션을

찾기 위해 귀납적 수단을 이용하는 방식을 통해 지원이 이루어진다. 이 소프트웨어는 훌륭하게 작동할 뿐만 아니라 우리의 툴체인(toolchain)에 매우 잘 맞으며, 프로그래머가 새로운 하드웨어를 위해 소프트웨어를 조정하고 기술을 최적화시키는 과정이 간편하다. 또한 게임 엔진 내의 모델을 (미리)보는 과정이 매우 빠르게 이루어지기에 아티스트의 작업시간을 줄일 수 있다.

**스킨 메쉬**

메쉬를 렌더링하는 과정에서 스키닝(skinning)을 하는 것은 오늘날 널리 쓰이는 방식이다. 이 기술에서 모델의 스켈레톤은 계층별로 나뉜 본으로 이루어져 있으며, 스켈레톤의 조인트는 메쉬의 정점에 붙어있게(attach) 된다.("스킨")





A 'skin' mesh attached to a skeleton. The colors show to which joints the triangles are attached.

메쉬를 애니메이션하기 위해서는 스켈레톤만 움직여주면 되는데, 왜냐하면 조인트의 움직임에 따라 거기에 붙어있는 메쉬의 정점이 변형되기 때문이다. 메쉬의 정점은 다수의 조인트에 붙이는 것이 가능하다. 정점은 최소한 하나의 조인트에 붙여줄 필요가 있으며(그렇지 않을 경우에는 다른 메쉬는 움직이는데 해당 정점은 그 자리에 그대로 고정되어 있을 것이다.), 오늘날의 실제 사람과 같은 캐릭터의 애니메이션을 위해서는 정점 당 최대 4 개의 조인트까지 붙여줄 필요가 있다. 정점마다 정확히 하나의 조인트만을 부여하는 것은 '리지드 스키닝(Rigid Skinning)'이라고 불린다. 정점에 다수의 조인트를 붙이는 것은 '스무스 스키닝(Smooth Skinning)'이라고 불리며, 정점에 얼마나 많은 개수의 조인트가 영향을 주는가를 지정할 수 있다.

PS2 타이틀은 하드웨어가 스무스 스키닝을 지원하지 않는 관계로 대개 리지드 스키닝 방식을 이용한다. PSP 는 스무스 스키닝을 하드웨어 단계에서 지원하는 관계로 PSP 타이틀은 이 기술을 사용한다. 캐릭터에 사용되는 조인트의 수는 게임에 필요한 디테일의

정도(level of detail)에 따라 달라지게 된다. PS2 와 PSP 타이틀에서는 대개 40 에서 50 조인트 정도가 사용되지만, 차세대 콘솔에서는 손가락, 머리카락, 옷 등도 애니메이션이 되어 있기에 조인트의 수가 100 개 또는 150 개도 쉽게 넘어서게 된다.

100 개의 조인트를 사용하는 캐릭터를 생각해보자. 이 스킨 메쉬를 렌더하기 위해서는 메모리상의 모든 조인트 행렬(joint matrix)를 불러오게 된다. 그러면 메쉬의 모든 정점을 돌아볼 수 있으며, 새 정점의 위치를 계산하거나 정점에 영향을 미치는 조인트 행렬을 살펴볼 수도 있다. 모든 정점의 위치를 정확하게 잡게 되면 전체 메쉬를 한 번(one pass)에 그릴 수 있다. 모든 스킨 프로세싱을 CPU 에서 다루게 되는 이러한 방식은 소프트웨어 스키닝이라고 한다. 이것은 간단한 솔루션이지만 스키닝 중에 CPU 가 다른 작업을 할 수 없다는 것과 새로운 정점을 만들면 GPU 로 보내기 전 위치를 정확히 잡아줘야 한다는 문제가 있다. 그렇지만 소프트웨어 스키닝은 사용할 수 있는 조인트의 수에 제한이 없으며, 메쉬에 사용되는 조인트가 50 개든 250 개든 관계 없이 쉽게 다룰 수 있다.

성능을 향상시키고 CPU의 부하를 줄이기 위해 많은 GPU가 하드웨어 스키닝을 지원한다. PC를 예로 들면, 스키닝은 정점 셰이더에서도 이루어지게 된다. 가장 유명한 스키닝 방식은 바로 행렬 팔레트 스키닝(Matrix Palette Skinning) [1]이다. 이 방식을 사용하면 조인트 행렬이 셰이더의 상수 테이블(constant table)에 저장되게 되며, 모든 정점은 조인트 행렬을 얻기 위해 상수 테이블 내에서 인덱스(색인)가 지정되게 된다. 그래픽 하드웨어에는 정점 셰이더에서 사용할 수 있는 상수 레지스터(constant register)의 최대 개수가 정해져 있기 때문에 한 번의 렌더 과정(render pass)에서 사용할 수 있는 조인트의 최대 개수가 영향을 받게 된다. 다른 행렬을 사용하기 위해서는 셰이더에 새로운 상수 레지스터를 보낼 필요가 있다. 상수 레지스터의 수는 대개 정점 셰이더가 필요로 하는 일반 상수인 카메라, 광원 정보, 시간, 바람/안개 값 등의 사항이 예약되어있다. 그 외의 상수는 조인트 행렬에 사용이 가능하다. 조인트 행렬은 4 x 4 행렬이지만, 마지막 행이 항상 [0 0 0 1]T 이기 때문에 필요한 것은 오로지 4 x 3 행렬에 지나지 않으며, 이로써 셰이더 패스당 33%의 추가 조인트를 사용할 수 있다.

현존 하드웨어(정점 셰이더 2.0 을 사용하는 것들)에서 상수 테이블은 256 개의 4 차원 벡터를 포함한다. 다른 파라메터를 위해 사용되는 상수의 수에 따라 이것의 양은 70~80 개 정도의 조인트로 이루어진다. 오래된 그래픽 카드의 상수 테이블은 오로지 98 개의 4 차원 벡터만을 포함하고 있으며 조인트 역시 20 개에서 28 개로 제한되어 있다.(그렇지만 이론적으로는 96/3=32 가 되어 32 조인트를 사용할 수 있다.)

만약 메쉬에 사용된 조인트의 수가 정점 셰이더의 조인트 제한을 넘어선다면 메쉬를 분할하여 각 부분들을 개별적으로 렌더할 필요가 있다. 메쉬를 나눌 때는 대개 최소한의 셰이더 패스를 사용하는데, 왜냐하면 셰이더에 상수를 올리는 것은 값비싼 연산이기 때문이다.

휴대용 기기인 PlayStation Portable(PSP)은 PC 와 비교하면 전혀 다른 구조를 띄고 있다. 하드웨어 자체가 PC 와는 매우 다르기 때문이다. 즉, 만약 모델에 많은 양의 조인트를 사용하고 싶다면 메쉬를 분할하는데 다른 알고리즘이 필요하다는 것이다.

제멋대로인 메쉬를 최적화하여 분할하는 것은 NP-complete 문제(실제 크기의 메쉬로는 해결할 수 없다는 것이다.)이지만 귀납적 접근 방법을 통해 실제로 훌륭하게 실행되는 솔루션을 만들어낼 수 있다. 그렇지만 이러한 툴을 만들기 전에 툴 체인에서 이 알고리즘을 어디에 위치시킬 것인지에 대해 결정할 필요가 있다.

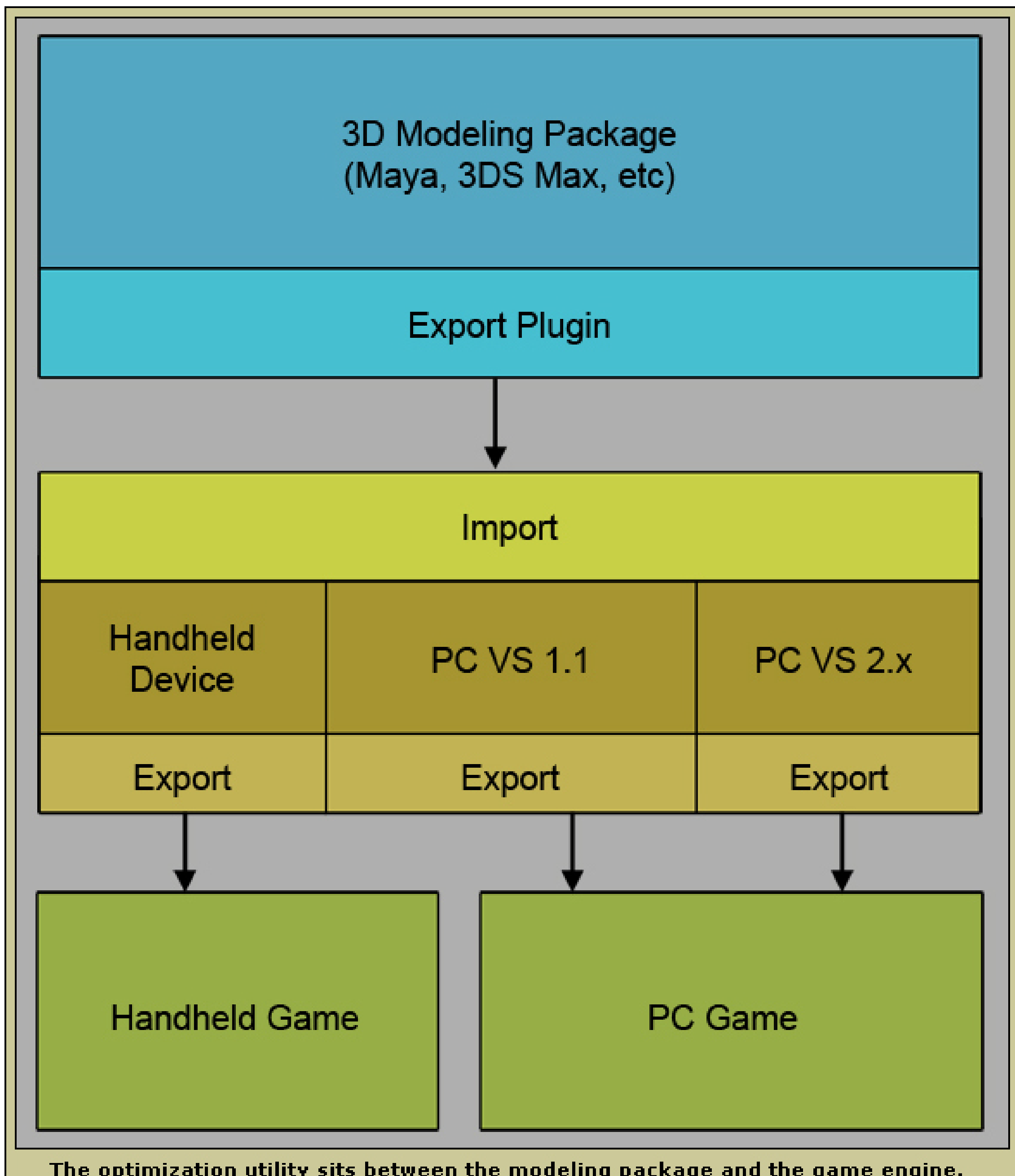
## 스킨 메쉬 추출하기

스킨 메쉬는 일반적으로 아티스트에 의해 Maya 나 3D Studio Max 같은 3D 모델링 패키지를 통해 만들어져서 게임이 인식할 수 있는 포맷으로 추출되게 된다. Wishbone Games 에서 우리는 Maya 를 PC 관련 하드웨어에 알맞게 제작된 서드파티의 플러그인과 함께 사용했다. 그러나 PSP 프로젝트를 시작한 후에는 PSP 하드웨어에 맞는 변화를 줄 필요가 있음을 깨달았다. 우리는 PSP 분야를 관리할 수 있는 개별

유틸리티를 만들기로 결정했고, 이것은 모델링 패키지와 게임 엔진의 사이에 위치하게 되었다.(그림 2)

이 툴을 제작에 있어 중요 디자인 사항은 아래와 같다.

1. 여러 하드웨어 플랫폼을 하나의 동일한 유틸리티를 이용하여 지원할 수 있어야 한다.
2. 새로운 하드웨어나 기존 하드웨어의 발전에 맞춰 확장이 쉬워야 한다.
3. 서로 다른 속도를 내는 다양한 최적화 기법이 있어야 한다. 빠르게 작업을 확인할 수 있는 것은 아티스트에게 있어 매우 중요하지만, 최종 추출 작업이나 야간 빌드는 성능 향상을 위해 메쉬의 최적화 작업에 더 많은 시간을 투여해도 문제가 없다.

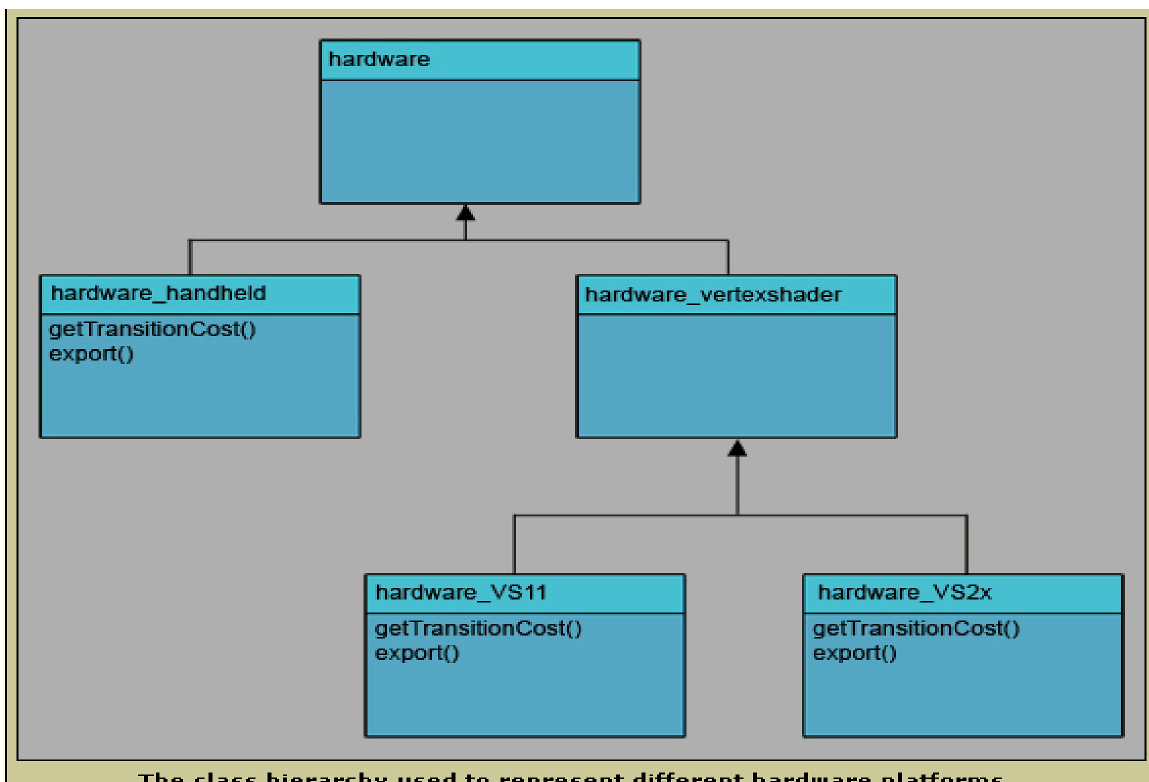


The optimization utility sits between the modeling package and the game engine.

여러 하드웨어 플랫폼을 지원하는 경우에는 일정한 툴체인을 지니는 것이 도움이 된다. 그렇기에 모델링 패키지에서 작업물을 추출한 후에는 언제나 최적화 프로그램을 거칠 필요가 있다. 일부 목표 하드웨어는 이 프로그램의 사용이 다른 하드웨어보다 어려울 수 있다. 그렇지만 최적화된 메쉬를 만들어낼 수 있는 변환 유틸리티를 모든 플랫폼에서 사용할 수 있으며, 목표가 되는 특정 하드웨어 구조의 이점을 취할 수 있다면 이상적일 것이다. 하드웨어 플랫폼을 위한

기능(입력 파일 포맷을 읽는 것 등)은 전부 공유되게 되며, 모든 기능을 하나의 유틸리티에 넣는 것은 소프트웨어의 빌드에 따른 불일치를 피할 수 있게 해준다.

디자인에서 고려해야 할 두 번째 사항은 바로 확장성이다. 새로운 하드웨어가 출시되면 그 하드웨어는 고유의 속성을 지니기 마련이다. 어쩌면 전체 셰이더 상수를 올리는 것이 더 쉬워졌을지도 모르며, 새로운 기능이 추가되었을 수도 있다. 이러한 새 하드웨어는 메쉬를 렌더하는 방식을 완전히 바꾸도록 요구할지도 모른다. 프로그래머는 필요할 때에 이런 새 하드웨어로 추출(export)할 수 있는 새로운 방법을 빨리 찾아낼 수 있어야만 한다. 게임 개발 과정에서 기존의 방식이 아닌 새로운 방식을 도입하는 일은 종종 있는 일이다. 몇몇 하드웨어를 아우르며 작업하게 되는 경우에는 특정 플랫폼에서는 무엇이 잘 작동하는지에 대한 새로운 통찰력을 지니게 될 수도 있으며, 흥미로운 글을 읽고 새로운 기법을 시험해보고자 하는 마음이 들지도 모른다. 이것은 새로운 서브클래스를 만드는 것만으로 쉽게 해낼 수 있다. 이것이 어떤 구조로 이루어지는지에 대해서는 3 번 그림에서 볼 수 있다.



The class hierarchy used to represent different hardware platforms.

세 번째 디자인 고려 사항은 바로 속도이다. 모델을 제작할 때 아티스트들은 작업물을 빠르게 (미리)볼 수 있어야 한다. 프로그래머가 빠른 컴파일 속도를 원하듯, 아티스트 역시 빠른 추출 속도가 필요하다. 아티스트가 캐릭터 모델을 작업하고 있는 도중이라면 렌더링 속도보다는 실행 속도가 더욱 중요한 요소라고 할 수 있다. 여러 종류의 최적화 기법을 지원하면 최적화에 걸리는 시간을 조절 할 수 있게 된다. 완전히 최적화된 모델은 야간 빌드나 최종 추출 작업에서 만들어야 할 것이다.

## 솔루션 작성하기

목표 플랫폼(Xbox, PC, PSP, 그 외), 최적화 옵션(질 중심 또는 속도 중심), 입력 파일과 출력 파일 등을 나타내는 커맨드 라인 옵션이 존재하는 개별 유틸리티 프로그램이 스킨 분할을 지원하기 위해 사용된다. 이 프로그램은 널리 알려진 OBJ 파일 포맷과 흡사한 ASCII 텍스트 파일을 읽어 들인 후 메쉬를 분할하고 출력 파일에 새로운 메쉬를 작성한다. 이 출력 파일에는 하드웨어에서 작동되어야 할 커맨드의 목록 등이 포함되어 있다. 하드웨어에 따라서 출력 파일은 달라질 수 있다.

이 데이터들은 메모리에서 읽혀지게 되며 목표 하드웨어와 독립적으로 진행되게 된다. 모든 정점은 최대 4 개의 조인트를 지니며 트라이앵글(삼각형)은 정확히 3 개의 정점을 지닌다. 즉, 최악의 상황에는 단순한 삼각형을 렌더하기 위해 12 개의 개별 조인트가 필요하다는 것이다. 렌더할 수 있는 최소 크기의 물체가 삼격형인 관계로 렌더를 할 때 최대 12 개의 조인트를 모두 사용할 수 있도록 하고 싶을 것이다. 삼각형에 사용되는 개별 조인트의 목록을 만들 때 오름차순으로 조인트 인덱스(색인)을 작성하여 "5 2 4"는 "2 4 5"와 동일한 것이 되게 된다. 이것인 메쉬 내의 트라이앵글 중 다수가 동일한 조인트 인덱스를 지닌다는 것과 흡사하며, 이러한 트라이앵글 그룹(triangle-group)은 한 번(one pass)에 렌더할 수 있다. 이러한 트라이앵글 그룹은 파티션이라고 부른다.

이러한 문제를 해결하게 되면 이 파티션들을 GPU 로 보내어 최대의 성능을 끌어내는 것이 가능해진다. 바로 이 부분에서 하드웨어 구조의 표현이 빛을 발하게 된다. 우선은 목표 하드웨어의 관련 분야를 시뮬레이트하는 오브젝트를 만든다. PC 에서는 이 오브젝트는 제한된

수의 상수 레지스터를 지닌 정점 셰이더를 표현한다. 파티션을 렌더하는데 드는 비용을 측정하기 위해 하드웨어 오브젝트에 쿼리를 보낼 수 있다.

## 간단한 알고리즘

솔루션을 내기 위한 매우 간단한 방법은 바로 파티션 목록을 죽 살펴보며 이것을 솔루션에 추가하는 것이다. 이러한 방식을 통해 파티션의 초기 순서가 솔루션의 질을 결정하게 된다. 만약 파티션 목록이 완전하지 않다면 실망스러운 렌더링 속도로 이어질 수 있는 불필요한 작업을 해야 할지도 모른다.

파티션의 목록을 나열함으로써 결과물을 크게 향상시키는 것이 가능하다. 사용된 고유 조인트의 수('1','2','3','1 2','2 3') 또는 오름차순으로 정렬('1', '1 2', '2', '2 3', '3') 하는 것이 일반적이다. 여기에서 하드웨어 표현(hardware representation)을 사용하지 않고, 대신 지식에 기반하여 추측하고 실행한다는 것에 유의하라. 이 알고리즘의 가장 큰 장점은 속도에 있다. 파티션을 렌더하기 위해 하드웨어를 갱신하는데 필요한 자원(relative cost)은 들지 않고서, 알고리즘은 그저 정렬 순서가 옳다고 가정하고 받아들이게 된다. 이러한 가정은 동시에 단점이 되기도 하는데, 왜냐하면 솔루션의 질이 하드코드되어 정렬된 입력 데이터에만 의존하게 되기 때문이다. 부주의한(또는 나쁜) 정렬은 하위 최적화 솔루션(sub-optimal solution)을 발생시킬 수도 있다. 이 단순한 알고리즘은 쉽게 적용이 가능하며, 매우 빠른데다가(검색이 없다.) 만약 프로그래머가 훌륭하게 정렬하기만 한다면 실제로 좋은 솔루션이 될 수 있다.(앞서 언급된 2 개의 정렬순서 중 하나) 많은 스킨 분할 알고리즘이 이 기술을 사용한다. 이 알고리즘은 의사코드(pseudocode) 1 에서 볼 수 있다.

```
FUNCTION SimpleAlgorithm PARAM partitionList RETURNING solution
      CALL partitionList.Sort
      INIT solution
      FOR each partition in the partitionList
            CALL solution.Append with partition
      END FOR
END FUNCTION
```

최저가 삽입(Cheapest Insertion)

파티션들을 하드웨어로 전달하기 위한 순서를 제공하는 대신, 하드웨어에 쿼리를 보내 좋은 솔루션을 찾을 수도 있다. 일반적으로 우리는 하드웨어가 특정한 상태에 있을 때 파티션을 렌더링하는 것이 어떤 효과를 미치는지 알기를 원한다. 각 삽입의 영향을 최소화는 것을 *최저가 삽입(cheapest insertion)*이라고 부른다. 순수한 구현은 더 많은 시간이 걸리지만 일반적으로 간단한 알고리즘에 비해 솔루션의 질이 훨씬 뛰어나다.

최저가 삽입의 알고리즘은 파티션의 전체 목록을 살펴보고 모든 파티션을 솔루션 내의 최저가 위치에 삽입한다.("간단한 알고리즘(simple algorithm)"처럼 여기서 끝나진 않는다.) 파티션을 솔루션에 삽입하게 되면 전체 솔루션의 비용을 다시 측정해야 하는데, 삽입이 비용을 증가시킬 수도 있으며 삽입된 파티션이 처리된 후 파티션을 저장할 수도 있기 때문이다.

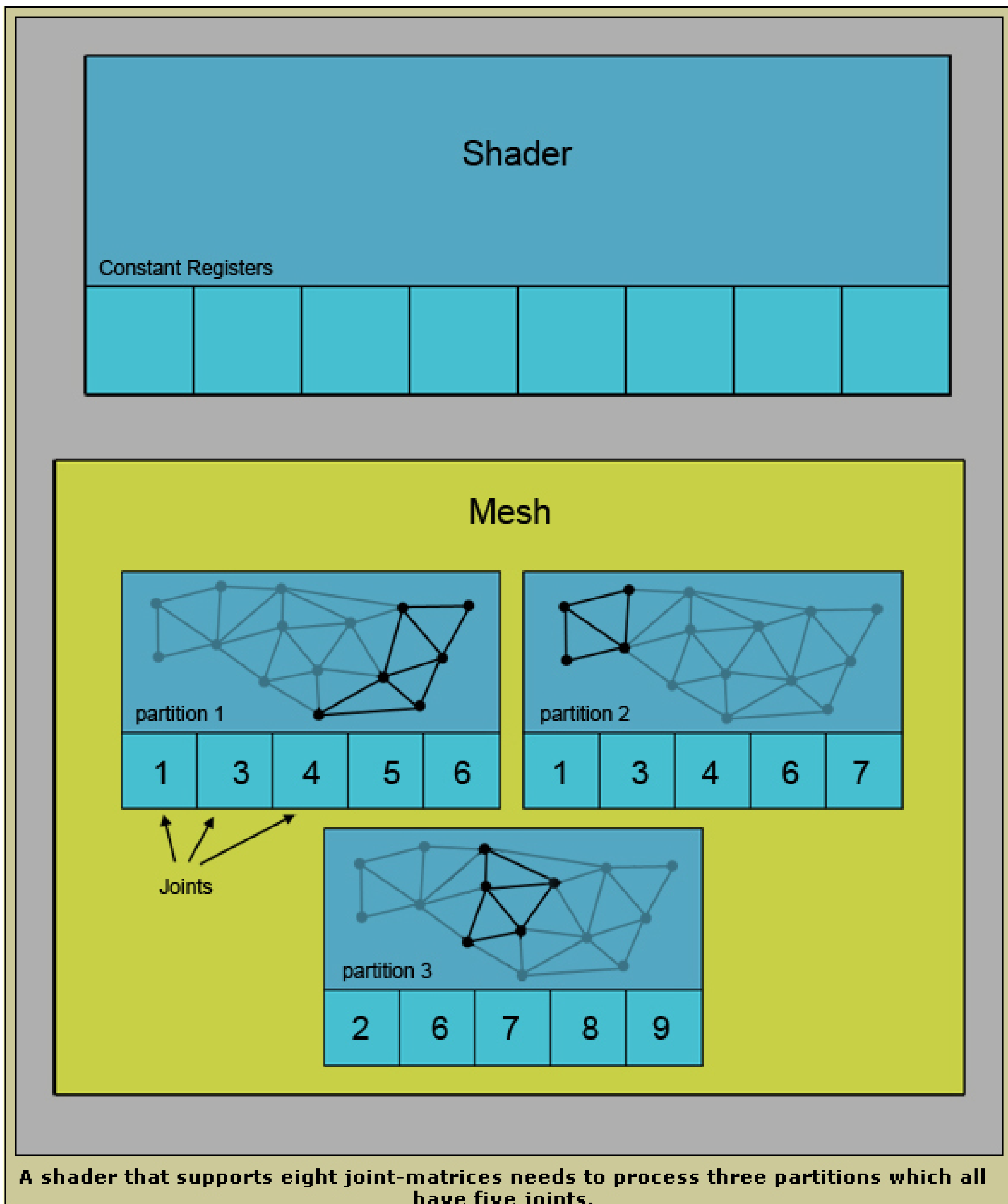
최저가 삽입은 간단한 알고리즘에 비해 훨씬 적은 범위를 다룸에도 어떤 파티션이 추가되었는가에 대해 매우 민감하다. 무작위 정렬을 이용한 결과는 훌륭한 정렬이 이루어진 간단한 알고리즘과 많은 차이가 나지만, 같은 정렬이라면 간단한 알고리즘이 최고의 결과를 가져온다.

이 알고리즘은 의사코드(pseudocode) 2 에서 볼 수 있다.

```
FUNCTION CheapestInsertion PARAM partitionList RETURNING solution
      CALL partitionList.Sort
      INIT solution
      FOR each partition in the partitionList
            SET bestInsertionCost to infinite
            FOR every insertion point in solution
                  CALL solution.GetInsertCost RETURNING insertionCost
                  IF insertionCost < bestInsertionCost
                        SET insertionPlace to current loop position
                  END IF
            END FOR
            IF bestInsertionCost < infinite
                  CALL solution.Insert with partition, insertionPlace
            ELSE
                  CALL solution.Append with partition
            END IF
      END FOR
END FUNCTION
```

예제

우리의 알고리즘을 설명하기 위하여 행렬 팔레트 스키닝이 가능한 셰이더를 지원하는 가상의 목표 하드웨어를 예로 들도록 하겠다. 한 번의 렌더링 과정에서 최대 8 개의 조인트 행렬을 사용하는 것이 가능하다.(이 셰이더는 8 개의 조인트 '슬롯(slot)'을 지니고 있다.) 예로 든 메쉬는 3 개의 파티션을 지니고 있으며, 각 파티션에는 5 개의 조인트가 사용되었다고 치자.



A shader that supports eight joint-matrices needs to process three partitions which all have five joints.

비용 계산(cost calculation)을 보여주는 동안 간단한 알고리즘을 이용하여 파티션을 풀도록 하겠다. 처음에는 하드웨어의 상태가 완전히 새로운 깨끗한 상태로 시작하게 된다. 우리의 목적은 5 개의 조인트 인덱스가 포함된 하나의 파티션을 렌더하는 것이다. 이 파티션을 렌더하는데 들어가는 "비용"은 5 개의 행렬을 올리는 것과 새로운 팔레트를 만드는데 드는 비용의 합이다. 두 번째 파티션 역시 5 개의 조인트 인덱스를 사용하지만, 이 중 4 개는 첫 번째 파티션과 중복된다. 이미 다른 파티션이 현재 팔레트에 존재하기 때문에 두 번째 파티션의 추가비용은 단 하나의 행렬을 올리는 것에 들어가는 비용 정도 밖에 되지 않는다. 전체 파티션은 팔레트에 들어가지 않기 때문에(3 개의 새 파티션이 필요하지만, 2 개만이 사용 가능한 상황) 하드웨어는 새로운 팔레트를 만들게 된다. 이 새 팔레트에서는 세 번째 파티션의 고유한 행렬 5 개를 올리게 된다. 모든 파티션을 처리하고 난 뒤에는 솔루션을 추출할 수 있게 된다. 최종적으로 추출된 파일의 내용은 아래에서 확인 가능하다.

```
Numpartitions 3
Partition 1
Numslots 5
SJ    1    1    # SJ: Slot Joint
SJ    2    3
SJ    3    4
SJ    4    5
SJ    5    6
Triangle list/strips/fan

Partition 2
Numslots 5
SJ    1    1
SJ    2    3
SJ    3    4
SJ    4    6
SJ    5    7
Triangle list/strips/fan

Partition 3
Numslots 5
SJ    1    2
SJ    2    6
SJ    3    7
SJ    4    8
SJ    5    9
Triangle list/strips/fan

shader "JointShader"
Numconstants 6
UC    1    1    # UC: Upload constant slot joint
UC    2    3
UC    3    4
UC    4    6
UC    5    7
UC    6    5
DP    1    # DP: Draw partition
DP    2
Numconstants 5
UC    1    2
UC    2    6
UC    3    7
UC    4    8
UC    5    9
DP    3
```

이득

추출한 메쉬 데이터를 최적화하기 위하여 하드웨어를 시뮬레이팅하는 것은 몇몇 중요한 이득으로 이어집니다. 하드웨어에 대한 정확한 이해는 한 곳에 위치하게 되고("하드웨어" 서브 클래스), 이것은 쉽게 코드 관리를 할 수 있게 해준다. 기술 프로그래머는 Xbox, PS2, PSP, PC 를 비롯한 여러 플랫폼에 기술을 구현시킬 수 있다. 또한

프로그래머가 동일한 하드웨어 플랫폼에 다른 구현 방식을 취할 수도 있다. 이러한 과정을 통해 어떻게 하드웨어를 최적화시켜서 구동할 수 있는지에 대한 새로운 통찰력을 얻을 수 있을지도 모른다.

추상화된 하드웨어(abstracted hardware)로 작업하는 것의 가장 큰 이점은 바로 조인트의 속성에 대한 특별하거나 명확한 지식이 필요하지 않다는 것이다. 우리는 하드웨어 상태를 바꾸는데 필요한 비용을 측정하는 기능을 추가했으며, 여러 정렬을 시도해보며 좋은 솔루션을 탐색했다. 이것은 프로그래머의 일을 크게 줄여주며, 컴퓨터는 수많은 대안들 중에서 최고의 안을 선택하며 자신의 일을 훌륭하게 수행한다.

## 소프트웨어

소프트웨어는 확장이 쉬운 소프트웨어 유틸리티를 만들 수 있도록 코드가 구성되었다. 프로그램은 C++로 생성되었고, MSVC++이나 GNU C++을 이용하여 컴파일 할 수 있다. Wishbone Games 에서 우리는 다수의 하드웨어에 메쉬를 최적화하기 위하여 이 유틸리티를 사용하였다. PC 에서는 분할 작업이 상당히 쉽고 다른 기술을 이용해도 비슷한 결과가 나온다.(같은 수의 랜더 횟수) 그렇지만 콘솔에서는 더 손질을 많이 한 하드웨어 시뮬레이션을 통해 속도를 크게 향상시킬 수 있었다. 우리는 하드웨어로 전달되는 조인트 행렬 데이터를 60% 가량 줄일 수 있었고, 이로써 전체 성능을 향상시킬 수 있었다.

## 결론

차이가 존재하는 하드웨어 성능은 곧 스킨 메쉬를 추출함에 있어 다른 방식의 최적화 기술이 필요하다는 것을 의미한다. 우리의 프레임워크를 통해 새로운 하드웨어나 새로운 기술을 상당히 손쉽게 생성하거나 플러그인 하는 것이 가능했다. 단지 “하드웨어” 클래스의 서브 클래스를 만들어서 그 안에 특정 하드웨어와 관련된 내용을 넣어주기만 하면 되는 것이다.

현재 구현된 방법은 상당히 표준적인 솔루션을 생성하지만, 우리에게는 이것만으로도 충분했다. 만약 어려운 하드웨어 시뮬레이션을 한다면 비용 함수(cost function)는 *최저가 삽입*을

시행하기에 너무 많은 시간이 걸릴 것이다. 이럴 경우 파티션용 삽입 지점의 더 나은 추정치를 만들거나 시뮬레이티드 애닐링(simulated annealing) 또는 제네틱 알고리즘(genetic algorithm) 같은 다른 최적화 기술을 이용해볼 수 있을 것이다. 그렇지만 우리의 목적은 단지 소프트웨어가 제대로 잘 돌아가게 하는 것이다.

이 유틸리티는 스키닝 정보에 기반한 메쉬의 분할을 최적화하기 위해 만들어졌다. 그렇지만 다른 분할 기준을 설정하는 것도 가능하다. 예를 들어 재질(텍스쳐 또는 셰이더), 물리효과(physics), 프렉쳐(fracture) 포인트 등에 따라 분할을 진행할 수 있다. 또한 메쉬를 제거하여 렌더링을 가속시키거나 폴리곤 감소를 시행하는 다른 유틸리티를 만들 수도 있다. 이러한 루틴 역시 하드웨어에 기반을 두고 있으며, 캐시 크기를 비롯한 여러 사항에 기준을 두고 있다. PS2 를 예로 들면, 텍스쳐가 입혀지지 않은 폴리곤은 최적화를 위해 32 픽셀을 넘지 않는 넓이를 지니게 된다. 이러한 정확한 지식은 추출 유틸리티를 만들 때 일반적으로 쓰이지는 않겠지만, 새로운 루틴을 쉽게 플러그인 할 수 있는 툴을 제작할 때에는 이러한 지식을 사용하게 된다. 그러면 이 두 기술을 이용하여 모델을 추출하는 것이 가능하며, 렌더링 속도를 비교할 수 있다. 또한 폴리곤을 렌더링 하는데 들어가는 비용을 측정하는 것까지도 가능하다. 그렇지만 캐싱이나 다중 프로세서, 그리고 어려운 버스 시뮬레이션 모델(bus simulation model)을 이용하는 복잡한 시스템에서는 많은 문제를 일으킬 수도 있다. 그렇지만 이런 유틸리티를 만들고 공유한다면 많은 프로그래머들이 환영할 것이라고 확신할 수 있다.

PC 하드웨어를 위한 컨버젼 프레임워크는 이곳에서 다운로드 받을 수 있다.(소스코드와 실행 파일) 마지막으로 이 글을 감수해준 필자의 친구인 Erik van der Pluym에게 감사의 인사를 전하고자 한다.

---

*1. 스키닝. 구글에서 "Matrix Palette Skinning"에 대해 찾아볼 것.*

*2. PS2 Programming Optimisations, 25 쪽.*