Gama Network Presents:

Gamasutra.com

The Sh GPU Metaprogramming Toolkit

(GPU 메타프로그래밍 툴킷 Sh)

By Michael McCool

Gamasutra

July 16, 2004

URL: http://www.gamasutra.com/features/20040716/mccool_01.shtml

Sh 는 GPU 를 C++로 프로그래밍할 수 있도록 해주는 툴로, 워털루 대학(University of Waterloo)에서 개발한 공개 오픈소스 시스템이다. 이것은 기본적으로 GPU 쉐이더 프로그램을 컴파일하고 런타임 환경설정을 지원하는 C++ 라이브러리로 구성되어 있다. 이 라이브러리는 연산자 오버로딩을 이용하여 명쾌한 고급 레벨 API 를 구축한다. 그러므로 Sh 프로그램에서의 정의 방법은 C++에서와 같이 간단하면서, 전용 쉐이딩 랭귀지에서 쉐이더를 쓰는 것과 같이 풍부한 표현을 할 수 있다. 여기에 덧붙여 Sh 는 C++의 범위 규칙을 적용할 수 있어, C++의 클래스, 템플릿, 함수, 사용자 정의 타입 등 모든 기능을 이용하여 GPU 코드를 모듈화하고 조작할 수 있다. 또 Sh 프로그램을 호스트 응용프로그램에 결합하기 위해 따로 접합 코드를 사용할 필요가 없다: 이들은 그냥 응용프로그램의

확장기능인 것처럼 활용된다. 예를 들어, 쉐이더 파라미터들과 텍스처들을 변수로 선언하면 쉐이더 정의 시 바로 이용할 수 있도록 Sh 에서 나머지 내부적인 처리를 수행한다. Sh 는 복잡한 멀티패스 렌더링 알고리즘에서 쉐이딩 언어로 사용할 수 있고, 또 다른 한편으로는 (시뮬레이션과 같은) 범용 스트림 계산을 구현하는 데 사용할 수도 있다.

게임 개발자로서 Sh 에 관심을 두어야 할 이유는 무엇인가? 먼저 Sh 는 시중의 다른 실시간 쉐이딩 언어들보다 훨씬 더 강력하고 모듈화되어 있으며, 완성도가 높은 프로그래밍 시스템이다. 이것은 단지 쉐이딩 언어에 그치는 것이 아니다: 이것은 텍스처와 쉐이더 파라미터, 그리고 이들과 쉐이더 사이의 연관성을 추적할 수 있다. C++의 객체지향형 기능을 이용하면 쉐이더들 간에 코드를 공유할 수도 있고, 복잡한 알고리즘이나 데이터 표현을 포장(Encapsulate)하여 이들이 좀 더 쉽게 재사용될 수 있도록 할 수 있다. 또한 Sh 를 이용하면 사용자의 데이터를 쉐이더로 변환하도록 하는 (메타프로그래밍) 맞춤 컴파일러를 제작할 수도 있다. 둘째로, Sh 는 다른 쉐이딩 언어들에서 요구되는 접합 코드나 다른 많은 성가신 것들을 제거하고 생산성을 높일 수 있다. 셋째로, 시뮬레이션이나 AI 와 같은 다른 게임 엔진 계산 기능을 Sh 로 가속할 수 있다. Sh 는 CPU 와 GPU 에 모두 컴파일하기 때문에 이들 기능들을 Sh 로 쓴다고 해도 반드시 이들이 GPU 에서만 실행되는 것은 아니다. 심지어는 그것이 실행시에 결정되도록 할 수도 있다. 예를 들어 설치시에 GPU 와 CPU 의 프로파일을 만들어 실행시에 시스템 부하가 더 잘 분산되는 프로세서를 골라 시뮬레이션 부분을 실행할 수도 있다. 일반적으로 Sh 는 시스템의 모든 계산 기능을 공통의 인터페이스를 통해 개발자가 다 이용할 수 있도록 도와준다.

데이터 표현과 알고리즘을 포장하여 나타내는 Sh 의 기능에 대한 향후 연구를 통해 Sh 클래스나 함수라는 형태로 복잡한 알고리즘의 구현이 가능할 수 있기를 기대한다. Sh 는 또한 완전한 공개 프로그램이므로 쉐이더 컴파일러 연구자들에게 유용한 플랫폼이다. 끝으로 우리는 Sh 를

병렬 서버나 게임 플랫폼을 포함하여 다른 컴파일 대상들도 지원할 수 있도록 확장할 예정이다. Sh 는 플랫폼, 벤더, API 로부터 독립적인 개념적 모델을 갖고 있다. 이상적으로는 이것이 이기종간 포팅을 더욱 쉽게 해주고 코드의 재사용 가능성을 크게 높여줄 것이다.

Sh 의 아키텍쳐

먼저 Sh 의 구조를 살펴보자. 라이브러리는 **ShPoint3f**, **ShVector3f**, or **ShMatrix4x4f** 와 같은 클래스의 집합으로 이루어져 있다. 이들은 그래픽 유틸리티 라이브러리로 직접 사용될 수 있다. 다수의 연산자와 함수들이 또한 정의되어 있어 이들 클래스의 객체들에 대해 적용될 수 있다. 벡터의 합성과 감산, 내적과 외적, 행렬-벡터 또는 행렬-포인트 변환, 벡터의 정규화 등등이 지원된다. Sh 는 또한 스위즐링 (Swizzling, 다차원 좌표나 행렬 원소의 재배열이나 추출), 골라 쓰기 (Writemasking, 다차원좌표나 행렬의 일부 원소만 지정하기) 기능을 지원한다.

Sh 의 객체에 대한 연산은 두 가지 모드로 지정될 수 있다. 하나는 *즉시 모드(Immediate mode)*인데, 이것은 연산이 지정되는 즉시 실행되는 것으로, 디폴트 모드이다. 다른 하나는 *보류 모드(Retained mode)*로 연산을 순차적으로 실행하는 것이 아니라 Sh 가 이들을 *프로그램 오브젝트*에 기록한다. 보류 모드는 해당되는 코드 부분을 **SH_BEGIN_PROGRAM** 와 **SH_END** 키워드로 둘러싸는 방법으로 표시된다. 이렇게 기록된 연산 명령들은 지정된 컴파일 대상에 (Sh 가 호스트의 CPU 용으로 코드를 동적으로 생성할 수도 있지만 보통은 GPU 가 대상이 됨) 맞게 컴파일된다. 프로그램 오브젝트들은 GPU 의 버텍스와 프래그먼트 쉐이더에 로딩될 수 있는데, 그러면 표준 그래픽 API 를 가지고 렌더링 하는 데 영향을 줄 수 있다. 다른 대안으로는 그래픽 API 를 호출할 필요 없이 범용 계산을 위한 스트림 함수로 직접 사용하는 것이다.

코드를 동적으로 생성하는 기능에 덧붙여 Sh 는 텍스처와 스트림을 관리할 수도 있다. 텍스처는 어레이나 또는 다른 파라미터들과 마찬가지로 C++의

범위 규칙을 이용하여 Sh 프로그램에 연결된다. 이는 데이터 추상을 텍스처를 이용하여 구축할 수 있다는 것을 의미한다. 예를 들어 어떤 압축 텍스처 타입을 구축하고, 그것을 쉐이더 코드의 특정한 순차적 진행에 의해 압축해제를 하고자 한다고 가정하자. 이를 위해 압축 데이터를 저장하는 내장 텍스처 클래스를 포장(Encapsulate)하는 클래스를 선언하고, 이를 호출하는 쉐이더 내에 필요한 코드를 삽입하기 위한 접근 연산을 재정의 할 수 있을 것이다. 이렇게 정의한 새로운 클래스가 내장 텍스처와 동일한 인터페이스를 지원한다면 내장 텍스처가 사용되는 곳에서는 어디나 사용될 수 있을 것이다.

스트림은 GPU 에서 범용 연산을 지원하기 위해 사용된다. 스트림 프로그램은 쉐이더와 유사하다. 특정 개수만큼의 입력을 특정 개수의 출력으로 매핑한다. Sh 의 스트림 오브젝트는 텍스처와 유사하다. 이들은 스트림 프로그램에 의해 생성되거나 호출될 수 있는 메모리 내의 일련의 데이터들을 가리킨다. 스트림 프로그램은 단순 연산자나 함수 호출 신택스를 이용하여 스트림에 적용할 수 있다. 또는 스티림을 개별적인 데이터의 채널들로 역분해하거나 재구성하는 것이 가능하다. 공유 서브스트림, 파라미터의 입력 값으로의 변환이나 그 반대로의 변환, 프로그램 콤포지션, 커링 등 고급 기능들을 지원하는 복잡한 스트림 신택스도 제공된다.

사례: 블린-퐁 쉐이더(Blinn-Phong Shader)

Sh 를 소개하는 데 가장 간편한 방법은 몇 개의 사례들을 살펴보는 것이다. 아래의 코드는 버텍스 쉐이더와 프래그먼트 쉐이더를 정의하는 것을 통해 단일 포인트 소스용의 블린-퐁 쉐이더를 (쉐이더 프로그램 분야의 “Hello World”라고 할 만한 가장 간단한 쉐이더) 정의하고 있다. 이 쉐이더는 또한 버텍스들을 조명을 위해 가시 공간으로 변환하고, 렌더링을 위해 디바이스 공간으로 변환한다. 그림 1 은 이 쉐이더를 이용해 렌더링한 결과이다.

그림 1: 블린-퐁 조명 모델; 기본형과 텍스처 맵을 적용한 경우

제일 먼저 변환 행렬과 조명 모델의 파라미터들을 지정하는 글로벌 변수를 몇 개 정의한다:

```
ShMatrix4x4f modelview;   // MCS to VCS transformation
ShMatrix4x4f perspective; // VCS to DCS transformation

ShColor3f phong_kd; // diffuse color
ShColor3f phong_ks; // specular color
ShAttrib1f phong_spec_exp;      // specular exponent
ShPoint3f phong_light_position; // VCS light position
ShColor3f phong_light_color;    // light source color

ShProgram phong_vert, phong_frag;
```

버텍스 쉐이더와 프래그먼트 쉐이더 자체를 초기화 함수 내에 구축한다:

```
void phong_init () {
  // Create vertex shader
  phong_vert = SH_BEGIN_PROGRAM("gpu:vertex") {
    // Declare shader inputs
    ShInputNormal3f nm; // normal vector (MCS)
    ShInputPosition3f pm; // position (MCS)

    // Declare shader outputs
    ShOutputNormal3f nv;   // normal (VCS)
    ShOutputVector3f lv;   // light-vector (VCS)
    ShOutputVector3f vv;   // view vector (VCS)
    ShOutputColor3f ec;    // irradiance
    ShOutputPosition4f pd; // position (HDCS)

    // Specify shader computations
    ShPoint3f pv = (modelview | pm)(0,1,2);
    vv = normalize(-pv);
    lv = normalize(phong_light_position - pv);
    nv = normalize(modelview | nm);
    ec = phong_light_color * pos(nv|lv);
    pd = perspective | pv;
  } SH_END; // End of vertex shader

  // Create fragment shader
  phong_frag = SH_BEGIN_PROGRAM("gpu:fragment") {
    // Declare shader inputs
    ShInputNormal3f nv; // normal (VCS)
    ShInputVector3f lv; // light-vector (VCS)
    ShInputVector3f vv; // view vector (VCS)
    ShInputColor3f ec;  // irradiance

    // Declare shader outputs
    ShOutputColor3f fc; // fragment color

    // Specify shader computations
    vv = normalize(vv);
    lv = normalize(lv);
    nv = normalize(nv);
    ShVector3f hv = normalize(lv + vv);
    fc = phong_kd * ec;
    fc += phong_ks * pow(pos(hv|nv), phong_spec_exp);
  } SH_END; // End of fragment shader
} // End of phong_init
```

Sh 쉐이더 프로그램의 정의 부분을 SH_BEGIN_PROGRAM 와 SH_END 키워드로 둘러싸고 있는 것을 볼 수 있다. SH_BEGIN_PROGRAM 은 기록된 일련의 연산들을 나타내는 프로그램 오브젝트를 돌려준다. 프로그램 오브젝트에 대한 입출력은 각각 Sh 타입 인스턴스 상에서 Input 과 Output 프리픽스로 표시된다. "|" 연산자는 행렬의 내적과 행렬 곱셈에 사용된다. 그러나 내적에는 dot 함수를, 그리고 행렬 곱셈에는 “*”를 대신 사용할 수도 있다.

이렇게 정의된 *phong_vert* 와 *phong_frag* 프로그램 오브젝트는 shBind API 를 써서 GPU 의 프래그먼트 쉐이딩 유닛과 버텍스에 로딩할 수 있다. 이제 보통의 그래픽 API 를 써서 기하학적 형체를 지정하면 쉐이더들이 그 형체에 적용될 것이다. 현재는 Sh 가 OpenGL 만을 지원하는데, DirectX 바인딩도 지금 작업 중에 있으므로 곧 지원될 것이다. 로딩된 쉐이더들을 위해 그래픽 API 에서 정확한 버텍스 속성들을 설정하여야 한다. 그리고 프래그먼트 쉐이더와 버텍스 쉐이더의 쌍도 각각의 입출력 값과 일관성이 있어야 한다. 선언의 타입과 순서에 따라 쉐이더에 대한 입력이 어떻게 버텍스의 속성들로 매핑 되는지를 결정하는 몇 가지의 규칙이 있다. 또는 프로그램 오브젝트에게 인터페이스 바인딩에 대한 설명을 사람이 읽을 수 있는 문구로 답하도록 질의할 수도 있다.

phong_kd 와 같이 쉐이딩 된 모든 버텍스와 프래그먼트에 걸쳐 동일한 값을 갖는 “단일” 파라미터들은 쉐이더 정의에 의해 간단히 직접 참조할 수 있다. 이들 파리미터들을 설정하기 위해 추가의 접합 코드가 필요하지 않으며, 값을 수정하려면 (쉐이더 정의 바깥에서) 간단히 할당하기만 하면 된다. 어떤 파라미터가 어떤 쉐이더 프로그램에 연결되느냐는 C++의 범위 규칙에 의해 결정된다. 예를 들어, 파라미터 데이터를 클래스의 멤버로 만들고, 쉐이더 프로그램 오브젝트를 멤버에 대한 함수로 정의할 수 있다. 그러면 쉐이더 프로그램을 생성하는 멤버 함수가 데이터 멤버들을 골라내고 포장된 쉐이더를 만들어낼 수 있을 것이다. 일반적으로 Sh 는 C++와 명쾌하게 통합되도록 디자인되어 있으며 대부분의 C++ 모듈 구성이 Sh

프로그램과 함께 사용될 수 있다. 이러한 통합을 통해, 그리고 C++가 Sh 프로그램을 실행시에 임의로 조작할 수 있는 덕분에 다른 많은 프로그래밍 기법들이 가능해진다.

만약 이 쉐이더에 대한 텍스처 맵이 필요하다면 *phong_kd* 용의 **ShColor3f** 대신 **ShTexture2D<ShColor3f>**를 사용할 수도 있다. 그러면 텍스처 좌표를 전달하고 텍스처 오브젝트에 인덱스를 넣기 위해 쉐이더의 정의를 수정해야만 한다. 텍스처를 바인딩 하는 것은 단일 파라미터를 바인딩 하는 것과 완전히 같은 방법으로 이루어진다. C++의 객체 지향형 특성을 이용하여 데이터 추상을 만들어 낼 수 있는 것이다.

다음 코드에서는 파라미터들을 클래스의 데이터 멤버로 포장하고, 템플릿 아규먼트와 생성시간 아규먼트를 사용하여 쉐이더를 파라미터화 한다. 또 템플릿 클래스를 이용하여 버텍스 쉐이더의 출력 값과 프래그먼트 쉐이더의 입력 값을 조절한다. (Sh 타입을 선언하는 데 더 범용이며 템플릿에 기반한 방법도 부수적으로 포함되어 있다. 이것을 이용하면 임의의 길이의 다차원좌표를 선언할 수 있다.) 끝으로 C++의 제어 콘스트럭트를 이용하여 쉐이더 코드들을 조작하고 있다: 이 경우 복수의 광원을 지원하기 위해 루프를 풀어내고, C++ 어레이를 이용하여 복수 광원의 속성들을 임시 저장한다. 또는 보통의 C++ 함수를 이용하여 함수들을 쉐이더 코드로 구현할 수도 있다. 이로써 노멀라이즈와 같은 표준 라이브러리 함수들이 (그리고 사실 연산자들이) 어떻게 구현되었느냐를 대략 설명하였다. 이 쉐이더를 이용한 렌더링 결과가 그림 1에 표시되어 있다.

```
class BlinnPhong {
  public:
    // Declare parameters and textures as data members
    ShTexture2D<ShColor3f> kd;
    ShTexture2D<ShColor3f> ks;
    ShAttrib1f spec_exp;
    ShPoint3f light_position[NLIGHTS];
    ShColor3f light_color[NLIGHTS];

    // Declare I/O type to coordinate vertex and
fragment        shaders
    template <ShBindingType IO> struct VertFrag {
      ShPoint<4,IO,float> pv;   // position (VCS)
      ShTexCoord<2,IO,float> u; // texture coordinate
      ShNormal<3,IO,float> nv;  // normal (VCS)
      ShColor<3,IO,float> ec;   // total irradiance
    };

    // Declare program objects for shaders
    ShProgram vert, frag;

    // Constructor: parameterized by texture resolution
    BlinnPhong (int res) : kd(res,res), ks(res,res) {

      // Create vertex shader
      vert = SH_BEGIN_PROGRAM("gpu:vertex") {
        // Declare shader inputs
        ShInputNormal3f nm;   // normal vector (MCS)
        ShInputTexCoord2f u;  // texture coordinate
        ShInputPosition3f pm; // position (MCS)

        // Declare shader outputs
        VertFrag<SH_OUTPUT> vf;
        ShOutputPosition4f pd; // position (HDCS)

        // Specify shader computations
        vf.pv = modelview | pm;
        vf.u = u;
        vf.nv = normalize(modelview | nm);
        pd = perspective | vf.pv;
        for (int i=0; i<NLIGHTS; i++) {
          ShVector3f lv =
            normalize(light_position[i] - vf.pv(0,1,2));
          vf.ec += light_color[i] * pos(vf.nv|lv);
        }
      } SH_END; // End of vertex shader

      // Create fragment shader
      frag = SH_BEGIN_PROGRAM("gpu:fragment") {
        // Declare shader inputs
        VertFrag<SH_INPUT> vf;

        // Declare shader outputs
        ShOutputColor3f fc;     // fragment color
```

프래그먼트 쉐이더에서 kd(vf.u) 나 ks(vf.u) 에서 볼 수 있는 것처럼 텍스처 읽기 연산이 "()"를 이용해 이루어지고 있는 것을 볼 수 있다. 이 연산은 각 좌표의 텍스처 좌표가 0 과 1 사이의 값을 갖도록 정규화된 제표 함수인 것으로 텍스처를 간주한다. Sh 는 또 텍스처 검색을 위해 "[]" 연산도 지원한다. 이것 역시 마찬가지의 방식으로 작동하는데 각 정수마다 텍셀을 부여한다. "[]" 검색은 텍스처가 데이터 스트럭쳐를 (예컨대, 레이 트레이서 가속기 같은 것) 임시 저장하는 어레이로 사용되는 경우 유용하다. Sh 는 또 사각형 텍스처, 1D 및 3D 텍스처, 큐브 텍스처 용의 추가 텍스처 타입도 여러 개 지원한다.

Sh 는 입력 값을 쓰거나 출력 값을 읽는 데에도 사용될 수 있다. 입력 값을 쓰는 것은 원본 데이터를 수정하지는 않는다: Sh 입력은 그 값에 의해 패스된다. 이러한 연산을 지원하지 않는 후방 기능에 매핑하는 경우 Sh 는 자동적으로 추가의 템포러리를 도입한다. 템포러리는 (자동 도입된 것 포함) 그 값이 할당되기 전에 사용되는 경우 항상 0 으로 그 값이 초기화된다. 이러한 초기화는 입력 코드를 단순화하기 위한 편의상 포함된 것이다. 예를 들어, += 연산을 0 으로 초기화된 출력 값에 사용하는 것은, 어떤 광원이 제일 처음인지를 신경 쓰지 않으면서 몇 개의 광원을 집적하여 하나의 출력 색상을 만들어내는 경우 대단히 유용하다.

그 외 몇 개의 쉐이더 예들이 그림 2 에 나타나 있다. 이들 예의 다수는 Sh 에 내장된 펄린(Perlin)과 월리(Worley)의 노이즈 함수를 사용하고 있다. 예를 들어 우드는 2 차 함수에 펄린 노이즈를 추가하고, 그 결과를 텍스처 맵에 저장되어 있는 주기적 톱니함수에 적용한다. 월리 노이즈 함수는 절차상 생성된 피쳐 포인트(Feature point)의 (우리의 경우는 불안정 그리드 사용) 집합에 k 번째로 가장 가까운 거리에 기반한 것이다. 월리 노이즈 함수를 계산하는 라이브러리 함수는 각각 다른 거리 함수와 함께 파라미터화 할 수 있다. 여기에 예로 나타낸 기린 쉐이더는 월리 함수로

맨하탄 거리 단위를 쓰고, 임계 함수를 이용해 가장 가까운 피쳐 포인트(Feature point)에 그 거리를 보낸다.

그림 2: 쉐이더의 또 다른 사례. 다수의 사례들에서 Sh 에 내장된 펄린과 월리의 노이즈 함수를 사용한다.

스트림 함수

스트림 함수는 gpu:stream 나 cpu:stream 를 대상으로 컴파일된 쉐이더들의 일반적인 형태이다. 스트림과 채널은 스트림 함수에 의해 호출될 수 있는 일련의 데이터를 담고 있는 콘테이너들이다. 채널은 기본적 다차원좌표 데이터 타입의 데이터들을 저장하며, 스트림은 채널들의 특정한 조합을 나타낸다. 스트림 함수 응용프로그램과 채널들을 이용한 스트림 구성을 위해 특수 연산을 적용한다: 스트림 함수 응용프로그램을 위해서는 “<<”를, 스트림 구성을 위해서는 “&”를 사용한다. 예를 들어 세 개의 입력 채널과 (포인트, 노멀, 컬러) 두 개의 출력 채널을 (컬러, 스칼라) 갖는 스트림 함수 f 가 있다고 가정하자. 이 경우 다음과 같은 방법으로 어떤 채널들을 선언할 수 있다:

```
ShChannel<ShPoint3f> p;
ShChannel<ShNormal3f> n;
ShChannel<ShColor3f> c1, c2;
ShChannel<ShAttrib1f> d;
```

그러면 다음과 같은 방법으로 스트림을 생성할 수 있다:

```
ShStream input_s = (p & n & c1);
ShStream output_s = (c2 & d);
```

채널들의 데이터를 초기화 한 후, 출력 스트림의 데이터를 생성하기 위해 스트림 함수 f 를 적용한다:

```
output_s = f << input_s;
```

필요하지 않다면 중간 스트림을 만들지 않고 그냥 "&" 연산자를 라인 내에서 사용할 수 있다:

```
(c2 & d) = f << (p & n & c1);
```

"<<" 연산자는 단일한 다차원좌표에 프로그램을 적용하기 위해 사용된다. 이것은 모든 원소들이 값을 갖는 채널인 것처럼 처리된다. 부분 계산도 (Partial evaluation, Currying) 지원되므로 프로그램에 입력 값을 한꺼번에 제공하지 않아도 된다. 다음은 위에서 나타낸 표현과 동일한 효과를 갖는다:

```
ShProgram g = f << p;
(c2 & d) = g << (n & c1);
```

위의 예에서 부분 계산을 위해, 데이터를 복사하는 것을 회피하려는 목적으로 채널 p 는 두 번째 라인에서 스트림 함수가 실제 실행될 때만 읽혀진다. 이러한 "읽기 연기"는 입력 값이 스트림 채널이 아니라 다차원

좌표일 경우에도 적용될 수 있다. 이것은 사실 흥미로운 특징이다: “<<”를 이용하면 “변동하는” 입력 속성들을 “일정한” 파라미터로 변환할 수 있다는 것이다! 이것은 유용한 연산이므로 “>>” 연산자를 이용하여 파라미터를 입력 값으로 변환하는 역함수도 정의한다. 여기에 어떤 프로그램을 왼쪽에 놓고 그 프로그램이 사용하는 파라미터들은 오른쪽에 놓는다. 그 결과 파라미터에 대한 의존성이 제거되고 추가적인 입력을 갖는 프로그램을 얻을 수 있다. 예를 들어, (텍스처 좌표와 같이) 아주 많은 곳에서 사용되는 값이 있을 때, 이것을 여기저기 전달하는 것이 아니라 글로벌 파라미터로 선언하고, 이를 나중에 입력 값으로 변환하는 것이다.

다른 프로그램 오브젝트를 만들기 위해 "<<" 와 "&" 연산자를 프로그램 오브젝트에 직접 적용하는 것도 가능하다. "<<" 연산자는 하나의 프로그램 오브젝트로부터 출력을 받아 다른 프로그램 오브젝트로 입력하여 새로운 프로그램 오브젝트를 만들어낸다. 달리 말하면 합성 함수 연산을 실행한다. "&" 연산자는 두 개의 프로그램 오브젝트로부터 모든 입력, 출력, 연산을 통합하여 새로운 프로그램 오브젝트를 만들어낸다. Sh 구문의 구조상 이것은 (두 개의 서로 다른 명칭 범위에 있는) 두 프로그램 오브젝트들의 소스 코드를 통합하는 것과 같은 효과를 갖는다. 이러한 연산을 몇 번이고 반복한 결과 얻어지는 합성 프로그램 오브젝트는 최적화를 포함하여 완전한 컴파일 수순을 거치게 되어 최적화된 단일한 상태로 단순화된다.

이러한 연산 기능은 대단히 유용한데, 기존에 존재하는 프로그램 오브젝트들을 작은 접합 프로그램들로 통합하는 경우에 특히 도움이 된다. 예를 들어, “<<”를 이용하여 프로그램의 입력과 출력을 재배치할 수 있고, 출력을 삭제하거나 (이것은 프로그램 특화라고 불리우는 연산인데 옵티마이저가 모든 불필요한 계산 과정을 삭제한다), 입력 값을 텍스처 접근으로 대체할 수도 있다. Sh 라이브러리에는 이와 같이 통상 사용되는 접합 프로그램들을 위한 수 많은 제너레이터들이 포함되어 있다.

그림 3 은 스트림 프로세싱을 이용한 응용프로그램의 예이다. 이 그림은 스트림 프로그램을 이용하여 각 입자의 상태를 갱신하고 쉐이더를 이용하여 갱신된 입자의 상태를 비쥬얼 렌더링으로 변환하는 GPU 상에서, 입자 시스템 시뮬레이션을 한 결과를 보여주고 있다. 이 응용프로그램에서 입자의 상태를 갱신하는 코드는 우리가 최근 발표한 논문 SIGGRAPH[3]에 포함되어 있다. 그 논문에는 퐁(Phong) 조명 모델을 프로그램 조작 연산을 (쉐이더 대수) 이용하여 우드 쉐이더로 변환하는 예도 설명되어 있다.

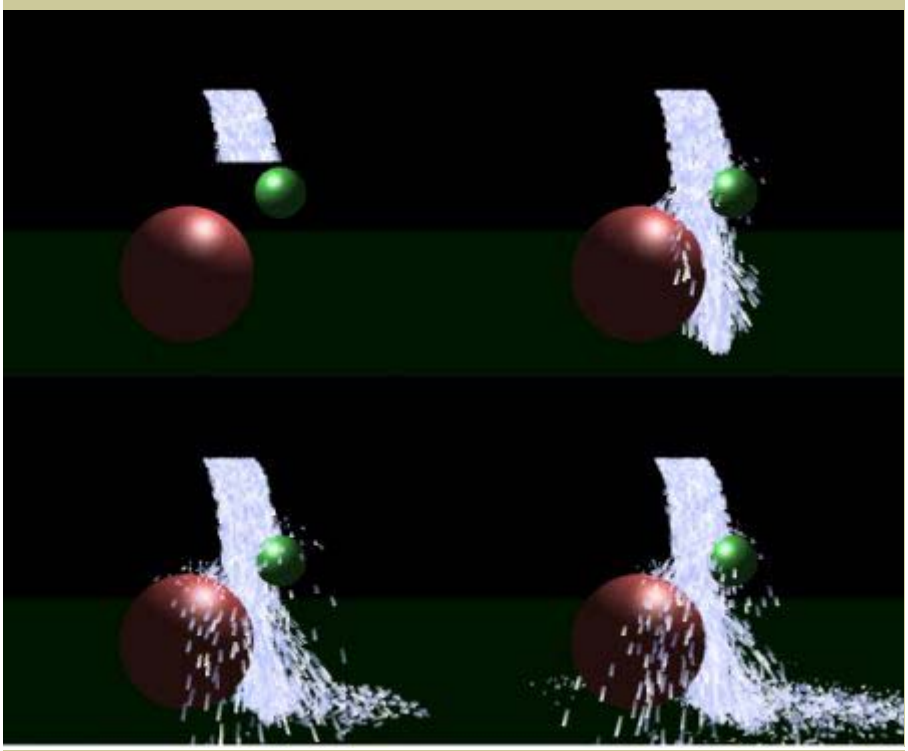

그림 3: 스트림 프로세싱을 이용하여 구현된 입자 시스템 시뮬레이션

향후 과제

Sh 는 계속 개선되는 과정에 있다. 연구 프로토타입 단계는 이미 넘어섰고, 이제는 강력하고 유연한 사양을 요구하는 쉐이더 용으로도 사용할 수 있다. 그러나 현재 우리는 스트림 컴퓨팅과 대규모 상용 수준의 게임 개발 프로젝트들을 더 잘 지원하기 위해 필요한 몇 가지 기능들을 추가하고 있는 중이다 (애셋 관리 시스템과의 통합 기능, 파일 표출화(Externalization) 기능, 그리고 추가적인 플랫폼 지원 등). 공식적으로 Sh 는 아직 알파 단계에 있지만 2004 년 8 월 베타 출시를 위해 마지막 테스트 단계를 거치고 있다. 베타 단계에 있다고 해서 Sh 가 불안정하다는 것을 뜻하지는 않는다. 다만 모든 최종적인 기능들을 다 갖추고 있지 않을 뿐이다. 우리는 게임 업계의 동료들로부터 평가를 청취하고, Sh 가 상용의 게임 프로젝트에 사용되는 데 장애물이 될 만한 미비 기능이 있는지를 찾아내는 데 큰 관심을 가지고 있다. 우리는 올 해 연말 이전에 그런 문제점들을 모두 해결하고자 최대의 노력을 기울이고 있다.

Sh 는 C++ 호스트 응용프로그램과 긴밀히 통합되어 있다는 점, 그리고 스트림 컴퓨팅을 직접 지원한다는 점에서 다른 쉐이딩 언어들로부터 차별성을 갖고 있다. 이러한 두 가지 특성은 복잡한 알고리즘을 구현하기 위해 CPU 와 GPU 의 제어와 계산 기능을 통합하는 응용프로그램에 초점을 맞춘 것이다. 그러나 리덕션이나 인덱싱과 같이 Sh 에 의해 직접 지원되지 않는 추가의 스트림 연산들도 일부 스트림 프로세싱 알고리즘을 구현하는 데 잠재적으로 용도가 있을 것이다. 그러나 GPU 에서 이들 멀티패스 연산을 효율적으로 구현하기 위해서는 유연한 버퍼 사용과 그래픽 가속기의 메모리를 관리할 수 있는 드라이버가 요구된다. 이 부분은 아직 불확실한 것들이 많이 남아 있다. 드라이버 문제가 해결된다는 전제 하에 이러한 기능들도 조만간 추가될 것이다.

우리는 더 많은 플랫폼을 지원하기 위한 노력도 기울이고 있다. 현재는 Sh 에서 ATI 와 Nvidia 의 플로팅포인트 GPU 와 호스트 CPU 를 (스트림

함수들이 CPU 코드로 동적으로 컴파일 됨) 지원하고 있다. 게임 플랫폼과 MPI 를 이용한 병렬 분산 메모리 클러스터를 지원하는 것을 현재 추진하고 있는 중이다. Sh 의 구문은 그러한 플랫폼에 효율적으로 매핑하는 것을 가능하기 위해 의도적으로 단순하게 만들어졌다. 그러나 이들 기기들은 GPU 와는 너무나 다른 아키텍쳐를 가지고 있기 때문에 추가적인 개발이 필요할 것이다.

우리의 궁극적 목적은 스트림 컴퓨팅과 쉐이더를 모두 지원하는 모든 플랫폼에서 단일한 컴퓨팅 모델을 제공하는 것이다. 그리고 그 위에 유용한 알고리즘 라이브러리들을 구축하는 것이다. 우리는 (게임 플랫폼 연결 부분 등) 개발에 라이센스를 지급하는 일부 부분을 제외하고 가능한 많은 부분을 공개 소스로 유지하려고 하고 있다.

추가 읽을 거리

AK Peters 는 "*Metaprogramming GPUs with Sh* "라는 2004 년 8 월 Sh 시스템에 관한 책을 출간할 예정이다 [2]. 이 책에는 상세한 사용자 튜터리얼, 레퍼런스 매뉴얼, Sh SourceForge 웹사이트로부터 구할 수 있는 공개 소스의 내부적인 부분에 대한 안내 등이 포함되어 있다. 이 웹사이트에는 샘플 쉐이더, 연구 논문 [3, 4], 비디오 등 추가적인 정보들도 포함하고 있다.

Sh 제작자들은 프로그래밍 콘테스트를 열 계획도 가지고 있다. ATI 와 NVIDIA 로부터 각각 두 개씩, 네 개의 고급 비디오 카드가 상품으로 제공될 예정이다. Sh 의 멋진 기능들을 최대한 잘 활용하는 프로그램들에게 상품이 수여될 것이다. 그러나 프로그램이 쉐이더에만 한정되지는 않을 것이다. 이에 대한 자세한 내용은 Sh 웹사이트를 참조하기 바란다 [1].

참고문헌

1. Sh Web Site, http://libsh.org

2. Michael McCool and Stefanus Du Toit, *Metaprogramming GPUs with Sh*, AK Peters, 2004, http://www.akpeters.com

3. Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan and Kevin Moule, Shader Algebra, ACM Transactions on Graphics (Proceedings of SIGGRAPH), August 2004.

4. Michael McCool, Zheng Qin, and Tiberiu Popa, *Shader Metaprogramming*, Proceedings of SIGGRAPH/Eurographics Conference on Graphics Hardware, September 2002, pp. 57-68.