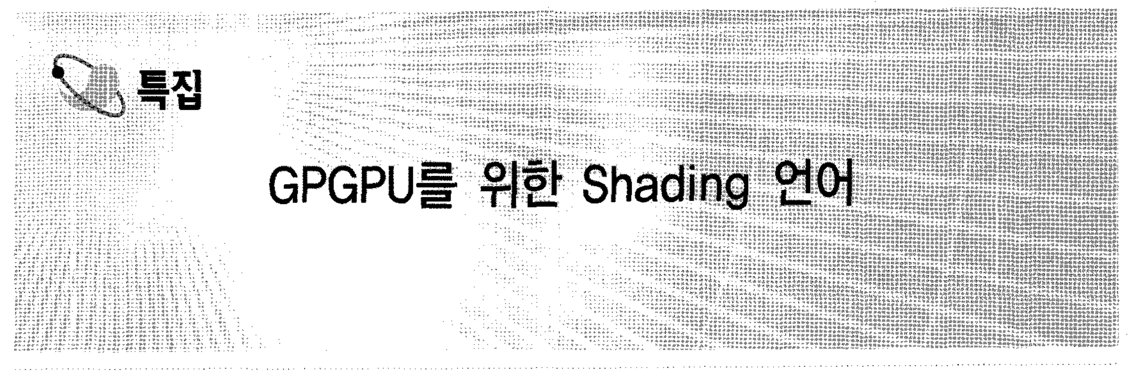


GPGPU를 위한 Shading 언어

이만희 · 박인규 (인하대학교)

I. 서 론

1990년대 중반 퍼스널 컴퓨터에서 3차원 그래픽 처리를 위한 가속기 하드웨어가 발표된 이후 일반 사용자용 GPU(Graphics Processing Unit)의 계산 성능은 CPU보다 빠른 속도로 발전해왔다. 이미 2003년경부터 부동소수점 (floating-point) 연산속도는 GPU가 CPU를 앞질렀고 NVIDIA의 GTX 280의 경우 933 GFLOPS(giga floating-point operations per second)로 같은 시대의 CPU에 비하여 9배의 성능 차이를 보인다[1]. 또한, GPU의 성능 향상과 더불어 고수준 shading language 언어를 이용한 프로그래밍이 가능하게 되었다. 이를 기반으로 GPU 내부의 기능을 변경하여 사용할 수 있게 됨에 따라 기존의 3차원 그래픽 처리뿐만 아니라 다양한 분야에서 GPU의 대용량 병렬처리를 사용할 수 있게 되었다[2].

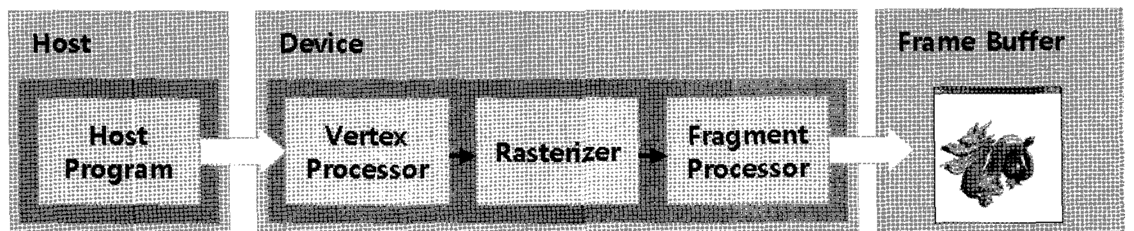
한편, 2007년 2월 NVIDIA에서 발표된 CUDA[1] 환경은 기존의 3차원 그래픽스 파이프라인 구조에 종속되지 않고 GPU 자체를 병렬처리를 위한 보조 프로세서 (co-processor)로의 사용을 가능하게 하였다. 최근 이러한 GPU의 대용량 병렬처리 능력을 이용하여 일반적인 영상처리 분야에 많이 사용되어온 2차원 필터링이나 영상 압축, 그리고 윈도우 기반의 특징 추출 알고리즘[3]을 가속하는 많은 연구가 진행되었다. 그리고 이러한 영상처리와 컴퓨터 비전분야의 여러 알고리즘들을 통합한 라이브러리들도 활발히 개발되어지고 있다.

그러나 비디오의 메모리 공간에서 자유로운 읽기 및 쓰기 등의 연산이 필요한 알고리즘이 요구되는 분야에서는 CUDA를 이용하여 눈부신 성능향상을 기대할 수 있지만, 기존의 그래픽스 파이프라인을 유지한 구조에서 내부 연산의 수정을 통하여 알고리즘이 구현될 수 있을 경우 오히려 고전적인 shader를 이용하는 것이 더욱 효과적일 수 있고 이러한 이유로 shader를 이용한 멀티미디어 처리 역시 꾸준한 관심을 얻고 있다.

본 기고에서는 shader의 기본적인 개념에 대하여 알아보고 shading 언어의 종류와 특징, 그리고 실제 shading 언어를 활용하여 멀티미디어 처리를 수행한 사례를 제시하고자 한다.

II. Shader

<그림 1>에 전통적인 고정된 (fixed) 그래픽



<그림 1> 전통적인 고정된 그래픽스 파이프라인

스 파이프라인을 단순화하여 도시하였다. 호스트 프로그램으로부터 넘겨받은 삼각형들의 3차원 정점 데이터는 vertex processor의 정점 단위의(per-vertex) 연산 과정에서 투영 변환(projective transformation)을 거쳐 2차원의 뷰포트(viewport) 좌표계로 변환된다. 동시에 반사 특성과 외부 조명 속성 및 퐁(phong) 조명 모델에 의해 자신의 색상이 결정된다. 그 이후 래스터(rasterize) 과정에서 각 삼각형 내부의 모든 화소들이 찾아지며, 이들은 fragment processor의 화소 단위의(per-pixel) 연산 과정을 거쳐 각각 화소의 색이 결정되어 프레임 버퍼에 기록된다.

그러나 이러한 기존의 고정된 그래픽스 파이프라인에서의 정형적인 연산으로는 보다 다양한 그래픽 효과의 구현이 제한적이므로 사용자들은 vertex processor와 fragment processor 부분의 programmability를 요구하게 되었고 그 결과 2000년 DirectX 8.0 버전에서 이를 지원하는 shader model 1.0이 발표되어 프로그래밍 가능한(programmable) 파이프라인 구조가 탄생하게 되었다. 그 후2002년에 발표된 shader model 2.0과 2004년에 발표된 shader model 3.0에서는 사용할 수 있는 명령어의 개수가 점차 증가되었고 2007년에 명령어 개수의 제한이 없어지고 geometry shader가 포함된 shader model 4.0이 발표되었다.

1. Vertex shader

Vertex shader는 <그림 1>에 도시한 바와 같이 그래픽스 파이프라인의 가장 앞부분에 위치하여 정점 단위의 연산을 수행하는 shader 모델을 의미한다. Vertex shader에서는 기본적으로 3차원 정점을 입력받아 4×4 행렬과 곱하여 좌표계의 변환을 수행하고 vertex의 조명을 계산하여 변환된 좌표와 함께 출력한다. 이와 같이 vertex shader 내부에서는 대부분 기하 변환과 관련된 연산이 이루어지므로 GPGPU의 응용분야에 따라 적절히 사용하거나 아무런 일도 하지 않을 수도 있다. 예를 들어 영상처리에서의 vertex shader의 활용도는 fragment shader에 비해 떨어진다.

2. Fragment shader (pixel shader)

Fragment shader는 <그림 1>에서 보는바와 같이 그래픽스 파이프라인에서 rasterize 단계 이후에 위치하는 화소 단위의 연산 과정을 의미하고 pixel shader라고도 불린다. Fragment shader는 최종적으로 화면에 표현되는 화소의 색을 결정하는 단계이므로 출력의 속성이 일반적인 영상처리의 출력과 유사하게 된다. 또한, 일반적인 GPGPU 응용분야에서도 처리할 데이터를 텍스처 형식으로 설정한 후, fragment shader에

서 이에 접근하여 data-parallel한 방식으로 대용량의 데이터를 나누어 처리한 후 그 결과를 프레임 버퍼 객체(Frame Buffer Object, FBO)라고 불리는 메모리 공간에 출력하는 방식을 취한다. 이와 같은 렌더링 방식을 off-screen 렌더링이라고 한다. 이와 같이 GPGPU를 위해 가장 많이 활용되는 부분이 바로 fragment shader이다.

3. Geometry shader

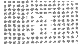
Geometry shader는 shader model 4.0에서 처음 포함되었으며 vertex shader와 rasterize 과정 사이에 위치하여 draw primitive 단위로 연산이 이루어지는 shader 모델을 의미한다. 즉, geometry shader의 입력과 출력은 삼각형의 집합이 되고 geometry shader 내부에서는 삼각형의 위치나 색, 텍스처 좌표들을 변경하거나 삼각형을 지우거나 또는 새로운 삼각형을 추가할 수 있다. Geometry shader는 Cg 2.0 버전[1]과 DirectX 10.0 버전의 HLSL(high-level shading language)[4]을 이용하여 개발할 수 있다. GLSL (OpenGL shading language)[5]에서는 아직 정식으로 포함되지 않았지만 NVIDIA의 OpenGL extension을 통하여 개발할 수 있다. Geometry shader를 이용하게 되면 progressive mesh와 같은 3차원 모델 LOD(Level-Of-Detail)의 효율적 구현이 가능하며, bump 매핑의 모든 연산을 GPU상에서 수행할 수 있고, stencil shadow volume을 이용하여 3차원 모델의 그림자를 빠르게 계산할 수 있는 등의 고수준 그래픽 처리의 GPU 구현이 보다 용이하다.

III. Shading Language

Shader 프로그래밍을 위한 고수준 프로그램 언어로 NVIDIA의 Cg, DirectX의 HLSL, OpenGL의 GLSL등이 존재한다. 본 장에서는 이러한 shading 언어들의 특징에 대하여 기술한다. 이들은 C문법과 유사한 고수준 언어로서, <그림 2>에 Cg와 GLSL, HLSL의 간단한 예제 코드를 도시하였다.

```
/* CG vertex shader */
struct VS_INPUT {
    float4 pos : POSITION;
    float4 color : COLOR0;
};

struct VS_OUTPUT {
    float4 pos : POSITION;
    float4 color : COLOR0;
}

void VS_main ( VS_INPUT In,
               out VS_OUTPUT Out,
               uniform float4x4 mvp : state.matrix.mvp )
{
    Out.pos = mul ( mvp, In.pos );
    Out.color = In.color;
}

/* GLSL vertex shader */
void main ( void )
{
    gl_Position = gl_ModelViewProjectionMatrix
                  * gl_Vertex;
    gl_FrontColor = gl_Color;
}

/* HLSL vertex shader */
float4x4 WorldViewProj : WORLDVIEWPROJ;

struct VS_INPUT {
    float4 pos : POSITION;
    float4 color : COLOR;
};

struct VS_OUTPUT {
    float4 pos : POSITION;
    float4 color : COLOR;
};

VS_OUTPUT VS_main ( const VS_INPUT In )
{
    VS_OUTPUT Out;
    Out.pos = mul ( In.pos, WorldViewProj );
    Out.color = In.color;
    return Out;
}
```

〈그림 2〉 Cg, GLSL, HLSL 예제



1. Cg

Cg는 C for Graphics의 약자로 2002년 NVIDIA에서 개발된 shading 언어이다. NVIDIA는 프로그래밍이 가능한 그래픽스 파이프라인을 지원하는 GPU인 GeForce3을 개발하였지만 하드웨어에 종속적인 저수준 어셈블리 언어를 이용하여 소프트웨어 개발이 이루어졌기 때문에 쉽게 접근하기 힘들었다. 이러한 단점을 극복하기 위하여 C 언어와 문법이 유사하고 하드웨어와 독립적인 고수준 shading 언어인 Cg가 제안되었다. 그 후 2006년 발표된 1.5 버전에서는 GLSL과 HLSL과의 교차 컴파일(cross-compile)을 지원하고 2007년에 발표된 2.0 버전에서는 shader model 4.0의 geometry shader를 지원하게 되었다. 2009년 4월 현재 DirectX10과 GLSL geometry profile을 지원하고 몇몇 버그를 수정한 2.2 버전이 발표되었다.

2. GLSL

GLSL은 OpenGL Shading Language의 약자로서 그 이름에서 알 수 있듯이 OpenGL에서 shader를 사용하기 위한 shading 언어이다. 2004년 OpenGL 2.0 버전에 정식으로 포함되었고 Cg와 마찬가지로 C 언어와 유사한 문법을 갖는 고수준 shading 언어이다. 정식 버전으로는 2006년 8월에 발표된 OpenGL 2.1 버전의 GLSL 1.2 버전이 가장 최신 버전이고 2006년 11월에 GLSL에서 geometry shader를 사용할 수 있는 NVIDIA extension인 EXT_geometry_shader4가 발표되었다.

3. HLSL

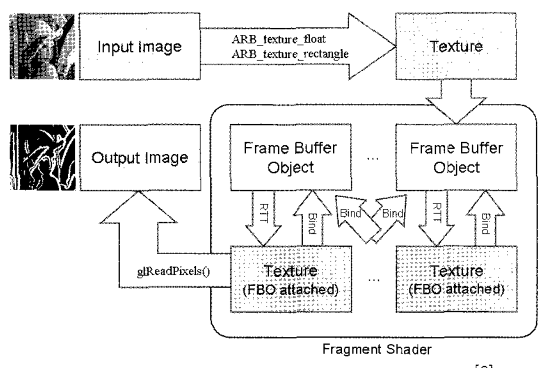
HLSL은 High Level Shader Language의 약자로 DirectX를 위한 shading 언어이다. 2000년 DirectX 8 버전에서 shader model 1.0이 시작되었고 초기에는 어셈블리어와 C 언어를 사용하여 vertex shader만을 제어할 수 있었다. 기본적으로 Cg와 매우 유사한 형태를 갖고 현재 shader model 4.0을 지원하는 Direct3D 10 버전의 HLSL이 가장 최신 버전이다.

IV. Shader를 이용한 멀티미디어 처리 사례

데이터를 고속으로 병렬 처리하기 위하여 shader를 사용하는 경우, 대부분 입력 데이터를 처리하여 결과를 프레임 버퍼 객체에 기록하는 off-screen 렌더링 방식이 일반적이다. 본 장에서는 위와 같은 일반적인 처리 구조를 소개하고, 이를 기반으로 하는 응용 사례를 제시한다.

1. Shader 기반의 멀티미디어 처리 프레임워크

2차원 필터링과 같은 일반적인 영상처리의 경우 render-to-texture를 활용한 off-screen



〈그림 3〉 GPU 기반 영상처리 프레임워크[6]

렌더링 방법을 활용하여 효율적으로 shader를 사용할 수 있다. 우선 입력 영상의 해상도와 동일한 크기를 갖는 frame buffer를 생성한 후 각각의 화소가 fragment shader에서 처리될 수 있도록 화면 전체 크기의 사각형을 생성한다. 이때 입력 영상은 텍스처 데이터로 지정되어 비디오 메모리에 저장되고 vertex shader에서는 기본적인 행렬변환을 수행한다. 그 후 fragment shader에서 각각의 화소에 대하여 임의의 영상처리 과정을 거친 후 그 결과는 PBO(pixel buffer object)나 FBO(frame buffer object) 형식의 비디오 메모리 공간에 저장되고 최종적으로 시스템 메모리로 복사된다. <그림 3>은 위와 같은 GPU 기반 영상처리 프레임워크의 기본 구조를 보여주고 있다. 또한 영상처리가 아닌 데이터의 처리 역시 같은 프레임워크로 매핑될 수 있다.

2. DWT(Discrete Wavelet Transform)

JPEG2000의 인코딩과 디코딩에 이용되는 변환 과정중의 하나인 DWT의 경우도 위와 같은 프레임워크를 이용하여 shader로 구현될 수 있다[6]. 일반적인 콘볼루션 기반의 DWT인 경우 두 번의 렌더링을 통하여 가로방향과 세로방향의 DWT 결과를 얻을 수 있고 결과 영상들은 모두 FBO를 통하여 비디오 메모리에 저장된다. 이때 DWT의 level이 증가할 경우 FBO에 저장되어있는 이전 결과를 그대로 텍스처 형식의 입력 영상으로 사용할 수 있어 메모리 이동에 대한 오버헤드를 줄일 수 있다.

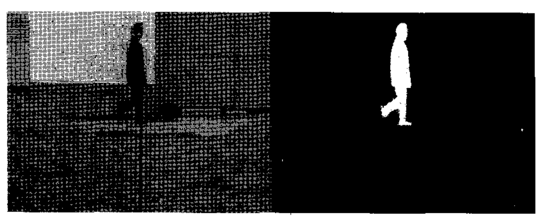
3. 3차원 물체 자세 추정(pose estimation)

3차원 물체 자세 추정은 컴퓨터 비전 분야에서 잘 알려진 문제 중의 하나이고 깊이 영상 기반의 3차원 물체 자세 추정의 경우 GPU의 병렬 연산을 최대로 사용할 수 있는 좋은 예제이다 [7]. 우선 입력된 3차원 거리 영상으로부터 shader를 이용하여 GPU상에서 median 필터와 edge 검출, 그리고 EDT(euclidean distance transform)를 수행한다. 그 후 fragment shader에서 수많은 표본 영상들과 입력 영상간의 병렬 정합을 수행하며 이 때 정의된 오차함수가 화소 단위로 계산된다. 병렬 정합 과정에서는 downhill simplex 최적화 기법이 GPU상에 구현되어 수많은 예제 자세와 입력 영상간의 최적의 평행 이동을 찾게 되고, 최종적으로 최소의 오차를 가지는 예제를 선택함으로써 6-자유도의 최적화 문제를 해결하게 된다. GLSL을 이용한 GPU 코드는 GTX280에서 쿼드 코어 CPU보다 약 115배 고속으로 수행됨을 확인하였다.

〈그림 4〉 움직이는 카메라로부터 검출된 배경[9]

4. 물체 검출

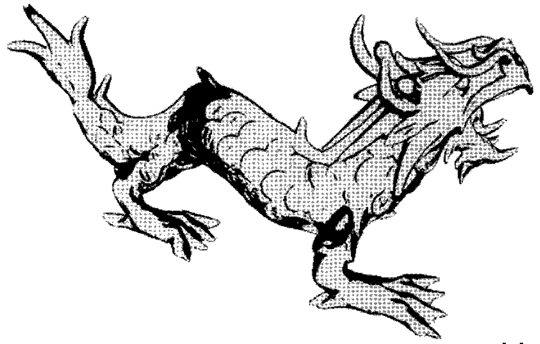
실내 환경에서의 벽 검출은 3차원 환경 모델링이나 SLAM(simultaneous localization and mapping)과 같은 응용프로그램의 중요한 부분 중의 하나이다. Moradi[8]는 스테레오 비전 환경에서 shader를 이용하여 실시간으로 벽을 검출



〈그림 5〉 동영상으로부터 움직이는 물체 검출[10]

할 수 있는 효과적인 방법을 제안하였다. 우선 스테레오 카메라에서 벽의 모서리 부분과 물체에 대한 3차원 point cloud를 얻고 연산량을 줄이기 위하여 view frustum culling을 수행한다. 그 후 가상의 벽을 생성하고 3차원 점들이 벽의 앞쪽이나 뒤쪽 어느 방향에 존재하는지를 계산하여 현재 가상의 벽이 유효한지를 검사한다. 위의 과정에서 view frustum culling은 그래픽스 파이프라인을 통하여 자동으로 계산되고 3차원 점들을 가상의 벽으로 투영하는 과정은 fragment shader에서 쉽게 구현할 수 있다. 실험 결과 CPU의 경우 평균 240ms가 소요되었고 GPU의 경우 평균 16.3ms가 소요되어 약 15배 정도의 수행시간을 단축하였다.

움직이는 카메라로부터 배경과 카메라의 움직임을 예측하는 것은 많은 계산량을 요구한다. Qian[9]은 sliding window 기반의 기법을 shader에 적용하여 입력 동영상에서 고속으로 배경과 카메라의 움직임을 계산하는 방법을 제안하였다 <그림 4>. 우선 카메라의 이동은 2D parametric transformation 이라 가정하고 RANSAC(random sample consensus)을 이용하여 두 프레임 사이의 homography를 예측한다. Sliding window의 기준 프레임은 입력 동영상의 중간 프레임으로 정하고 sequence 프레임을 기준 프레임으로 warping하여 움직임 보정을 계산한다. 실제 구현과정에서는 영상 warping과 배경 계산의 두 단계로 나뉘고 입력 영상은 텍스처 메모리를 이용하여 연산을 수행한다. 실험 결과 sliding window의 크기와 연산 방법에 따라 GPU의 경우 10~18fps의 수행속도를 보여주었고 CPU에 비하여 12~15배 정도 수행시간을 단축하였다.



〈그림 6〉 Shader를 이용한 실시간 카툰 렌더링[11]

위와 유사하게 동영상으로부터 움직이는 물체를 검출하는 것 또한 컴퓨터 비전 분야에서 중요한 주제이다. FUKUI[10]는 입력 동영상에서 배경을 분리하고 background subtraction을 이용하여 움직이는 물체를 검출하는 방법을 shader를 이용하여 실시간으로 처리하였다<그림 5>. 또한 배경의 명암이 바뀔 경우 움직이는 물체를 제대로 검출할 수 없기 때문에 변환표를 이용하여 영상을 보정하였고 CIELAB 색 공간을 이용하여 그림자를 제거하였다.

5. 고품질 고속 렌더링

많은 량의 데이터를 실시간으로 고품질 렌더링하는 것은 비디오게임과 같은 분야에서 매우 중요한 일이다. Livny[11]는 shader에서 3차원 모델로부터 LOD를 이용하여 실시간 카툰 렌더링을 수행하는 방법을 제안하였다<그림 6>. 우선 입력 모델의 표면 메쉬를 부드럽게 표현하기

〈그림 7〉 Shader를 이용한 EWA splatting 렌더링 결과[12]

위하여 삼각형 내부에서 법선벡터만 보간하는 것이 아니라 삼각형 내부의 정점들의 위치 또한 vertex shader에서 변경하여 매끄러운 모델의 외곽선을 계산한다. Fragment shader에서는 각 화소에 해당되는 표면 위치의 법선벡터와 시점 벡터를 이용하여 카툰 컬러에 대한 실루엣 커브를 찾아 카툰 컬러에 적용하고 조명을 고려하여 색의 경계부분을 찾아 검은색으로 매핑 한다. 실험결과 3609K의 정점을 갖고 7218K의 삼각형을 갖는 dragon 모델의 경우 평균 27.1ms의 수행시간을 보여줘 30fps 이상의 실시간 처리가 가능한 것을 확인할 수 있었다.

Ren[12]은 EWA (elliptical weighted average) splatting을 이용하여 고품질의 point based 렌더링을 shader에서 수행할 수 있는 기법을 제안하였다<그림 7>. 우선 EWA splatting을 shader에서 구현하기 위하여 두 단계의 렌더링으로 구분하고 첫 번째 단계에서 모든 surfel들을 깊이 버퍼에 렌더링하여 visibility를 계산한다. 그 후 물체 공간상의 EWA resampling 필터를 생성하여 텍스처가 입혀진 polygon을 화면 공간상으로 투영시키고 화면 공간상의 EWA resampling 필터를 계산하여 물체를 렌더링 한다. 이때 첫 번째 단계에서 시선 방향으로의 깊이 계산과 두 번째 단계에서 surfel polygon의 정점 위치 계산은 vertex shader에서 수행되고 pixel shader에서는 화면 공간상에서의 가중치 합 계산과 알파 채널을 이용한 알파 블렌딩을 수행한다.

V. 결 론

본 기고에서는 GPGPU를 위한 shader가 무엇인지, 그리고 shading language의 종류에는 어떤 것들이 있는지 알아보고 실제 멀티미디어 처리에 shader가 어떻게 사용되었는지 영상처리의 경우를 중심으로 살펴보았다.

최초의 shader 모델 1.0이 발표된 후 현재까지 shading 언어는 꾸준히 발전해 왔고 GPU 기반의 통합 계산 플랫폼인 CUDA 역시 마찬가지로 알파 버전의 발표 후 3년 동안 많은 발전을 이루었다. GPU를 특수한 목적으로 사용하기 위한 방법으로 두 가지가 존재하지만 두 가지 방법의 사용 목적에 차이가 존재하기 때문에 어느 하나가 소멸되지는 않을 것으로 보인다. 두 가지 방법의 차이를 이해하고 자신의 사용 목적을 정확히 파악하여 적응적으로 선택하는 것이 멀티미디어 분야에서의 최선의 GPGPU (General Purpose GPU) 활용이라고 할 수 있다.

참고문헌

[1] http://www.nvidia.com/

[2] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell, “A Survey of General-Purpose Computation on Graphics Hardware,” Computer Graphics Forum, No.26, pp.80-113, March, 2007.

[3] Y. Luo and R. Duraiswami, "Canny Edge Detection on NVIDIA CUDA," Proc. Workshop on Visual Computer Vision on GPU's (CVGPU), June, 2008.

[4] http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx

[5] R. Rost, OpenGL Shading Language Second Edition, Addison-Wesley, 2006.

[6] 이만희, 박인규, 원석진, 조성대, "GPU를 이용한 DWT 및 JPEG2000의 고속 연산," 전자공학회 논문지, Vol.44-SP, No.6, pp.9-15, 2007년 11월.

[7] M. Germann, M. D. Breitenstein, I. K. Park, and H. Pfister, "Automatic pose estimation for range images on the GPU," Proc. Sixth International Conference on 3-D Digital Imaging and Modeling (3DIM07), pp.81-88, October, 2007.

[8] H. Moradi, E. Kwon, D.N. Sohn, and J.H. Han, "A Real-Time GPU-Based Wall Detection Algorithm for Mapping and Navigation in Indoor Environments," Lecture Notes in Computer Science, Vol.4558, pp.1072-1077, July, 2007.

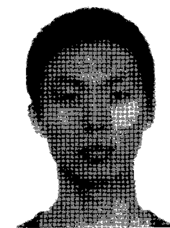
[9] Y. Qian and G. Medioni, "A GPU-based implementation of motion detection from a moving platform," Proc. Workshop on Visual Computer Vision on GPU's (CVGPU), June, 2008.

[10] S. Fukui, Y. Iwahori, and R. Woodham, "GPU based extraction of moving objects without shadows under intensity changes," Proc. IEEE World Congress on Evolutionary Computation, pp.4165-4172, June, 2008.

[11] Y. Livny, M. Press, and J. El-Sana, "Interactive GPU-based adaptive cartoon-style rendering," The Visual Computer, Vol.24, No.4, pp.239-247, March, 2008.

[12] L. Ren, H. Pfister, and M. Zwicker, "Object Space EWA Surface Splatting : A Hardware Accelerate Approach to High Quality Point Rendering," Proc. EUROGRAPHICS 2002, Vol.21, No.3, pp.461-470, September, 2002.

저자소개



이 만 희

2006년 2월 인하대학교 컴퓨터공학과 공학사
2008년 8월 인하대학교 정보공학과 공학석사
2008년 9월 ~ 현재 인하대학교 정보공학과 박사과정
2007년 4월 ~ 2008년 2월 한국전자통신연구원(ETRI) 위촉연구원

주관심 분야 : GPGPU, 영상기반 모델링 및 렌더링

박 인 규

1995년 2월 서울대학교 제어계측공학과 공학사
1997년 2월 서울대학교 제어계측공학과 공학석사
2001년 8월 서울대학교 전기컴퓨터공학부 공학박사
2001년 9월 ~ 2004년 3월 삼성종합기술원 멀티미디어 랩 전문연구원
2007년 1월 ~ 2008년 2월 Mitsubishi Electric Research Laboratories(MERL) 방문연구원
2004년 3월 ~ 현재 인하대학교 정보통신공학부 조교수

주관심 분야 : 컴퓨터 그래픽스 및 비전 (영상기반 3차원 형상 모델링 및 렌더링, computational photography), GPGPU