한국컴퓨터그래픽스학회 Vol. 14, No. 1, P. 17~25

프로그래밍 가능한 GPU를 이용한 포토 모자이크

강동완 윤경현

중앙대학교 공과대학 컴퓨터공학과

dongwann@cglab.cse.cau.ac.kr, khyoon@cau.ac.kr

Photomosaic using a programmable GPU

Dongwann Kang Kyung-Hyun Yoon

Dept. of Computer Science & Engineering, Chung-Ang Univ.

요 약

본 논문은 프로그래밍 가능한 GPU를 이용한 포토 모자이크 생성 방법을 제안한다. 그래픽스 파이프라인을 통해 포토 모자이크를 생성할 수 있도록 정점을 디자인하고, 타일로 사용할 영상 데이터베이스의 텍스처 표현을 제시한다. 정점 셰이더에서는 텍스처에 저장된 입력 영상과 타일 영상들을 이용해 최적 타일을 찾고, 프래그먼트 셰이더는 이것을 프레임 버퍼에 그림으로써 포토 모자이크를 생성한다. 본 논문에서 제안한 방법은 최적 타일을 찾는 기존의 포토 모자이크 알고리즘에 비해 월등히 빠른 결과를 보여준다.

Abstract

We proposed the method for photomosaic generation using a programmable GPU. We design vertices to generate a photomosaic through a graphics pipeline and suggest a texture representation of an image database whice is used for tile. Both the source image and the tiles are stored to texture, which are matched by a vertex shader and drawn by a fragment shader. This is much faster than several techniques which achieve the best match for each tile.

키워드: 타일 매칭, 비사실적 렌더링, GPGPU

Keywords: Tile Matching, Non-Photorealistic Rendering, GPGPU

제 1 절 서론

포토 모자이크는 입력 영상을 작은 사진 타일들의 모자이크로 나타내는 표현양식이다. Silvers[1] 에 의해 제안된 포토 모자이크 기법은 일반적으로 다음 단계들을 거쳐 생성된다:

1. 입력 영상을 사진 타일이 대체할 사각형들로 나눈다.
2. 데이터베이스에서 각 사각형과 가장 유사한 영상을 찾는다.
3. 가장 유사한 것으로 찾아진 영상들로 각 사각형을 대체한다.

Silvers가 포토 모자이크 기법을 제안한 이후, Finkelstein 과 Range[2] 는 사각형 이외의 형태의 타일(예를 들면, 육각형 모양의)을 사용하고 색상 보정 기법을 적용한 포토 모자이크 기법을 제안했다. 또한 Kim 과 Pellacini[3] 는 임의의 모양의 타일들을 사용할 수 있도록 포토 모자이크 기법을 더욱 확장시켰다. 이 외에도 비디오 모자이크[4] 와 스택커블 모자이크[5] 등의 다양한 기법들이 개발되었다.

Tran[6] 은 포토 모자이크의 *효과*와 *비용*을 측정하는 방법을 제안하였다. *효과*는 입력 영상과 포토 모자이크 영상 간의 유사성을 이용해 평가하였고, *비용*은 포토 모자이크 알고리즘의 실행 시간을 이용해 평가하였다. 일반적으로 방대한 데이터베이스는 입력 영상에 더욱 근접한 영상을 찾을 확률을 높여주기 때

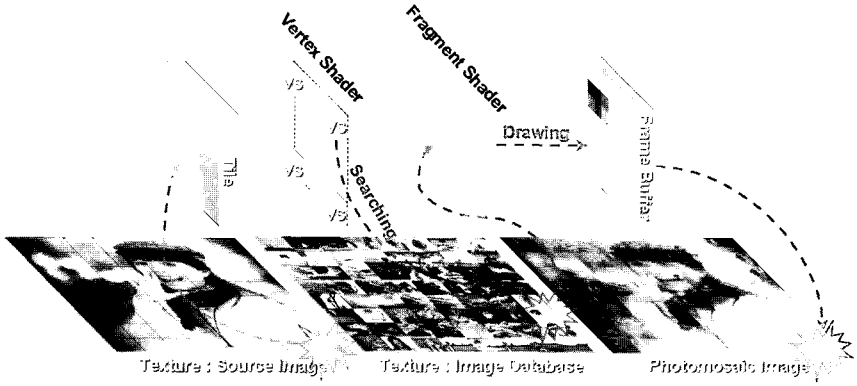


그림 1: 본 논문의 포토 모자이크 프레임워크: 정점 세이더는 데이터베이스 영상이 담긴 텍스처에서 가장 유사한 영상을 찾는다. 프래그먼트 세이더는 정점 세이더가 선택한 텍스처 좌표를 받아 프레임 버퍼에 해당 텍셀을 그린다.

문에 효과적인 포토 모자이크를 생성하는데 도움을 준다. 데이터베이스에서 유사한 영상을 찾는 고차원 검색은 포토 모자이크 알고리즘의 주된 병목 지점이다. 따라서 효율적 검색은 포토 모자이크 알고리즘의 디자인에서 중요한 이슈에 해당한다.

또한 Tran은 포토 모자이크 알고리즘의 평가 항목으로 *타일의 다양성*을 제안하기도 하였다. 타일의 다양성을 증가시켜야 한다는 그의 아이디어는 이후의 여러 포토 모자이크 애플리케이션들에서 다양한 방법으로 적용되었다.

Blasi 와 *Petralia*[7] 는 선형검색보다 월등히 빠른 antipole 트리 구조[8] 를 이용해 포토 모자이크의 검색 속도 향상을 달성했다. 하지만 antipole 트리 구조는 근사 검색을 하기 때문에, 이것을 이용한 포토 모자이크 결과는 최적 타일 검색을 하는 포토 모자이크 알고리즘에 비하여 효과가 떨어진다.

본 논문은, 포토 모자이크 생성 비용을 줄이기 위해, GPU의 병렬 처리를 이용한 검색을 통해 포토 모자이크를 생성한다. 포토 모자이크의 각 타일 검색은 서로 독립적으로 수행되기 때문에 병렬 알고리즘에 매우 적합하다. 본 알고리즘은 전역 검색(exhaustive search)을 이용한다. 따라서 전역 검색을 사용한 기존 알고리즘들에 비해 현저히 빠르며, 가장 효과적인(Tran의 평가방법에 따른) 결과를 생성한다.

제 2 절 알고리즘

2.1 알고리즘의 개요

앞에서 설명한 일반적인 포토 모자이크 알고리즘의 단계 2와 3은 각각 GPU의 서로 다른 단계에 배정된다. 그림 1과 같이, 유사한 영상을 검색하는 것은 정점 세이더에서, 검색한 영상을 그리는 것은 프래그먼트 세이더에서 수행된다. 정점 세이더는 텍스처에 저장된 입력 영상을 고려하여, 데이터베이스가 저장된 다른 텍스처에서 유사한 영상을 검색한다. 프래그먼트 세이더는 정점 세이더가 선택한 영상의 텍스처 좌표를 전달받아 프레임버퍼에 해당 텍셀을 그린다. 이 과정에서 세이더들이 참조할 텍스처들과 정점들은 CPU에서 선처리 된다.

다음 절에서는 알고리즘의 세부를 살펴본다.

2.2 텍스처 표현

입력 영상을 담은 텍스처는 OpenGL이나 Direct3D와 같은 3D API를 이용해 영상을 로딩함으로써 손쉽게 만들 수 있다.

현재의 GPU가 수백장 혹은 그 이상의 텍스처들을 지원하지 않기 때문에, 타일 영상 데이터베이스를 텍스처로 변환하는 것은 조금 더 복잡하다. 데이터베이스는 제한된 수와 크기의 텍스처로 변환되어야 한다. 포토 모자이크의 타일 영상은 일반적으로 매우 작으므로, 데이터베이스내의 영상들의 해상도는 포토 모자이크의 타일 크기만큼 줄어들어도 무방하다. 본 알고리즘에서는 해상도를 낮춰 정규화한 영상들을 하나의 텍스처로 조합하였다(그림 2). 비록 최근의 GPU들이 텍스처 크기에 대한 엄격한 제약(2의 승수 크기와 같은)을 강요하지는 않지만, 정밀도를 위해 텍스처의 크기는 2의 승수로 주어지는 것이 바람직하다.

2.3 정점 구성

정점 세이더는 입력으로 정점들을 필요로 하기 때문에, 정점은 포토 모자이크의 사각 타일들의 각 코너에서 생성되어야 한다. 각각의 타일에서의 최적 영상 선택은 정점 세이더에서의 검색을 필요로 하기 때문에, 그림 3(a)와 같이 타일당 하나의 정점이 필요한 것처럼 보인다. 타일의 네 코너중 한 코너에 위치한 정점에만 접근하는 것은 효율적으로 보이지만, 이것은 중요한 문제

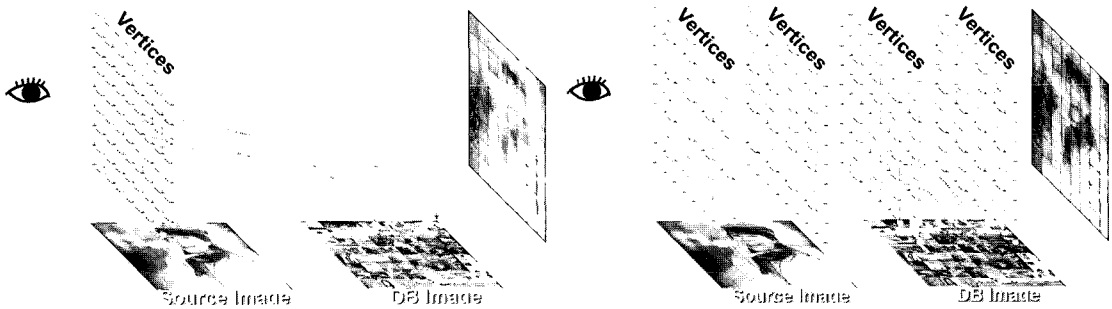


그림 3: (a) '타일당 하나의 정점(검색)'은 잘못된 참조를 야기한다. (b) 이 문제를 해결하기 위해, 타일들이 정점을 공유하지 못하도록 하였다((b)는 정점을 공유하지 않는 타일 구조를 보여주는 한 예시일 뿐이며, 그림과 같이 인접 타일 간의 깊이 값이 서로 달라야 하거나 몇 개의 층으로 구성되어야 할 필요는 없음을 알린다.).

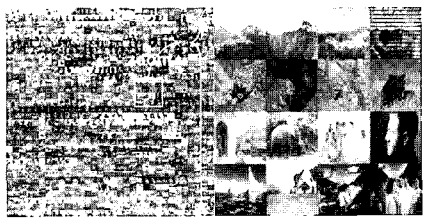


그림 2: 영상 데이터베이스 텍스처 : 1,024장의 영상을 담고 있는 데이터베이스의 텍스처 표현. 데이터베이스의 모든 영상은 정규화되어 텍스처를 구성한다.

를 야기한다. 정점 셰이더가 그것의 결과(색상이나, 노말, 텍스처 좌표 등의)를 프래그먼트 셰이더에 전달하때, 그 값은 정점들 사이에서 보간된다. 따라서 각 정점 셰이더가 검색한 유사한 영상의 텍스처 좌표는 정점들 사이에서 보간되어 전달되므로, 검색한 영상과 무관한 텍셀을 지시하게 된다. 따라서 그림 3(a)의 정점 구조는 사용될 수 없다.

이 문제를 해결하기 위해 본 논문에서는 타일들이 정점들을 공유하지 않도록 하는 정점 구조를 구성하였다(그림 3(b)). 이 구조에서는 각 타일마다 네개의 코너에 위치한 정점들에서 동일한 검색을 수행한다. 이것은 비효율적이지만, 정점 셰이더가 정확한 텍스처 좌표를 프래그먼트 셰이더에 전달하게끔 한다.

정점 셰이더는 입력 영상이 담긴 텍스처에 접근함으로써 입력 영상에서의 현재 타일에 해당하는 영역을 얻는다. 이것은 각 정점들의 텍스처 좌표를 $(x/width,\ y/height)$으로 설정함으로써 가능하다.

위와 같은 텍스처 좌표에서는 동일한 텍스처 좌표가 서로 다른 타일에 해당되는 정점들에 배정된다(그림 4). 따라서 텍스처 좌표만으로는 어느 타일의 정점인지 알 수 없으므로, 해당 정점들은 서로 구분되어져야 한다. 이를 위해, 본 논문에서는 각 타일의 네 코너의 정점들에 미리 지정된 서로 다른 색상을 부여했다. 이 색상들은 정점의 타일 내 상대적 위치를 지시해주는 플래그 역할을 한다.

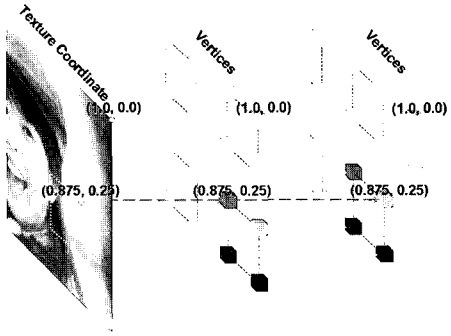


그림 4: 서로 다른 타일에 해당하는 정점에 동일한 텍스처 좌표가 부여되기 때문에, 정점의 타일 내 상대적 위치를 구분하기 위한 플래그로써 미리 지정된 서로 다른 색상이 사용되었다.

2.4 정점 셰이더를 이용한 영상 검색

정점 셰이더는 입력 영상의 텍스처 좌표 v를 이용해 텍셀에 접근할 수 있다. 그리고 유닛 타일 사이즈 u와 텍스처 좌표계에서의 텍셀 사이즈 t를 이용해 현재의 정점에 대응하는 타일의 텍셀들의 좌표를 구할 수 있다. 타일의 좌측 상단 정점을 기준으로 구한 b는 다음과 같다:

$$b = (v_x + t_x \cdot i, v_y + t_y \cdot j)(0 \le i, j < u). \quad (1)$$

다음으로, 정점 세이더는 유사한 방법으로 타일 크기와 데이터베이스 텍스처 좌표, 유닛 텍셀의 크기를 이용해 영상 데이터베이스에 접근한다.

이제 세이더는 입력 영상의 대상 타일과 가장 유사한 영상을 데이터베이스에서 찾을 준비가 되었다. 포토 모자이크를 위한 영상 검색에는 많은 방법들이 적용되어질 수 있다. 예를 들어, 웨이블릿 변환[9, 10] 을 이용한 빠른 다중 해상도 영상 검색 방법[11] 이 포토 모자이크의 확장 기법인 비디오 모자이크[4] 에 적용된 바 있고, antipole 트리구조[8] 가 포토 모자이크의 속도 향상을 위해 사용되기도 하였다. 하지만 이 방법들은 근사 결과를 도출하는데 그친다. 영상과 같은 다차원 데이터의 효율적 검색은 매우 어렵기 때문에, 이 알고리즘들의 포토 모자이크 효과(Tran이 제안한 입력과 결과의 유사도)는 상대적으로 낮다.

본 논문에서는 전역 검색(exhaustive search)을 이용하여 가장 유사한 영상을 찾았다. 전체 데이터베이스 텍스처(b)에 접근하여 입력 영상의 각 타일(a)로부터의 L_2거리인 $d_{a,b}$가 가장 작은 것을 찾았다.

$$c_{i,j} = \sqrt{(r^a_{i,j} - r^b_{i,j})^2 + (g^a_{i,j} - g^b_{i,j})^2 + (b^a_{i,j} - b^b_{i,j})^2}. \quad (2)$$

$$d_{a,b} = \sqrt{\sum_{i=1}^{w}\sum_{j=1}^{h}(c_{i,j})^2}. \quad (3)$$

각 타일의 네 코너에 대응하는 프래그먼트로 전달되는 텍스처 좌표는 데이터베이스내 가장 유사한 영상의 네 코너의 좌표이다. 따라서 검색된 영상의 텍셀들은 정확히 보간되어 프래그먼트 세이더로 전달된다.

본 논문의 알고리즘은 중복되는 타일 선택에 대해 고려하지 않았다. 하지만 타일의 중복을 줄이는 것은 포토 모자이크 응용 단계에서 중요하게 여겨지기도 한다. 본 논문은 우선순위 큐를 이용해 인접한 타일들이 서로 다른 n의 n번째 유사 영상을 선택하게 함으로써 중복된 타일의 수를 줄일 수 있도록 하는 방법을 제공한다. 본 논문의 병렬 알고리즘에서는 인접한 타일에서의 검색 결과를 알수 없기 때문에, 이 방법은 완벽한 중복 회피를 보장하지는 못한다.

그림 5는 영상 검색을 위해 본 논문에서 사용한 GLSL 코드이다.

2.5 프래그먼트 세이더를 이용한 드로잉

프래그먼트 세이더는 영상 데이터베이스에서 선택된 타일을 프레임버퍼에 그림으로써 포토 모자이크를 완성한다. 원래 포토

```
uniform sampler2D srcTex, DBTex;
//source image and database texture
uniform vec2 TexelSizeDB, TexelSizeSrc;
//texel size of database and source image
uniform int nTileSize, nDBCountPerSide;
//real size of tile, # of image per a side of db texture
uniform float fDBStep, fMAX_GL_FLOAT;
//size of a tile on db texture, maximum value of GL_FLOAT
uniform vec2 unitTileSize;
//size of a tile on source image texture
varying vec2 DBindex;
//texture coordinates of best match image

void main()
{
   ivec2 ij, ab;
   vec2 offsetDB, temp, corner;
   vec3 sampledColorSrc, sampledColorDB, tempVec;
   float minDistance, fDistance;
   gl_TexCoord[0] = gl_MultiTexCoord0;
   minDistance = fMAX_GL_FLOAT;

   corner = vec2(0,0);
   if(gl_Color.r == 1)
      corner.s = -unitTileSize.s;
   if(gl_Color.g == 1)
      corner.t = -unitTileSize.t;
   ab = ivec2(0,0);
   for(; ab.y<nDBCountPerSide; ab.y += 1){
      ab = ivec2(0,ab.y);
      for(; ab.x<nDBCountPerSide; ab.x += 1){
         fDistance = 0.;
         offsetDB = ab*fDBStep;
         ij = ivec2(0,0);
         for(; ij.y<nTileSize; ij.y += 1){
            ij = ivec2(0,ij.y);
            for(; ij.x<nTileSize; ij.x += 1){
               temp = gl_TexCoord[0] + ij*TexelSizeSrc + corner;
               sampledColorSrc = texture2D(srcTex,temp);

               temp = offsetDB + ij*TexelSizeDB;
               sampledColorDB = texture2D(DBTex,temp);

               tempVec = sampledColorSrc - sampledColorDB;
               fDistance += dot(tempVec,tempVec);
               if(fDistance > minDistance){
                  ij = ivec2(nTileSize,nTileSize);
                  break;
               }
            }
         }
         if(fDistance <= minDistance){
            minDistance = fDistance;
            DBindex = offsetDB;
         }
      }
   }
   if(gl_Color.r == 1)
      DBindex.s += fDBStep;
   if(gl_Color.g == 1)
      DBindex.t += fDBStep;
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

그림 5: 유사 영상 검색을 위한 정점 세이더의 GLSL 코드

```
uniform sampler2D DBTex, srcTex;
//source image and database texture
uniform float fBlending;
//blending rate - k
varying vec2 DBindex;
//texture coordinates of best match image

void main()
{
   vec4 colorDB, colorSrc;

   colorDB = texture2D(DBTex, DBindex.st);
   colorSrc = texture2D(srcTex, gl_TexCoord[0].st);
   colorDB += (colorSrc-colorDB)*fBlending;
   colorDB.a = 1.;

   gl_FragColor = colorDB;
}
```

그림 6: 검색한 타일을 프레임 버퍼에 그리는 프래그먼트 셰이더의 GLSL 코드.

모자이크는 데이터베이스에서 선택한 타일을 수정없이 사용하는 것을 원칙으로 하지만 최근의 알고리즘들은 보다 효과적인 포토 모자이크 결과를 만들기 위해 각 타일의 색상 보정을 허용한다. Finkelstein과 Range[2] 는 선택된 타일에 대한 히스토그램 변환을 통해 보다 입력 영상에 가까운 결과를 얻는 방법을 제안했다. 이 방법이 후처리 과정에서 이미 선택된 타일을 대상으로 적용된다면, 히스토그램 변환을 거침으로써 더 입력 영상에 가까워지는 영상이 데이터베이스내에 존재할 가능성이 있다. 따라서 가장 효과적인 결과를 얻기 위해서는 매번의 검색단계에서 데이터베이스의 모든 영상에 대해 히스토그램 변환을 수행하여야 한다.하지만 이것은 지나치게 많은 비용을 요구한다.

본 논문에서는 다음 수식과 같은 간단한 색상 혼합 방법을 이용하였다:

$$c_{frame\_buffer} = k \cdot c_{source} + (1-k) \cdot c_{best\_match}. \quad (4)$$

사용자는 위의 식의 k에 적절한 값을 줌으로써 혼합 비율을 조절할 수 있다. k가 0이라면, 데이터베이스로부터 선택한 타일의 색상은 변하지 않는다. 하지만 k가 1이라면, 타일은 입력 영상과 동일해진다. 이 방법은 후처리단계에서 적용되더라도 그 결과가 항상 최적 매칭임을 보장한다. 또한 프래그먼트 셰이더에서 간단히 구현될 수 있다.

그림 6은 검색한 타일을 그리는 과정을 수행하는 GLSL 코드이다.

제 3 절 결과

본 알고리즘은 Microsoft Windows상에서 표준 OpenGL API와 OpenGL 셰이딩 언어(GLSL[12])를 이용해 C++로 구현되었다. 2.3장의 프로그램은 조건 분기와 반복문을 사용하고 있다. 따라서 본 알고리즘은 셰이더 모델 3.0 혹은 그 이상의 버전을 필요로 한다. 본 논문의 모든 실험은 Pentium IV 2.4 GHz, 4 GB RAM, NVIDIA GeForce 8800 GT GPU를 이용해 수행되었다.

본 논문에서는 제안한 알고리즘과 다른 포토 모자이크 알고리즘의 성능을 비교하였다. Tran[6] 의 연구에서처럼 입력 영상과 결과 영상의 L_2거리($[0, 255]$로 정규화 함)를 이용해 *효과*를 평가했고, 실행 시간(초)으로 *비용*을 평가했다. 비교된 포토 모자이크 알고리즘은 C++로 구현되었다:

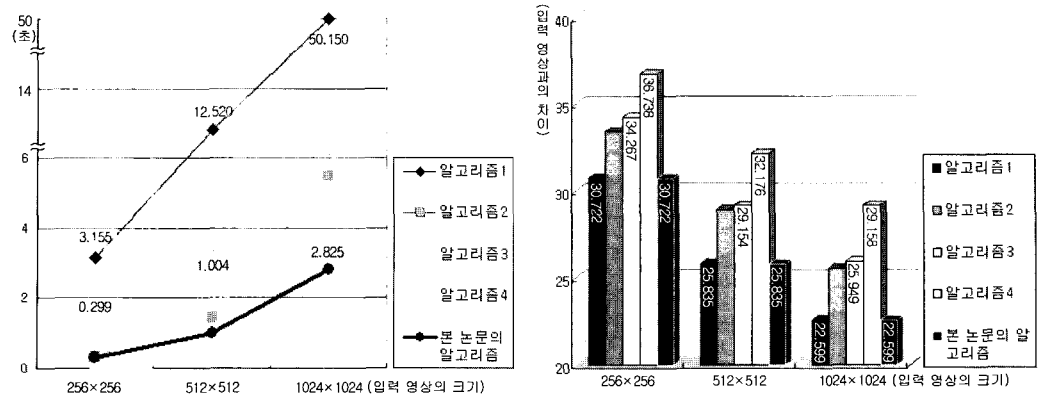
1. 최소 L_2 거리를 갖는 영상을 전역 검색을 통해 찾는 알고리즘.
2. 낮은 해상도(4×4 픽셀)의 타일들로 알고리즘 1을 수행한 알고리즘.
3. Antipole 트리 자료구조를 이용한 Blasi 와 Petralia[7] 의 알고리즘.
4. 낮은 해상도(4×4 픽셀)의 타일들로 알고리즘 3을 수행한 알고리즘.

그림 7은 입력 영상의 크기에 대한 각 알고리즘의 성능을 보여준다. 16×16 크기의 타일과 4,096장의 영상 데이터베이스가 사용되었고, 색상 혼합과 중복 타일 제거는 적용되지 않았다. 이 실험은 본 논문에서 제안한 알고리즘의 영상 크기 증가에 따른 수행 시간의 변화 폭이 상대적으로 작음을 보여준다.

본 논문에서 제안한 알고리즘은 최적 매칭을 보장하지만 같은 결과를 생성하는 알고리즘 1에 비해 훨씬 빠르다. 의심의 여지없이 알고리즘 4는 가장 빠르지만, 효과면에서는 가장 취약하다. 알고리즘 3은 알고리즘 4에 비해 더 나은 효과를 보여주지만, 낮은 해상도에서 수행되는 알고리즘 2보다 좋지 못하다.

그림 8은 데이터베이스의 크기와 성능간의 관계를 보여준다. 16×16크기의 타일과 512×512 크기의 입력 영상이 사용되었고, 색상 혼합과 중복 타일 제거는 사용되지 않았다. 데이터베이스의 크기가 증가할수록 비용이 증가하지만, 효과도 나아짐을 알 수 있다. 본 알고리즘은 알고리즘 1에 비해 훨씬 낮은 비용으로 최적 매칭의 결과를 제공한다. 알고리즘 2와 속도면에서 비슷하지만, 데이터베이스의 크기가 증가할수록 더 나은 속도를 보여준다. 비록 알고리즘 4가 본 논문에서 제안한 알고리즘보다 월등히 빠르지만, 데이터베이스 크기가 증가할수록 효과면에서의 격차는 점점 커진다. 그림 9는 그림 8에서 16,384장의 데이터베이스를 이용해 실험한 포토 모자이크 결과들을 보여준다.

그림 10은 색상 혼합 비율을 조절하는 변수 k에 따른 변화를 보여준다. k가 증가할수록, 결과는 입력 영상에 가까워지지만, 다소 인위적인 느낌을 준다.



(a) 수행 시간

(b) 입력 영상과의 차이

그림 7: 입력 영상의 크기에 따른 다양한 알고리즘의 성능 비교

그림 11은 중복 타일 감소 효과를 보여준다. 본 논문에서 제안한 중복 타일 제거 방법은 완벽하지 않지만, 나쁘지 않은 결과를 만들어낸다.

제 4 절 결론

본 논문에서는 GPU를 이용해 병렬적으로 검색을 수행함으로써 최적 매칭을 보장하는 기존 알고리즘에 비해 월등히 빠르게 포토 모자이크를 생성했다. 비록 antipole 트리를 이용하는 근사 검색 알고리즘[7] 보다 빠르지는 않지만, 본 알고리즘은 실시간을 요구하지 않는 포토 모자이크의 특성상 효과와 비용의 측면에서 적절한 균형을 제공한다.

본 알고리즘의 구현은 각 타일마다 네번의 동일한 검색을 필요로 한다. 최근에 발표된 Direct3D 10[13] 은 정점 세이더와 프래그먼트 세이더 사이에서 새롭게 추가된 기하 세이더를 제공하는데, 이것은 새로운 정점을 만드는 기능을 제공한다. 이 기능을 이용한다면 동일한 검색을 반복하는 것을 피할 수 있으므로 개선된 성능을 얻을 수 있을것으로 보인다. 또한 기하 세이더는 기하 프리미티브에 접근 가능하므로, 중복 타일의 제거에도 기여할 수 있을 것이다.

본 논문은 단순 선형 검색을 수행햐였지만, 고차원 데이터의 빠르고 '정확한' 검색 방법을 GPU 상에서 구현한다면, 포토 모자이크 성능의 비약적 향상을 얻을 수 있을것이다.

Acknowledgements

이 논문은 2007년도 정부(과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(No.R01-2005-000-10940-0).

참고 문헌

[1] R. Silvers and M. Hawley, *Photomosaics*. New York, NY, USA: Henry Holt and Co., Inc., 1997.

[2] A. Finkelstein and M. Range, "Image mosaics," in *EP '98/RIDT '98: Proceedings of the 7th International Conference on Electronic Publishing, Held Jointly with the 4th International Conference on Raster Imaging and Digital Typography*. London, UK: Springer-Verlag, 1998, pp. 11–22.

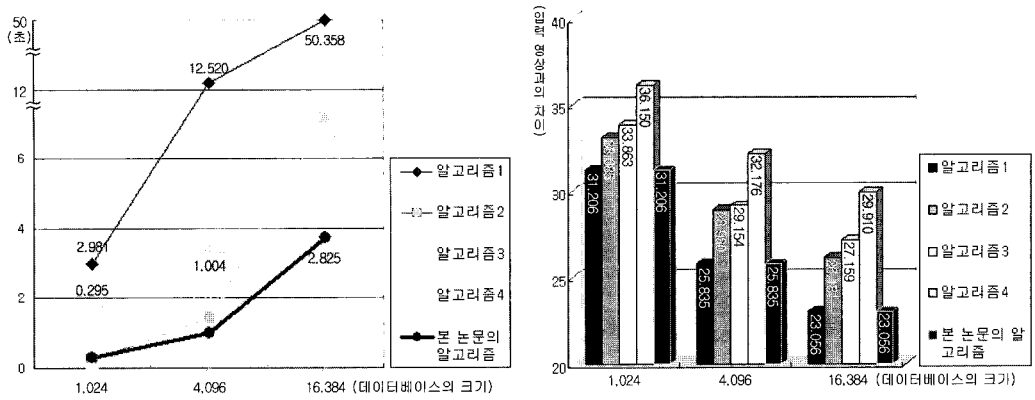
[3] J. Kim and F. Pellacini, "Jigsaw image mosaics," in *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 2002, pp. 657–664.

[4] A. W. Klein, T. Grant, A. Finkelstein, and M. F. Cohen, "Video mosaics," in *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*. New York, NY, USA: ACM, 2002, pp. 21–28.

[5] J. Park, K. Yoon, and S. Ryoo, "Multi-layered stack mosaic with rotatable objects," in *Computer Graphics International*, 2006, pp. 12–23.

[6] N. Tran, "Generating photomosaics: an empirical study," in *SAC '99: Proceedings of the 1999 ACM symposium on Applied computing*. New York, NY, USA: ACM, 1999, pp. 105–109.

[7] G. D. Blasi and P. Maria, "Fast photomosaic," in *WSCG '05: Poster Proceedings of the 13th International Conference in*



(a) 수행 시간

(b) 입력 영상과의 차이

그림 8: 데이터베이스의 크기에 따른 다양한 알고리즘의 성능 비교

Central Europe on Computer Graphics, Visualization and Computer Vision'2005, 2005, pp. 15–16.

[8] D. Cantone, A. Ferro, A. Pulvirenti, D. R. Recupero, and D. Shasha, "Antipole tree indexing to support range search and k-nearest neighbor search in metric spaces," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 4, pp. 535–550, 2005.

[9] E. J. Stollnitz, T. D. DeRose, and D. H. Salesin, "Wavelets for computer graphics: A primer, part 1," *IEEE Comput. Graph. Appl.*, vol. 15, no. 3, pp. 76–84, 1995.

[10] E. J. Stollnitz, T. D. DeRose, and D. H. Salesin, "Wavelets for computer graphics: A primer, part 2," *IEEE Comput. Graph. Appl.*, vol. 15, no. 4, pp. 75–85, 1995.

[11] C. E. Jacobs, A. Finkelstein, and D. H. Salesin, "Fast multiresolution image querying," in *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1995, pp. 277–286.

[12] R. J. Rost, *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006.

[13] D. Blythe, "The direct3d 10 system," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 724–734, 2006.

(a) 본 논문에서 제안한 알고리즘과 알고리즘 1의 결과

(b) 알고리즘 2

(c) 알고리즘 3

(d) 알고리즘 4

그림 9: 16,384개의 데이터베이스를 이용해 그림 8에서 실험한 포토 모자이크 결과들. 16×16 크기의 타일과 512×512 크기의 입력 영상이 사용되었고, 색상 혼합과 중복 타일 제거는 사용되지 않았다.

(a) $k = 0.0$ (b) $k = 0.33$

그림 10: 색상 혼합 효과: 512×512 크기의 입력 영상과 16×16 크기의 타일, 4,096장의 영상 데이터베이스가 사용되었고, 중복 타일 제거는 사용되지 않았다.

(a) 중복 타일 미제거. (b) 중복 타일 제거.

그림 11: 중복 타일 제거 효과: $1,024 \times 1,024$ 크기의 입력 영상과 16×16 크기의 타일, 16,384장의 영상 데이터베이스가 사용되었고, 색상 혼합은 사용되지 않았다.