DXT 압축 알고리즘

김성익(http://gamecode.org)

2009.10.11

Spirit of Flame
3D RealTime Graphics Programming Study
Kasa

DXT 압축

- 빠른 디코딩이 가능한 압축 알고리즘
  - 실시간 디코딩이 가능
  - 거의 모든 비디오 카드에서 실시간으로 디코딩
  - 대역폭을 적게 사용하여 텍스처 메모리 대역폭의 한계를 높여줌
- 최적화를 위해서 사용
  - 묻지도 따지지도 말고 사용
- 컨버터를 사용 오프라인 압축
  - DDS 포맷으로 저장
  - Nvidia 의 포토샵 플러그인
  - DirectX 기본 컨버터 (DirectX Texture Tool)
  - ATI의 compressonator
- 알고리즘에 대해 몰라도 됨. 사용시 DXT1, DXT3, DXT5의 특성만 고려

갑자기 왠 DXT 압축 알고리즘?

- 절차적 텍스처
  - 런타임에서 텍스처 생성
  - 멀티코어를 활용, 백그라운드 압축
- 런타임 텍스처 합성
  - 사례 : Allegorithmic "Substance Air"
    - a new genetation of texture
    - http://www.allegorithmic.com
    - 감동적~
  - 새로운 시대 예감

- DXT 손실 특성을 알자~

DXT 알고리즘 특징

- 4x4 단위로 블럭화
  - 한 텍셀 read 시 해당 텍셀이 포함된 16픽셀이 내부 비디오 캐시로 이동
  - 실제 렌더링시 Bilinear Filtering 등 아주 근접한 텍스처를 읽어들이는 경향이 있음
  - 렌더링 특성상 연속된 텍셀을 읽어들이는 경향
  - 다음에 해당 블럭 중에서 읽어들일 확률이 매우 높음 (cache hit!!)

- 압축률이 높다
  - dxt1 기준으로 보면 4x4 이미지를 8바이트로 압축
  - 비압축 16비트 텍스처의 경우 2*16 => 32 바이트가 필요 (1/4 압축)
    (X8R8G8B8 텍스처에 메모리를 가지고 있는 경우와 비교하면 1/8 수준)

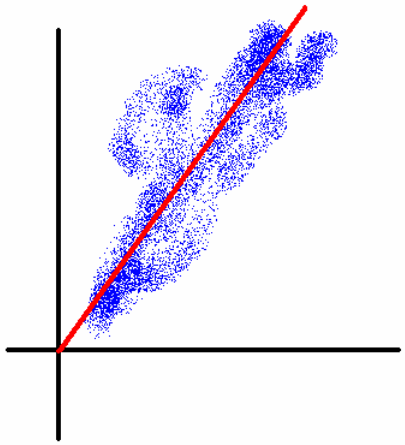
- 손실 압축
  - 실제적인 이미지 손실이 생김 (이미지의 특성과 연관)
  - 시각적으로 납득할만한 수준 (거의 대부분 게임에서 사용)
  - 특성이 있음

- 아주 특별한 경우를 제외하고는 무조건 사용 !!

DXT1 기본 알고리즘(1)

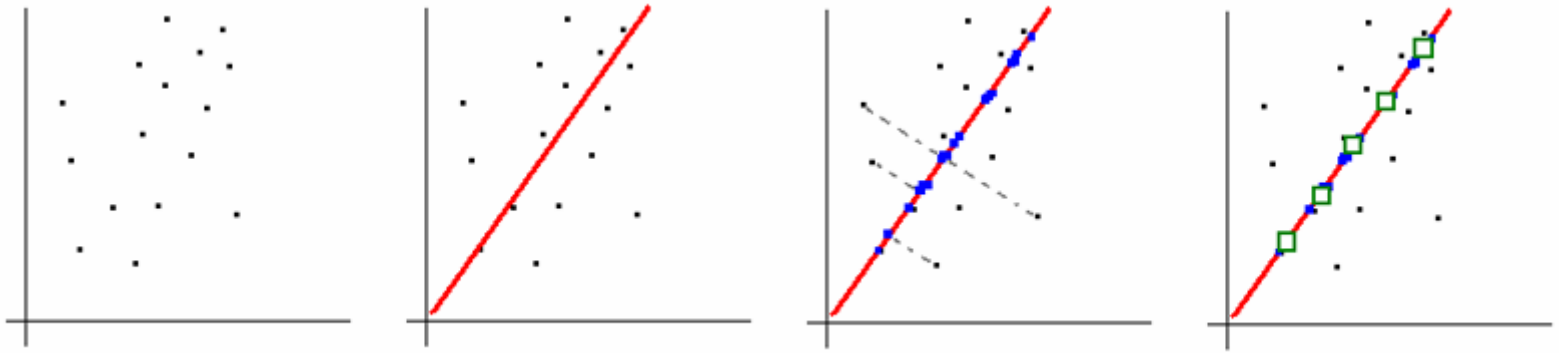
- 4x4 한블럭
  - 8 바이트
  - 2 바이트 + 2바이트 + 4바이트
  - 16비트 + 16비트 + 32비트
  - 16비트 + 16비트 + (2비트 + 2비트 + … + 2비트)
  - RGB1(565:16) + RGB2(565:16) + T00(2) + T01(2) + T02(2) + T03(2) + T10(2) + T12(2) + .. + T33(2)

- PCA 알고리즘 (주성분 분석)
  - 입력된 셈플로 고유 벡터(Eigen Vector)를 구함
  - 주성분 = 고유벡터 (분포 성향)

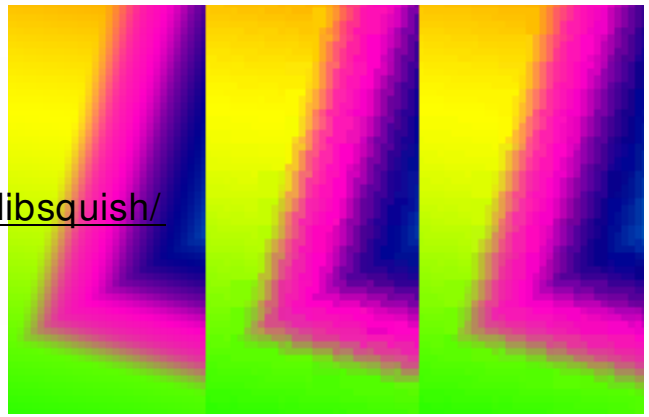


DXT1 기본 알고리즘(2)

- 16개의 셈플로 두 개의 양단 칼라값 구함
  - PCA 활용 축
  - 각 칼라는 3차원 벡터 (RGB)
- 입력된 두 값을 토대로 선형 보간
  - 가까운 값으로 할당



- 선택에 따라 퀄리티차이 발생
  - RangeFit
  - ClusterFit
  - 등등
  - 참고 : http://code.google.com/p/libsquish/



DXT1 기본 알고리즘(3)

- Pseudo Code

```
unsigned short RGB0 = *(unsigned short*)&ptr[0];
unsigned short RGB1 = *(unsigned short*)&ptr[2];
unsigned long table = *(unsigned long*)&ptr[4];

float weight[4] = { 0/3.0f, 3/3.0f, 1/3.0f, 2/3.0f };

for(y=0, i=0; y<4; y++)
{
    for(x=0; x<4; x++, i+=2)
    {
        int idx = (table>>i) & 3;
        c[y][x] = lerp(RGB0, RGB1, weight[idx]);
    }
}
```

양단 값이 많은 특성을 활용하기 위해 가중치값의 순서를 섞어 놓은 듯???
(0, 1이 RGB0,RGB1 이 되도록 구성)

- 필요한 경우 예외적으로 검정색을 별도로 디코딩
  - 이 경우 수치적으로 RGB0 >= RGB1
  - DXT1에 알파를 사용하는 경우 BLACK이 아니라 알파 0 가 컬러

```
float weight[3] = { 0/2.0f, 2/2.0f, 1/2.0f, };

for(y=0, i=0; y<4; y++)
{
    for(x=0; x<4; x++, i+2)
    {
        int idx = (table>>i) & 3;
        c[y][x] = idx == 3 ? BLACK : lerp(RGB0, RGB1, weight[idx]);
    }
}
```

DXT3/DX5 알고리즘(1)

- DXT 3
  - RGB 압축은 DXT1과 동일
  - 16 픽셀의 알파값을 4비트씩 선형적으로 (8바이트)
- DXT5
  - 알파채널은 따로 압축 (RGB와 유사)
  - 8바이트
    - 1바이트 + 1바이트 + 6바이트
    - 8Bit + 8Bit + 3Bit + 3bit ... + 3Bit
    - ALPHA0(8) + ALPHA1(8) + T00(3) + T01(3) + .. + T32(3) + T33(3)
    - RGB 디코딩과 비슷 (8레벨로 나눔)
  - Pseudo Code

```
float weight[8] = { 0/7.0f, 7/7.0f, 1/7.0f, 2/7.0f, 3/7.0f, 4/7.0f, 5/7.0f, 6/7.0f };

for(y=0, i=0; y<4; y++)
{
    for(x=0; x<4; x++, i+2)
    {
        int idx = ...;
        a[y][x] = lerp(ALPHA0, ALPHA1, weight[idx]);
    }
}
```

DXT3/DX5 알고리즘(2)

- RGB에서 BLACK 칼라처럼 0, 255 예외 방식 존재
  - 마찬가지로 ALPHA1이 ALPHA0 보다 같거나 큰 경우 8개 값을 6+2로 분리해서 예외 코드
- Pseudo Code

```
float weight[6] = { 0/5.0f, 5/5.0f, 1/5.0f, 2/5.0f, 3/5.0f, 4/5.0f };

for(y=0, i=0; y<4; y++)
{
    for(x=0; x<4; x++, i+2)
    {
        int idx = ...;
        if (idx >= 6)
            a[y][x] = idx == 6 ? 0 : 255;
        else
            a[y][x] = lerp(ALPHA0, ALPHA1, weight[idx]);
    }
}
```

기타

- 인코딩시 키 값을 어떤 값으로 잡느냐에 따라 에러(퀄리티) 달라짐
  - 압축 알고리즘을 구현(혹은 선택)한다면 신중할 필요가 있음
- 손실 특성 대비
  - 한 블럭이 선형적인 특성이 있어서 노멀 텍스처로 사용할 경우 파워 손실
    - 노멀라이즈 필요
    - 번지 소프트 HALO 케이스
      - 노멀맵 DXT5 압축 사용, RGB중 한 채널 사용하지 않고, 알파에 그 채널의 값을 채움
      - 알파는 비교적 손실이 작음
- 절차 텍스처 생성시 DXT 포맷으로 직접 생성
  - 랜덤 이미지 : 16개 값을 만들어 인코딩할 필요없이 DXT 상태로 랜덤하게 생성
- 참고문서 (Official)
  - http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_s3tc.txt