3D Game Programming 42
-Pixel Shader

afewhee@gmail.com

- Pixel Shader 개요
- Simple Pixel Shader
- Pixel Shader 응용
  - 단색(Monotone) 효과
  - Blur(흐림) 효과

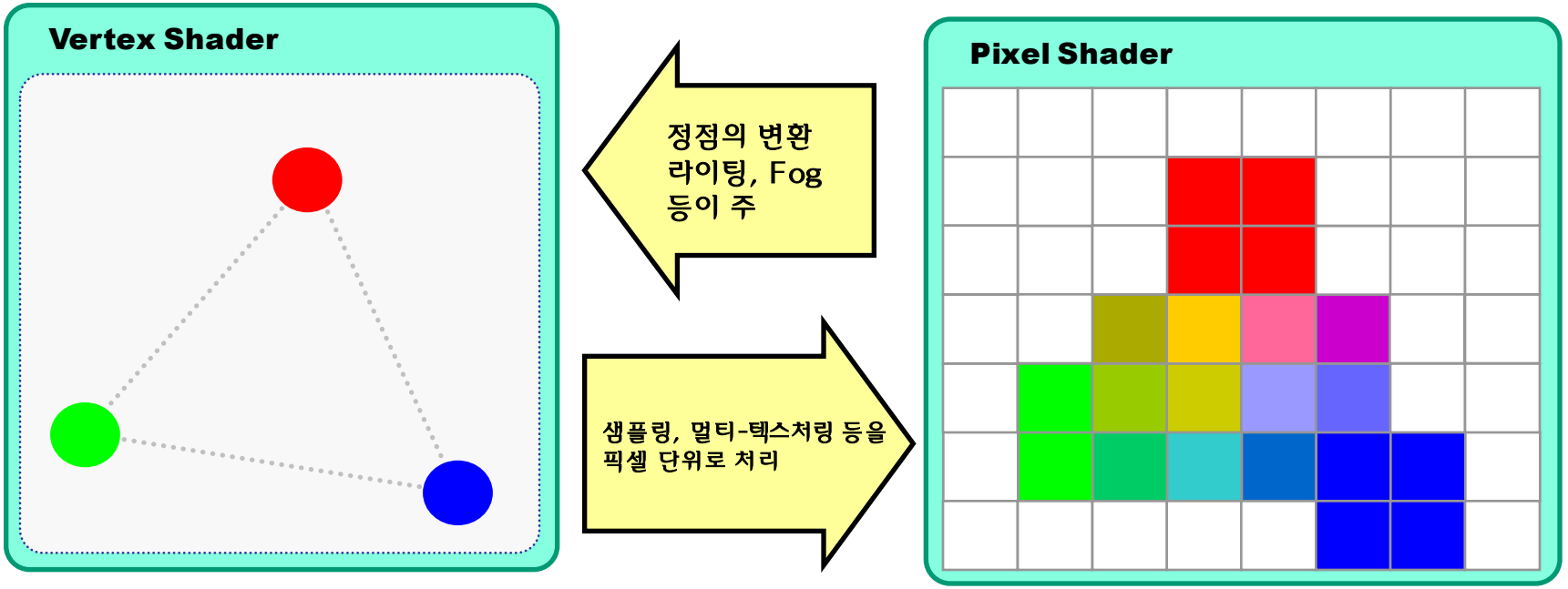
- Vertex Shader Version Maximum number of instruction slots
  - ps_1_1 128
  - ps_2_0 256
  - ps_2_x 256
  - ps_3_0 512 minimum, 최대 하드웨어 의존

- Pixel Shader Version Maximum number of instruction slots
  - ps_1_1 4 texture + 8 arithmetic
  - ps_1_2 4 texture + 8 arithmetic
  - ps_1_3 4 texture + 8 arithmetic
  - ps_1_4 6 texture + 8 arithmetic per phase
  - ps_2_0 32 texture + 64 arithmetic
  - ps_2_x 96 minimum, 최대 하드웨어 의존
  - ps_3_0 512 minimum, 최대 하드웨어 의존

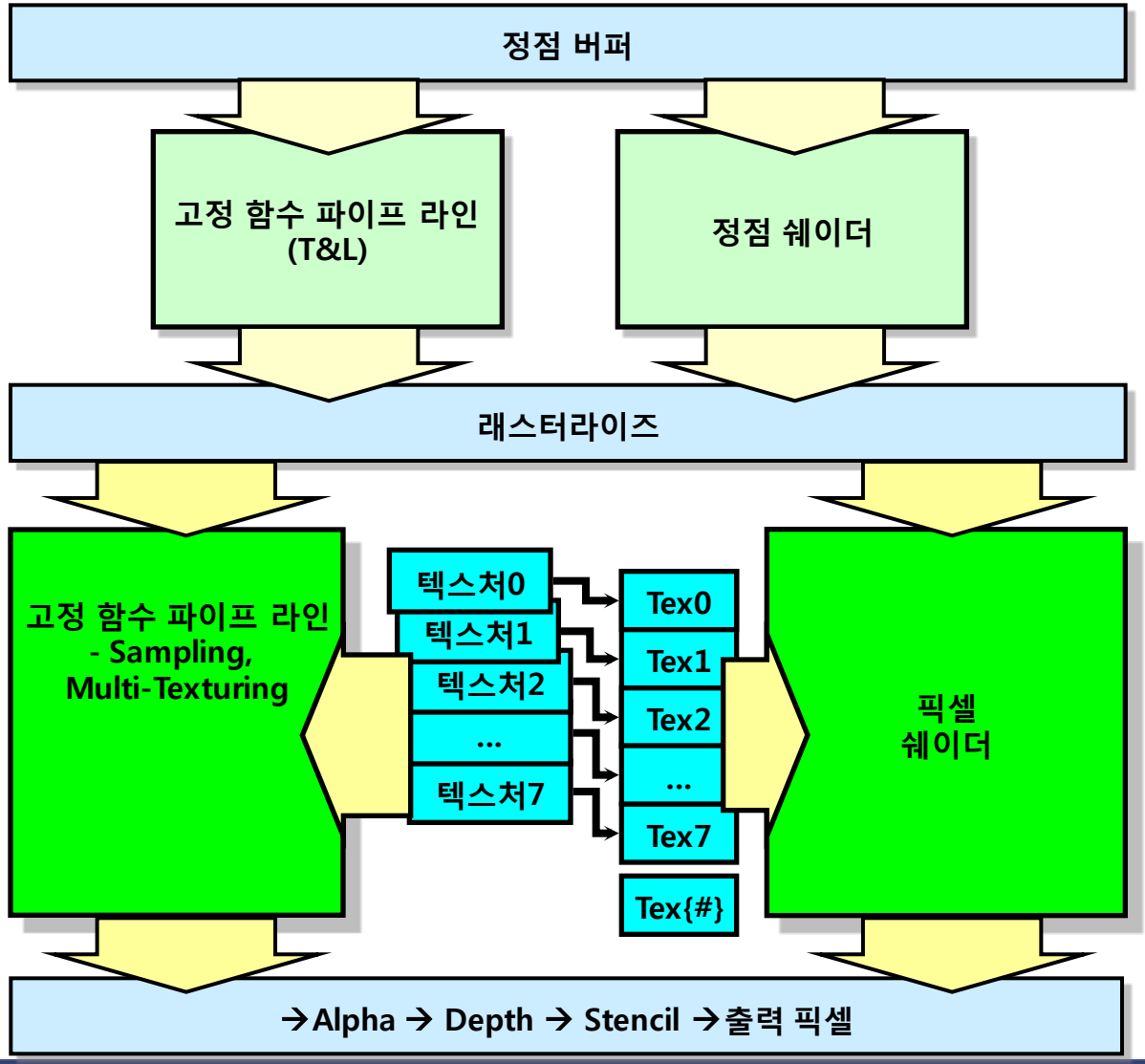
픽셀 쉐이더와 정점 쉐이더 비교

- 정점 쉐이더: 정점의 변환, 라이팅, 포그 등을 정점 단위에서 처리
- 픽셀 쉐이더: 정점의 래스터라이즈 변환 후 색상, 텍스처 샘플링, 멀티 텍스처링 등 픽셀 단위에 대한 처리

- 정점 처리는 비교적 단순해서 대부분의 그래픽 카드가 정점 쉐이더 지원
- 픽셀 처리는 메모리와 처리에 대한 비용이 커서 픽셀 쉐이더 지원은 그래픽 카드마다 차이가 심함
- 픽셀 쉐이더는 명령어 수도 정점 쉐이더보다 훨씬 적음



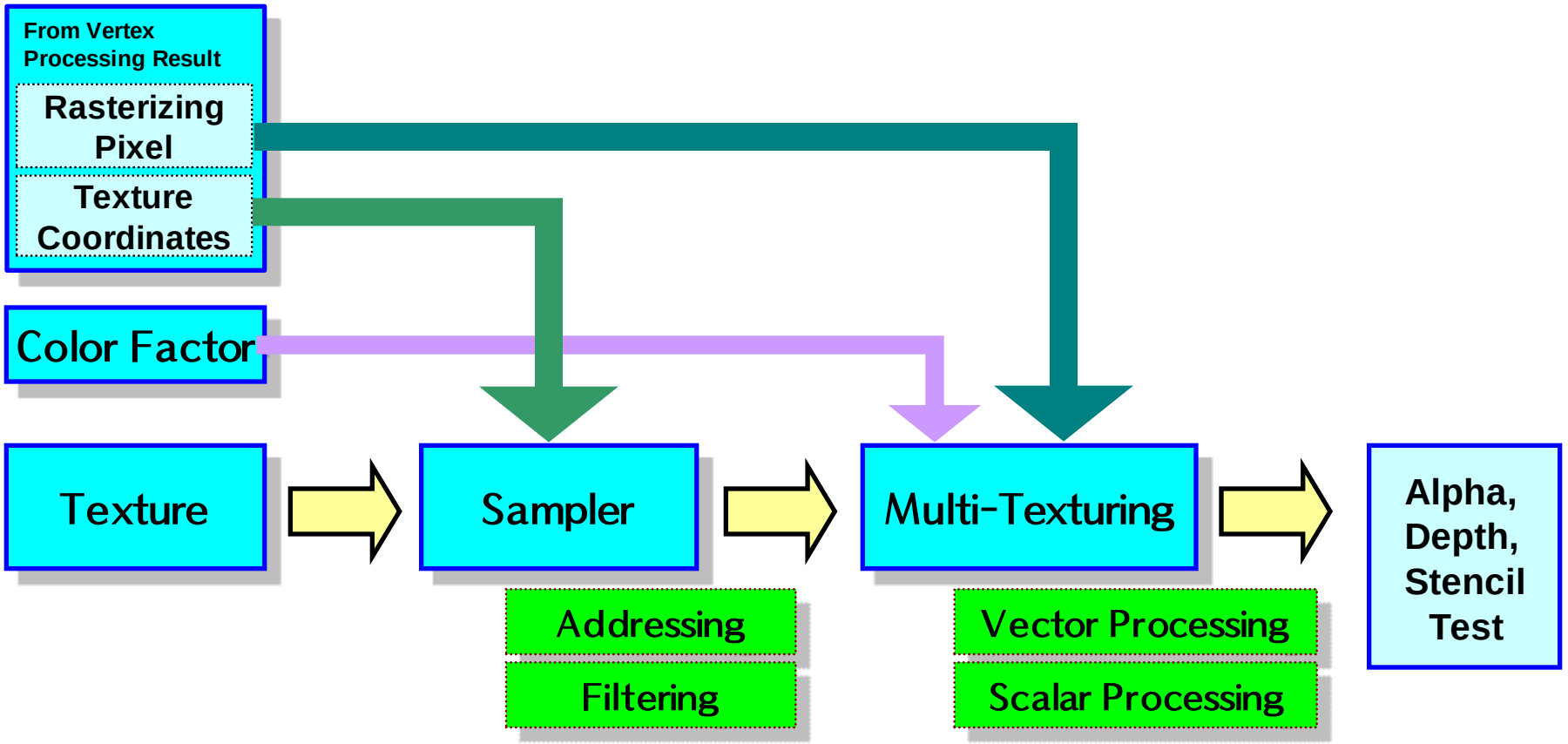
- Pixel Shader 위치와 의미



❖쉐이더 명령어
저 수준 또는 고 수준 명령어로 Sampling, Multi-Texturing을 처리

❖처리(Processing)의 독립성
래스터라이즈로 만든 픽셀이 정점 처리(Vertex Processing)에서 고정 함수 사용으로 처리되었거나 쉐이더 사용으로 처리하던 간에 이 둘을 구분하지 않음

- 고정 함수 파이프 라인 Pixel Processing



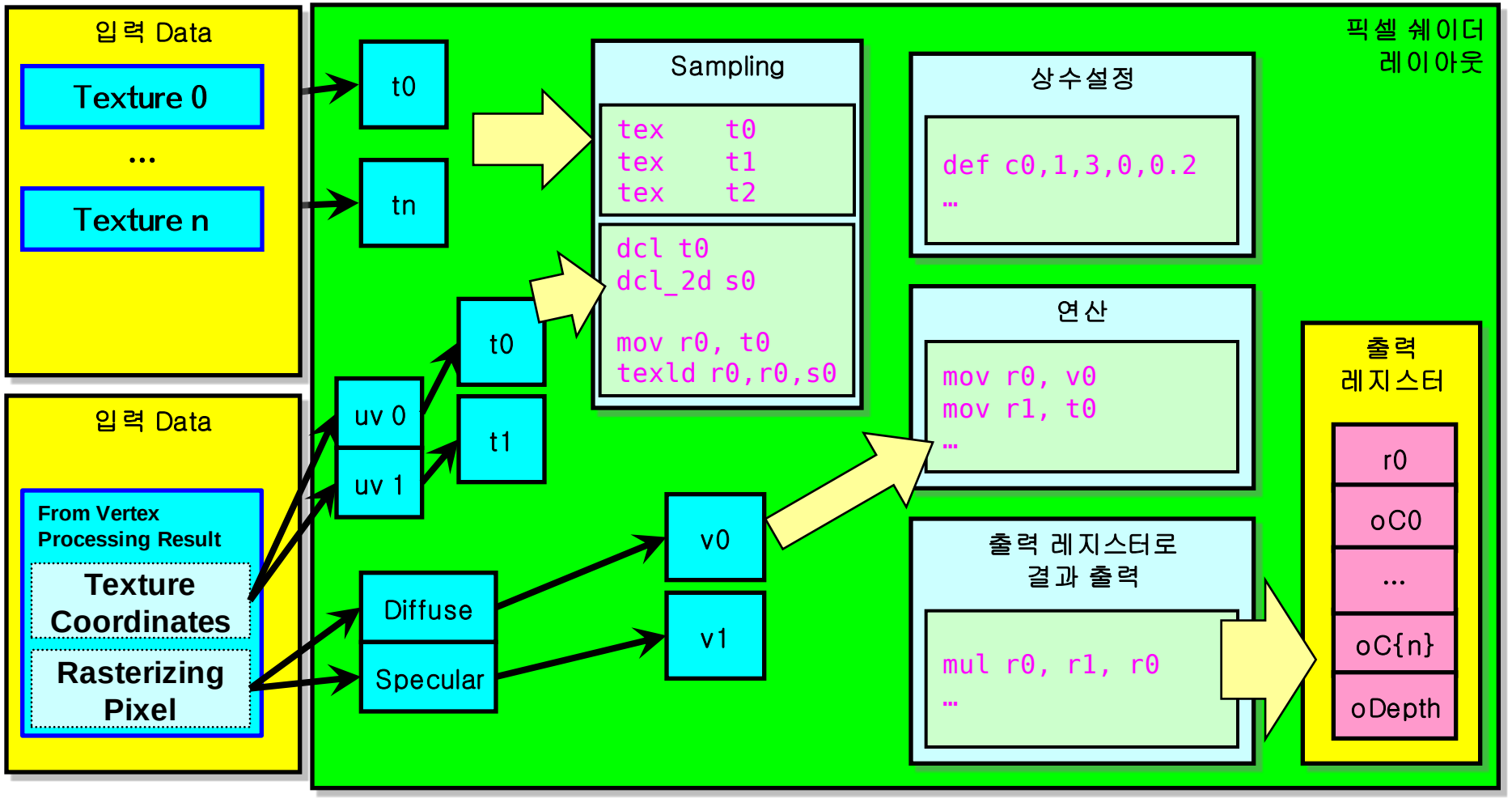
- Pixel Shader
  - 고정 함수 파이프 라인의 Sampling과 Multi-Texturing의 Vector, Scalar Processing을 처리

- Assembly 형식의 저 수준 픽셀 쉐이더

```
; Simple Pixel Shader

ps_1_1                  // 픽셀 쉐이더 버전

tex     t0              // 0번째 텍스처를 Sampling 해서 t0에 저장
mov     r0, t0          // 출력 레지스터(r0)에 복사
```

- 픽셀 쉐이더 명령어는 버전마다 차이가 심함

● 픽셀 쉐이더 레이아웃 (Pixel Shader Layout)



- 저 수준 픽셀 쉐이더 프로그램 방법
  - 쉐이더 명령어 버전 선언
    - ps_1_1 ➔ 픽셀 쉐이더 명령어 버전은 1.1
  - 상수 레지스터 설정
    - def c0, 1.0, 3.0, 0.0, 2
    - def c12 , 1.0, 0.0, 0.0, 1
  - Sampling (명령어)
    - tex t0, or texld t0 ➔ 0번 Stage 텍스처 샘플링 t0 레지스터에 저장
  - 픽셀 처리에 대한 연산
    - 산술, 논리 연산
    - 연산의 과정에서 상수 레지스터(c#)은 임시 변수로 사용 불가
    - 임시 변수는 임시 변수 레지스터(r#) 이용
  - 출력 레지스터에 결과 저장
    - mov r0, r1 // ps_1_x: r0는 출력 레지스터
    - mov oC{#}, r0 // ps_2_0: #:0~3 (색상 저장)
    - mov oDepth, r2 ... // ps_2_0: Depth 저장

픽셀 쉐이더 이용 방법

쉐이더 코드와 객체 생성 단계

- 어셈블리 형식의 저 수준 쉐이더 코드를 Text로 작성
- 저 수준 쉐이더 코드 컴파일: D3DXAssembleShader()
- 픽셀 쉐이더 객체 생성: pDevice->CreatePixelShader()

렌더링 단계

- 렌더링 머신(장치: 디바이스)에게 픽셀 쉐이더 사용을 통보
- 쉐이더 상수 설정
- 렌더링
- 장치의 픽셀 쉐이더 사용 해제

- 저 수준 픽셀 쉐이더 컴파일(Assemble)
  - 저급 언어인 어셈블리어 컴파일 담당은 어셈블러
  - 저 수준 쉐이더 컴파일은 DirectX의 D3DXAssembleShader() 함수

Ex)

```
DWORD dwFlags = 0;
#if defined( _DEBUG ) || defined( DEBUG )
    dwFlags |= D3DXSHADER_DEBUG;
#endif

LPD3DXBUFFER    pShd  = NULL;
LPD3DXBUFFER    pErr  = NULL;
INT     iLen    = strlen(sShader);

hr = D3DXAssembleShader(sShader, iLen, NULL, NULL, dwFlags, &pShd, &pErr);

// Error
if ( FAILED(hr) )
{
    if(pErr)
    {
        MessageBox( GetActiveWindow(), (char*)pErr->GetBufferPointer(), "Err", 0);
        pErr->Release();
    }
    return -1;
}
```

  - 문법 (Syntax) 에러: D3DXAssembleShader() 함수의 에러 반환 객체의 버퍼에 대한 내용을 char형 포인터로 변환해서 사용
    - (char*)pErr->GetBufferPointer()

- 픽셀 쉐이더 객체 (IDirect3DPixelShader9)
  - 컴파일 한 쉐이더 명령어를 사용하기 위한 객체
  - pDevice->CreatePixelShader(...);

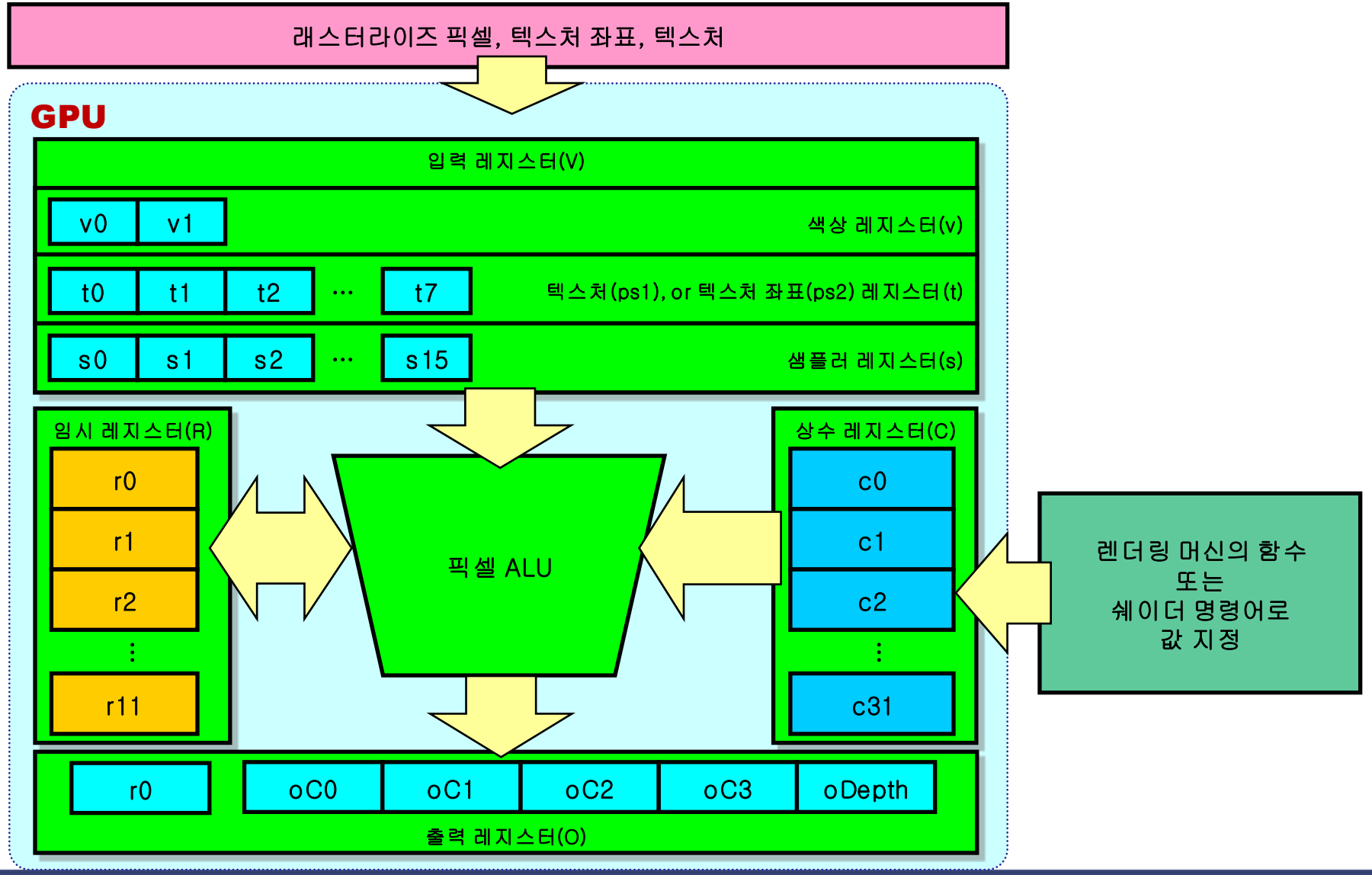
Ex)

```
// Compile
D3DXAssembleShader(…, &pShd, &pErr);

// 픽셀 쉐이더 객체 생성과 정점 선언 객체 생성
hr = m_pDev->CreatePixelShader(
                    (DWORD*)pShd->GetBufferPointer()
                    , &m_pVs);
if ( FAILED(hr) )
   return -1;
```


- 렌더링 단계
  - 렌더링 머신(장치: 디바이스)에게 픽셀 쉐이더 사용을 통보
    - pDevice->SetPixelShader()
  - 텍스처 설정: 고정 함수 파이프 라인과 동일
    - pDevice->SetTexture(0, pTexture);
  - 쉐이더 상수 설정
    - pDevice->SetPixelShaderConstantF(상수 레지스터 이름(시작), 변수 주소, 크기) ➔ GPU의 상수 레지스터 설정
  - 렌더링
    - pDevice->DrawPrimitive()
  - 장치의 픽셀 쉐이더 사용 해제
    - pDevice->SetPixelShader(NULL)

- 상수 레지스터 설정 시 주의 사항
  - 형식은 정점 쉐이더와 동일
  - 상수 레지스터 범위 인지 : c0 ~ c31 (ps.2.0)
  - c를 생략하고 숫자만 전달:
    - 4에 변수의 값을 설정할 경우 pDevice->SetPixelShaderConstantF(4, ..., ...)
  - 변수 주소: 대부분 float* 형으로 캐스팅
  - 크기
    - CPU는 자료의 처리가 INT형이 기본 이듯, 쉐이더는 float4(float 4개의 연속 메모리)가 기본
    - r, g, b, a 인 float형 4개를 전달할 때 크기는 1
    - x, y float형 2개를 전달할 때도 크기는 1
  - 상수 레지스터 사용 범위
    - 크기를 1보다 큰 값을 전달하면 해당 상수 레지스터에서부터 크기 만큼 상수 레지스터 점유

- 픽셀 쉐이더 가상 머신



- 레지스터 (CRTV: constant, temporary, texture, color)
  - 4 종류: 입력, 임시, 상수, 출력 레지스터
  - "레지스터+index" 가 이름
  - ex) v1 ➔"v1" 자체가 레지스터 이름.

- 입력 레지스터 (Input Register)
  - Color Register : v#. v0-Diffuse, v1-Specular
  - Texture Register: t#
  - Sampler Register: s#

- 임시 레지스터 (Temporary Register)
  - r로 시작. r0~r(1, 2, 6, 12, 32)
  - 연산의 결과를 임시로 저장하는 데 사용
  - 쉐이더 버전 1_4 이하에서는 출력 레지스터와 겸용으로 사용

- 상수 레지스터 (Constant Register)
  - c로 시작. c0~c31, ~c255(ps_2_0). 이외 i, b,
  - def: 상수 설정에 사용
  - 외부에서 사용자가 값을 설정 ➔pDev->SetPixelConstantF()

- 출력 레지스터
  - o(영문자 small 'o')로 시작
  - ps_1_x 에서는 임시 레지스터 r0가 출력 레지스터
  - oC0: Render Target. oC1~oC7 : Multi-Element Texture
  - oDepth: Depth

1.3 Pixel Shader Coding 기초 - 주요 명령어

- 쉐이더 버전:
  ps_1_1, ps_1_2, …

- 상수 레지스터(Constant Register) 설정
  - 종류: float, int, bool
  - 정의 : def 상수 레지스터 이름, value 0, value 1, value 2, value 3

  Ex)

```
def c0, -0.5, 0.5, 0, 0.6
def c1, 0,0,0,0 def c2, 1,1,1,1
```

  - 상수 레지스터는 외부에서 설정 가능
    - pDevice->SetPixelConstant{F|I|B}("c를 제외한 인덱스", 변수 주소, 크기);

  Ex)

```
D3DXCOLOR p(r,g,b,a);
pDevice->SetPixelConstantF( 2, (float*)&p, 1); // c2에 float형 상수 설정
```

- Sampling

  Ex)

```
tex t0
```

- 연산 명령어 문법 - 정점 쉐이더와 동일
  - Assembly의 Mnemonic 형태
  - operator dest, source1, source2, source3
  - 모든 연산은 float4를 r, g, b, a로 나누어서 연산
  - dest operand는 임시(r#), 출력(o…)레지스터만 가능
  - dest는 상수(c#), 입력(v#)는 불가

- 주석: ';' 또는 '//'

- 산술 명령어: ps_2_0 이상 지원이 되어야 편리

1.3 Pixel Shader Coding 기초 - 주요 명령어

- tex: Sampling - 텍스처에서 색상 추출
  - tex t{n} - ps_1_1~3
    n-stage 색상을 텍스처에서 Sampling하고 t0에 저장
  - texld r{m}, t{n} : ps_1_4
    n-텍스처좌표에서 m-stage 텍스처 Sampling

- mov: copy
  - mov r2, r0;
    r2.x = r0.x;
    r2.y = r0.y;
    r2.z = r0.z;
    r2.w = r0.w;
- add: 덧셈
  - add r2, r0, r1
    r2.x = r0.x + r1.x;
    r2.y = r0.y + r1.y;
    r2.z = r0.z + r1.z;
    r2.w = r0.w + r1.w;

- sub: 뺄셈
  - sub r2, r0, r1
    r2.x = r0.x - r1.x
    r2.y = r0.y - r1.y
    r2.z = r0.z - r1.z
    r2.w = r0.w - r1.w

- mul: 곱셈
  - mul r2, r0, r1
    r2.x = r0.x * r1.x;
    r2.y = r0.y * r1.y;
    r2.z = r0.z * r1.z;
    r2.w = r0.w * r1.w;

- pow: 멱수-승수 구하기(x^y). ps_2_0 이상
  - pow r2, r0, r1;
    r2 = pow(abs(r0), r1);
    r2 = exp(r1 * log(r0))

- exp; 2의 승수 2^x. ps_2_0

```
def  c10, -1, 0.5, 0., 0.
mov  r0, c10
exp  r0.x, r0.x
exp  r0.y, r0.y
exp  r0.z, r0.z
exp  r0.w, r0.w
```

- log: 2의 지수 . ps_2_0

log dst, src; src=0 -> dst = -FLT_MAX

```
def  c10, 4, 3., 2., 2.
mov  r0, c10
log  r0.x, r0.x
log  r0.y, r0.y
log  r0.z, r0.z
log  r0.w, r0.w
```


- frc : Fragment. ps_2_0
  - frc dst, src;
    dst.x = src.x - floorf(src.x);
    dst.y = src.y - floorf(src.y);
    dst.z = src.z - floorf(src.z);
    dst.w = src.w - floorf(src.w);

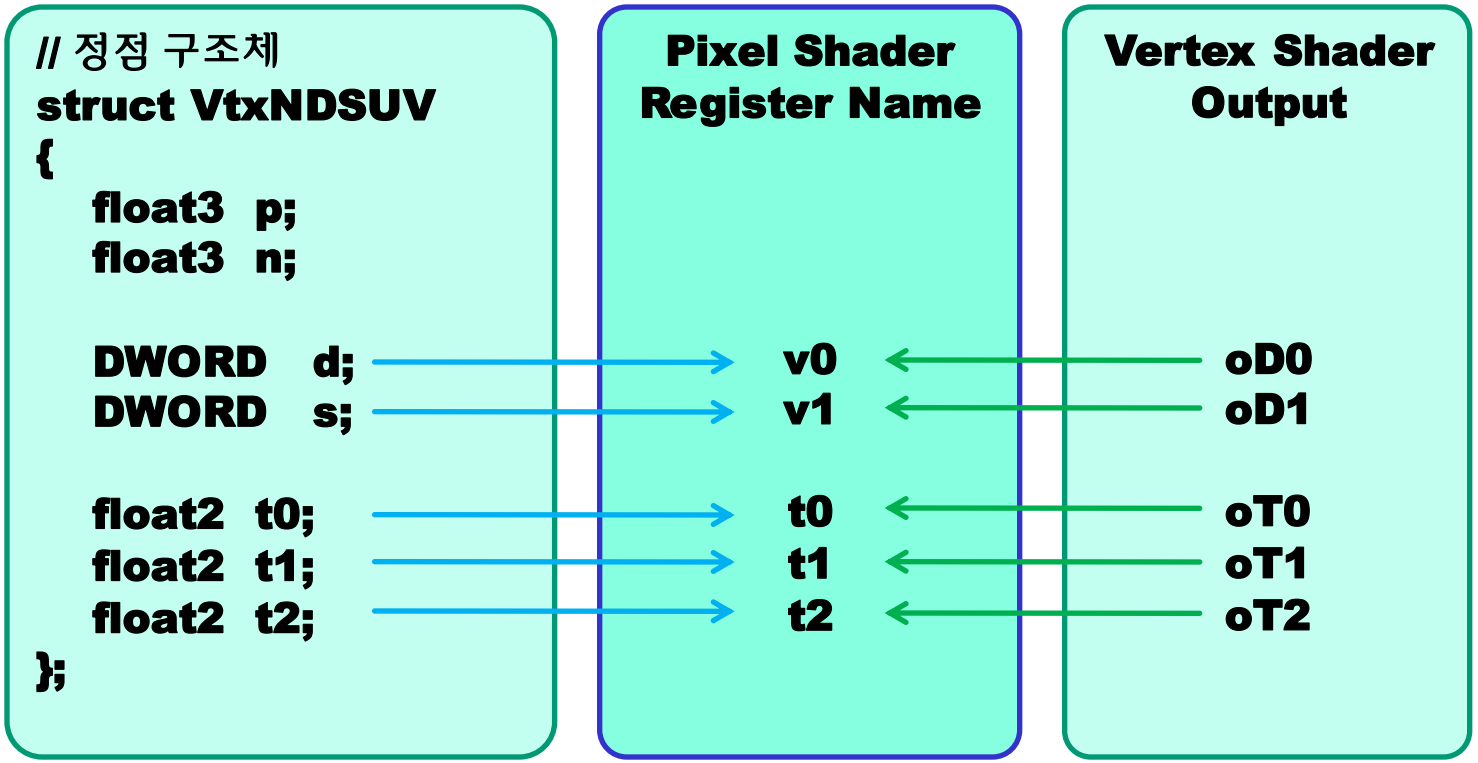
- rcp : 역수. ps_2_0
  - rcp dst, src;dst, src의 swizzle component 명시
  - src =0 -> dst = FLT_MAX;

```
def  c10, 4, 3., 2., 1.
mov  r0, c10
rcp  r0.x, r0.x
rcp  r0.y, r0.y
rcp  r0.z, r0.z
rcp  r0.w, r0.w
```

- rsq : 제곱근의 역수. ps_2_0
  - dst, src의 swizzle components 명시
  - src =0 이면 dst = FLT_MAX;

```
dp3 r3, r1, r2
rsq r3, r3.w
r3.x = r3.y = r3.z = r3.w = 1.f/sqrtf(r3.w);
```


- 고정 파이프 라인, 정점 쉐이더, 픽셀 쉐이더 입력 레지스터 대응 관계

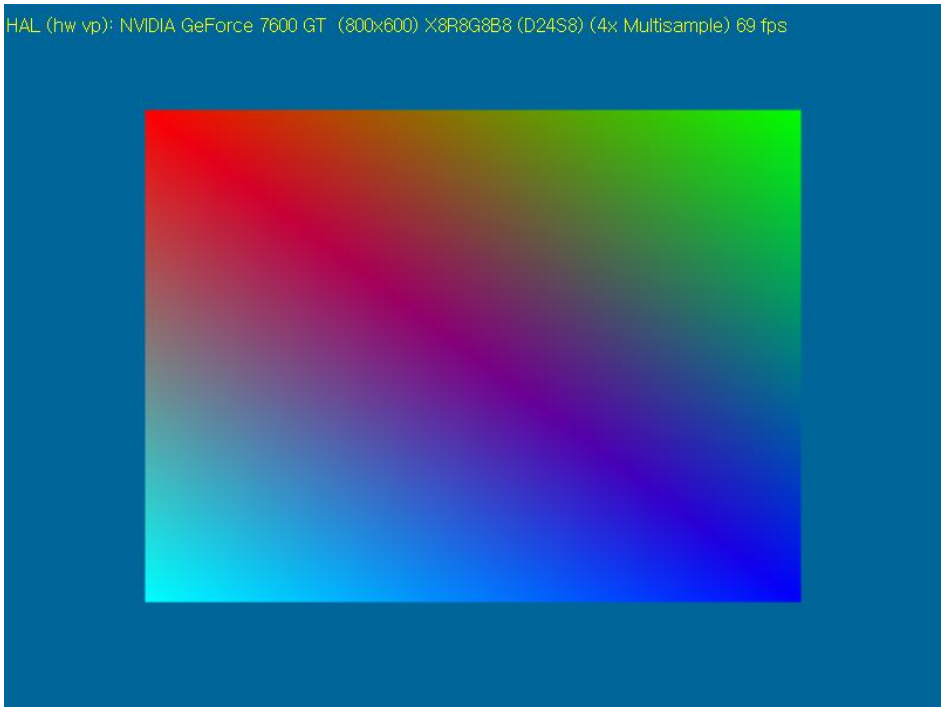


※ **t#** 레지스터는 **1.1~1.3**까지는 **Sampler** 객체에 의한 샘플링 데이터 저장 레지스터이고, **1.4** 버전은 텍스처 좌표에 대한 레지스터임

정점의 색상 출력

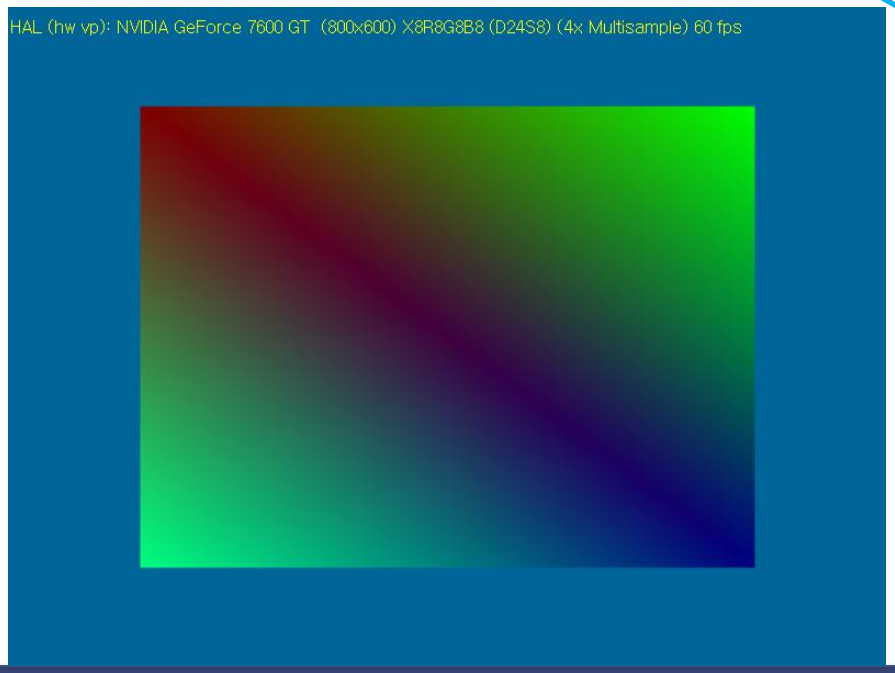
```
ps_1_1              ; 픽셀 쉐이더 버전 1.1
mov r0, v0          ; 입력된 정점의 색상을 출력
```

- 파이프라인에서 처리된 색상은 v0에 저장
- v0를 r0에 복사
- r0는 임시 레지스터인 동시에 출력 레지스터



- 정점의 색상과 상수 값 혼합

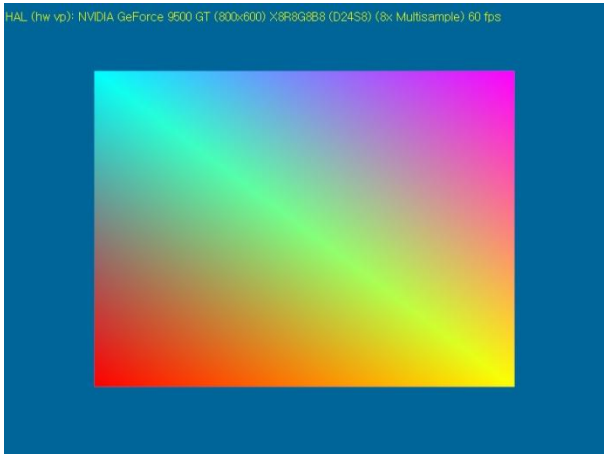
```
ps_1_1                      // 픽셀 쉐이더 버전 선언

def c0, 1, 1, 0.5f, 1       // 상수 레지스터 c0.rgba =(1,1,0.5f,1)

mul r0, c0, v0              // 상수 값과 색상 혼합
mul r0, r0, c1              // 외부에서 정한 상수 값과 색상 혼합
```




```
//상수 레지스터 값은 쉐이더 코드의 값이 우선
D3DXCOLOR color0(1,1,1,1);
m_pDev->SetPixelShaderConstantF(0,(float*)&color0,1);

D3DXCOLOR color1(0.5f,1,1,1);
m_pDev->SetPixelShaderConstantF(1,(float*)&color1,1);
```

- 반전, 흑백

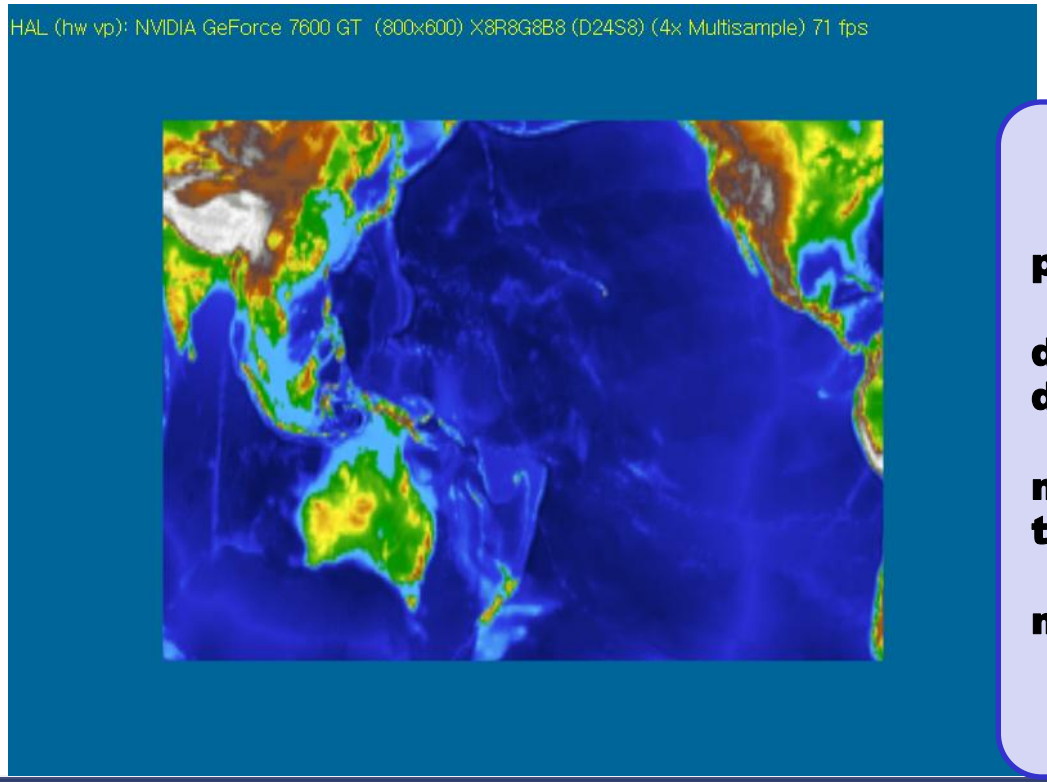
```
// reverse
ps_1_1                          // 픽셀 쉐이더 버전 선언

def c0, 1, 1, 1, 0              // 상수 레지스터 c0.rgba =(1,1,1,0)
sub r0, c0, v0                  // 반전 색상 = 1. - 색상
```




```
// monotone
ps_1_1                          // 픽셀 쉐이더 버전 선언

// 상수 레지스터 c1.rgba =(0.299, 0.587, 0.114,0)
def c1, 0.299, 0.587, 0.114, 0

// 흑백 r=g=b = r*299 + g*.587 + b * .114 = dot3(c1,v0)
dp3 r0, c1, v0
```

- 텍스처 Sampling

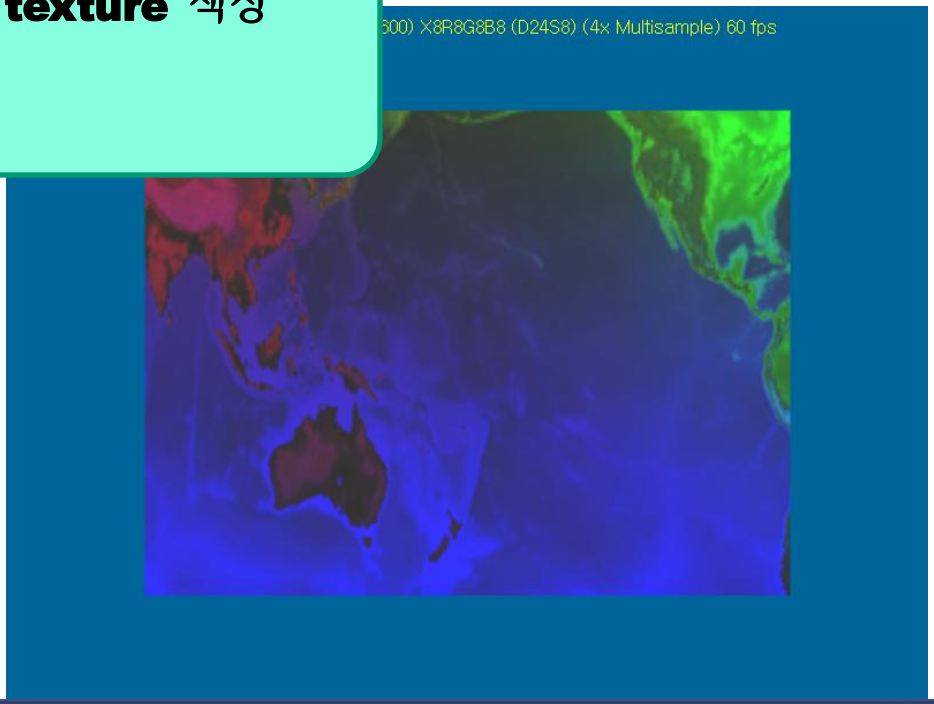
```
ps_1_1                  // 픽셀 쉐이더 버전 선언

tex t0                  // 0-stage 텍스처를 t0에 Sampling
mul r0, t0              // 출력 레지스터에 복사
```




```
ps_2_0              // 픽셀 쉐이더 버전 선언

dcl t0              // t0 텍스처 좌표 선언
dcl_2d s0           // 0-stage 샘플러 객체 선언

mov r0, t0          // 텍스처 좌표를 임시 레지스터에 복사
texld r0, r0, s0    // Sampling(색상 추출)

mov oC0, r0         // 출력 레지스터 oC0에 복사
```

- 정점 Diffuse + 텍스처 Sampling + 외부 상수

```
ps_1_1                          // 픽셀 쉐이더 버전 선언

def c0, 0.2, 0.2, 0.2, 0

tex t0                          // 0번 째 stage 텍스처를 t0에 Sampling

mul r0, c1, v0                  // 외부 상수 값과 Diffuse 색상 곱셈
mul r1, r0, t0                  // r1 = Diffuse * c1 * texture 색상
add r0, r1, c0                  // 출력= r1 + c0
```



- Multi-Texturing

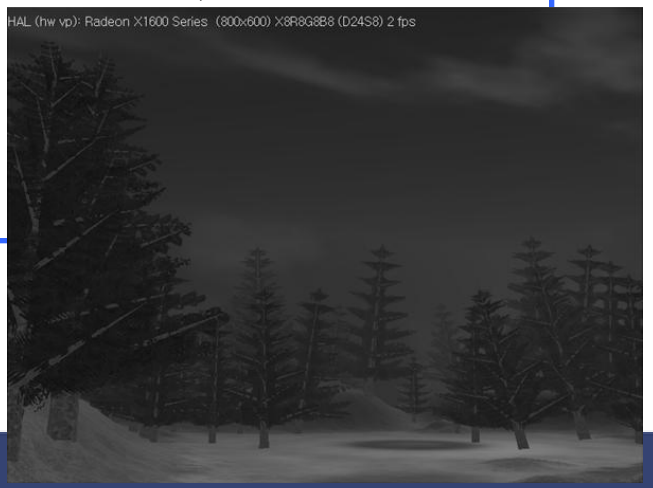
```
ps_1_4                  // 픽셀 쉐이더 버전 선언

texld r0, t0            // 0 번째 stage 텍스처 샘플링
texld r1, t0            // 1 번째 stage 텍스처 샘플링

add_x4 r2, r0, r1 // 두 텍스처 픽셀 값을 더하고 4배
mul r0, r2, v0          // 출력
```

```
ps_1_1

tex t0                  // 0 번째 stage 텍스처 샘플링
tex t1                  // 1 번째 stage 텍스처 샘플링

add_x4 r0, t0, t1 // 두 텍스처 픽셀 값을 더하고 4배
mul r0, r0, v0          // 출력
```


```
// ps1.1~4 까지는 텍스처 좌표 지정이 없으므로 이를 고정 함수 파이프 라인에서 지정
m_pDev->SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 0);
```

- 후면 버퍼의 색상을 이용한 단색(Monotone) 만들기
  - 후면버퍼의 색상을 직접 처리하면 렌더링의 속도 저하

```
IDirect3DSurface9* pSrc;
hr = m_pd3dDevice->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &pSrc);
D3DSURFACE_DESC dsc;
pSrc->GetDesc(&dsc);
D3DLOCKED_RECT  rc;
hr = pSrc->LockRect(&rc, NULL, 0);
DWORD* pColor = (DWORD*)rc.pBits;
for(int i=0; i<dsc.Width* dsc.Height; ++i)
{
    D3DXCOLOR color = pColor[i];
    FLOAT d = color.r * 0.299f + color.g * 0.587f + color.b * 0.114f;
    pColor[i] = D3DXCOLOR(d,d,d,1);
}
pSrc->UnlockRect();
pSrc->Release();
```



- 픽셀 쉐이더를 이용한 단색(Monotone) 만들기
  - 장면을 텍스처에 저장
  - Monotone을 픽셀 쉐이더를 작성해서 장면이 저장된 텍스처에 적용

장면을 텍스처에 그린다.

픽셀 쉐이더를 이용해 단색으로 만든다

화면 크기에 맞게 조정한다.

- 단색(Monotone) 만들기 – One Object Multi-texture

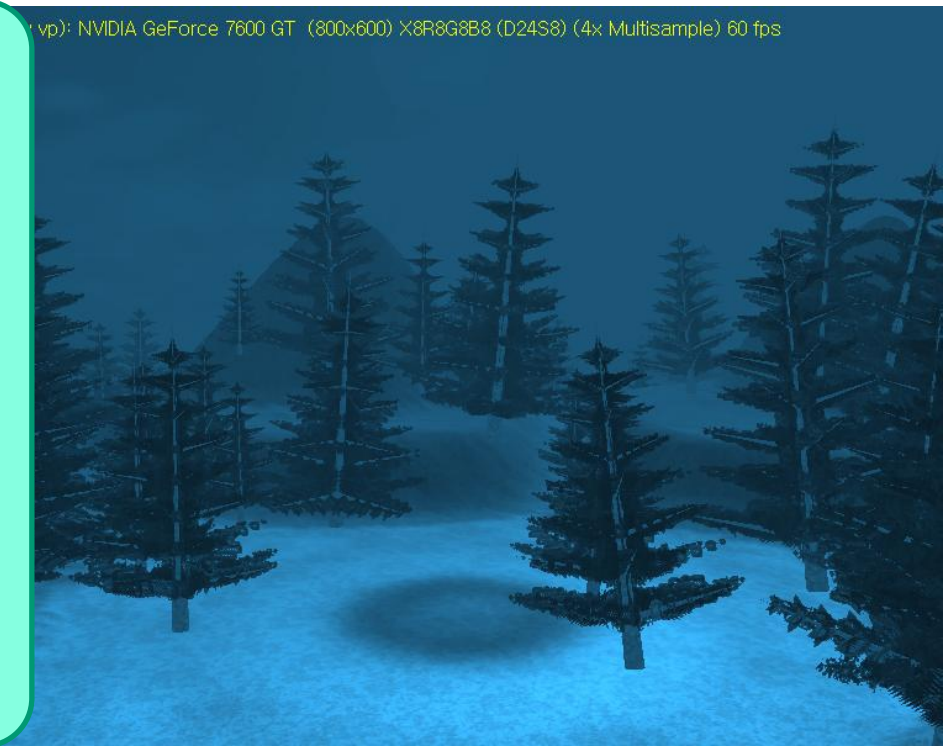
단색 = r*0.299 + g*0.5987 + b*0.114

```
ps_1_4                  // 픽셀 쉐이더 버전 선언

def c0, 0.299, 0.5987, 0.114, 0     // 색상 비중 값

texld r0, t0            // 0 번째 stage 텍스처 샘플링
texld r1, t0            // 1 번째 stage 텍스처 샘플링

add r2, r0, r1          // 두 텍스처 픽셀 값을 더한다.
mul r1, r2, v0          // 정점 프로세싱이 끝난 색상을 곱한다.

dp3_x2 r2, r1, c0 // 내적한 다음 2배
mov r0, r2              // rgb 에 값을 설정
mul r0, r0, c1          // 외부의 색상과 곱셈
```

- 단색(Monotone) 만들기 – Rendering Target Screen

단색 = r*0.299 + g*0.5987 + b*0.114

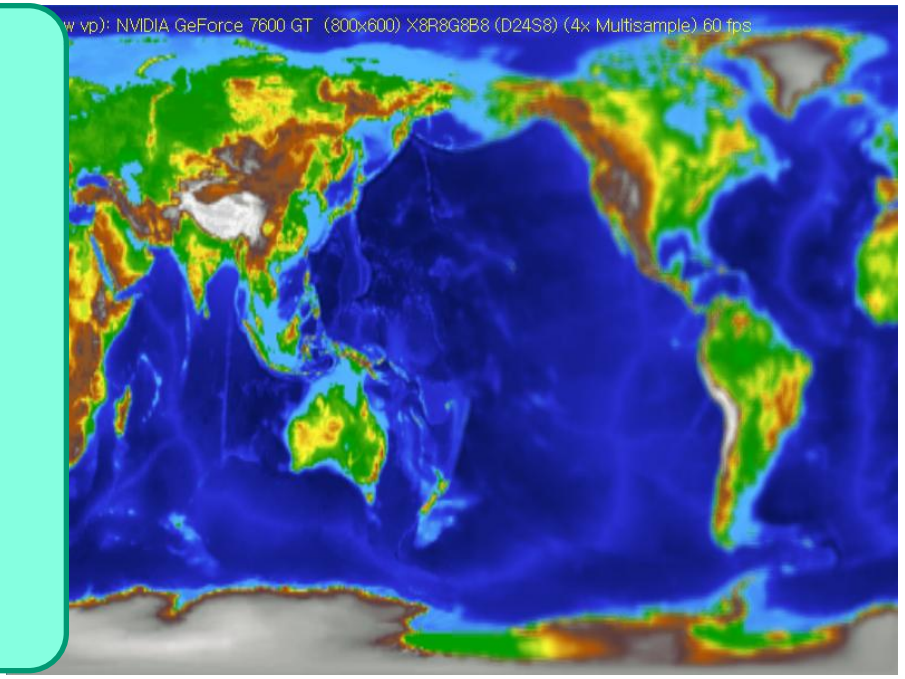
```
// 텍스처의 색상만 이용해서 쉐이더 코드를 간단하게 함

ps_1_1                  // 픽셀 쉐이더 버전 선언

def c0, 0.8, 0.9, 0.4, 0    // 색상 비중 값

tex t0                  // 0-stage 텍스처 샘플링

dp3 r0.rgba, t0, c1     // 내적으로 간단히 처리
mul r0, r0, c1          // 출력 = 텍스처*외부상수
```



- 흐림 효과
  - 인접한 픽셀과 자신의 픽셀에 가중치를 곱해서 최종 색상을 결정
  - 인접한 픽셀의 색상은 텍스처 좌표를 조정해서 얻음
  - HLSL을 이용하면 간단하게 하나의 텍스처를 여러 번 샘플링 해서 구현
  - 쉐이더 버전이 낮으면 2~3번 픽셀 쉐이더를 적용

```
// 정점 구조체
struct VtxDUV1
{
    VEC3 p;
    DWORD d;
    FLOAT u0,v0;
    FLOAT u1,v1;
    FLOAT u2,v2;
    FLOAT u3,v3;
    FLOAT u4,v4;
};
```

```
ps_1_4
def c0, 0.2f, 0.2f, 0.2f,
0.2f
texld r0, t0
texld r1, t1
texld r2, t2
texld r3, t3
texld r4, t4
add r5, r0, r1
add r5, r5, r2
add r5, r5, r3
add r5, r5, r4
mul r1, r5, c0
mov r0, r1
```



- Masking + 흐림 효과
  - 원근 감을 없애기 위해 색상을 단색으로 처리하고 마스킹을 이용해서 경계에서만 색 변화를 유도

```
ps_2_0
def c10, 0, 1, 0.8, 0    // Min, Max, 전체 밝기
def c11, .7, 0.9, .3, 0  // 색상 비중 값

dcl t0                   // t0 텍스처 좌표 선언
dcl_2d s0                // 0-stage 샘플러 객체 선언
dcl_2d s1                // 1-stage 샘플러 객체 선언

// Circle Image
mov r0, t0               // 샘플링 1-stage with 0 stage texture
coordinate
texld r0, r0, s1

// Render Target Image Sampling
mov r1, t0
add r1, r1, c0
texld r1, r1, s0

mov r2, t0
add r2, r2, c0
texld r2, r2, s0
...
// Multiple Masking Value
mul r1, r1, c20.x
mul r2, r2, c20.y
...
// Addi all Pixel
mov r10, r1
add r10, r10, r2
add r10, r10, r3
add r10, r10, r4
...
max r10.rgba, r10.rgba, c10.rrrr
//abs_sat r10, r10

dp3 r10, r10, c11        // 내적으로 간단히 단색 만듦
mul r10, r10, c16        // 외부 상수와 곱해서 최종 색상 출력
mul r10, r0, r10         // Multiple r0, r10
mul r10, r10, c10.zzz    // 전체 밝기 조정
mov oC0, r10
```

```
// Masking
 1, -1,  1
-1,  7, -1
 1, -1,  1
```