# 3D Game Programming 71
## - WebGL

afewhee@gmail.com

- 1. Develop environment and Rendering Context
- 2. Shading Language and Vertex Buffer
- 3. Primitive
- 4. Transform
- 5. Lighting 1
- 6. Lighting 2
- 7. Fog
- 8. Multi-texturing
- 9. Blending




# 1. Develop environment and Rendering Context

- HTML
  - HTML 구동 원리 이해, Layout용 간단한 CSS
  - canvas tag
- JavaScript
  - JavaScript code로 WebGL 작성
- WebGL library
  - OpenGL Method 활용
- GLSL
  - vertex/fragment shading language

- Editor:
  - EditPlus https://www.editplus.com/
- Web Browser
  - Explorer 11, Chrome, Opera
- Helpful Site
  - http://msdn.microsoft.com/en-us/library/ie/hh772398(v=vs.85).aspx
  - http://learningWebGL.com/blog/
  - https://developer.mozilla.org/en-US/docs/Web/WebGL/
  - http://www.w3schools.com/

- Code 작성
  - EditPlus
- Code Run
  - EditPlus -> Ctrl+B(자체 브라우저 실행)
- Detail Debugging and Design
  - Chrome->F12

- Rendering context
  - WebGL을 수행하는 rendering 관련 명령어 등의 문맥
  - canvas에서 얻음.
- TIMER
  - HTML은 정적 page. timer로 주기적으로 page 갱신 <- JavaScript 사용
- Vertex/Fragment Shader Compile
  - JavaScript 사용.
  - vertex shader: script type="x-shader/x-vertex"
  - fragment shader: script type="x-shader/x-fragment"
- Rendering Object
  - Geometry Object: VBO(Vertex buffer object) / IBO(index buffer object)
  - texture object 생성
- Rendering
  - back buffer clear
  - bind to pipe line the vbo, ibo, texture objects
  - draw primitive

- static canvas: BODY에 canvas tag 이용

```
<body . . .>
<canvas id="canvas01" width ="800" height="480"> </ canvas>
...
var native_win = document.getElementById("canvas01");
```

- dynamic canvas: JavaScript로 생성

```
var native_win = document.createElement("canvas");
native_win.width  = 800;
native_win.height = 480;
document.body.appendChild(native_win);
```

- Rendering Context 생성

```
gl = native_win.getContext("experimental-WebGL");
```

- Timer
  - setInterval("Rendering Function", 0);

- Web page

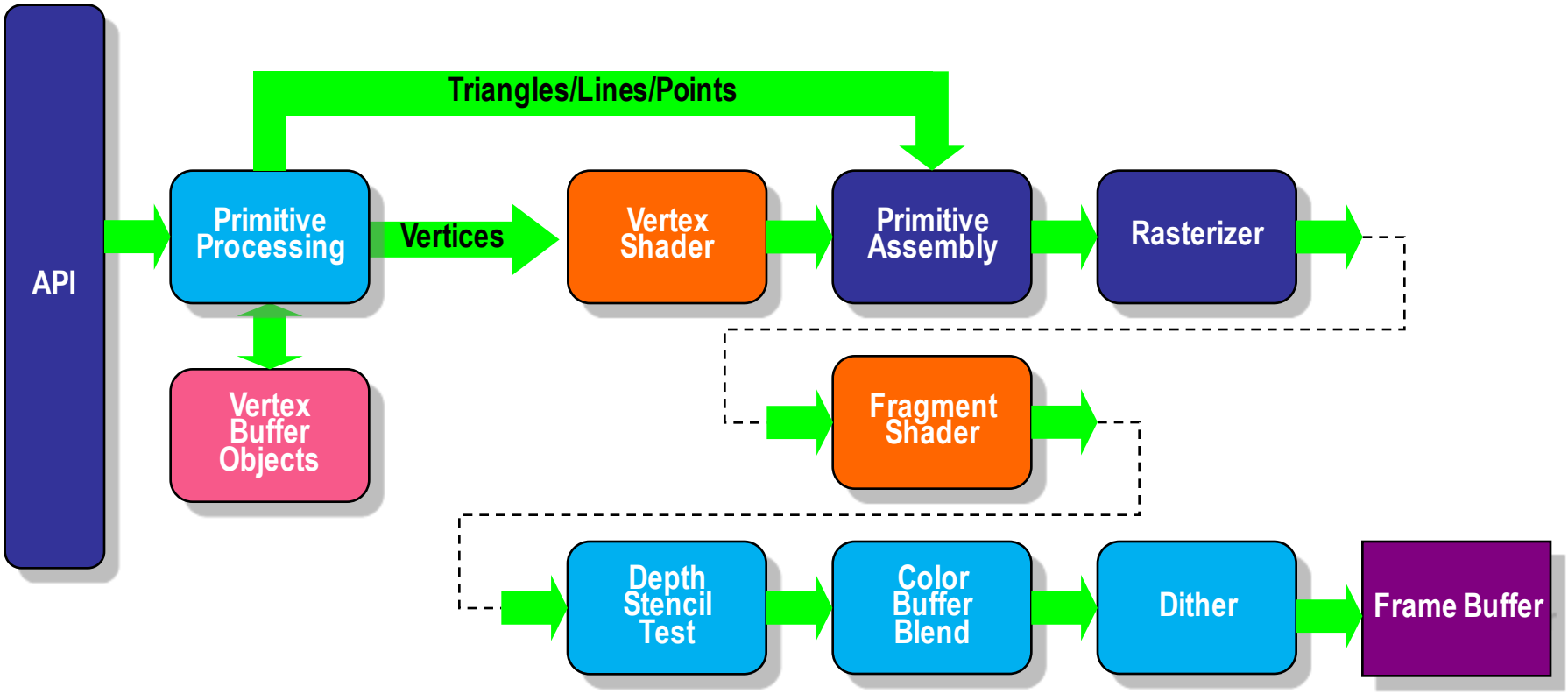
```
<html>
<head>
<script type="text/javascript">
var gl;                         // global context variable
…
function WebGLRender() { … }

function WebGLStart() {
        …
        // initialize gl context
        try {
                gl = native_win.getContext("experimental-webgl");
        }
        catch (e) {
        }

        setInterval("WebGLRender()", 0);
}
</script>
</head>

<body onload="WebGLStart();">
…
```

- CONSTANTS
  - 대문자로 시작
  - 문법: context object.CONSTANT_NAME
  - ex)
    - gl.DEPTH_TEST
    - gl.CULL_FACE

- Method
  - Camel Casing Notation, 소문자로 시작
  - 문법: context object.methodName
  - ex)
    - gl.clearColor
    - gl.viewport
    - gl.clear
    - gl.flush

- Context
  - gl = native_win.getContext("experimental-webgl");
- Timer
  - setInterval("WebGLRender()", 0);
- Rendering Method
  - gl.clearColor(r, g, b, a);
  - gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
  - gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
  - gl.flush();

# 2. Shading Language and Vertex Buffer



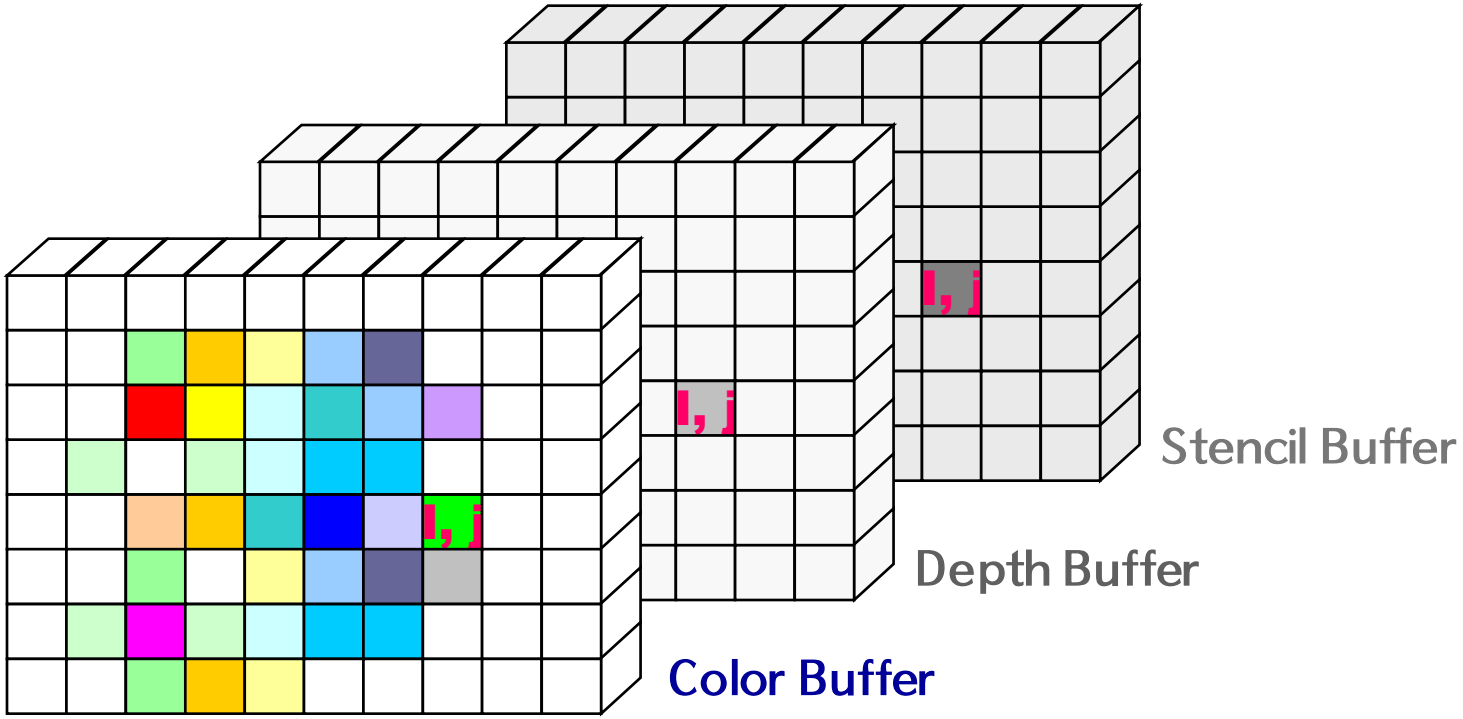

※ WegGL 1.x는 OpenGL 2.0에 대응

- Frame buffer
  - 비디오 장치에 frame을 출력하기 위한 버퍼
  - WebGL frame buffer : 속도를 위해 2중 버퍼링 사용
    - front buffer, back buffer

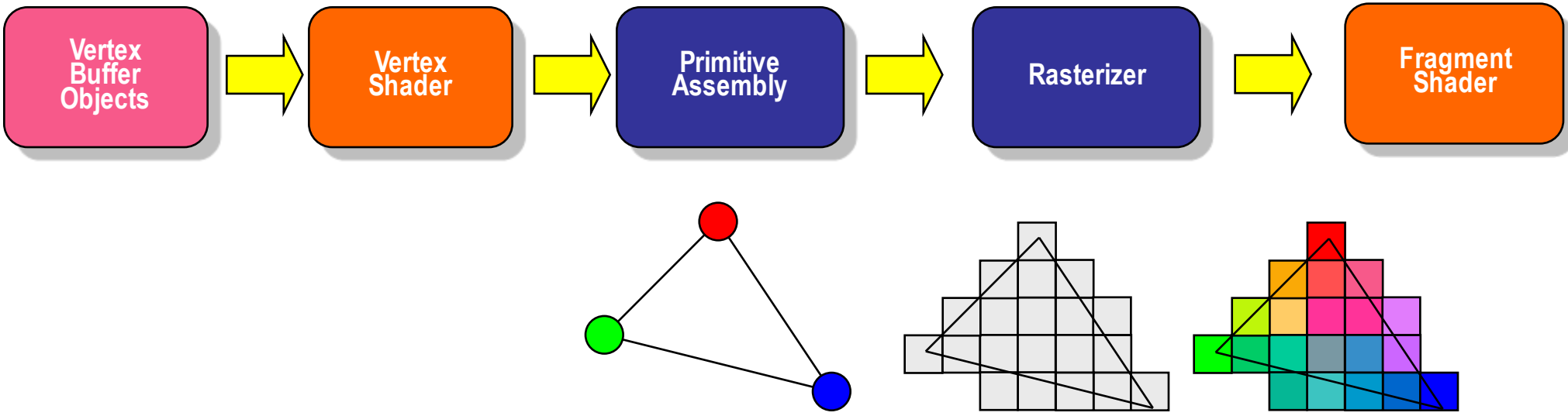
- Front Buffer
  - 현재 비디오 장치에 출력되는 frame을 저장하고 있는 버퍼
  - 접근 불가

- Back buffer
  - 현재화면 다음에 출력할 frame을 저장하고 있는 버퍼
  - Color, Depth, Stencil buffer
  - 프로그래밍 가능한 버퍼

- Color Buffer : Pixel 정보 저장
- Depth Buffer: Depth test(픽셀의 선후차성 검사)를 위한 깊이 값 저장
- Stencil buffer: 스텐실 마스크를 위한 스텐실 인덱스 저장

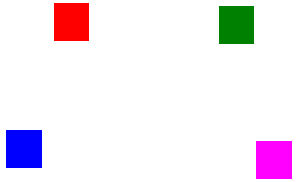



※ 3D 프로그래밍은 Back buffer의 Color, Depth Stencil buffer를 채우는 것

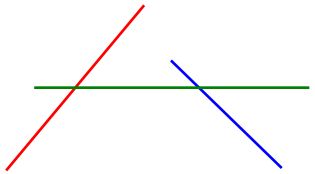


- Vertex Processing
  - Primitive Processing -> Rasterizer
- Primitive Processing
  - Primitive: 렌더링 구성 단위
  - Vertex Shader과정에서 정점 데이터 출력을 실행하고 수집 가능한 primitive를 결정
- Vertex Buffer Objects(VBO)
  - 프리미티브를 구성하는 버퍼
  - 렌더링 성능 향상 제공

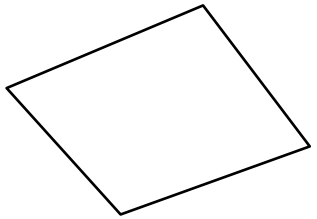

- Vertex Shader
  - 입력 받은 VBO, IBO 정보를 사용 Transform and Lighting (T&L) 수행
- Primitive Assembly
  - 프리미티브: 점, 라인, 삼각형 등 장면에 연출 되는 Geometry Object
- 프리미티브 종류
  - 기본: Point Sprite, Lines, Line Strip, Line Loop, Triangles, Triangle Strip, Triangle Fan
  - Indexed primitive: index 정보를 사용하는 프리미티브
- Rasterizer
  - Vertex Processing 에서 fragment processing으로 넘기는 픽셀(색상, 깊이, 스텐실 값을 결정하는)최종 과정

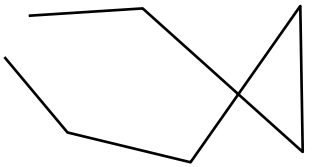
GL_POINTS



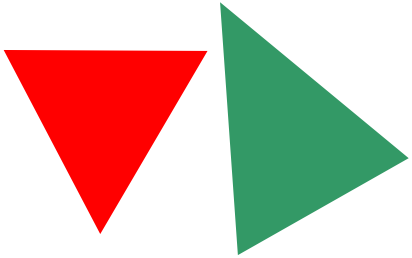
GL_LINES



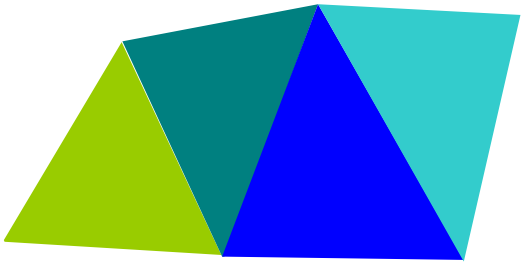
GL_LINE_LOOP



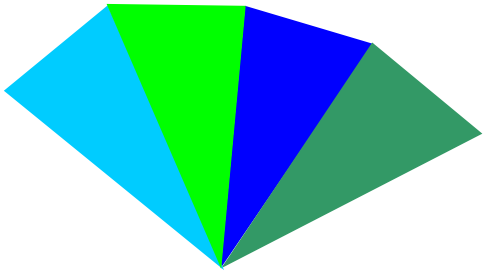
GL_LINE_STRIP



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

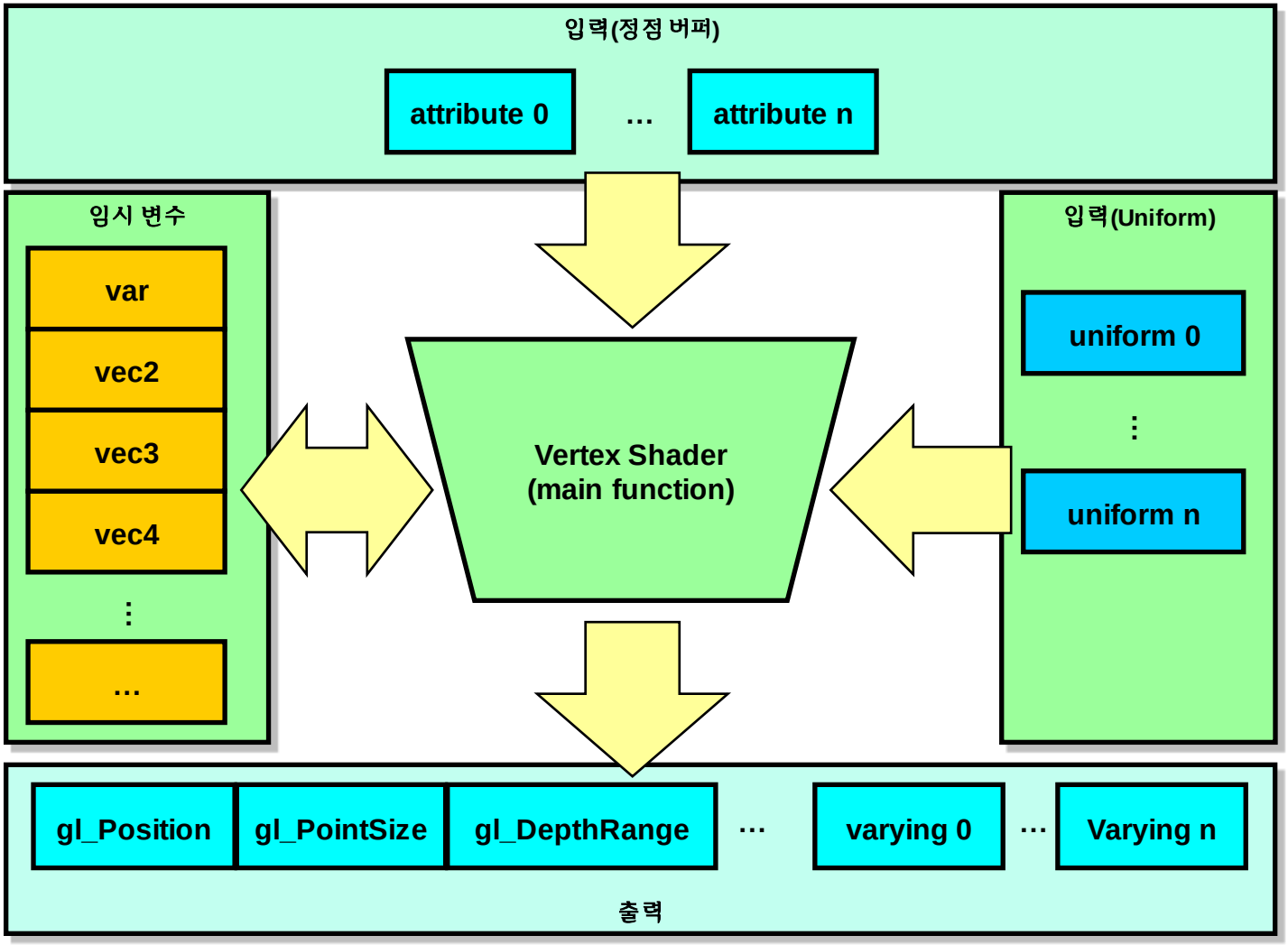
- Fragment Shader -> Dither

- Fragment Shader
  - Sampling
    - texture에서 pixel 정보 추출
    - 최소 n개의 sampling 가능
  - Multi texturing
    - Vertex processing 에서 입력 받은 색상 정보와 sampling에서 추출한 색상을 혼합하는 과정
  - Discard
    - 렌더링이 필요 없을 때 이후 과정 포기
    - discard 명령어 사용
    - Alpha test 대신 가능

- Depth Test
  - 레스터라이저에서 만든 depth값과 back buffer에 저장된 깊이 값 비교
- Stencil Test
  - 입력 받은 stencil 값과 back buffer에 저장된 스텐실 값 비교
- Color Buffer Blending
  - Fragment Shader에서 만든 픽셀(rgba)과 depth값과 back buffer에 저장된 픽셀과 혼합
- Dithering
  - 지원되지 않는 색상을 기본색을 조합해서 표현

- 목적: Transform & Lighting
  - Transform: 정점 위치, 법선, texture 좌표의 변환
  - Lighting: Lambert, Phong shading
  - Fog, Diffuse
  - Fragment shader 입력 값 결정

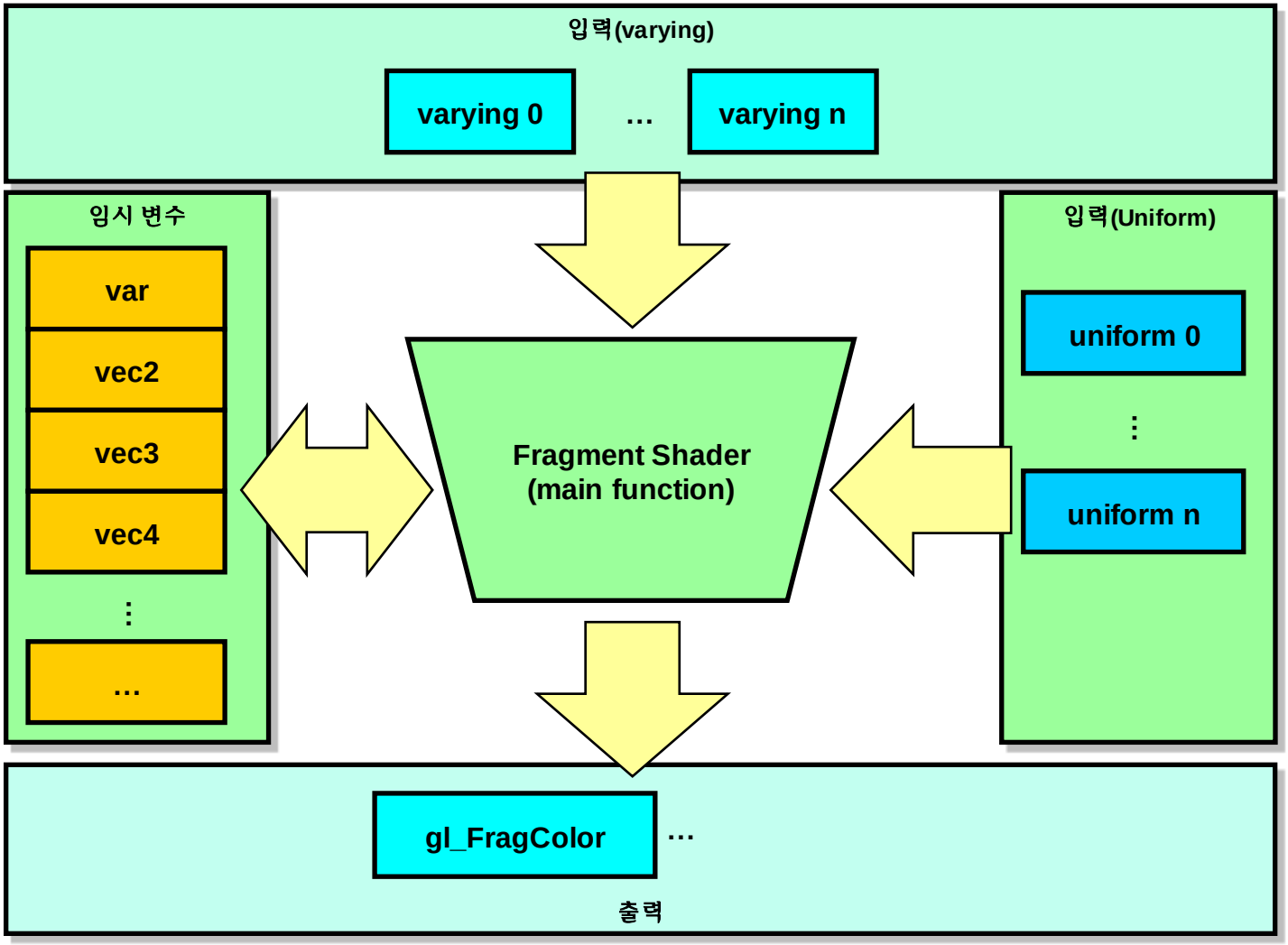
- 결과:
  - 정점 위치: gl_Position
  - Point sprite size : gl_PointSize
  - Fragment shader 입력 값: varying



- 입력
  - 정점 버퍼: attribute
  - 상태 머신 함수 : uniform
- 임시변수
  - var, int, vec2, vec3, vec4 형 변수
- 출력
  - Built-in variables: gl_로 시작. gl_Position, gl_PointSize…
  - For Fragment Variable: varying으로 시작
    - Ex)varying vec4 vr_diffuse
- main()
  - Vertex Shader, Fragment Shader에서 시작 함수

```
attribute vec3      at_pos;         // input position from rendering pipe line
attribute vec4      at_dif;         // input diffuse from rendering pipe line

varying   vec4       vr_dif;        // output to fragment processing

void main(void) {
        gl_Position = vec4(at_pos, 1.0);
        vr_dif = at_dif;
}
```

- 색상 버퍼 색상을 갱신하기 위한 전 단계
- Sampling: texture에서 pixel 정보 추출: texture2D()
- Multi-texturing
  - Color 연산: 색상 범위 [0, 1.0]
  - 가산 연산(덧셈): 밝아짐
  - 감산 연산: 뺄셈, 곱셈. 어두워짐
  - 범위가 0~1.0이기 때문에 곱셈하면 어두워짐
- discard: 렌더링이 필요 없을 때 이후 과정 포기
- 출력: gl_FragColor



- 입력
  - Vertex Shader: varying
  - 상태 머신 함수 : uniform
- 임시변수
  - var, int, vec2, vec3, vec4 형 변수
- 출력
  - Built-in variables: gl_FragColor
- main()
  - Vertex Shader, Fragment Shader에서 시작 함수

```
varying    vec4   vr_dif;              // diffuse. input from vertex processing
varying    vec2   vr_tex;              // texture coordinate from vertex processing
uniform    sampler2D us_tex;           // sampler for texture

void main(void) {
          vec4 ct0       = texture2D(us_tex, vr_tex);   // sampling
          gl_FragColor = ct0 * vr_dif* 2.0;             // 감산연산 * 2.0 배
}
```

- Shader 생성
  - Vertex Shader: gl.createShader(gl.VERTEX_SHADER)
  - Fragment Shader: gl.createShader(gl.FRAGMENT_SHADER)

- Compile
  - gl.shaderSource(shader, str);
  - gl.compileShader(shader);

- Program Object
  - Vertex/Fragment shader 실행자
  - Shader 연결: gl.attachShader()
  - Linking: vertex shader, fragment shader 조합:
    - gl.linkProgram(g_program);
  - Linking 검사
    - gl.getProgramParameter(g_program, gl.LINK_STATUS)


- VBO(Vertex buffer Object) 생성
  - wgb_pos = gl.createBuffer();
- VBO data Fill
  - gl.bindBuffer(gl.ARRAY_BUFFER, wgb_pos);
  - gl.bufferData(gl.ARRAY_BUFFER, new var32Array(vtx_pos), gl.STATIC_DRAW);
- Rendering
  - gl.enableVertexAttribArray(index);
    gl.bindBuffer(gl.ARRAY_BUFFER, wgb_pos);
    gl.vertexAttribPointer(index, 3, gl.var, false, 0, 0);
- 가장 많이 사용되는 VBO
  - position, normal, diffuse, texture coordinate


- Texture 생성

```
tex = gl.createTexture();
tex.image = new Image();
tex.image.src =  “image name";
tex.image.onload = function() { TextureOnLoad(tex_mario); }
```

- Pixel Fill, Filtering/Addressing: Texture 생성에서 결정

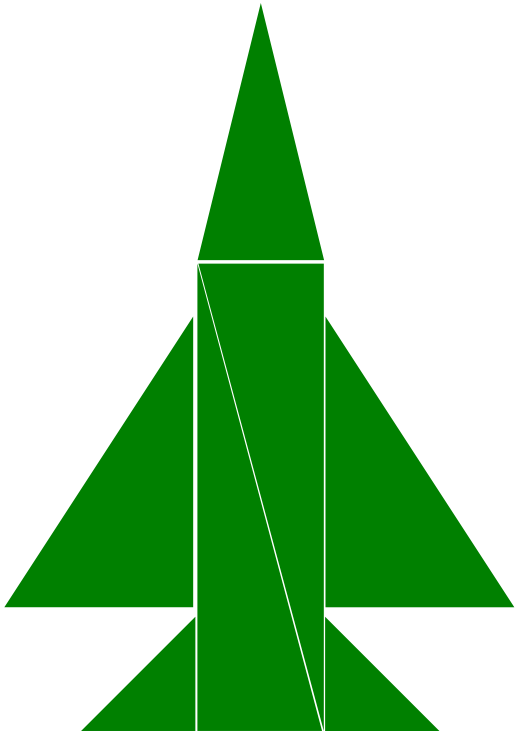
```
gl.bindTexture(gl.TEXTURE_2D, texture);

gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.LINEAR_MIPMAP_NEAREST);

gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
texture.image);

gl.generateMipmap(gl.TEXTURE_2D);
gl.bindTexture(gl.TEXTURE_2D, null);
```


- Rendering

```
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, tex_object);
gl.uniform1i(gl.getUniformLocation(programObject, "sampler_name"), 0);

gl.drawArrays(…);                              // draw..

// disable client-side capability
…
gl.bindTexture(gl.TEXTURE_2D, null);
```

- 상태 머신 설정
  - gl.Enable/Disable …
- Program Object
  - gl.useProgram(g_program)
- Attribute 활성화, binding VBO
  - gl.enableVertexAttribArray(0)
  - gl.bindBuffer(gl.ARRAY_BUFFER, wgb_pos)
  - gl.vertexAttribPointer(0, 3, gl.var, false, 0, 0)
- Texture binding
  - gl.activeTexture(gl.TEXTURE0);
  - gl.bindTexture(gl.TEXTURE_2D, tex_mario);
  - gl.uniform1i(gl.getUniformLocation(g_program, "us_tex"), 0);
- Rendering
  - gl.drawArrays(…)
- Program Object, VBO, Texture 연결 해제
  - gl.disableVertexAttribArray(n);
  - gl.bindTexture(gl.TEXTURE_2D, null);
  - gl.useProgram(null);


- 제시한 sample 코드에서 gl method와 관련한 함수들을 모두 찾고 이들의 의미를 적어보자.
- 제시한 sample 코드를 가지고 사각형 4개를 생성해보자. 4개의 사각형은 red, green, blue, magenta 색상을 주어보자.(텍스처는 적용할 필요 없음)
- 제시한 sample 코드를 가지고 다음 그림과 같은 비행기를 만들어보자.(색상은 자유)



# 3. Primitive

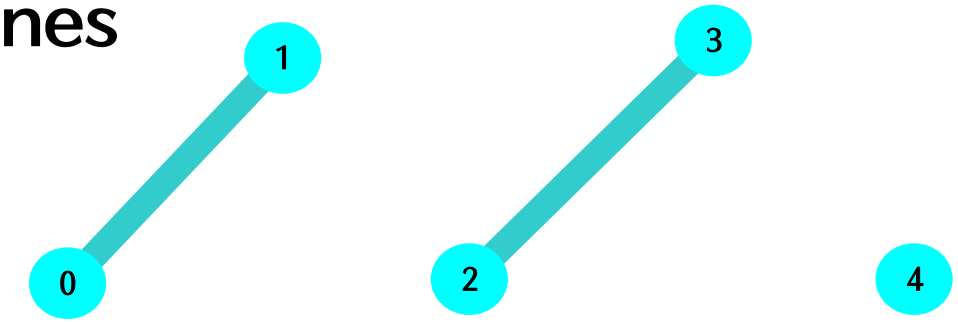
- 프리미티브 (Primitive)
  - 그래픽 출력의 기본 단위 ➔ 생명체의 세포
  - 정점을 조합해서 Geometry를 결정
- 종류
  - 점(Point), 선(Line), 삼각형(Triangle), 사각형(Quad: OpenGL)
- 타입
  - List, Loop, Strip, Fan
- Geometry
  - 점: Point List
  - 선: Line List, Line Strip, Line Loop
  - 삼각형: Triangle List, Triangle Strip, Triangle Fan

- List - Lines
  - 반직선의 집합
  - 정점 수 = n 개 Line * 2
  - gl.drawArrays(gl.LINES, …), gl.drawElements(gl.LINES, …)

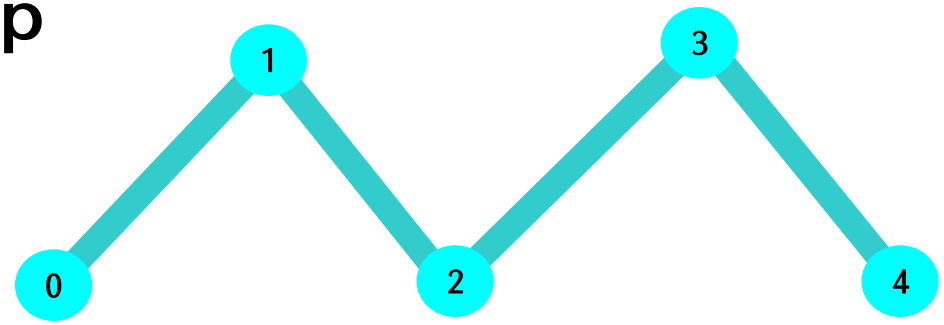
- Line Strip
  - 연속된 Line 집합
  - Line의 시작점은 이전 Line의 마지막 점
  - 정점 수 = n 개 Line +1
  - gl.drawArrays(gl.LINE_STRIP, …), gl.drawElements(gl. LINE_STRIP, …)

- Line Loop
  - 폐쇠된 Line 생성
  - Strip에서 마지막의 정점이 시작과 연결
  - gl.drawArrays(gl.LINE_LOOP, …) , gl.drawElements(gl. LINE_LOOP, …)

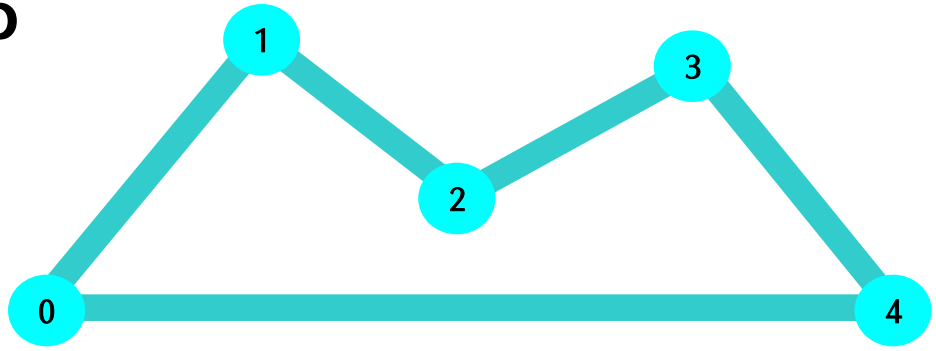

- List - Lines




- Line Strip




- Line Loop

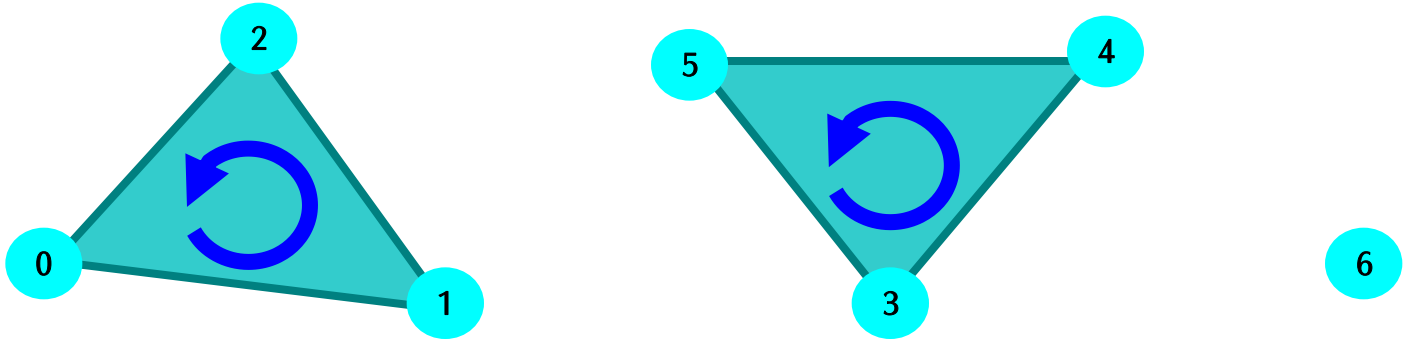


- List - Triangles
  - 삼각형들의 집합
  - 정점 수 = n 개 Triangle * 3
  - gl.drawArrays(gl.TRIANGLES, …), gl.drawElements(gl. TRIANGLES, …)

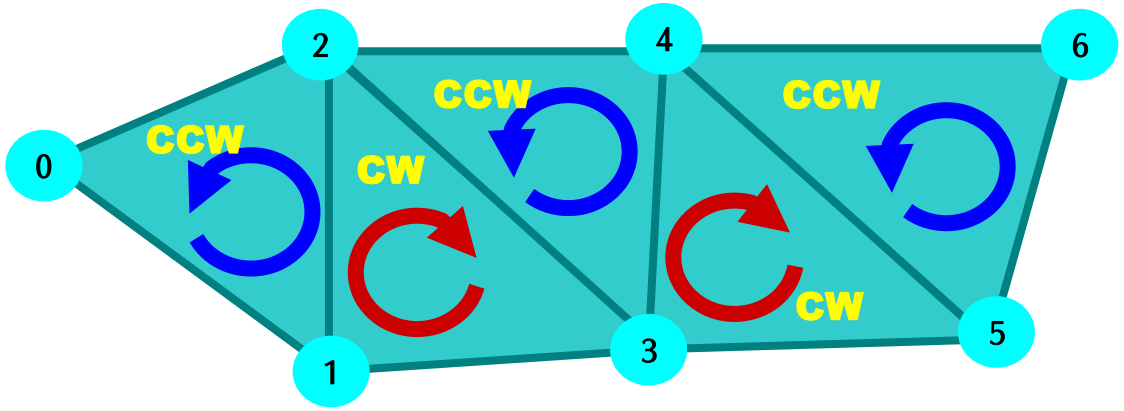
- Triangle Strip
  - 연속된 Triangle 집합
  - 렌더링 순서 0->1->2, 1->2->3, 2->3->4 , 3->4->5,…순으로 CCW와 CW를 번갈아 교대하면서 렌더링
  - 정점 수 = n 개 Triangle +2
  - gl.drawArrays(gl.TRIANGLE_STRIP, …), gl.drawElements(gl. TRIANGLE_STRIP, …)

- Triangle Fan
  - 부채꼴(Fan)과 같은 형태의 삼각형 리스트 구성
  - 모든 삼각형의 시작점은 최초 시작점
  - 렌더링 순서 0->1->2->3->4->5->6…
  - gl.drawArrays(gl.TRIANGLE_FAN, …), gl.drawElements(gl. TRIANGLE_FAN, …)

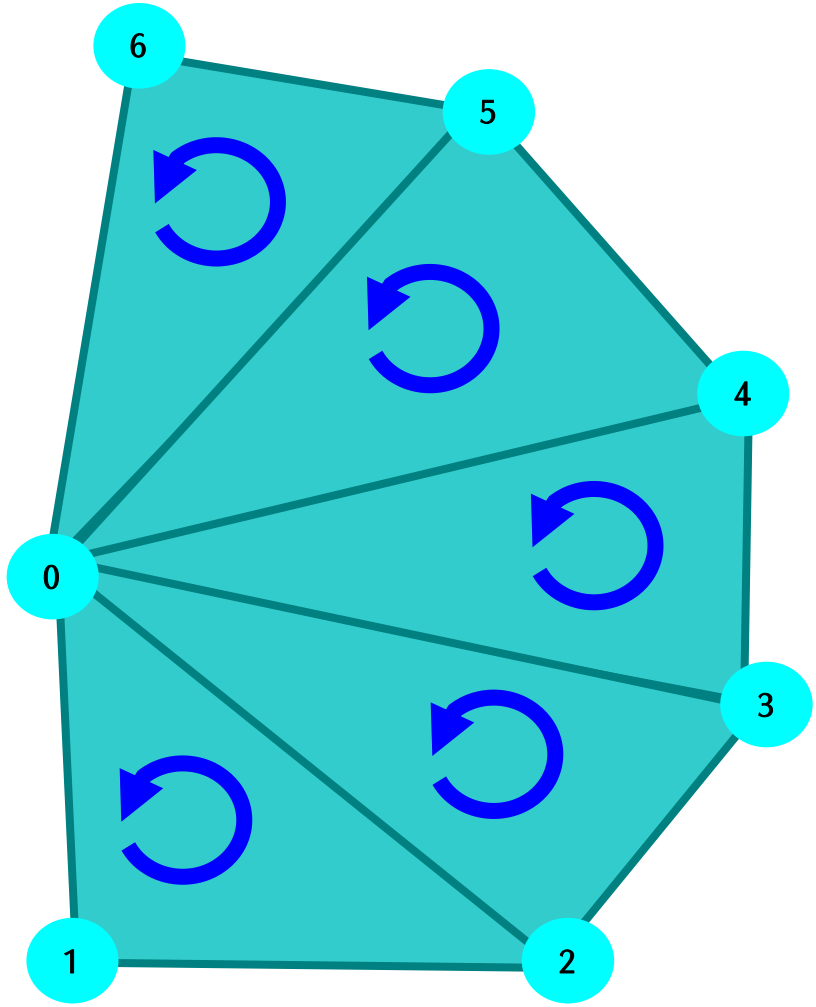
- List - Triangles

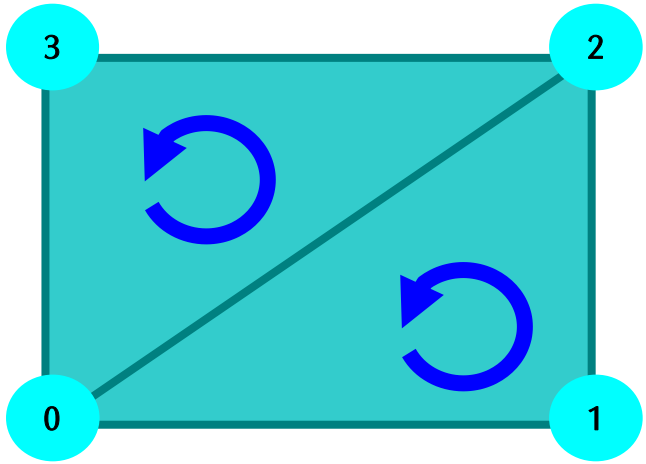



- Triangle Strip



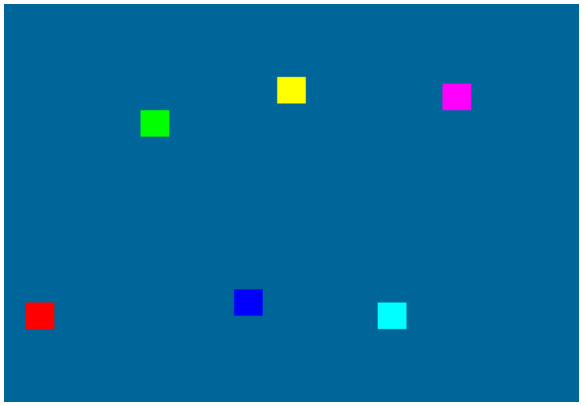
- Triangle Fan

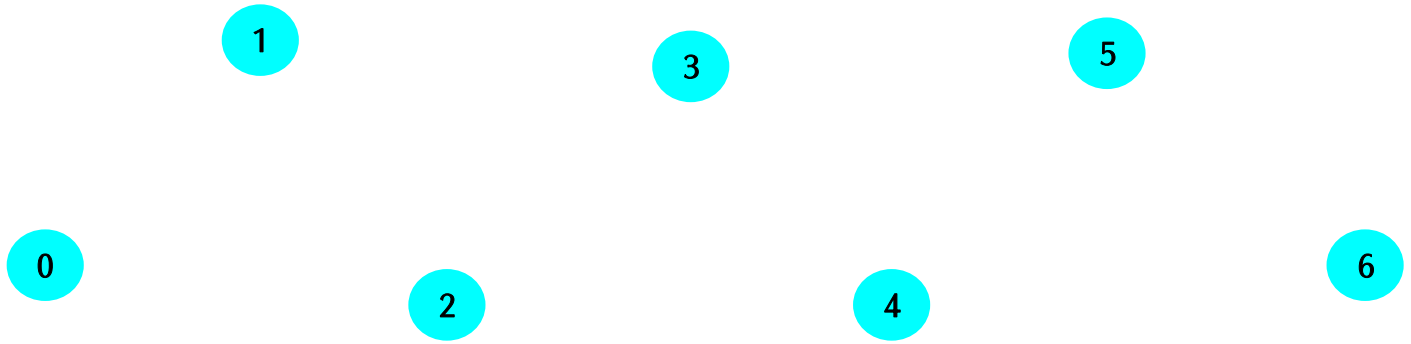


- 점에 대한 단 하나의 프리미티브
- 개별적인 점을 렌더링

- 필요한 정점의 수 = 프리미티브 수
- gl.drawArrays(gl.POINTS, …)





- Index 사용
  - Geometry Object를 Triangle Strip, Fan의 방식으로 그릴 수 있다면 List보다 1/3 정도 메모리만 필요
  - 대부분의 Geometry Object 패턴은 대부분 List로 그려야만 하는데 Index List를 사용하면 메모리 절약 가능
  - 정점을 0번부터 차례로 설정, 삼각형을 구성하는 인덱스 리스트 작성, 인덱스에 맞게 프리미티브를 렌더링

  Ex) Triangle Strip을 인덱스로 바꾸어 보기
  v0->v1->v2 ➔ index(0,1,2)
  v1->v2->v3 ➔ index(1,2,3)
  v2->v3->v4 ➔ index(2,3,4)
  v3->v4->v5 ➔ index(3,4 5)

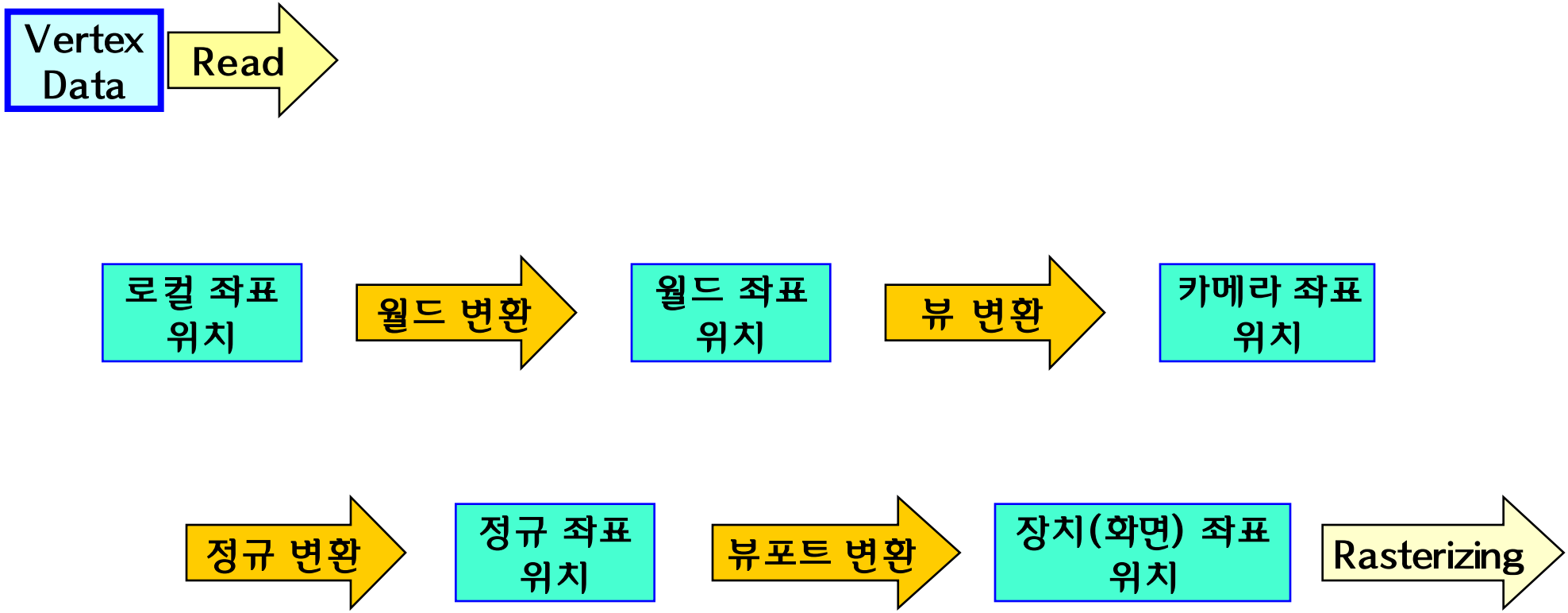
- 사용 메모리 크기:
  - Strip : ~프리미티브 수 * 정점 사이즈
  - Index List: 프리미티브 수 * 정점 사이즈 + Index List Size(~0) (← 정점 사이즈에 비해 작아음)
  - 인덱스 리스트가 WORD(16bit:2Byte) 이면 하나의 삼각형을 구성하는데 6Byte가 필요
  - 정점은 위치를 포함하므로 3개의 정점은 최소 3* sizeof(var3) = 3* 4 * 3 = 36 Byte. 게임에서는 보통 텍스쳐 좌표와 법선 벡터가 포함되므로 3 * (sizeof(position) + sizeof(normal) + sizeof(texture coord)) = 3 * 4 * (3 + 3 + 2) = 96Byte
  - 인데스와 정점의 크기는 최대 6/36 = 1/6 보다 작고 일반적인 경우에는 6/96~0.07 가 되므로 인덱스를 사용하면 거의 Strip 방식에서 사용하는 메모리 크기와 비슷
  - 적은 메모리를 사용하면 속도의 이득이 있으므로 Index List를 사용하면 Strip, Fan과 비슷한 렌더링 속도

- 생성
  - VBO와 동일, Index 정보를 저장
  - wgb_idx = gl.createBuffer();
- IBO data Fill
  - gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, wgb_idx);
  - gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(vtx_idx), gl.STATIC_DRAW);
- Rendering
  - gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, wgb_idx);
  - gl.drawElements …
  - 해제
    - gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);

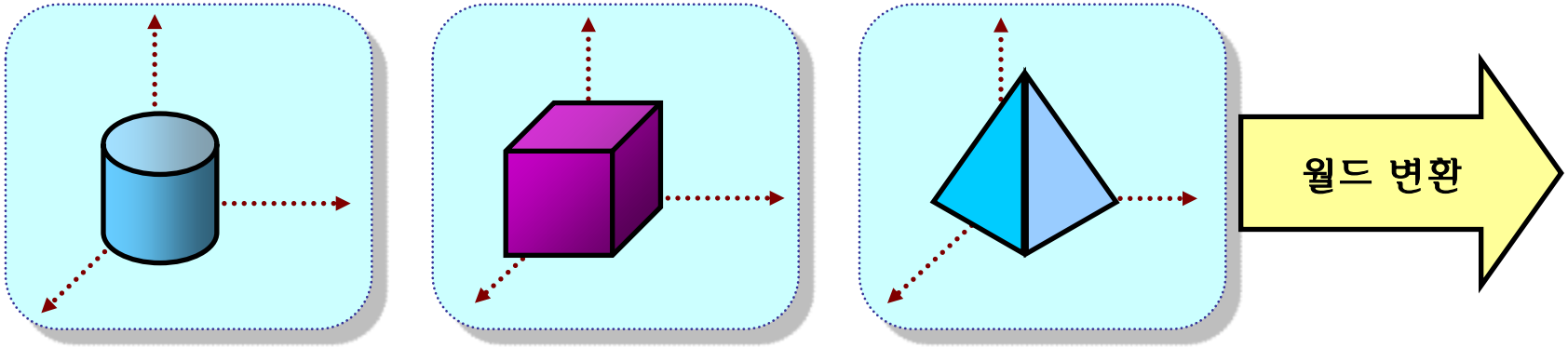
- X, Y, Z 지시선을 그려보자.
- 직육면체를 Triangle List로 작성해보자.
- 위의 직육면체를 Index Buffer를 사용해보자.
- 삼각형에 대한 인덱스용 구조체를 작성하고, 이를 토대로 앞서 만든 직육면체를 UP를 이용해서 렌더링 해보자.
- 직육면체를 여러 개 이용해서 로봇을 만들어 보자.

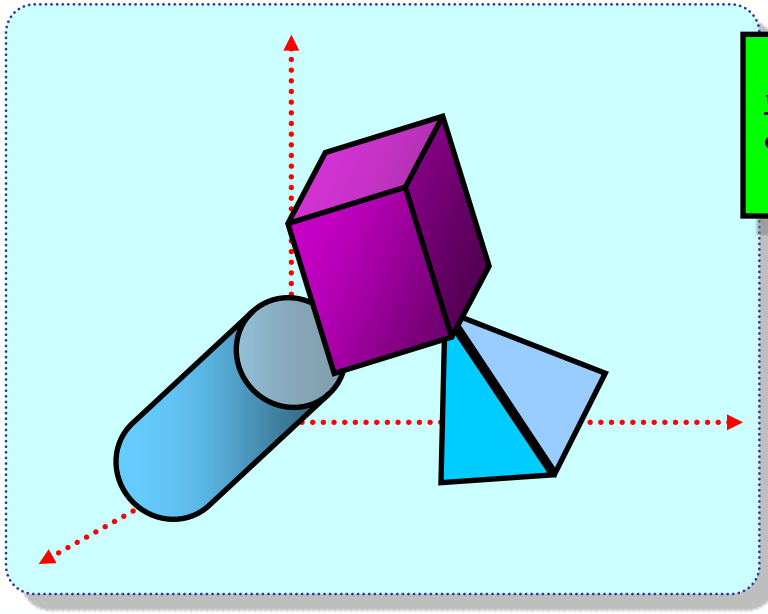
# 4. Transform

- 기하 변환 (Geometry transform)
  - 좌표와 관련된 지오메트리의 정점 또는 텍스처 좌표의 변환
  - 회전 : rotation
  - 크기 : scaling
  - 평행 이동 : translation
- Pipe line transform
  - 파이프라인에서 기하 변환 수행
  - World transform: 월드 변환
  - Viewing transform: 뷰 변환 == 카메라 변환
  - Projection transform: 정규 변환
  - Viewport: 뷰포트 변환 == 장치 변환



각각의 로컬 좌표계(모델 좌표계)에 만들어진 Geometry



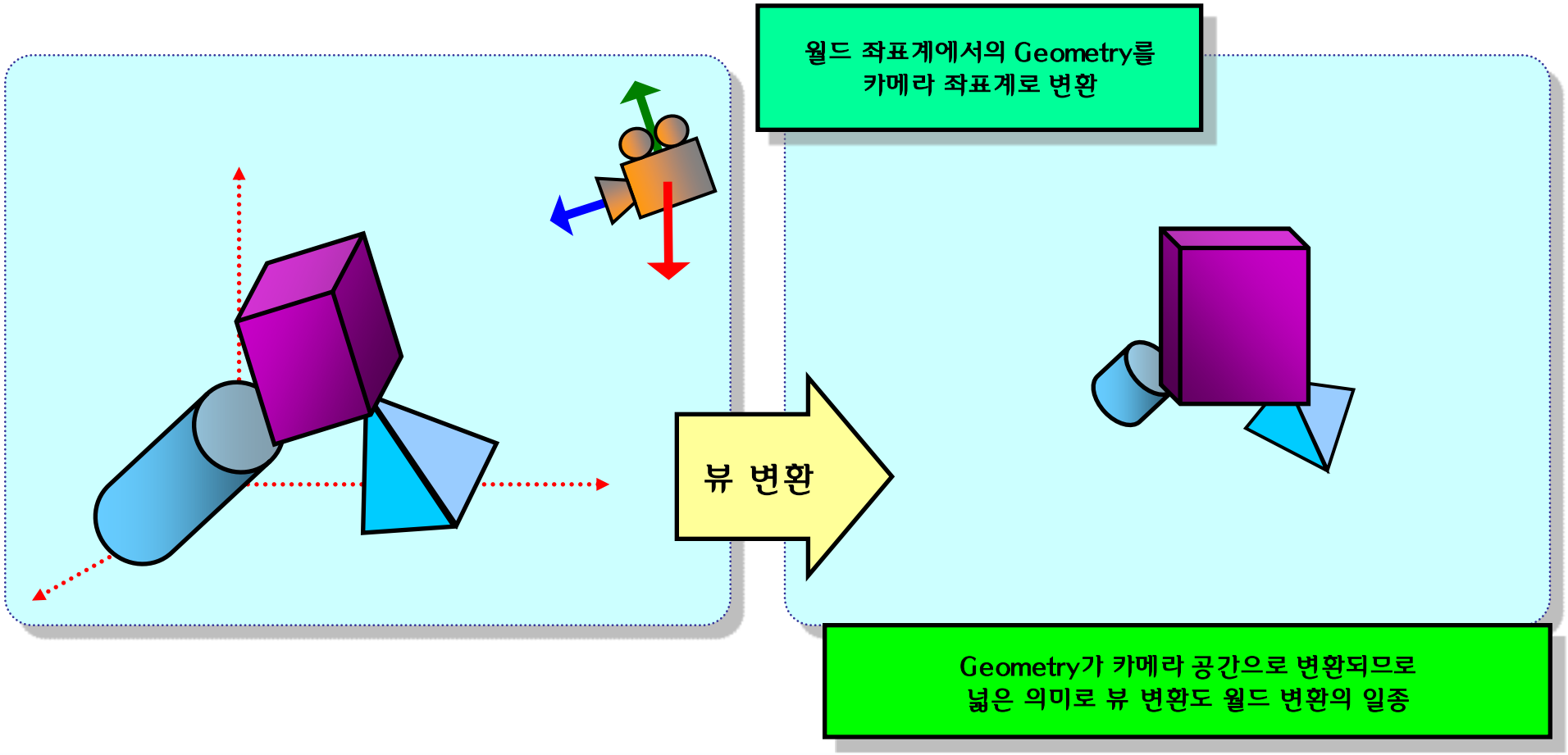
모델 좌표계에서 만들어진 Geometry는 크기, 회전,
이동 변환을 거쳐 월드 좌표계의 정점으로 변환

모델 좌표계: Geometry Object를 구성하기 위한 좌표계

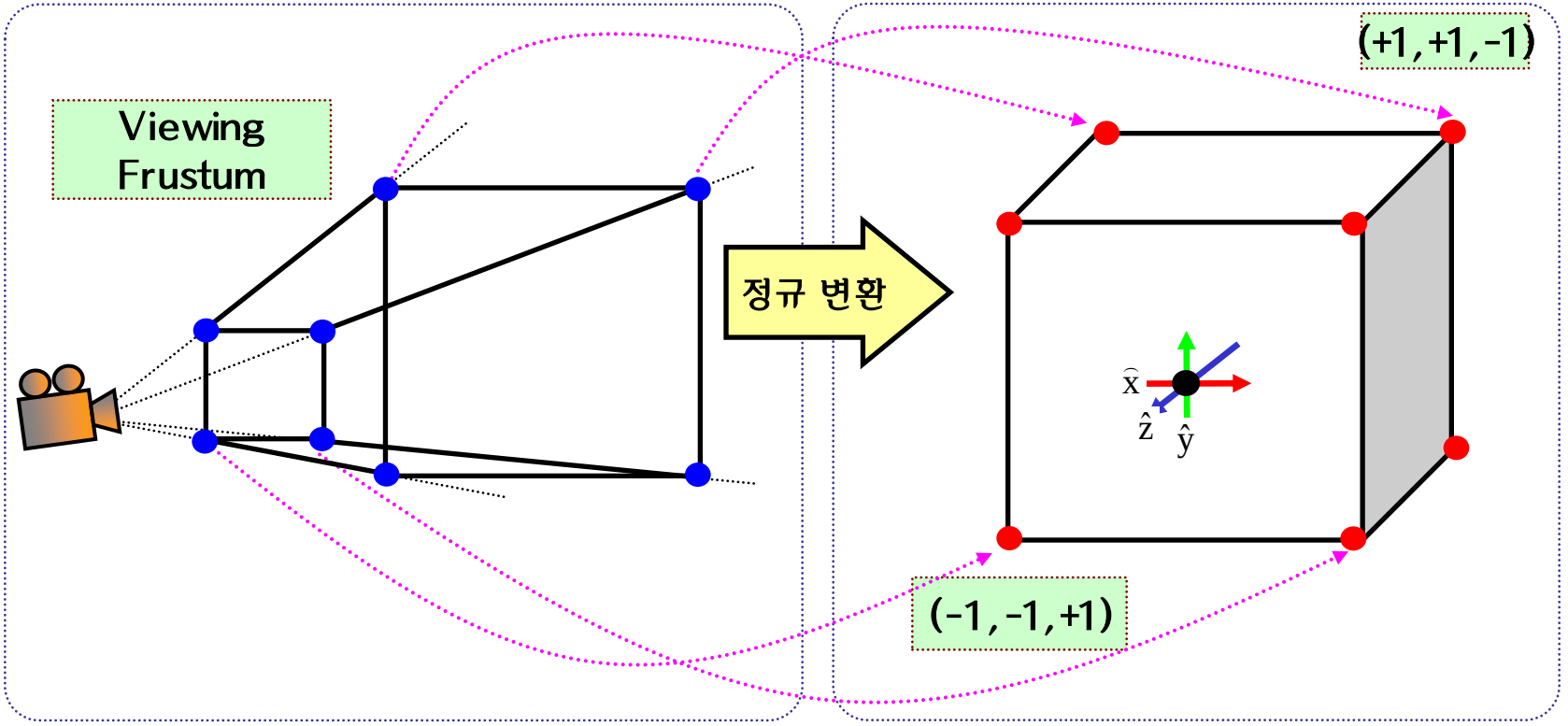
월드 좌표계: 3D 장면을 연출 하기 위한 좌표계

- 뷰잉 변환 (View Transform) ➔ Camera 좌표계로 변환

- 뷰 변환 행렬 적용
  - 뷰 행렬 만들기: D3DXMatrixLookAtLH()
  - 상태 머신 설정: pDevice->SetTrasform(D3DTS_VIEW, &View_Matrix)

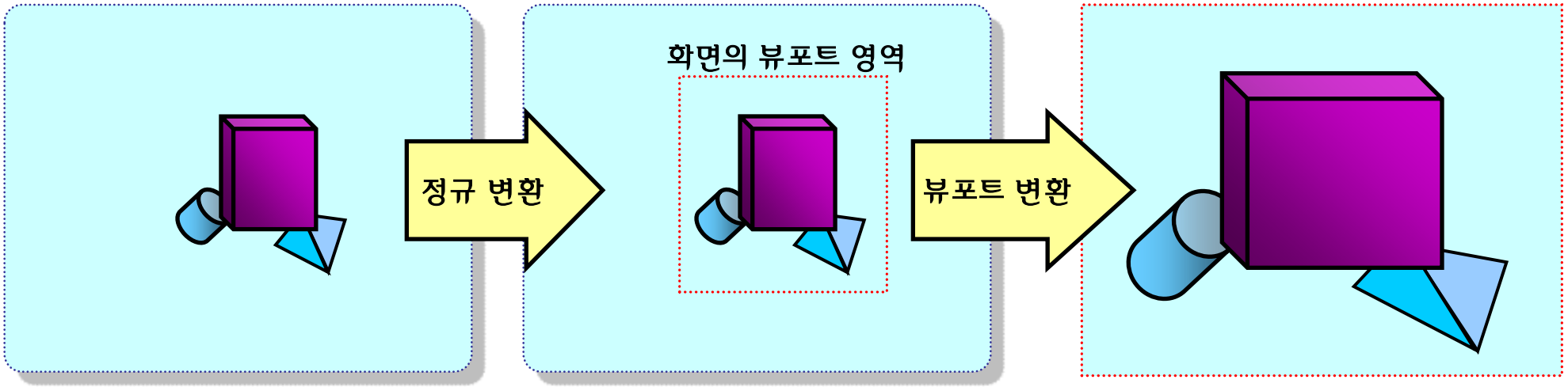


- 정규 변환 (View Transform) ➔ 투영 변환(Projection Transform)
  - Camera 좌표계의 Geometry를 투영 평면으로 변환
  - 카메라의 뷰 체적(View Volume) 안의 Geometry는 [-1,1] 범위의 값을 가짐
  - 장치에 독립




뷰 체적(View Volume)과 정규변환

- 뷰포트 변환
  - 장치 의존 변환
  - 해당 윈도우의 클리핑 영역에 맞게 변환
  - 뷰포트를 설정하지 않으면 Device의 가로, 세로 폭이 해당 화면의 폭과 높이로 설정
  - Vertex Shader가 아닌 파이프라인에서 진행



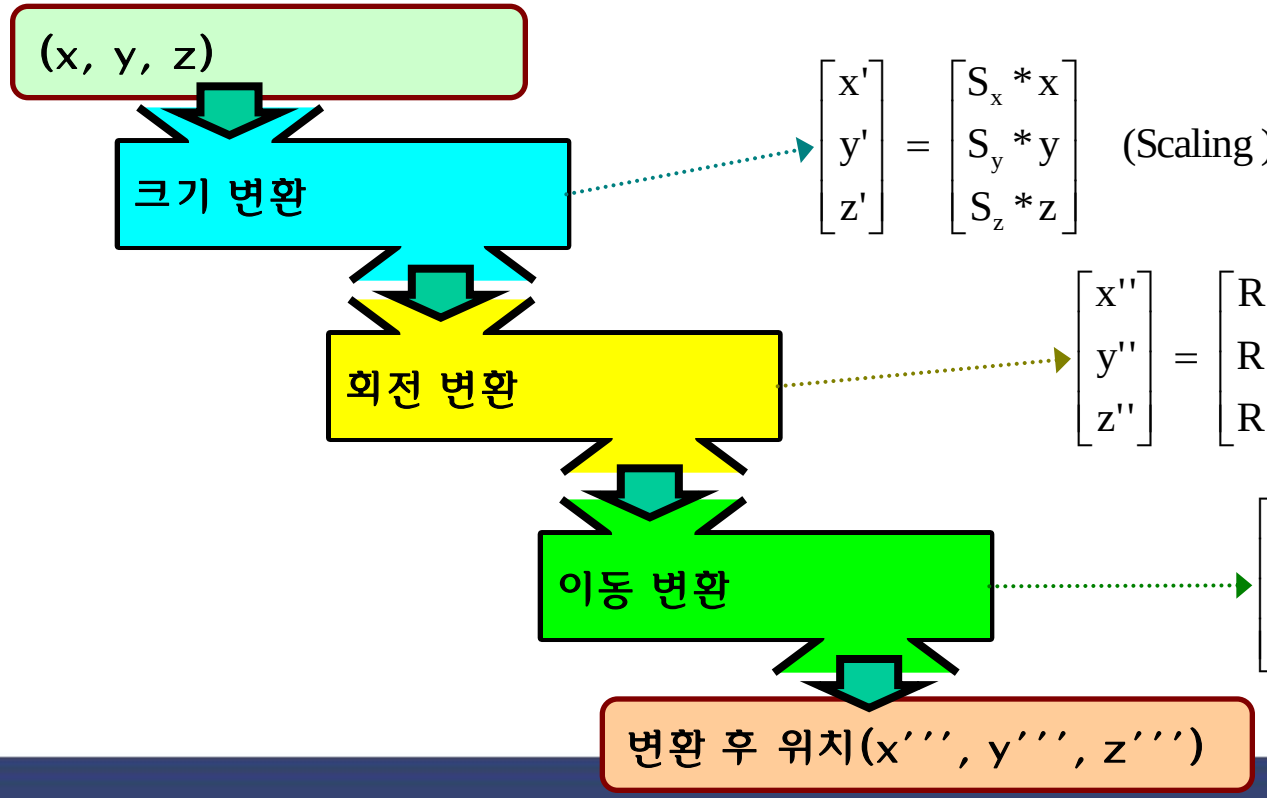
- 4x4 행렬 사용
  - 4x4 행렬을 사용하면 기하 변환의 회전, 크기, 이동(평행 이동)을 한 번에 처리가 가능
  - mat4, mat3, mat2 형 행렬 사용
- 벡터와 곱셈
  - Row major 행렬 : 행렬 * 벡터
  - Column major 행렬: 벡터 * 행렬
- attribute 확장
  - vec3 -> vec4 자동으로 w= 1.0 추가
  - vec2 -> vec4 자동으로 z=0.0, w= 1.0 추가


```
// shader
attribute vec4      at_pos;
uniform   mat4      um_Wld;
uniform   mat4      um_Viw;
uniform   mat4      um_Prj;

void main() {
vec4 pos    = um_Wld * at_pos;      // world transform
pos         = um_Viw * pos;         // view transform
pos         = um_Prj * pos;         // projection transform
gl_Position = pos;



// Rendering...
gl.useProgram(g_program);

gl.uniformMatrix4fv(gl.getUniformLocation(g_program, "um_Wld"), false, um_Wld);
gl.uniformMatrix4fv(gl.getUniformLocation(g_program, "um_Viw"), false, um_Viw);
gl.uniformMatrix4fv(gl.getUniformLocation(g_program, "um_Prj"), false, um_Prj);
```


- 3x3 행렬에 대한 기하 변환

$$[x \quad y \quad z]\begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}$$

- 3x3 행렬에 대한 정점 변환
  - 여러 단계를 거침




$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} S_x * x \\ S_y * y \\ S_z * z \end{bmatrix} \quad \text{(Scaling)}$$

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix}\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \quad \text{(Rotation)}$$

$$\begin{bmatrix} x''' \\ y''' \\ z''' \end{bmatrix} = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} + \begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} \quad \text{(Translation)}$$

- 4x4 행렬을 이용한 기하 변환
  - 하나의 행렬로 계산 되므로 계산하는 단계가 적음

$$[x' \quad y' \quad z' \quad w'] = [x \quad y \quad z \quad 1] \begin{bmatrix} R_{11} & R_{12} & R_{13} & M_{14} \\ R_{21} & R_{22} & R_{23} & M_{24} \\ R_{31} & R_{32} & R_{33} & M_{34} \\ T_x & T_y & T_z & M_{44} \end{bmatrix}$$

$$x' = x*R_{11} + y*R_{21} + z*R_{31} + T_x$$
$$y' = x*R_{12} + y*R_{22} + z*R_{32} + T_y$$
$$z' = x*R_{13} + y*R_{23} + z*R_{33} + T_z$$
$$w' = x*R_{14} + y*R_{24} + z*R_{34} + R_{44}$$
$$(\text{if } R_{14} \leftarrow 0 \text{ and } R_{24} \leftarrow 0 \text{ and } R_{34} \leftarrow 0 \text{ and } R_{44} \leftarrow 1 \text{ then } w' = 1)$$

$$x'/=w', x'/=w', x'/=w', x'/=w'$$
$$[x' \quad y' \quad z' \quad w'] = [x'/w' \quad y'/w' \quad z'/w' \quad 1]$$

$$\overleftrightarrow{M} = \begin{bmatrix} S_x*R_{11} & S_x*R_{12} & S_x*R_{13} & 0 \\ S_y*R_{21} & S_y*R_{22} & S_y*R_{23} & 0 \\ S_z*R_{31} & S_z*R_{32} & S_z*R_{33} & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

- 정점의 평행 이동(Translation)
  - 정점의 위치를 상대적으로 이동
  - V′ = V(x,y,z)+T(x,y,z)
    = (Vx+Tx, Vy+Ty, Vz+Tz)

- 행렬을 이용한 변환

$$\begin{bmatrix} V_x & V_y & V_z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix} = \begin{bmatrix} V_x + T_x & V_y + T_y & V_z + T_z & 1 \end{bmatrix}$$

- 이동 행렬

$$T(t) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

$$T^{-1}(t) = T(-t) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -t_x & -t_y & -t_z & 1 \end{bmatrix}$$

- 크기변환 (Scaling)
  - 정점의 위치에 scalr 배을 적용한 것
  - V′ = S(x,y,z) ⊗ V(x,y,z)
    = (Vx*Sx, Vy*Sy, Vz*Sz)

- 행렬을 이용한 변환

$$\begin{bmatrix} V_x & V_y & V_z & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} V_x * S_x & V_y * S_y & V_z * S_z & 1 \end{bmatrix}$$

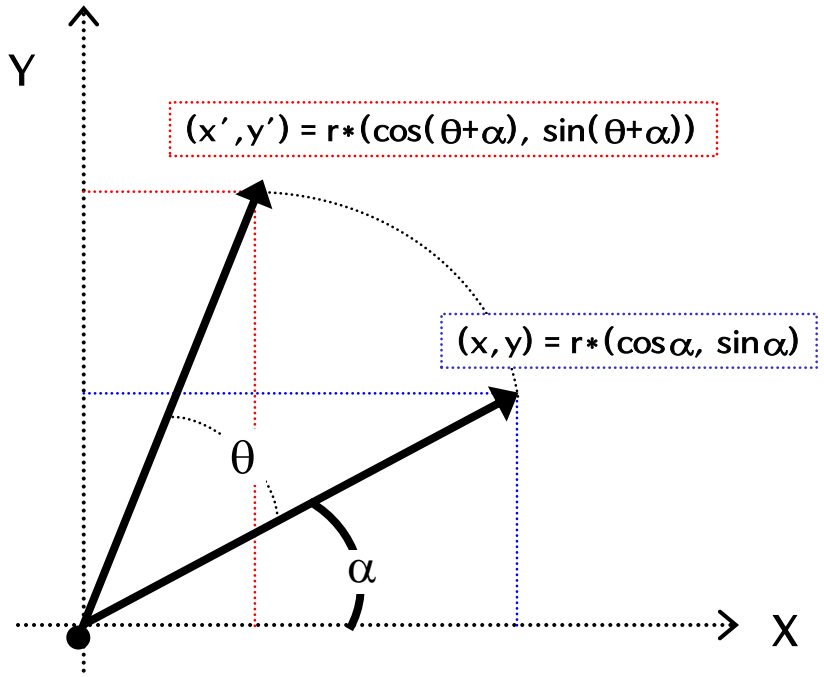
- 크기 변환 행렬 (Scaling Matrix)

$$S(s) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S^{-1}(s) = S(\frac{1}{s_x} \quad \frac{1}{s_y} \quad \frac{1}{s_z}) = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 & 0 \\ 0 & 0 & \frac{1}{s_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 정점의 회전(Rotation)
  - 좌표 축에 대한 회전 ➔ 구하기 쉬움
  - 임의의 축에 대한 회전 ➔ Quaternion 사용

- 2차원 회전 변환




$$x' = r*\cos(\theta+\alpha)$$
$$= r*\cos\theta\cos\alpha - r*\sin\theta\sin\alpha$$
$$= \cos\theta*(r*\cos\alpha) - \sin\theta*(r*\sin\alpha)$$
$$= \cos\theta*x - \sin\theta*y$$

$$y' = r*\sin(\theta+\alpha) = r*\sin\theta\cos\alpha + r*\cos\theta\sin\alpha$$
$$= \sin\theta*r*\cos\alpha + \cos\theta*r*\sin\alpha$$
$$= \sin\theta*x + \cos\theta*y$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} r*\cos(\theta+\alpha) \\ r*\sin(\theta+\alpha) \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

$$\begin{pmatrix} x' & y' \end{pmatrix} = \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

- X, Y, Z축에 대한 회전 행렬

$$M_{RotX}(\theta)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & \sin\theta & 0\\ 0 & -\sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

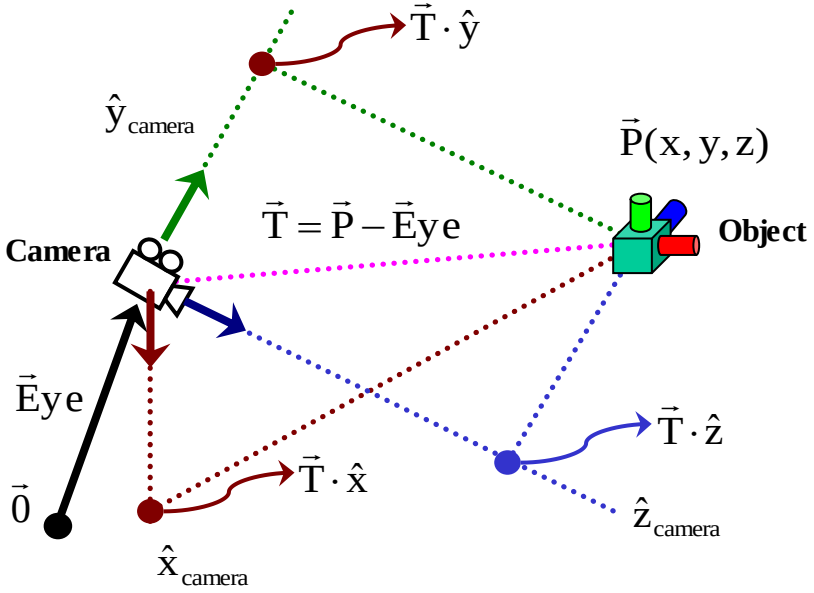
$$M_{RotY}(\theta)=\begin{bmatrix}\cos\theta & 0 & -\sin\theta & 0\\ 0 & 1 & 0 & 0\\ \sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$M_{RotZ}(\theta)=\begin{bmatrix}\cos\theta & \sin\theta & 0 & 0\\ -\sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

- 회전 행렬의 역 행렬
  - 각도에 반대 각도(-θ)를 적용한 행렬
  - 전치. Transpose(R) = Inverse(R) ← Orthogonal Matrix 특징

- 임의의 축에 대한 회전
  - 행렬의 순서가 바뀌면 두 행렬의 곱 M*N≠N*M이므로 정점의 변환을 회전행렬에서 적용할 때 회전에 대한 행렬을 연속적으로 적용할 때 문제가 발생 →짐벌락(Gimbal Lock)
  - 각도에 대해서 누적으로 하고 임의의 축에 대한 회전으로 변경 →Quaternion 사용

- Affine변환
  - $T'_i = \Sigma M_{ij} T_j + V_i$
  - 3D 프로그램의 정점의 변환은 전부 Affine 변환
  - 아핀 변환은 이전의 특성을 그대로 유지
    - 평행선은 변환 후 평행선에 대응, 유한한 점은 이전의 유한 점에 대응
  - 이동, 회전, 크기, 대칭, 밀림(Shear) 변환은 아핀 변환의 한 종류
  - 단순한 이동, 회전, 대칭 변환의 경우 각도, 길이, 평행관계가 그대로 유지
    ➔ 선분의 길이, 각도는 변환 후에도 유지



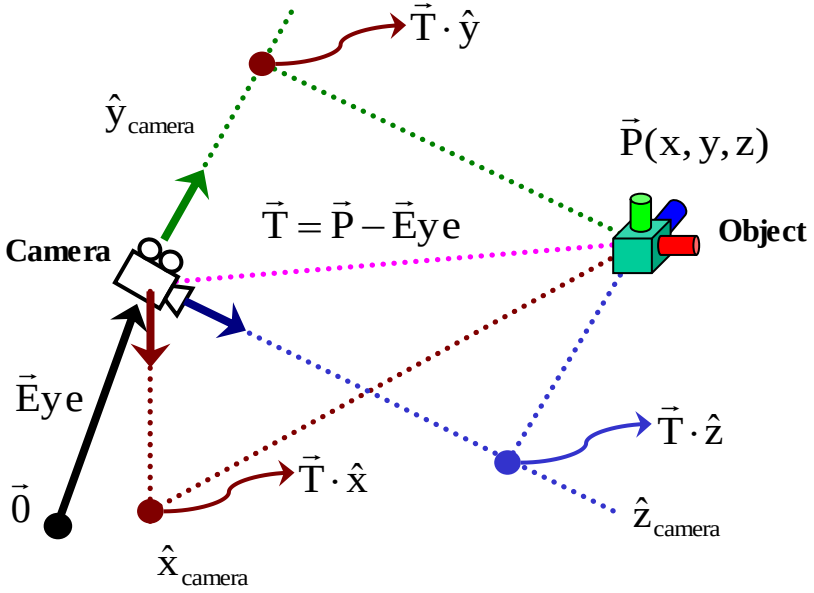

$$(x',y',z',1)=(x,y,z,1)\begin{pmatrix}\hat{x}_x & \hat{y}_x & \hat{z}_x & 0\\ \hat{x}_y & \hat{y}_y & \hat{z}_y & 0\\ \hat{x}_z & \hat{y}_z & \hat{z}_z & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

$$+(x,y,z,1)\begin{pmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ -\vec{Eye}\cdot\hat{x} & -\vec{Eye}\cdot\hat{y} & -\vec{Eye}\cdot\hat{z} & 1\end{pmatrix}$$

$$(x',y',z',1)=(x,y,z,1)\begin{pmatrix}\hat{x}_x & \hat{y}_x & \hat{z}_x & 0\\ \hat{x}_y & \hat{y}_y & \hat{z}_y & 0\\ \hat{x}_z & \hat{y}_z & \hat{z}_z & 0\\ -\vec{Eye}\cdot\hat{x} & -\vec{Eye}\cdot\hat{y} & -\vec{Eye}\cdot\hat{z} & 1\end{pmatrix}$$

$$M_{view}=\begin{pmatrix}\hat{x}_x & \hat{y}_x & \hat{z}_x & 0\\ \hat{x}_y & \hat{y}_y & \hat{z}_y & 0\\ \hat{x}_z & \hat{y}_z & \hat{z}_z & 0\\ -\vec{Eye}\cdot\hat{x} & -\vec{Eye}\cdot\hat{y} & -\vec{Eye}\cdot\hat{z} & 1\end{pmatrix}$$

$$ViewMatrix^{-1}=\begin{pmatrix}\hat{x}_x & \hat{x}_y & \hat{x}_z & 0\\ \hat{y}_x & \hat{y}_y & \hat{y}_z & 0\\ \hat{z}_x & \hat{z}_y & \hat{z}_z & 0\\ \vec{Eye}_x & \vec{Eye}_y & \vec{Eye}_z & 1\end{pmatrix}$$




$$(x',y',z',1)=(x,y,z,1)\begin{pmatrix}\hat{x}_x & \hat{y}_x & \hat{z}_x & 0\\ \hat{x}_y & \hat{y}_y & \hat{z}_y & 0\\ \hat{x}_z & \hat{y}_z & \hat{z}_z & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

$$+(x,y,z,1)\begin{pmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ -\vec{Eye}\cdot\hat{x} & -\vec{Eye}\cdot\hat{y} & -\vec{Eye}\cdot\hat{z} & 1\end{pmatrix}$$

$$(x',y',z',1)=(x,y,z,1)\begin{pmatrix}\hat{x}_x & \hat{y}_x & \hat{z}_x & 0\\ \hat{x}_y & \hat{y}_y & \hat{z}_y & 0\\ \hat{x}_z & \hat{y}_z & \hat{z}_z & 0\\ -\vec{Eye}\cdot\hat{x} & -\vec{Eye}\cdot\hat{y} & -\vec{Eye}\cdot\hat{z} & 1\end{pmatrix}$$

$$M_{view}=\begin{pmatrix}\hat{x}_x & \hat{y}_x & \hat{z}_x & 0\\ \hat{x}_y & \hat{y}_y & \hat{z}_y & 0\\ \hat{x}_z & \hat{y}_z & \hat{z}_z & 0\\ -\vec{Eye}\cdot\hat{x} & -\vec{Eye}\cdot\hat{y} & -\vec{Eye}\cdot\hat{z} & 1\end{pmatrix}$$

$$ViewMatrix^{-1}=\begin{pmatrix}\hat{x}_x & \hat{x}_y & \hat{x}_z & 0\\ \hat{y}_x & \hat{y}_y & \hat{y}_z & 0\\ \hat{z}_x & \hat{z}_y & \hat{z}_z & 0\\ \vec{Eye}_x & \vec{Eye}_y & \vec{Eye}_z & 1\end{pmatrix}$$

- 뷰 행렬 계산

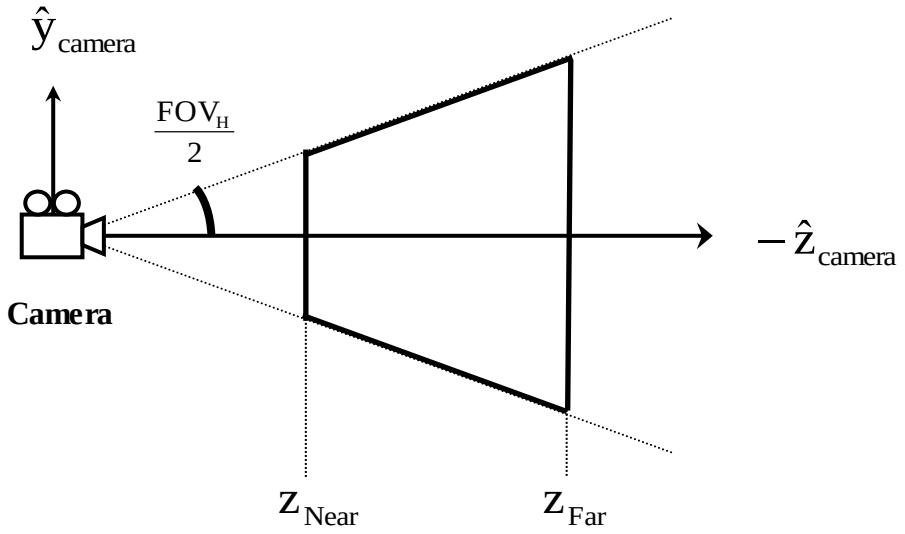
- 카메라의 X, Y, Z축을 구한다.
  Zaxis = 카메라가 보고 있는 지점 위치(Look) – 카메라의 위치(Eye);
  Zaxis = normalize(Zaxis);
  Xaxis = Cross(Up, Zaxis);
  Xaxis = normalize(Xaxis);
  Yaxis = Cross( Zaxis, Xaxis);

- 뷰 행렬의 _41, _42, _43 값을 정한다.
  _41 = -dot(Xaxis, eye)
  _42 = -dot(Yaxis, eye)
  _43 = -dot(Zaxis, eye)

- 뷰 행렬을 설정한다.

| | | | |
|---|---|---|---|
| Xaxis.x | Yaxis.x | Zaxis.x | 0 |
| Xaxis.y | Yaxis.y | Zaxis.y | 0 |
| Xaxis.z | Yaxis.z | Zaxis.z | 0 |
| -dot(Xaxis, eye) | -dot(Yaxis, eye) | -dot(Zaxis, eye) | 1 |


카메라의 **FOV**




$$\begin{pmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & Q & -1 \\ 0 & 0 & M & 0 \end{pmatrix}$$

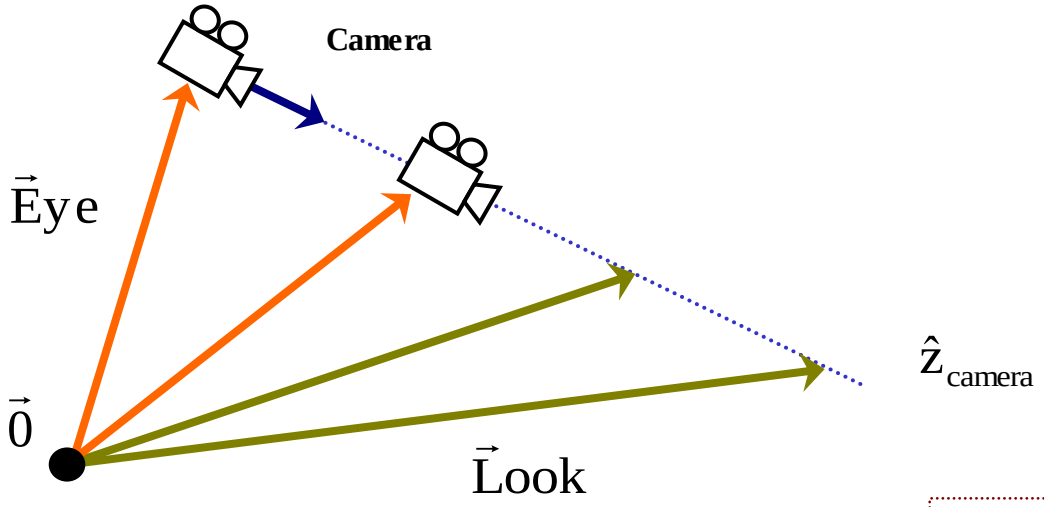
$$, \; h = \cot(\frac{FOV_H}{2}), \; w = \frac{h}{Aspect}$$

$$, \; Q = -\frac{z_{Far} + z_{Near}}{z_{Far} - z_{Near}}, \; M = -\frac{2 * z_{Far} * z_{Near}}{z_{Far} - z_{Near}}$$

- 투영 행렬 계산

```
var Near_Plane;
var Far_Plane;
var FOV; //Field of View
var Aspect = ScreenWidth/ScreenHeight;
var h, w, Q;
h = cot(FOV/2.f);
w = h/ Aspect;
Q = -(Far_Plane + Near Plane) /( Far_Plane - Near_Plane);
M = -2 * Far_Plane * Near Plane /( Far_Plane - Near_Plane);

um_prj = [   w, 0,   0,    0,
             0, h,   0,    0,
             0, 0,   Q,   -1,
             0, 0,   M,   0);
```

```
w = 2/(Right-Left);
h = 2/(Top-Bottom);
Q = 2/(Near-Far);
41= (Left+Right)/(Left-Right);
42= (Bottom+Top)/(Bottom-Top);
43= (Near+Far)/(Near-Far);

um_prj = [   w, 0,   0,    0,
             0, h,   0,    0,
             0, 0,   Q,    0,
             0, 0,   M,   1);
```




카메라의 위치: Eye Vector
카메라가 보고 있는 지점: Look Vector
카메라의 이동 스피드: Speed

```
var3 vcZ = Look – Eye;
Normalize(vcZ);
Eye  += vcZ * Speed;
Look += vcZ * Speed;
```

- Scissor(가위) 판정을 활성화 해당 영역만 clear
- Viewport, Scissor 영역을 동일하게 설정하면 화면 분할 효과

```
gl.enable(gl.SCISSOR_TEST);
gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
gl.scissor (0, 0, gl.viewportWidth, gl.viewportHeight);
```

- 제시한 sample 코드를 활용해서 카메라를 구성해보자.
  - 카메라 좌우 회전: left, right Key
  - 카메라 상하 회전: up, down Key

- 월드 행렬을 사용해서 이전에 만든 로봇을 키보드의 동작에 따라 움직여 보자.

# 5. Lighting 1

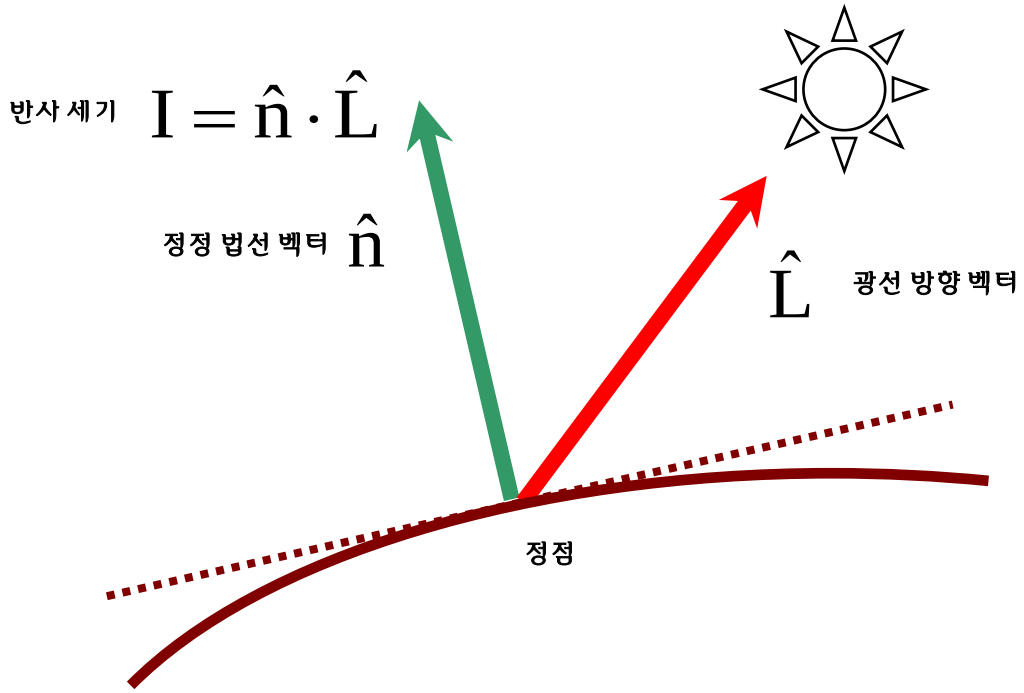
- Lambert(램버트) 확산
- 반사의 세기 = 정점의 위치에서 바라보는 광원의 방향과 정점의 법선 벡터의 내적
- 반사에 대한 색상의 밝기(Intensity) = Dot(N, L)

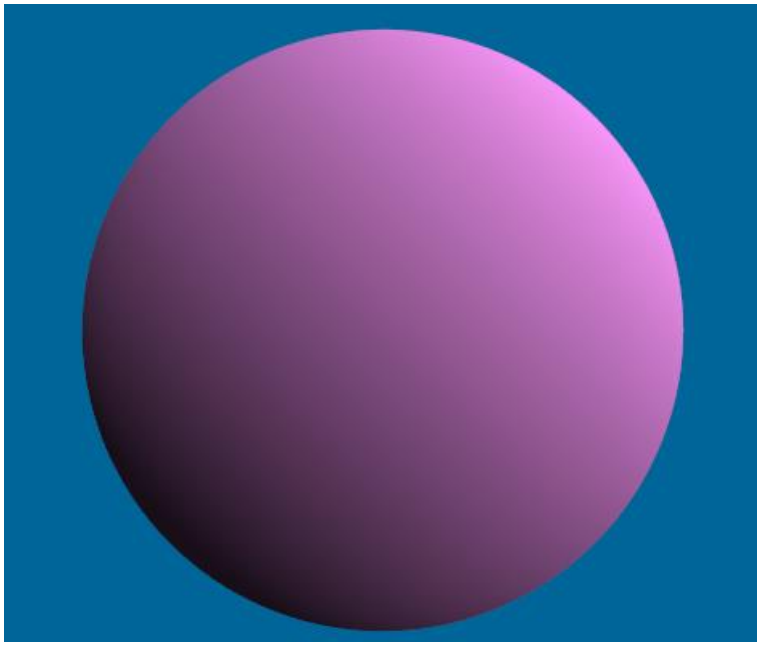
- 여러 조명이 있을 경우 내적 값 합산

$$I = \sum_i \hat{n} \cdot \hat{L}_i \times (\text{Vertex Diffuse} \otimes L_i\text{'s Diffuse Color})$$

$$A \otimes B = (A_1, A_2, A_3, A_4) \otimes (B_1, B_2, B_3, B_4) = (A_1 * B_1,\ A_2 * B_2,\ A_3 * B_3,\ A_4 * B_4)$$

- 램버트 확산
  - 반사 세기 = Dot(정점의 법선(N), 빛의 방향(L))
  - 내적의 범위가 [-1, 1] 이므로 반사 세기 범위를 [0,1]로 변경
  - 반사 세기 = (1+ Dot( N, L))/2



- Phong, Blinn-Phong 반사
- 반사의 세기 = 정점의 위치에서 광원의 반사 방향, 정점에서 카메라 위치 시선 방향 에 대한 방향의 내적 결과에 Highlight 승수

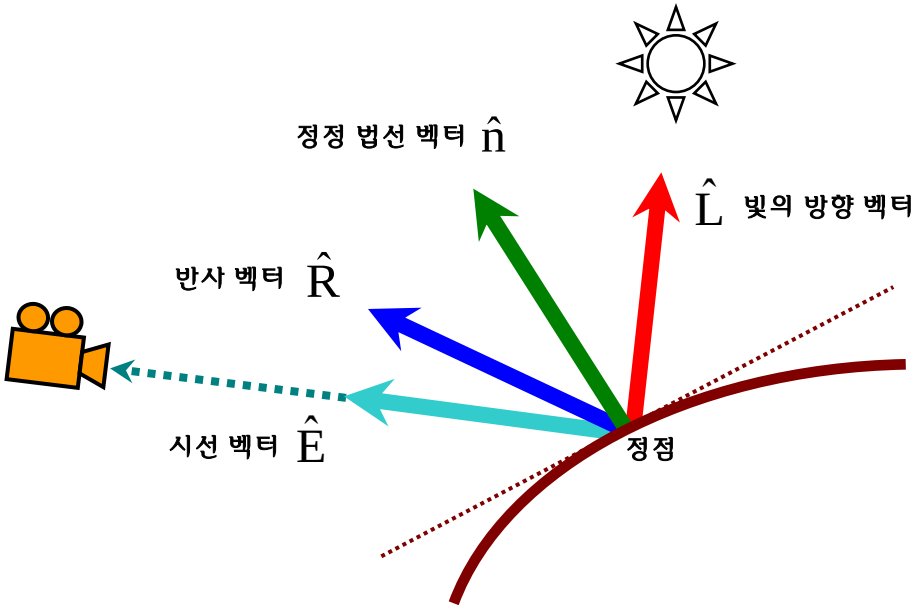
- 공식: Intensity = Dot(E, L) : E 정점에서 시선 방향

- 여러 조명이 있을 경우 각각의 Intensity를 더함.

$$I = \sum_{i} \left( \hat{E} \cdot \hat{R}_i \right)^{Power} \times (Specular \otimes L_{Specular-i} Color)$$

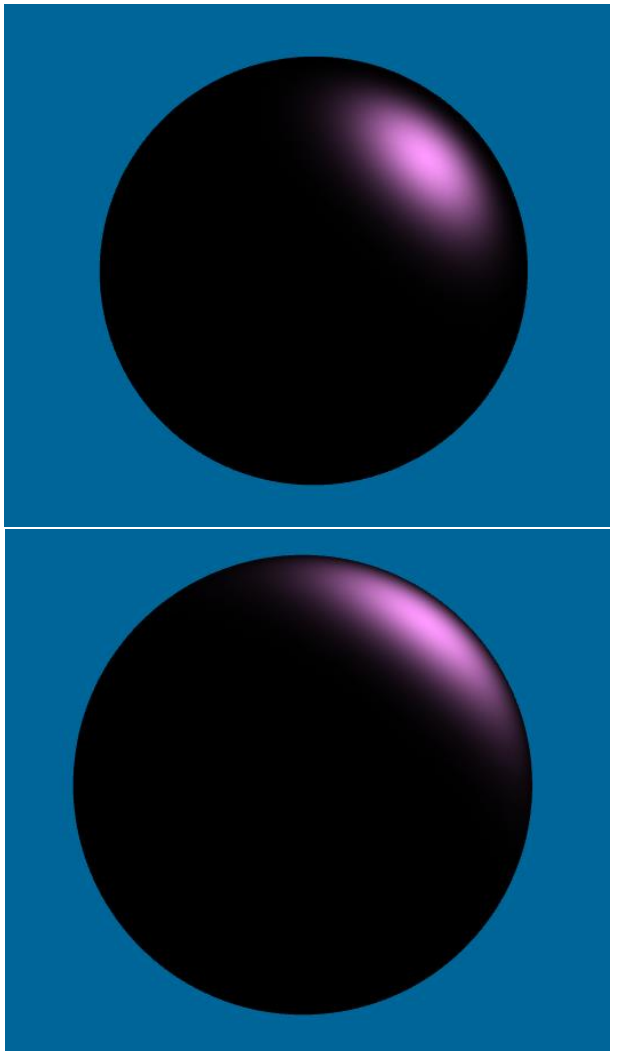
# 5.2 Phong Reflection

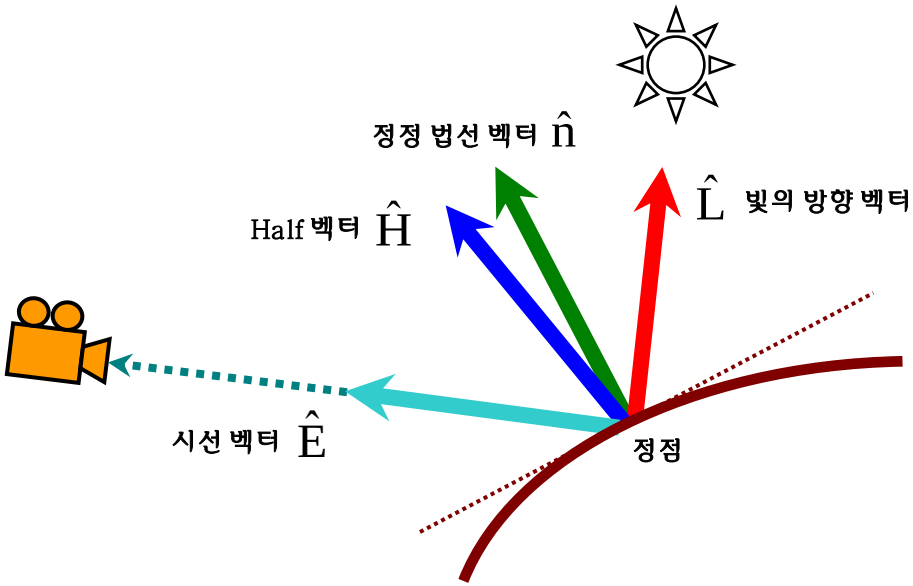



$$\hat{R} = 2\left(\hat{L} \cdot \hat{n}\right)\hat{n} - \hat{L}$$

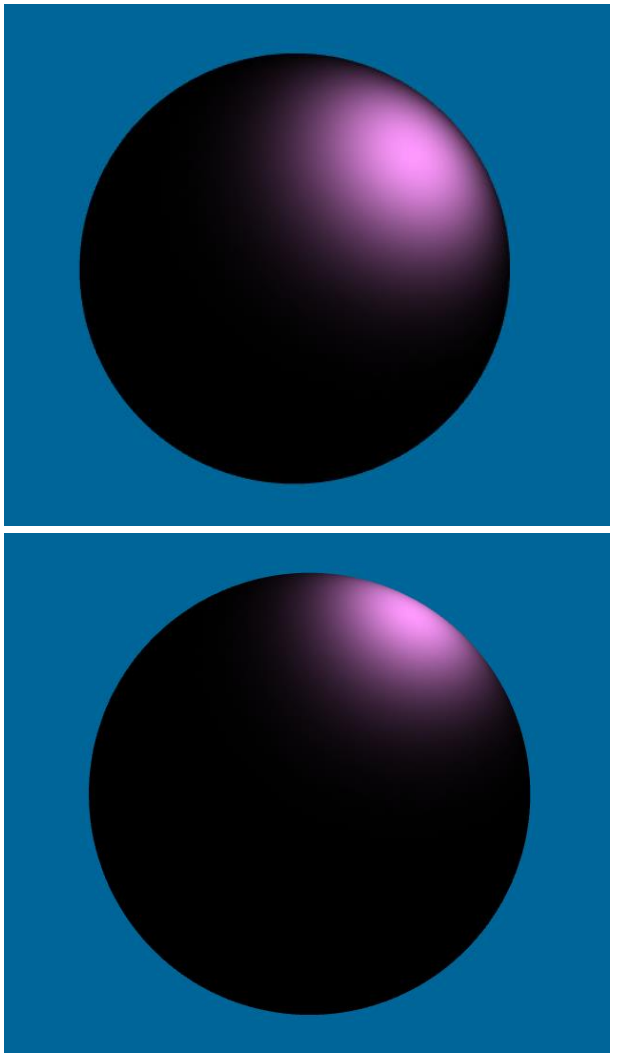
$$\text{Phong 반사} = \left(\hat{E} \cdot \hat{R}\right)^{\text{Power}}$$



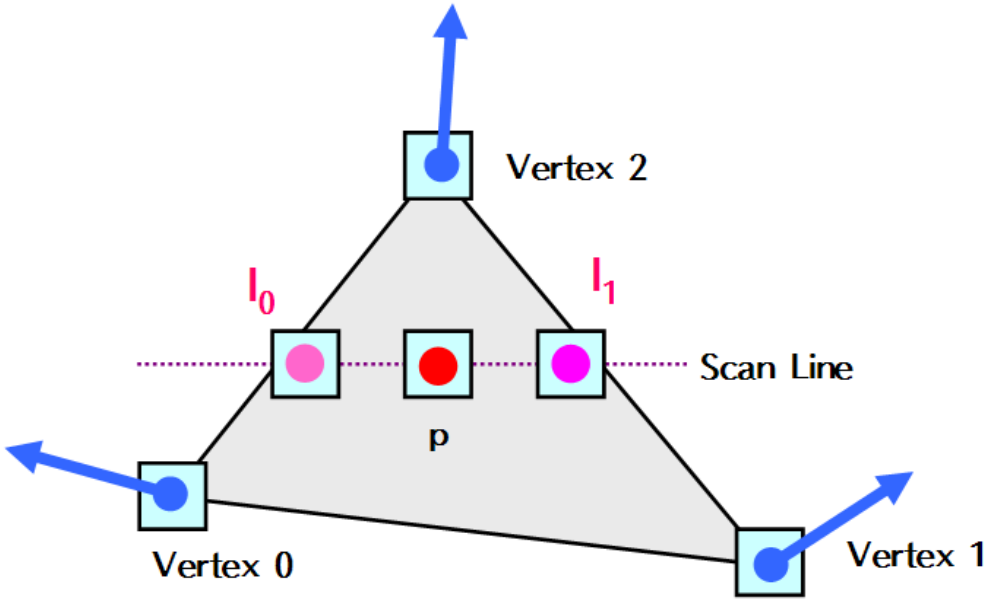


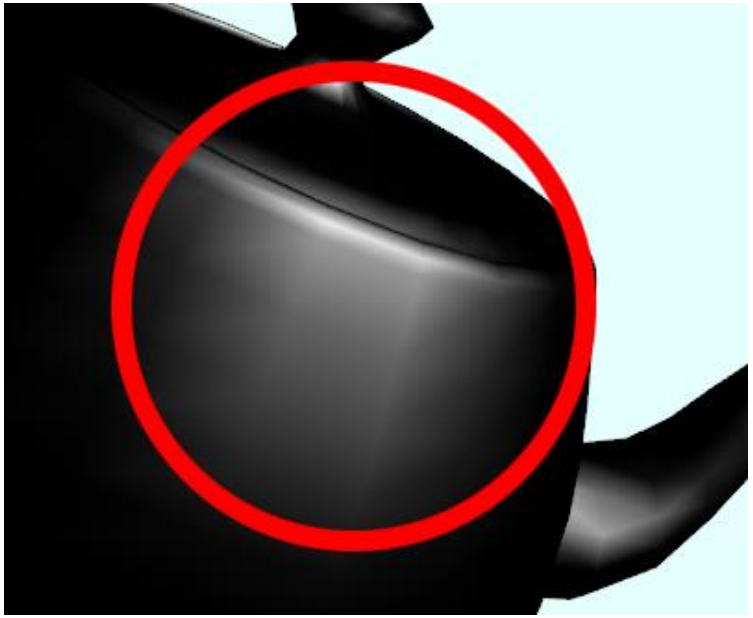

$$\hat{H} = \frac{\hat{E} + \hat{L}}{|\hat{E} + \hat{L}|}$$

$$\text{Blinn-Phong 반사} = \left(\hat{n} \cdot \hat{H}\right)^{\text{Power}}$$



- Vertex Process에서 Lighting 계산
- 레스터라이저는 선형 보간으로 픽셀을 생성
- 정점의 숫자가 적으면 사이의 보간에 의한 조명 효과 감소

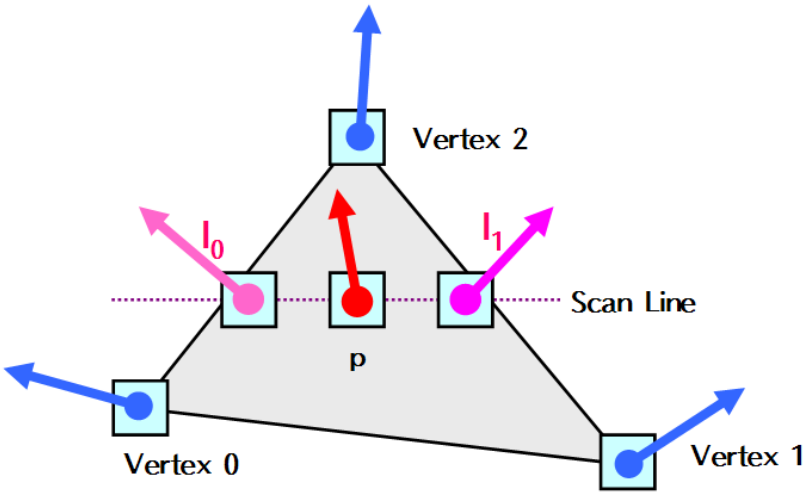


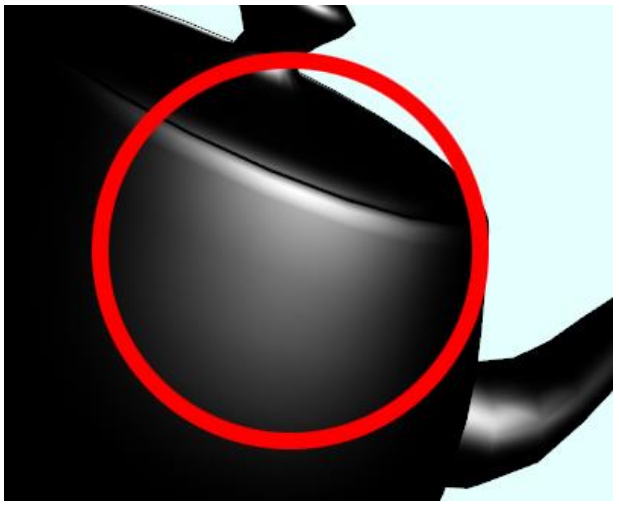



- Fragment Process에서 Lighting 계산
- 픽셀 단위에서 조명을 계산하기 때문에 정점의 숫자가 적어도 조명 효과를 만듦

- Vertex Shader에서 전달하는 법선 벡터, 시선 벡터 등을 사용하며 이들은 보간을 한 결과이므로 반드시 정규화(normalize)해서 사용

- 정규화 안 하면 Vertex Process Lighting과 동일



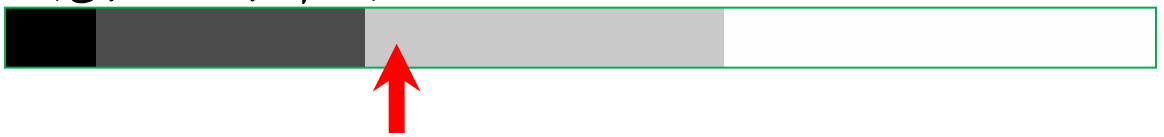



- 제시한 sample 코드를 활용해서 라이팅 3개를 추가해보자.

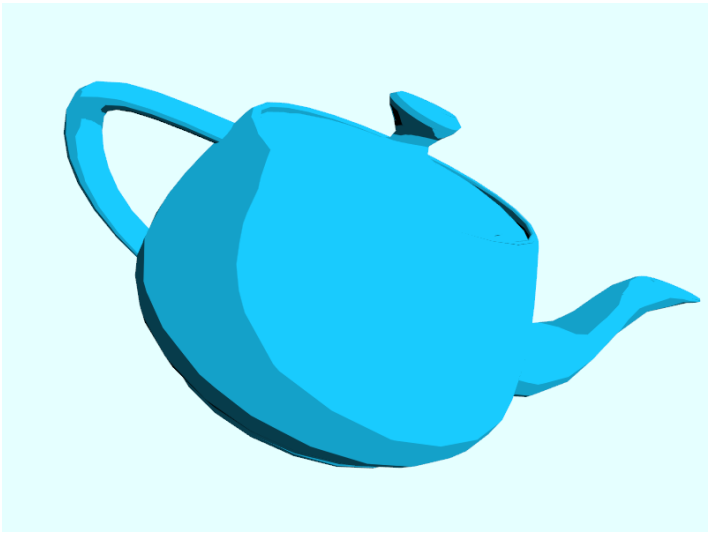
# 6. Lighting 2

- Lambert 반사 결과를 Discrete 로 표현
  - 반사의 세기를 [0,1]로 만들고 이들 텍스처 좌표로 활용
- if문을 사용한 방법
  - 추가 리소스 필요 없음
  - 색상의 경계에서 Aliasing 현상 발생
- 카툰 텍스처를 사용한 방법
  - 필터링을 적용하면 색상의 경계가 부드럽게 표현

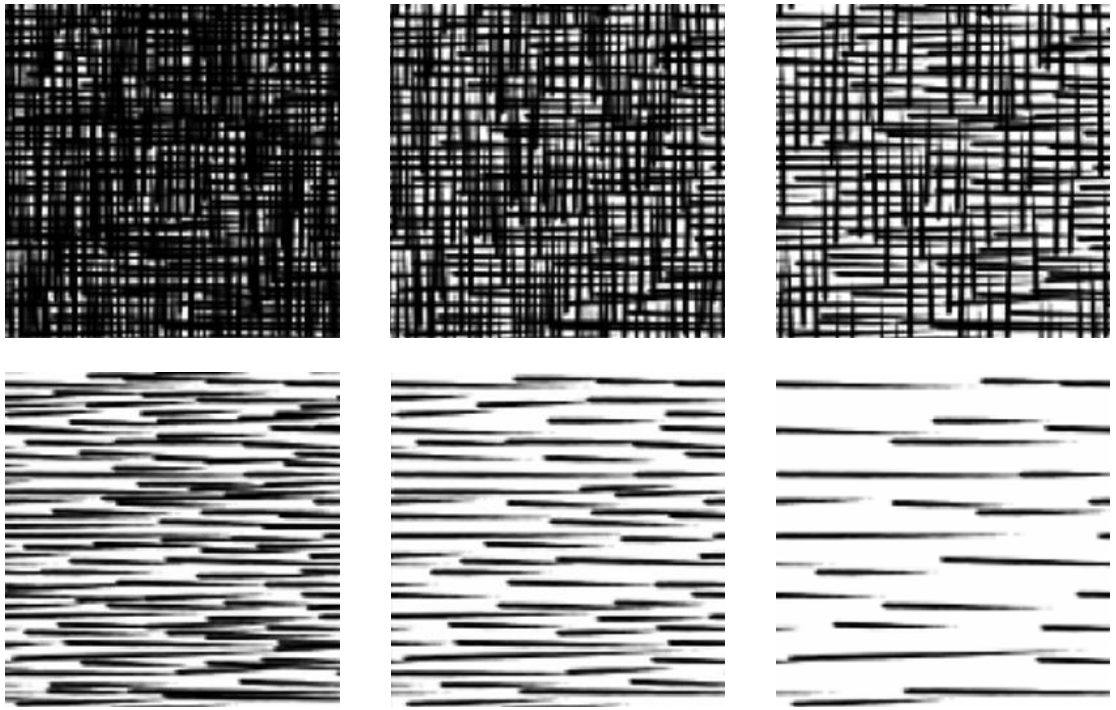
<카툰(Cartoon) 텍스처>

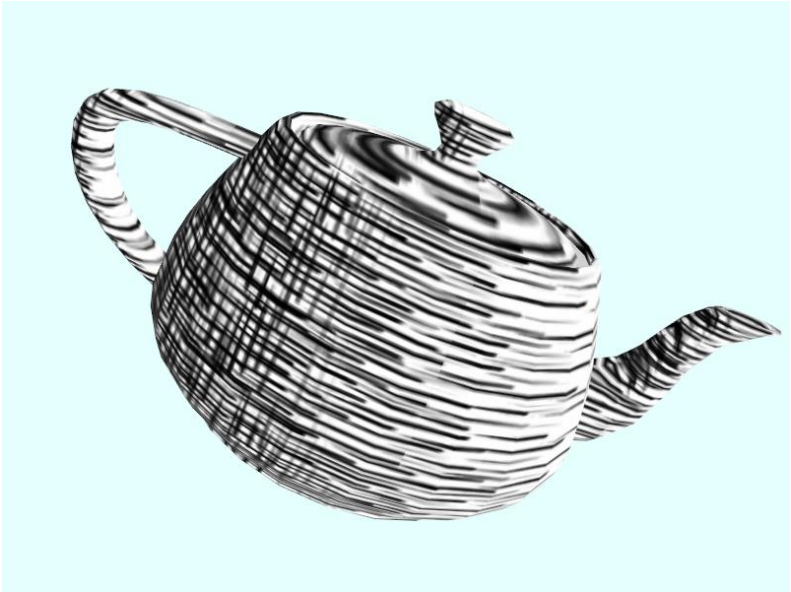


반사의 세기 = 텍스처 좌표



- Lambert 반사의 세기를 텍스처로 표현
- 반사의 세기가 적용할 텍스처의 각 단계별 비중

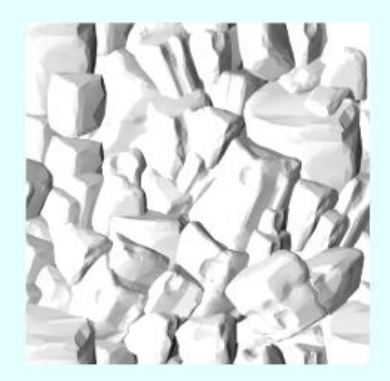




- 저 폴리곤에서 고(高) 폴리곤 조명효과 생성
  - Normal Texture: 법선 정보가 저장된 텍스처
- Fragment Shader에서 처리
  - Texture의 rgb를 법선 xyz로 변환 과정만 추가 되고 나머지 조명 계산은 동등
- rgb -> xyz
  - 색상은 [0.1] , 법선은 [-1,+1] 이므로 법선 = 색상*2.0-1.0
- Lambert 반사로 bump 밝기 설정



- Pixel rgb to xyz
  - vec3 ct_nor = texture2D(us_nor, vr_tex ).xyz * 2.0 - 1.0;
- Operation to lighting direction
  - float bump = (dot(lgt_dir, ct_nor) + 1.0) * 0.5;



- 최종 색상 = diffuse map color * bump intensity

- 인터넷에서 범프 텍스처 30개를 찾아서 적용해보자.




# 7. Fog


- 카메라의 위치와 z 방향에 따른 포그
- 거리 크기에 대한 농도 결정
  - 카메라에서 멀어지면 포그 농도 짙어짐
  - Vertex Shader 월드 변환 후에 거리 계산 및 농도 결정

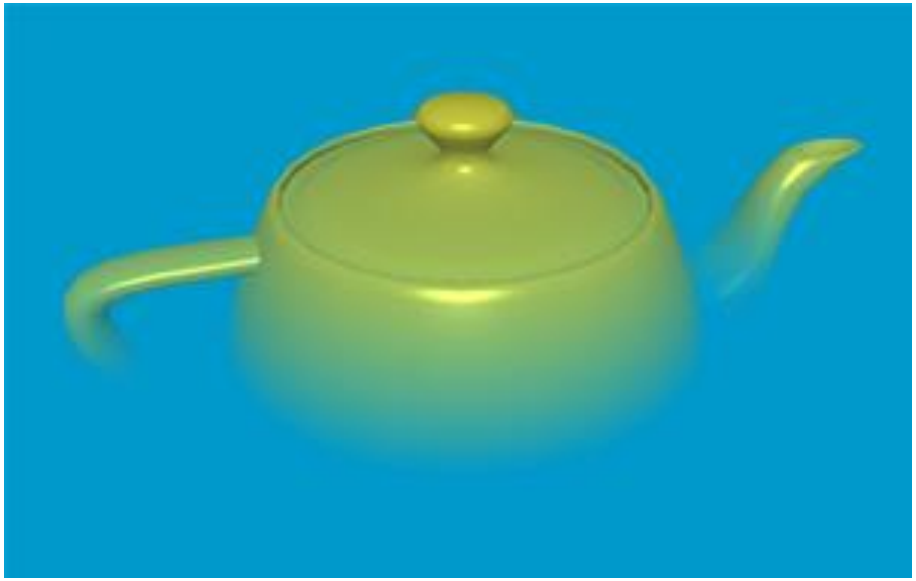
```
void main() {
        vec4 pos     = um_Wld * at_pos;

        vr_eye       = uf_cam - vec3(pos);
        float range = length(vr_eye);
        vr_fog       = range/(fog_rng.y - fog_rng.x);
        vr_fog                          = clamp(vr_fog, 0.0, 1.0);
```

- Z에 대한 농도 결정
  - 카메라 z축의 비례에 포그 농도가 결정
  - Vertex Shader 뷰 변환 후에 내적을 사용 농도 결정

```
void main() {
        vec4 pos    = um_Wld * at_pos;
        pos         = um_Viw * pos;

        vr_fog      = -pos.z/(fog_rng.y - fog_rng.x);       // fog has dependency to view transform
        vr_fog      = clamp(vr_fog, 0.0, 1.0);
```


- Geometry 위치로 포그 농도 결정
  - Vertex Shader 월드 변환 후 농도 결정

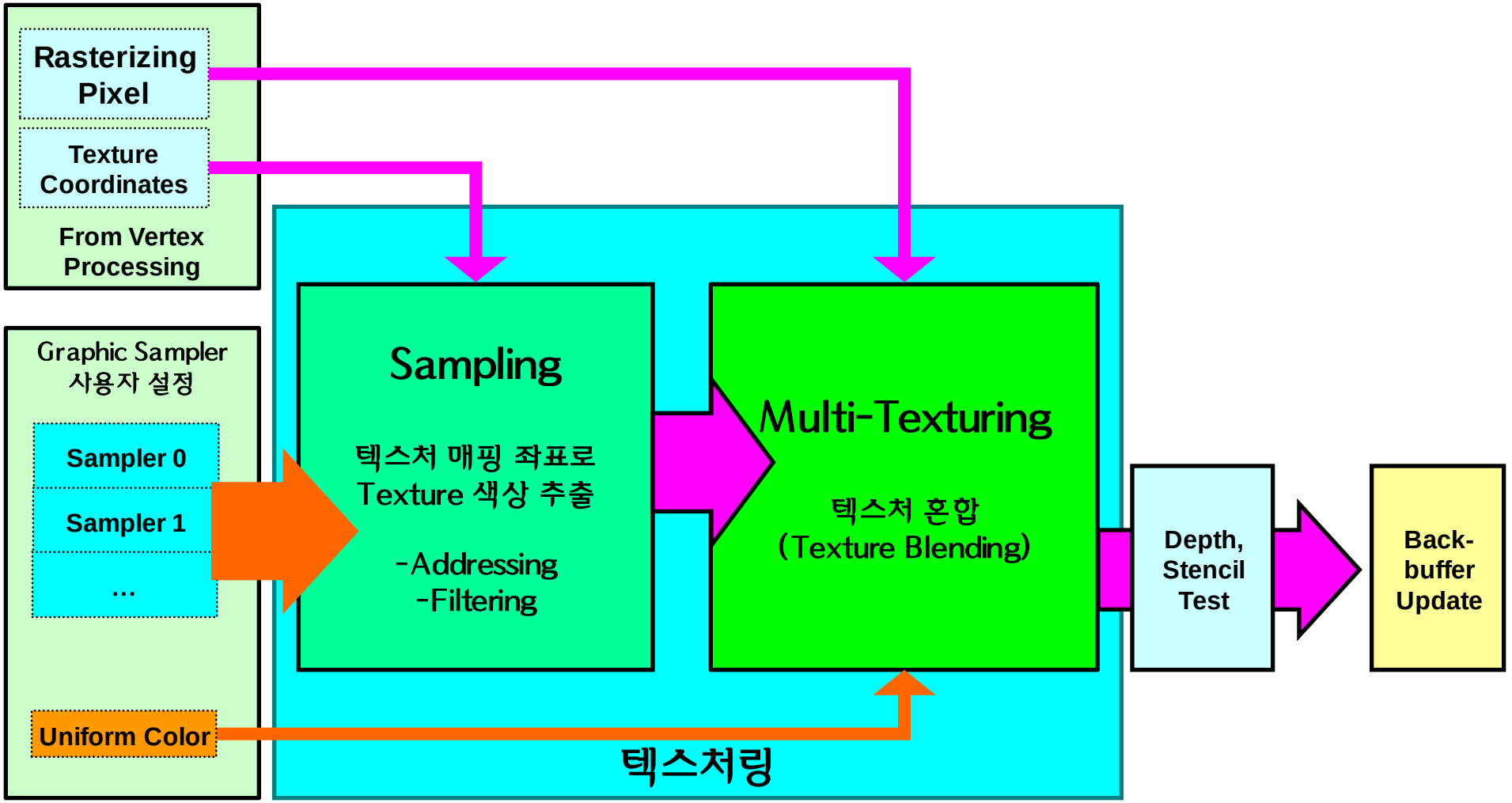
```
void main() {
        vec4 pos     = um_Wld * at_pos;

        vr_fog       = 1.0 - pos.y/fog_rng;
        vr_fog       = clamp(vr_fog, 0.0, 1.0);
```



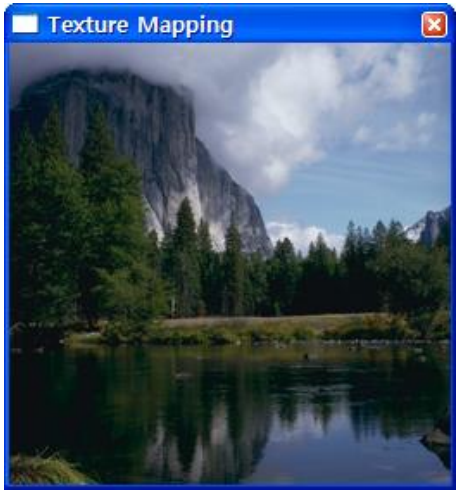
- Vertex Shader에서 농도(weight) 결정
- Fragment Shader에서 최종 색상 결정
  - 최종 색상 = fragment processing color*(1.0 - 포그 농도) + 포그 색상 * 포그 농도
- 포그 색상은 clear 색상과 일치해야 자연스러움


# 8. Texturing

- Texturing
  - 폴리곤(Polygon) 만으로도 가상 세계를 표현
    - Detail이 높아질 수록 컴퓨터의 자원이 많이 필요
  - 텍스처링
    - 폴리곤을 줄이고 오브젝트의 질감을 높이기 위해 이미지를 폴리곤에 적용하는 매핑(Mapping) 기술
  - 샘플링(Sampling)
    - 조건에 맞는 색상을 이미지에서 추출하는 방법
- 텍스처링 기술
  - Filtering
  - Addressing
  - Multi-Texturing



- 의미
  - 화면에 출력되는 픽셀이 확대/축소 될 때 픽셀을 보정하는 방법
- 적용 대상
  - Minification filtering : 텍스처가 확대 됐을 때 Sampling 방법
  - Magnification filtering: 텍스처가 축소 됐을 때 Sampling 방법
  - MIP map filtering
    - 카메라에서 멀리 떨어진 객체는 낮은 해상도의 텍스처에서, 가까이에 있는 객체는 높은 해상도의 텍스처에서 Sampling 방법
    - MIP map texture: 원본 텍스처에서 보다 작은 여러 해상도의 텍스처
- 종류
  - Nearest
  - Linear
  - MIP map


화면에 1:1로 출력

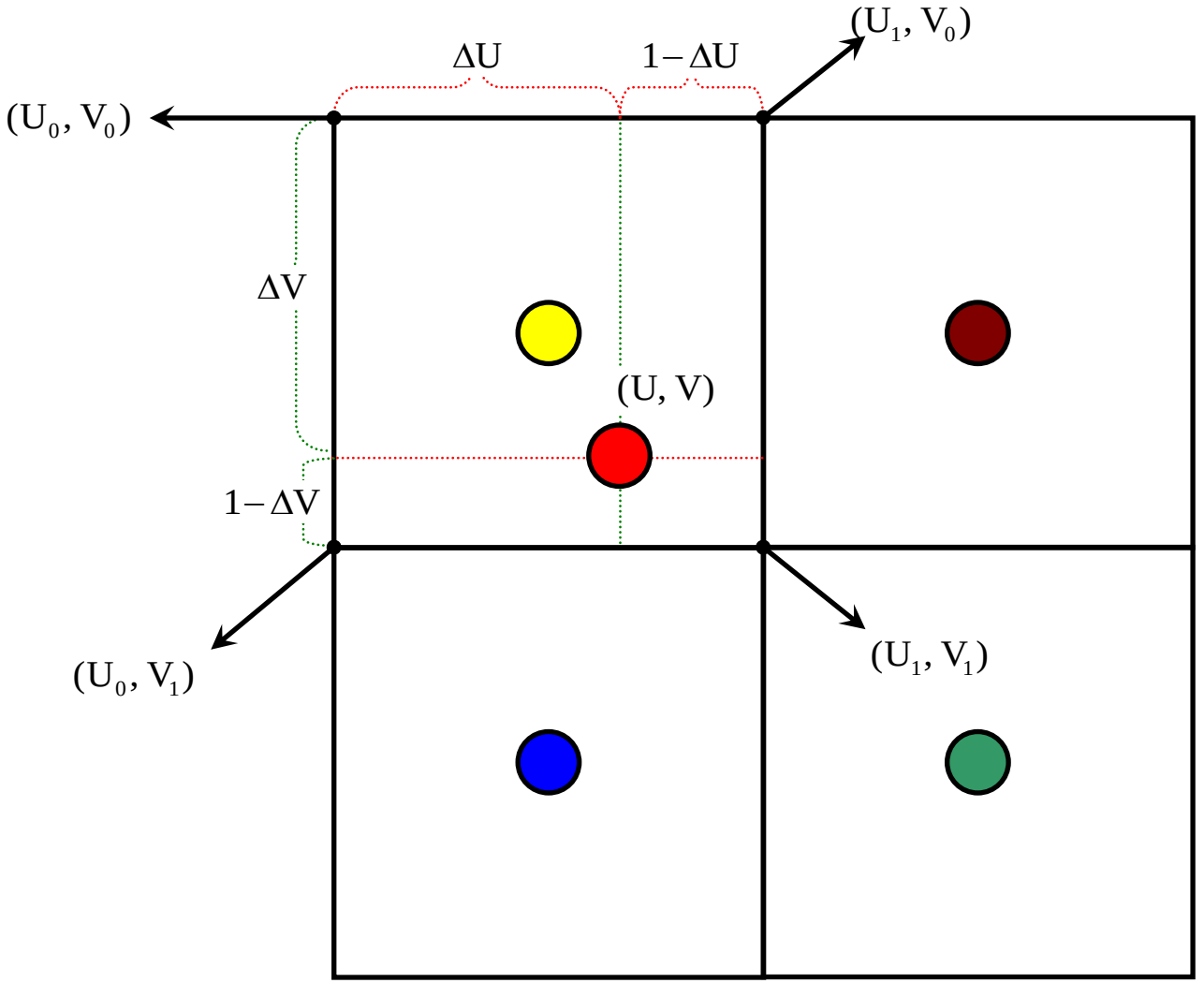

기하변환에 의한 텍스처 왜곡

MIP(multum in parvo)-map Pyramid

- Nearest (근접점 샘플링: nearest-point sampling)
  - 해당 UV좌표에서 가장 가까운 픽셀을 샘플링
  - 가장 빠름
  - 텍스처의 경계에서 문제 발생

- LINEAR (선형 필터링: bilinear interpolation filtering)
  - 해당 UV좌표에서 가장 가까운 네 픽셀을 샘플링, 가중치를 적용해서 픽셀을 결정
  - 대부분의 3D 프로그래머가 선호

- MIP 필터링
  - MIP map을 사용한 필터링




$$w_{00}(U_0, V_0) = (1-\Delta U)(1-\Delta V)$$

$$w_{10}(U_1, V_0) = (1-\Delta U)(\Delta V)$$

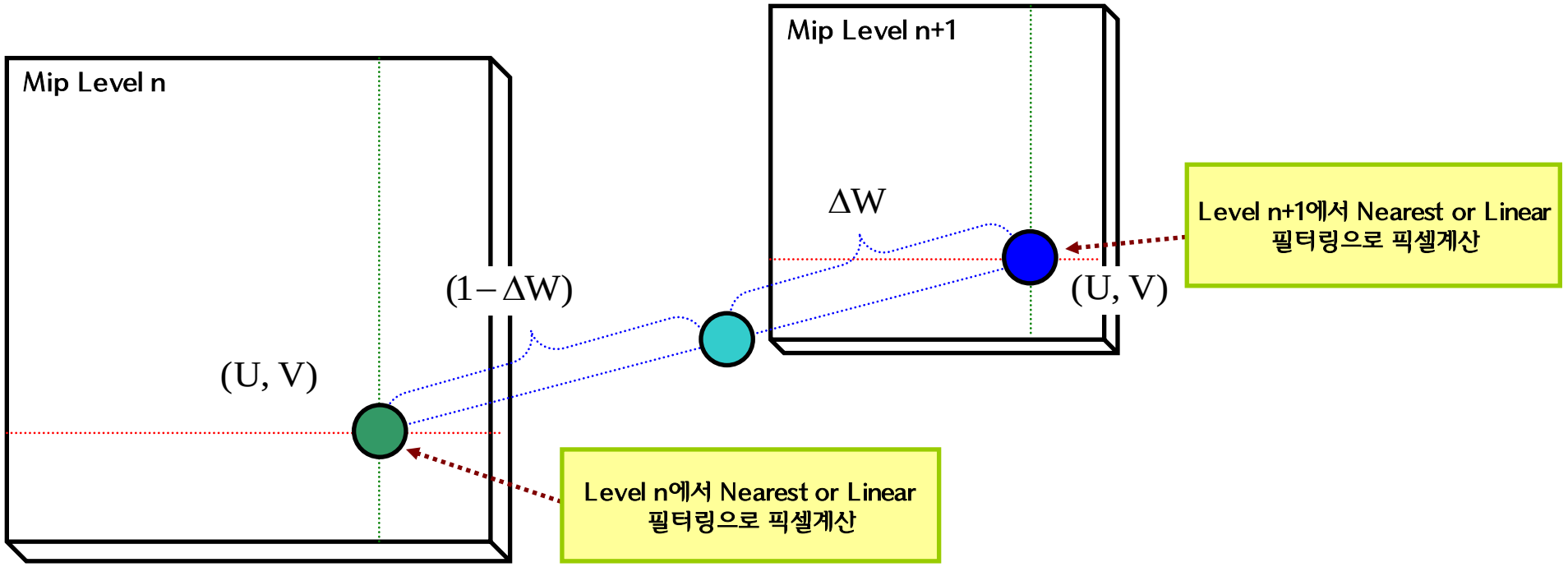
$$w_{11}(U_1, V_1) = (1-\Delta U)(1-\Delta V)$$

$$w_{01}(U_0, V_1) = (\Delta U)(1-\Delta V)$$

$$\begin{aligned} Pixel &= \sum w_i * P_i \\ &= w_{00} * Pixel(U_0, V_0) \\ &\quad + w_{10} * Pixel(U_1, V_0) \\ &\quad + w_{01} * Pixel(U_0, V_1) \\ &\quad + w_{11} * Pixel(U_1, V_1) \end{aligned}$$

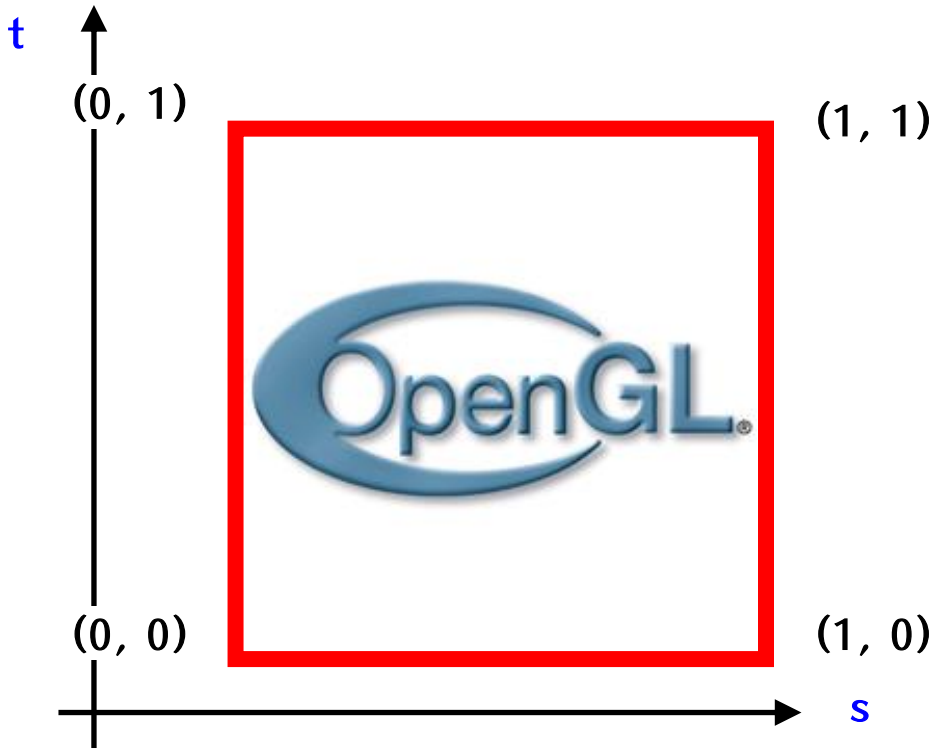
# 9.2 MIP Filtering

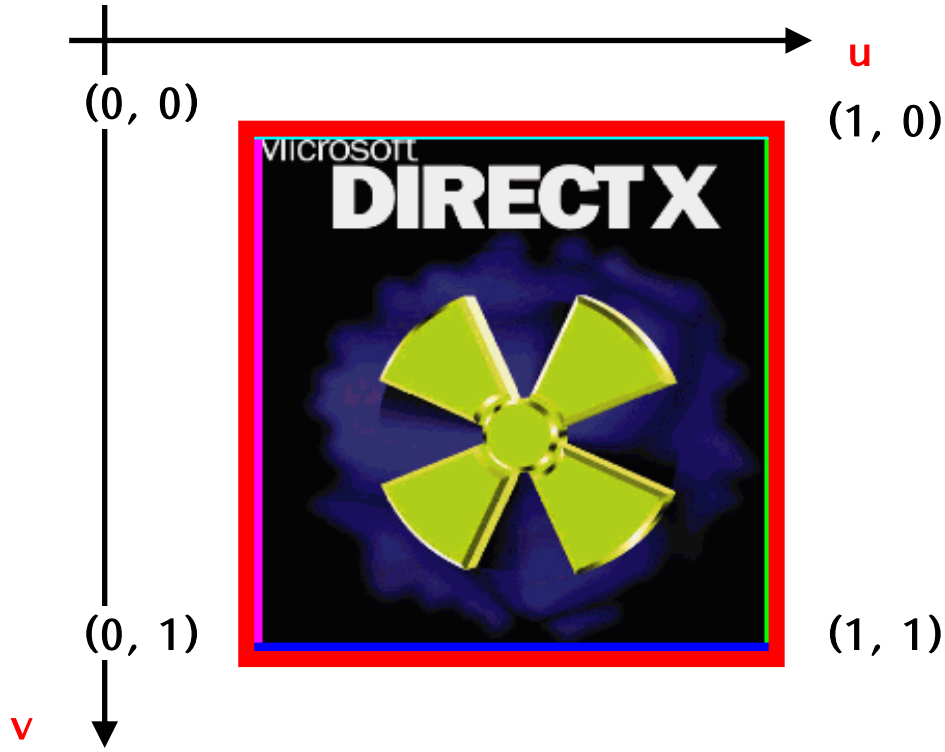





$$Pixel = (1-\Delta w) * Pixel_{level-n} + \Delta w * Pixel_{level-n+1}$$

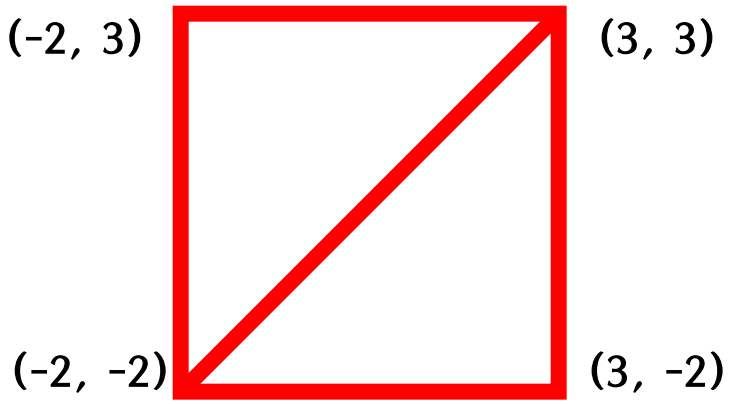
- 텍스처 좌표계
  - ST 좌표계: 수학의 오른손 좌표계 사용. 좌 하단(0,0), 우 상단 (1,1). OpenGL
  - UV 좌표계: 왼손 좌표계. 좌 상단(0,0), 우 하단(1,1). Direct3D. y 값이 화면 좌표계와 동일
- 정규화
  - 모든 텍스처 좌표는 [0,1]의 범위를 가짐(정규화)
- Addressing
  - [0,1] 범위 이외 텍스처 좌표 값에 대한 sampling 위치를 결정하는 방법
- 종류
  - Repeat
  - Mirror
  - Clamp-to Edge

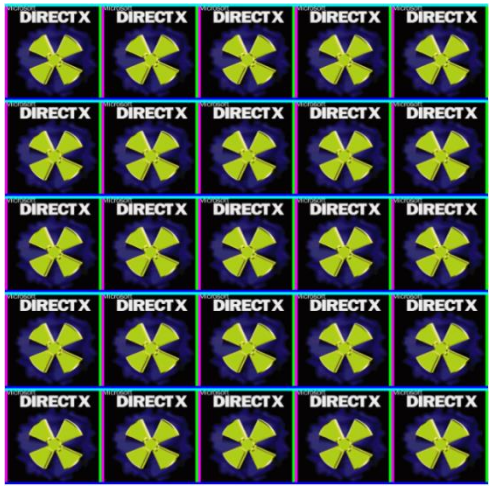



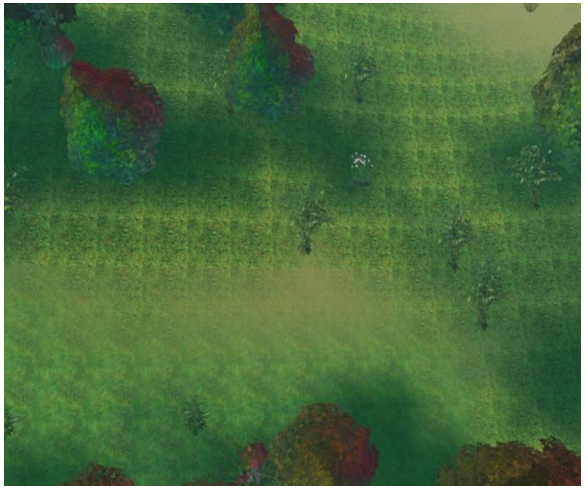



- Default Addressing
  - 1보다 크면 [0,1] 범위 안에 올 때까지 1씩 감소 시켜 결정
    Ex)2.4 → 0.4, 10.5 → 0.5
  - 0보다 작으면 [0,1] 범위 안에 올 때까지 1씩 증가 시켜 결정
    Ex) -1.0 → 0.0, -0.4 → 0.6
  - 실외 지형에서 타일링(Tiling)으로 자주 사용

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
```



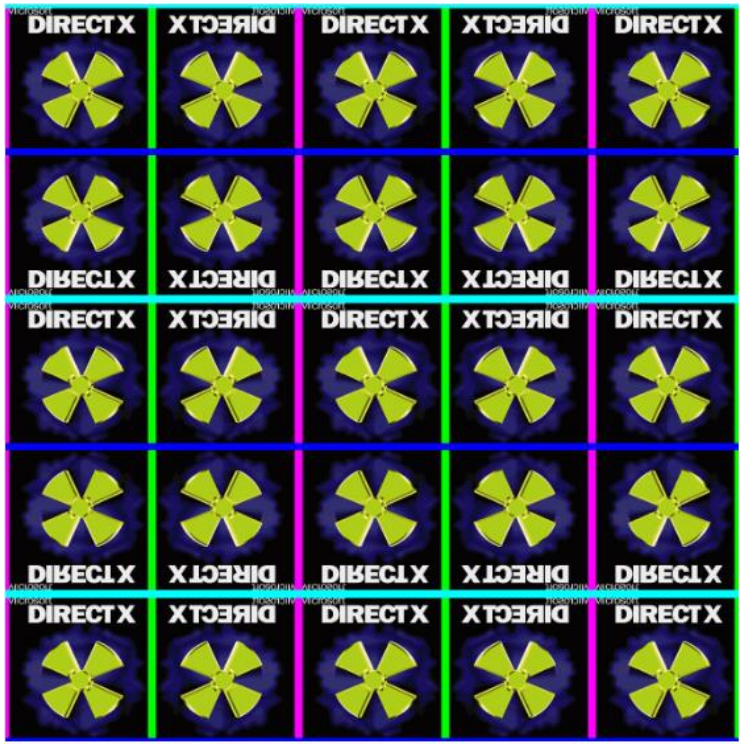



# 9.3 Addressing: Mirrored Repeat

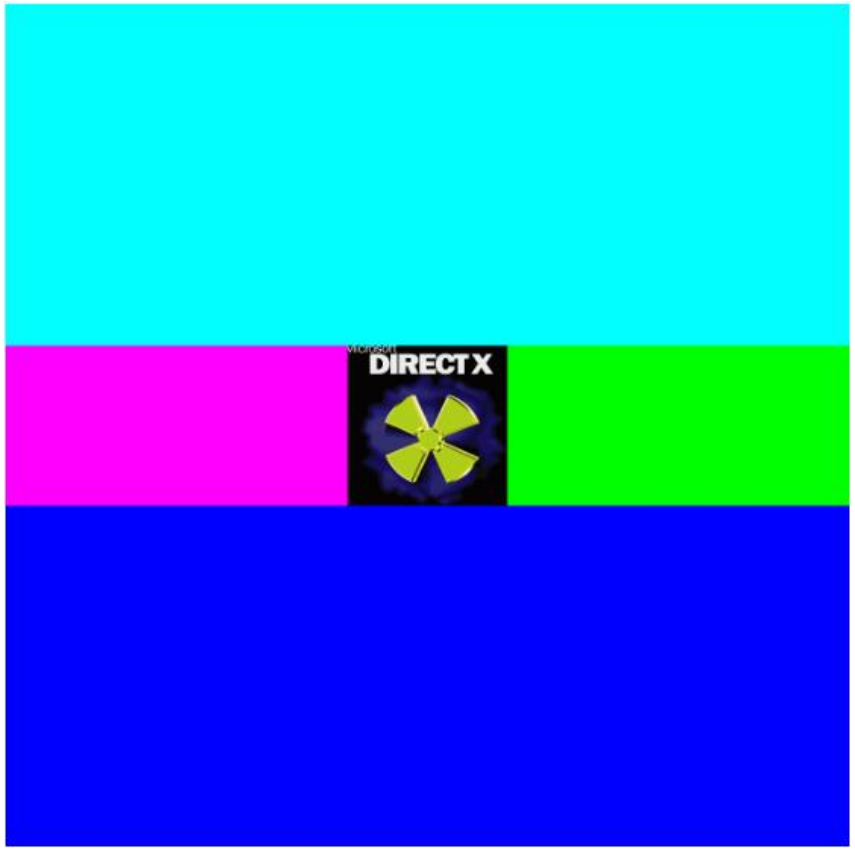
- Wrap과 비슷하나 거울처럼 텍스처를 적용
- 1.1→ 0.9, 2.0→0.0
- 경계가 자연스럽게 이어지는 텍스처 매핑에서 주로 사용

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.MIRRORED_REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.MIRRORED_REPEAT);
```



- [0,1]범위 밖의 색상은 마지막 픽셀 값으로 결정
- Clamp모드는 그림자 맵과 같이 불 필요하게 픽셀이 반복 되는 것을 막는데 주로 사용

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
```

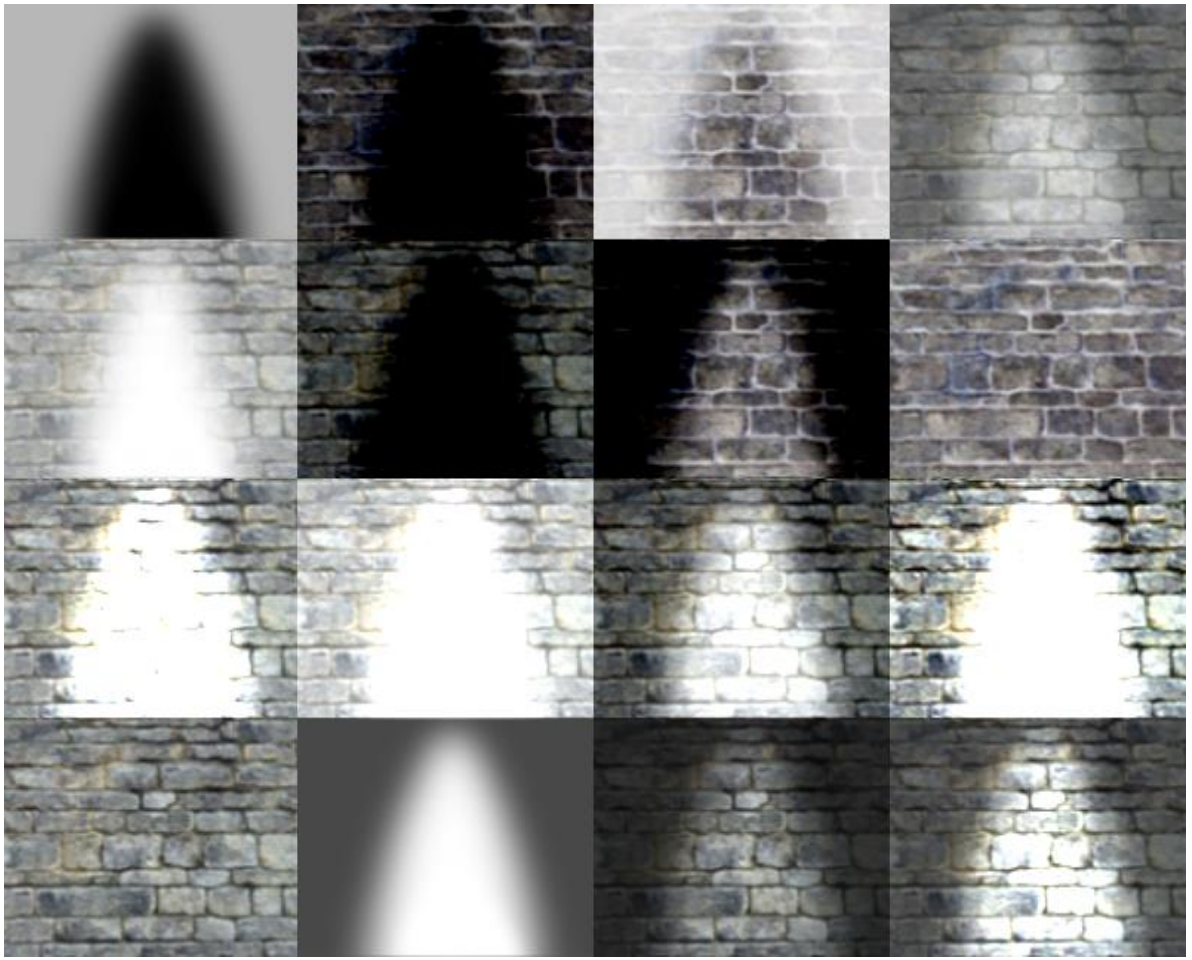


- Vertex processing Rasterizer에 의해 만들어진 픽셀, 사용자가 정의한 색상 ,Sampling한 텍스처 픽셀 등을 조합, 최종 픽셀을 만들어 가는 과정
- 픽셀의 R, G, B, A를 독립적으로 처리 가능
- 덧셈, 뺄셈, 곱셈을 사용한 가산/감산 연산 및 수학 함수를 사용, 픽셀을 조작
- 렌더링이 필요 없을 때 이후 과정 포기 가능 -> Discard

# 9.4 Multi-Texturing 예



```
ct0 = texture2D(us_t0, 텍스처 좌표);
ct1 = texture2D(us_t1, 텍스처 좌표);

최종 색상 = ct0 * ct1;              // Modulate
최종 색상 = ct0 * ct1 * 2.0;        // Modulate 2x
최종 색상 = ct0 * ct1 * 4.0;        // Modulate 4x
최종 색상 = ct0 + ct1;              // Add
최종 색상 = ct0 + ct1 - 0.5;        // Add signed
최종 색상 =(ct0 + ct1 - 0.5)*2.0; // Add signed 2x
최종 색상 = ct0 + ct1 - ct0*ct1;   // Add smooth
최종 색상 = ct0 - ct1;              // Sub
최종 색상 = ct1 - ct0;              // Sub
최종 색상 = 1.0 - ct0;              // Inverse ct0
최종 색상 = 1.0 - ct1;              // Inverse ct1
최종 색상 = 1.0 - (ct0 + ct1);      // Inverse (ct0+ct1)
최종 색상 = 1.0 - ct0 * ct1;        // Inverse signed
최종 색상 = (ct0 +ct1) * 0.5;       // Half
```



- 앞서 조사한 범프 텍스처 2장을 골라서 결합해서 블렌딩 해보자.

# 9. Blending

- Alpha Blending
  - 두 픽셀을 가중치에 따라 섞는 방법
  - Pixel의 Alpha 값을 해당 색상의 Weight와 동일
  - Shader 에서 [0,1] 범위 값을 가짐  1 완전 불투명 , 0: 완전 투명

- 종류
  - 멀티 텍스처링에 의한 블렌딩: Vertex, Material, Texture Alpha 사용
  - Alpha Blending: Frame Buffer or Render Target Alpha Blending

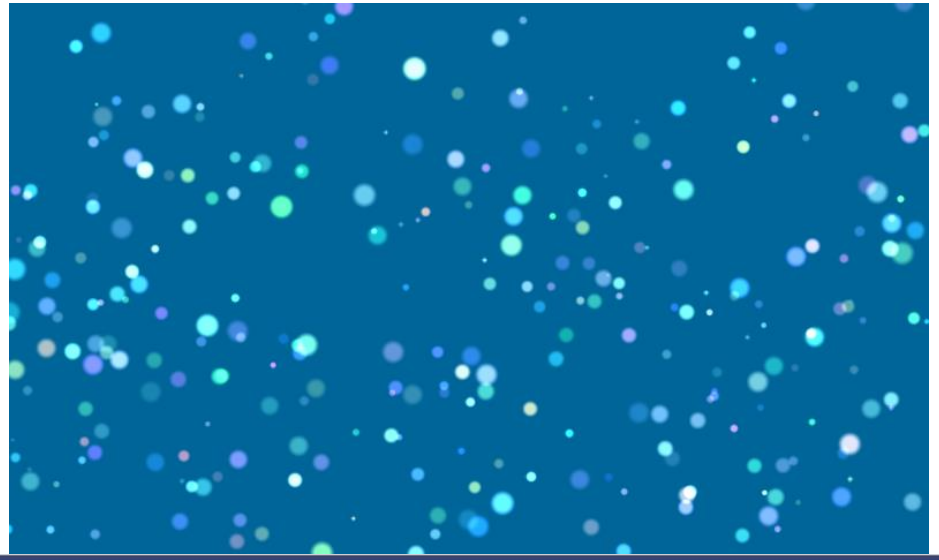
- Frame Buffer Alpha Blending
  - Back buffer에 쓰여져 있는 픽셀과 새로운 픽셀을 섞는 방법

  - 최종 픽셀 = Source Pixel(새로운 픽셀) ⊗ Source Blend Factor
    + Dest Pixel(이미 저장되어 있는 픽셀) ⊗ Dest Blend Factor
    ⊗ 은 Pixel의 R, G, B에 적용될 연산

- Default Blend Factor:
  - Source: SRC_ALPHA
  - Dest: INV_SRC_ALPHA
  - 주의: WebGL은 해당 없음

- Ex)

```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE);
gl.blendFunc(gl.ONE, gl.ONE);
gl.blendFunc(gl.BLEND_SRC_RGB, gl.BLEND_DST_RGB);
gl.enable(gl.BLEND);
```



- 분수 파티클을 만들어보자.
- 색상을 변화시킨 폭발 파티클을 구현해 보자.
- 화면 보호 프로그램을 만들어 보자.