WebGL 3D Draw and GLSL

WebGL

GLSL ES…

- 데이터, 변수 및 변수 유형
- 벡터, 행렬, 구조체, 배열 및 샘플러 유형
- 운영자, 제어 흐름 및 기능
- 속성, 균일 변수 및 가변 변수
- 정밀 한정자
- 전처리 기 및 지시문

```
// Vertex shader
attribute vec4 a_Position;
attribute vec4 a_Color;
uniform mat4 u_MvpMatrix;
varying vec4 v_Color;
void main() {
  gl_Position = u_MvpMatrix * a_Position;
  v_Color = a_Color;
}
```

```
// Fragment shader
#ifdef GLSL_ES
precision mediump float;
#endif
varying vec4 v_Color;
void main() {
  gl_FragColor = v_Color;
}
```

Variables are declared at the beginning of the code, and then the `main()` routine defines the entry point for the program.

WebGL

Variables

- The character set for variables names contains only the letters a–z, A–Z, the underscore (_), and the numbers 0–9.
- Numbers are not allowed to be used as the first character of variable names.
- The keywords shown in Table 6.1 and the reserved keywords shown in Table 6.2 are not allowed to be used as variable names. However, you can use them as part of the variable name, so the variable name `if` will result in error, but `iffy` will not.
- Variable names starting with `gl_`, `webgl_`, or `_webgl_` are reserved for use by OpenGL ES. No user-defined variable names may begin with them.

attribute	bool	break	bvec2	bvec3	bvec4
const	continue	discard	do	else	false
float	for	highp	if	in	inout
Int	invariant	ivec2	ivec3	ivec4	lowp
mat2	mat3	mat4	medium	out	precision
return	sampler2D	samplerCube	struct	true	uniform
varying	vec2	vec3	vec4	void	while

Vector Types and Matrix Types

GLSL ES supports vector and matrix data types which, as you have seen, are useful when dealing with computer graphics. Both these types contain multiple data elements. A vector type, which arranges data in a list, is useful for representing vertex coordinates or color data. A matrix arranges data in an array and is useful for representing transformation matrices. Figure 6.1 shows an example of both types.

$$\begin{pmatrix} 3 & 7 & 1 \end{pmatrix} \quad \begin{bmatrix} 3 & 7 & 1 \\ 1 & 5 & 3 \\ 4 & 0 & 7 \end{bmatrix}$$

```
vec3 v3 = vec3(1.0, 0.0, 0.5);  // sets v3 to(1.0, 0.0, 0.5)
vec2 v2 = vec2(v3);  // sets v2 to (1.0, 0.0) using the 1st and 2nd elements of v3
vec4 v4 = vec4(1.0); // sets v4 to (1.0, 1.0, 1.0, 1.0)
```

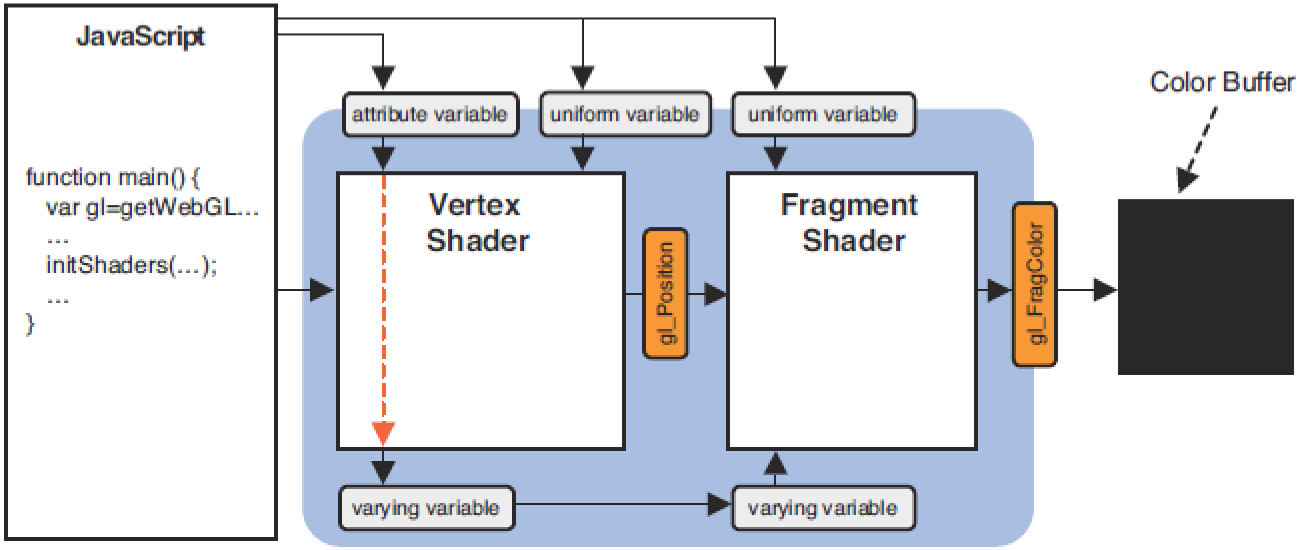
Category	Types in GLSL ES	Description
`Vector`	`vec2, vec3, vec4`	The data types for 2, 3, and 4 component vectors of floating point numbers
	`ivec2, ivec3, ivec4`	The data types for 2, 3, and 4 component vectors of integer numbers
	`bvec2, bvec3, bvec4`	The data types for 2, 3, and 4 component vectors of boolean values
`Matrix`	`mat2, mat3, mat4`	The data type for 2×2, 3×3, and 4×4 matrix of floating point numbers (with 4, 9, and 16 elements, respectively)

```
mat4 m4 = mat4 (  1.0,  2.0,  3.0,  4.0,
                  5.0,  6.0,  7.0,  8.0,
                  9.0, 10.0, 11.0, 12.0,
                 13.0, 14.0, 15.0, 16.0 );
```

$$\begin{bmatrix} 1.0 & 5.0 & 9.0 & 13.0 \\ 2.0 & 6.0 & 10.0 & 14.0 \\ 3.0 & 7.0 & 11.0 & 15.0 \\ 4.0 & 8.0 & 12.0 & 16.0 \end{bmatrix}$$

WebGL

Attribute, uniform, and varying variables



Attribute : Vertex shader에만 사용, 전역 변수로 선언
Uniform : 모두 사용 가능, 전역 변수 선언, 읽기 전용,
Varying : 전역 변수, vertex -> fragment 데이터 전달

WebGL

Precision Qualifiers

```
#ifdef GL_ES
precision mediump float;
#endif
```

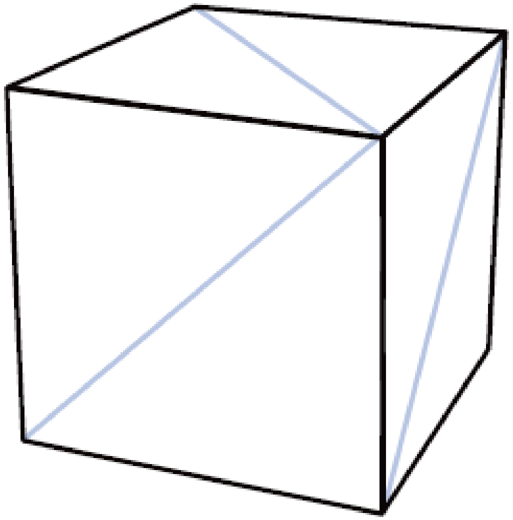
Precision Qualifiers	Descriptions	Default Range and Precision	
		Float	int
highp	High precision. The minimum precision required for a vertex shader.	$(-2^{62}, 2^{62})$ Precision: 2^{-16}	$(-2^{16}, 2^{16})$
mediump	Medium precision. The minimum precision required for a fragment shader. More than lowp, and less than highp.	$(-2^{14}, 2^{14})$ Precision: 2^{-10}	$(-2^{10}, 2^{10})$
lowp	Low precision. Less than mediump, but all colors can be represented.	$(-2, 2)$ Precision: 2^{-8}	$(-2^{8}, 2^{8})$

Type of Shader	Data Type	Default Precision
Vertex shader	int	highp
	float	highp
	sampler2D	lowp
	samplerCube	lowp
Fragment shader	int	medium
	float	**None**
	sampler2D	lowp
	samplerCube	lowp

WebGL

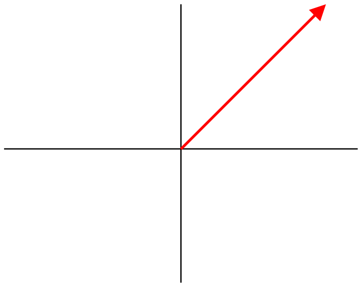
3D Objects

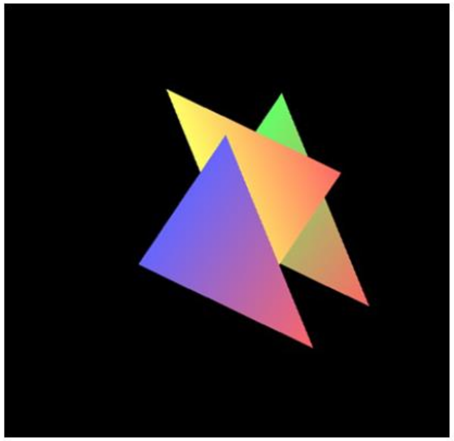
- 사용자 시작 3D 세계 표현
- 3D 공간 볼륨 제어
- 클리핑
- 전경 및 배경 객체 처리
- 2D 개체 그리기

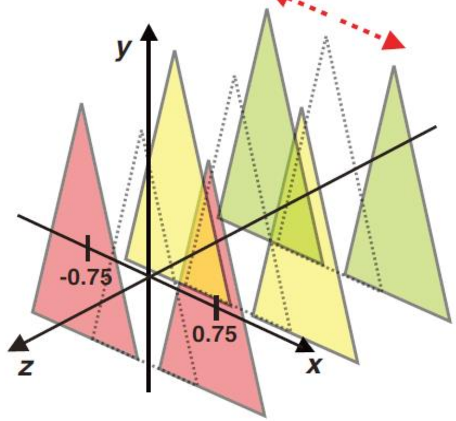


시각화 방향 지정

- + 깊이
- 보는 방향 / 보이는 범위

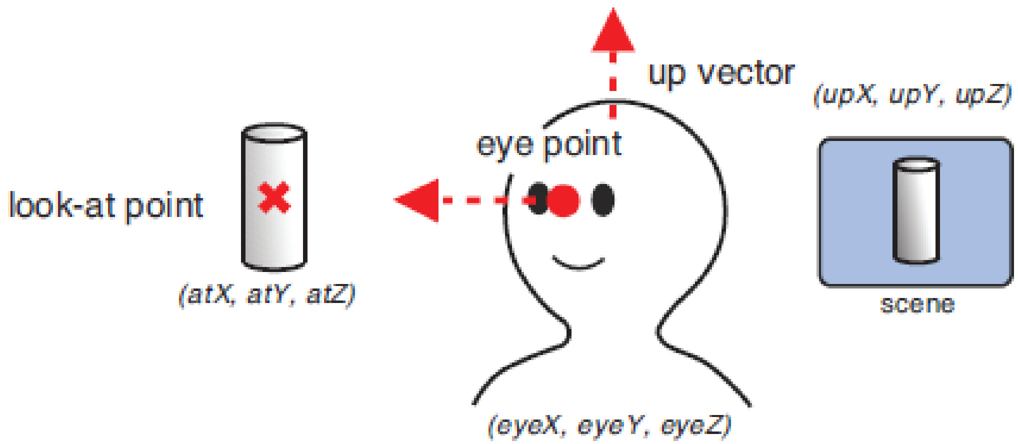


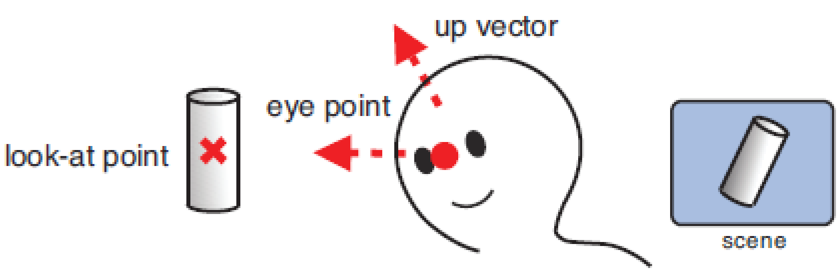




Eye Point, Look-At Point, and Up Direction

- **Eye Point : 3D 공간을 보는 시작점 (eyeX, eyeY, eyeZ)**
- **Look-at Point : 사용자가 보는 지점, 시선의 방향 결정 (atX, atY, atZ)**
- **Up direction : 시점에서 시점으로 보여지는 장면 위쪽 방향 결정 (upX, upY,upZ)**





Eye Point, Look-At Point, and Up Direction

- **cuon-matrix.js**

`Matrix4.setLookAt(eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ)`

Calculate the view matrix derived from the eye point (*eyeX*, *eyeY*, *eyeZ*), the look-at point (*atX*, *atY*, *atZ*), and the up direction (*upX*, *upY*, *upZ*). This view matrix is set up in the Matrix4 object. The look-at point is mapped to the center of the `<canvas>`.

Parameters	eyeX, eyeY, eyeZ	Specify the position of the eye point.
	atX, atY, atZ	Specify the position of the look-at point.
	upX, upY, upZ	Specify the up direction in the scene. If the up direction is along the positive y-axis, then (*upX*, *upY*, *upZ*) is (0, 1, 0).
Return value	None	

```
var initialViewMatrix = new Matrix4();
initialViewMatrix.setLookAt(0, 0, 0, 0, 0, -1, 0, 1, 0);
```

eye point   look-at point   up vector

WebGL

Eye Point, Look-At Point, and Up Direction

```
Matrix4.prototype.setLookAt =
 function(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)
{
  var e, fx, fy, fz, rlf, sx, sy, sz, rls, ux, uy, uz;

  fx = centerX - eyeX;
  fy = centerY - eyeY;
  fz = centerZ - eyeZ;

  // Normalize f.
  rlf = 1 / Math.sqrt(fx*fx + fy*fy + fz*fz);
  fx *= rlf;
  fy *= rlf;
  fz *= rlf;

  // Calculate cross product of f and up.
  sx = fy * upZ - fz * upY;
  sy = fz * upX - fx * upZ;
  sz = fx * upY - fy * upX;

  // Normalize s.
  rls = 1 / Math.sqrt(sx*sx + sy*sy + sz*sz);
  sx *= rls;
  sy *= rls;
  sz *= rls;

  // Calculate cross product of s and f.
  ux = sy * fz - sz * fy;
  uy = sz * fx - sx * fz;
  uz = sx * fy - sy * fx;

  // Set to this.
  e = this.elements;
  e[0] = sx;
  e[1] = ux;
  e[2] = -fx;
  e[3] = 0;

  e[4] = sy;
  e[5] = uy;
  e[6] = -fy;
  e[7] = 0;

  e[8] = sz;
  e[9] = uz;
  e[10] = -fz;
  e[11] = 0;

  e[12] = 0;
  e[13] = 0;
  e[14] = 0;
  e[15] = 1;

  // Translate.
  return this.translate(-eyeX, -eyeY, -eyeZ);
};
```

LookAtTriangle (html)

```html
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="utf-8" />
  <title>Look triangle from slant</title>
 </head>

 <body onload="main()">
  <canvas id="webgl" width="400" height="400">
  Please use a browser that supports "canvas"
  </canvas>

  <script src="webgl-utils.js"></script>
  <script src="webgl-debug.js"></script>
  <script src="cuon-utils.js"></script>
  <script src="cuon-matrix.js"></script>
  <script src="LookAtTriangles.js"></script>
 </body>
</html>
```

WebGL

LookAtTriangle (js)

```
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'attribute vec4 a_Color;\n' +
  'uniform mat4 u_ViewMatrix;\n' +
  'varying vec4 v_Color;\n' +
  'void main() {\n' +
  '  gl_Position = u_ViewMatrix * a_Position;\n' +
  '  v_Color = a_Color;\n' +
  '}\n';

// Fragment shader program
var FSHADER_SOURCE =
  '#ifdef GL_ES\n' +
  'precision mediump float;\n' +
  '#endif\n' +
  'varying vec4 v_Color;\n' +
  'void main() {\n' +
  '  gl_FragColor = v_Color;\n' +
  '}\n';
```

WebGL

LookAtTriangle (js)

```
function main() {
  // Retrieve <canvas> element
  var canvas = document.getElementById('webgl');

  // Get the rendering context for WebGL
  var gl = getWebGLContext(canvas);
  if (!gl) {
    console.log('Failed to get the rendering context for WebGL');
    return;
  }

  // Initialize shaders
  if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
    console.log('Failed to intialize shaders.');
    return;
  }

  var n = initVertexBuffers(gl);
  if (n < 0) {
    console.log('Failed to set the vertex information');
    return;
  }

  // Specify the color for clearing <canvas>
  gl.clearColor(0, 0, 0, 1);

  // Get the storage location of u_ViewMatrix
  var u_ViewMatrix = gl.getUniformLocation(gl.program, 'u_ViewMatrix');
  if (!u_ViewMatrix) {
    console.log('Failed to get the storage locations of u_ViewMatrix');
    return;
  }

  // Set the matrix to be used for to set the camera view
  var viewMatrix = new Matrix4();

  viewMatrix.setLookAt(0.20, 0.25, 0.25, 0, 0, 0, 0, 1, 0);

  gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);

  gl.clear(gl.COLOR_BUFFER_BIT);

  gl.drawArrays(gl.TRIANGLES, 0, n);
}
```

WebGL

LookAtTriangle (js)

```
function initVertexBuffers(gl) {
  var verticesColors = new Float32Array([
   // Vertex coordinates and color(RGBA)
    0.0,  0.5,  -0.4,  0.4,  1.0,  0.4,
   -0.5, -0.5,  -0.4,  0.4,  1.0,  0.4,
    0.5, -0.5,  -0.4,  1.0,  0.4,  0.4,
   // The back green one
    0.5,  0.4,  -0.2,  1.0,  0.4,  0.4,
   -0.5,  0.4,  -0.2,  1.0,  1.0,  0.4,
    0.0, -0.6,  -0.2,  1.0,  1.0,  0.4,
   // The middle yellow one
    0.0,  0.5,   0.0,  0.4,  0.4,  1.0,
   -0.5, -0.5,   0.0,  0.4,  0.4,  1.0,
    0.5, -0.5,   0.0,  1.0,  0.4,  0.4,
  // The front blue one
  ]);
  var n = 9;

  // Create a buffer object
  var vertexColorbuffer = gl.createBuffer();
  if (!vertexColorbuffer) {
    console.log('Failed to create the buffer object');
    return -1;
  }

  // Write the vertex coordinates and color to the buffer object
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorbuffer);
  gl.bufferData(gl.ARRAY_BUFFER, verticesColors, gl.STATIC_DRAW);

  var FSIZE = verticesColors.BYTES_PER_ELEMENT;
  var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
  if(a_Position < 0) {
    console.log('Failed to get the storage location of a_Position');
    return -1;
  }
  gl.vertexAttribPointer(a_Position, 3, gl.FLOAT, false, FSIZE * 6, 0);
  gl.enableVertexAttribArray(a_Position);

  var a_Color = gl.getAttribLocation(gl.program, 'a_Color');
  if(a_Color < 0) {
    console.log('Failed to get the storage location of a_Color');
    return -1;
  }
  gl.vertexAttribPointer(a_Color, 3, gl.FLOAT, false, FSIZE * 6, FSIZE * 3);
  gl.enableVertexAttribArray(a_Color);

  gl.bindBuffer(gl.ARRAY_BUFFER, null);

  return n;
}
```