7.버텍스 배열

7-1.버텍스 배열

입체 도형은 아주 많은 정점들과 관련 정보들로 구성된다. 그럴싸한 입체 도형 하나를 만들려 glVertex 함수를 수천, 수만번을 호출해야 하고 각 정점마다 색상이나 법선 정보도 필요하다. 정보가 많은 것은 어쩔 수 없는 일이지만 각 정보마다 함수를 일일이 호출하는 것은 낭비가 너무 심하다. 알다시피 함수 호출은 실행 파일 크기면에서나 속도면에서나 비용이 꽤 비싼 동작이다.

각 정점마다 달라지는 것은 좌표일 뿐이므로 이 좌표들을 하나의 배열에 모아 둔 후 루프를 돌리면서 배열 요소를 하나씩 전달하는 것이 구조적으로 훨씬 더 나은 방법이다. 프로그래밍의 기본을 아는 사람이라면 반복이 과도해질 때 당연히 이 방법을 떠올릴 것이다. OpenGL도 배열로 정점의 집합을 정의하는 방법을 공식적으로 지원한다. 먼저 다음 함수를 호출하여 배열을 사용하도록 설정한다.

void glEnableClientState(GLenum cap);
void glEnableClientState(GLenum cap);

배열을 사용하는 것은 OpenGL 서버인 그래픽 카드의 설정과는 상관이 없고 그래픽을 그리는 클라이언트인 CPU와 상관이 있으므로 glEnable 함수를 사용하지 않는다. 어떻게 그릴 것인가의 문제가 아니고 어떻게 정보를 전달할 것인가의 문제이므로 배열 사용 여부는 클라이언트의 설정일 뿐이다. 그래서 glEnable 함수 대신 glEnableClientState 함수를 사용한다. 인수로 어떤 배열을 사용할 것인가를 전달한다.

```
GL_COLOR_ARRAY
GL_EDGE_FLAG_ARRAY
GL_FOG_COORD_ARRAY
GL_INDEX_ARRAY
GL_NORMAL_ARRAY
GL_SECONDARY_COLOR_ARRAY
GL_TEXTURE_COORD_ARRAY
GL_VERTEX_ARRAY
```

정점뿐만 아니라 각 정점의 색상이나 법선 정보, 텍스처 좌표 등도 배열로 정의 가능하다. 입체 도형을 구성하기 위한 모든 정보들을 하나의 배열에 집약할 수 있다. 배열은 동일 타입 변수의 집합일 뿐이므로 일반적인 C 구문으로 작성한다. 정점 배열의 경우 GLfloat 타입의 일차원 배열을 선언하고 배열을 구성하는 x, y나 x,y,z 좌표를 죽 나열하면 된다.

```
GLfloat vert[] = { x1, y1, z1, x2, y2, z2, .... };
```

배열 요소는 필요한만큼 얼마든지 나열할 수 있다. 미리 정의해 놓은 이진 데이터를 파일이나 네트워크로 읽어들일 수도 있고 필요할 경우 일부를 약간 변형하는 것도 가능하다. 다음 함수는이 배열의 위치와 구조를 알려 준다. 대표적으로 정점 배열의 경우만 보자.

void glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid * pointer);

size는 한 좌표를 구성하는 요소의 개수이다. x, y 두개로 구성된 좌표이면 2를 지정하고 x, y, z 세 개로 구성된 좌표이면 3을 지정한다. 배열 자체가 일차원이므로 한 정점을 이루는 좌표가 몇 개씩 한 쌍인지를 밝히는 것이다. type은 배열 요소의 타입이다. 정수 좌표이면 GL_INT, 실수 좌표이면 GL_FLOAT 등과 같이 타입

을 밝힌다. 이 두 인수는 정점을 정의할 때 glVertex2i를 호출할 것인지 glVertext3f를 호출할 것인지를 결정한다.

stride는 배열 요소간의 간격을 지정하는데 모든 요소를 연이어 배치하는 것이 보통이므로 이 값은 대개의 경우 0이다. 이 인수가 어떻게 활용되는지는 잠시 후 실습해 볼 것이다. 마지막 인수 pointer는 배열이 정의된 실제 주소이다. 이 함수 호출에 의해 OpenGL은 정점 데이터가 어느 배열에 어떤 구조로 저장되어 있는지를 알게 된다. 색상 배열이나 법선 배열도 구조는 거의 동일하다.

void glColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid * pointer);
void glNormalPointer(GLenum type, GLsizei stride, const GLvoid * pointer);

다만 정보의 성격상 약간씩 달라지는 부분이 있는데 색상은 rgb, rgba 등으로 구성될 수 있으므로 size 인수가 있다. 단, size의 가능한 값은 3 또는 4뿐이라는 점에서 정점 배열과는 다르다. 법선은 무조건 xyz 세 개씩 한 쌍이므로 size 정보는 불필요하다. 배열을 정의한 후 다음 함수로 배열 요소를 하나씩 꺼낸다.

void glArrayElement(GLint i);

이 함수는 배열에서 i 번째 정점 좌표를 꺼내 해당 정보를 서버로 전달한다. size와 type 정보를 바탕으로 한번에 몇 개씩 꺼낼 것인가를 결정할 수 있다. 정점 배열이라면 glVertex 함수를 호출하고 색상 배열이라면 glColor를 호출할 것이다. 배열로부터 정점을 정의할 경우에는 glBegin과 glEnd 블록에서 이 함수를 호출하면 된다.

배열을 정의하고 사용하는 절차가 다소 복잡하다. 다음 예제는 배열을 사용하는 가장 기본적인 절차를 잘 보여주며 또한 여러 가지 대체적인 방법도 실험해 봄으로써 정점 배열이 왜 실용적인지를 설명한다. 피라미드를 구성하는 정점들을 배열로 정의해 두고 루프를 돌며 정점을 정의함으로써 입체 도형을 한번에 그린다.

VertexArray

```
#include <windows.h>
#include <gl/glut.h>
#include <stdio.h>

void DoDisplay();
void DoKeyboard(unsigned char key, int x, int y);
GLfloat xAngle, yAngle, zAngle;

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
      glutCreateWindow("OpenGL");
      glutDisplayFunc(DoDisplay);
      glutKeyboardFunc(DoKeyboard);
      glutMainLoop();
      return 0;
}

void DoKeyboard(unsigned char key, int x, int y)
{
      switch(key) {
      case 'a':yAngle += 2;break;
      case 'd':yAngle -= 2;break;
      case 'w':xAngle += 2;break;
      case 's':xAngle -= 2;break;
```

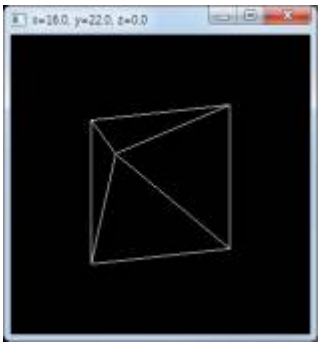
```
        case 'q':zAngle += 2;break;
        case 'e':zAngle -= 2;break;
        case 'z':xAngle = yAngle = zAngle = 0.0;break;
        }
        char info[128];
        sprintf(info, "x=%.1f, y=%.1f, z=%.1f", xAngle, yAngle, zAngle);
        glutSetWindowTitle(info);
        glutPostRedisplay();
}

void DoDisplay()
{
        static GLfloat vert[] = {
                0, 0, -0.8,             // 12시
                0.5, 0.5, 0,
                -0.5, 0.5, 0,

                0, 0, -0.8,             // 9시
                -0.5, 0.5, 0,
                -0.5, -0.5, 0,

                0, 0, -0.8,             // 6시
                -0.5, -0.5, 0,
                0.5, -0.5, 0,

                0, 0, -0.8,             // 3시
                0.5, -0.5, 0,
                0.5, 0.5, 0,
        };

        glEnableClientState(GL_VERTEX_ARRAY);
        glVertexPointer(3, GL_FLOAT, 0, vert);

        glClear(GL_COLOR_BUFFER_BIT);
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glRotatef(xAngle, 1.0f, 0.0f, 0.0f);
        glRotatef(yAngle, 0.0f, 1.0f, 0.0f);
        glRotatef(zAngle, 0.0f, 0.0f, 1.0f);

        glRectf(-0.5, 0.5, 0.5, -0.5);

        glBegin(GL_TRIANGLES);
        for (int i=0;i<sizeof(vert)/sizeof(vert[0]);i+=3) {
                glVertex3f(vert[i], vert[i+1], vert[i+2]);
        }
        glEnd();

        glPopMatrix();
        glFlush();
}
```

vert 배열에는 피라미드를 구성하는 각 삼각형의 정점 좌표들을 저장했다. 삼각형 하나당 3개씩의 정점이 필요하고 각 정점당 x, y, z 세 쌍씩 좌표가 필요하므로 총 정점의 개수는 12개이며 배열의 크기는 36이다. GL_TRIANGLE_FAN 모드를 사용하면 5개로도 가능하지만 인덱스를 사용하는 더 좋은 방법이 있으므로 여기

서는 그냥 각 삼각형을 구성하는 정점을 중복 정의했다. 매 함수 호출시마다 배열을 초기화할 필요는 없으므로 static으로 선언했는데 아예 전역으로 선언해도 상관없다.

배열을 선언한 후 정점 배열 기능을 사용하겠다고 선언한다. 그리고 glVertexPointer 함수로 vert 배열의 위치를 알려 주는데 GL_FLOAT 타입의 변수 3개씩 한쌍을 이루어 정점 하나에 대한 정보임을 명시한다. 아직 색상 정보가 없으므로 폴리곤 모드를 GL_LINE으로 설정하여 선만 그리도록 했다.

일단 glRectf 함수로 밑면 사각형을 먼저 그린다. 사각형은 모양이 다르므로 별도로 그렸는데 두 개의 삼각형으로 분할하면 이 역시 정점 배열에 포함시킬 수도 있다. 나머지 삼각형은 루프를 돌며 그린다. 예제의 코드는 아주 원론적이다. vert 배열의 선두에서 시작하여 3칸씩 건너뛰면서 i, i+1, i+2 번째 요소를 꺼내면 이 값들이 곧 x, y, z 좌표값이다. 좌표를 꺼내 glVertex 함수로 차례대로 전달한 것이다. 피라미드가 잘 그려질 것이다.



정점들이 배열에 정의되어 있으므로 배열을 인수로 취하는 glVertex 함수를 호출할 수도 있다. 3칸씩 건너뛰면서 x 좌표가 있는 선두 번지를 glVertex3fv 함수로 전달하면 나머지 y, z 값은 알아서 꺼내 쓸 것이다. 다음 코드도 결과는 동일하다.

```
glBegin(GL_TRIANGLES);
for (int i=0;i<sizeof(vert)/sizeof(vert[0]);i+=3) {
	glVertex3fv(&vert[i]);
}
glEnd();
```

이상의 두 코드는 어디까지나 배열에 좌표를 저장해 놓고 쓸 수 있다는 것을 보여주는 것 뿐이다. 좌표값을 인수로 직접 전달하지 않을 뿐 glVertex 함수를 매번 호출하는 것과 원론적으로 같은 코드이다. 함수 호출 코드가 반복되지 않으므로 메모리상의 이점은 있지만 루프 내부에서 함수를 여러 번 호출하기는 매 한가지이므로 속도상의 이점은 없다. 하지만 정점이 배열에 모여 있어 수정하기 편하다는 이점은 있다.

여기까지의 변화는 C 언어 문법 수준에서 충분히 납득이 될 것이다. 이제 OpenGL의 정점 배열 기능을 활용해 보자. glBegin ~ glEnd 부분을 다음 코드로 대체한다.

```
glBegin(GL_TRIANGLES);
for (int i=0;i<sizeof(vert)/sizeof(vert[0])/3;i++) {
	glArrayElement(i);
}
glEnd();
```

배열에서 좌표를 직접 꺼낼 필요없이 glArrayElement 함수로 정점의 순서값만 전달해 주면 된다. 이때 glArrayElement 함수의 인수는 배열의 첨자가 아니라 정점의 순서값임을 유의하자. 각 정점이 3개씩의 값으로 구성되어 있으므로 배열 크기의 1/3만큼만 루프를 돌면 된다. 배열을 활용하는 최종적인 코드는 다음 함수를 호출하는 것이다.

void glDrawArrays(GLenum mode, GLint first, GLsizei count);

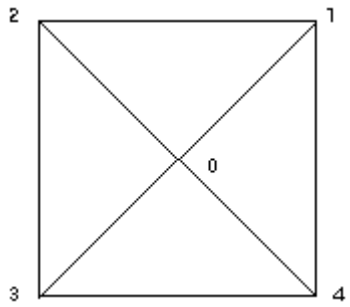
이 함수는 배열의 크기만 알려 주면 알아서 루프를 돌고 glBegin, glEnd 블록까지도 처리해 준다. mode 인수는 glBegin으로 전달할 그리기 모드이며 first는 배열의 첫번째 요소를 지정하되 통상 0이다. count는 배열에 저장된 정보의 개수이다. 이 경우 정점이 12개이므로 12로 전달한다. 다음 딱 한 줄로 피라미드를 그릴 수 있다.

```
glDrawArrays(GL_TRIANGLES, 0, sizeof(vert)/sizeof(vert[0])/3);
```

이 함수의 내부 동작은 앞에서 작성한 코드를 역으로 유추해 보면 쉽게 짐작할 수 있다. glBegin으로 그리기 블록을 시작하고 배열을 순회하면서 각 정점 좌표를 꺼내 glVertex 함수를 순서대로 호출할 것이다. 딱 하나의 함수 호출만으로 그리기를 수행하므로 메모리를 훨씬 덜 차지하고 속도도 월등히 빠르다.

7-2.배열 인덱스

앞 예제는 배열을 정점에 정의함으로써 코드가 훨씬 더 짧아졌고 속도도 빨라졌다. 그러나 아직 색상도 입히지 못했고 같은 정점이 여러 번 중복되어 있어 뭔가 허술해 보인다. 물론 문제점을 해결할 수 있는 명쾌한 방법들이 제공된다. 피라미드를 구성하는 삼각형들은 정점들을 공유하는데 이 정점들을 하나로 합쳐서 지정할 수 있다. 피라미드는 다음 다섯 개의 정점으로 구성된다.



각 정점들을 배열에 한번씩만 나열해 놓고 도형은 어느 정정들로 구성되는지 인덱스만 밝힘으로써 정의할 수 있다. 예를 들어 12시 삼각형은 0, 1, 2번 정점으로 구성되고 9시 삼각형은 0, 2, 3번 정점으로 구성된다. 정점들은 각 도형의 접점에 있으므로 중복은 어찌할 도리가 없다. 그러나 좌표는 3개의 실수로 구성되는데 비해 첨자는 고작 정수 1바이트밖에 안되므로 중복에 의한 부작용이 훨씬 덜하다. 정점의 인덱스 배열을 정의한 후 다음 함수로 도형을 그린다.

void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid * indices);

그리기 모드와 배열의 크기, 인덱스 배열의 요소 타입 그리고 인덱스 배열을 전달한다. 이 함수는 배열에서 인덱스를 꺼내고 인덱스가 가리키는 순서값로부터 정점의 좌표를 찾아 도형을 그린다.

VertexArrayIndex

```
void DoDisplay()
{
	static GLfloat vert[] = {
		0, 0, -0.8,		// 중앙
		0.5, 0.5, 0,		// 우상
		-0.5, 0.5, 0,		// 좌상
		-0.5, -0.5, 0,		// 좌하
```

```
        0.5, -0.5, 0,           // 우하
    };

    static GLubyte index[] = {
        0, 1, 2,                //12시
        0, 2, 3,                // 9시
        0, 3, 4,                // 6시
        0, 4, 1,                // 3시
    };

    glClear(GL_COLOR_BUFFER_BIT);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glRotatef(xAngle, 1.0f, 0.0f, 0.0f);
    glRotatef(yAngle, 0.0f, 1.0f, 0.0f);
    glRotatef(zAngle, 0.0f, 0.0f, 1.0f);

    glRectf(-0.5, 0.5, 0.5, -0.5);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, vert);

    glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_BYTE, index);

    glPopMatrix();
    glFlush();
}
```

실행 결과는 앞 예제와 완전히 동일하다. 그러나 중복되는 정점을 한번만 기록함으로써 개수가 5개로 줄어들었다. 대신 이 정점들이 어떻게 조합되어 삼각형을 구성하는지 인덱스 배열이 추가되었다. 당장은 큰 이득이 없는 것 같지만 좌표 외에 색상이나 텍스트 좌표 등의 정보가 더 들어가면 인덱스를 쓰는 것이 훨씬 더 유리하다.

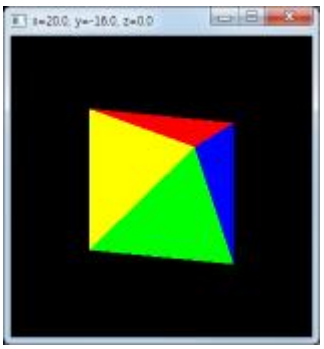
7-3.색상 배열

다음은 색상 배열도 적용하여 각 면에 색상을 입혀 보자. 정점 배열과 사용하는 방법이나 구성 원리는 거의 비슷하다. 각 정점에 해당하는 색상을 배열로 정의해 두고 위치를 가르쳐 준다. 면을 채울 것이므로 폴리곤 모드는 지정할 필요가 없되 대신 면끼리 전후 관계를 표현하기 위해 깊이 테스트는 해야 한다.

```
void DoDisplay()
{
    static GLfloat vert[] = {
        0, 0, -0.8,             // 중앙
        0.5, 0.5, 0,            // 우상
        -0.5, 0.5, 0,           // 좌상
        -0.5, -0.5, 0,          // 좌하
        0.5, -0.5, 0,           // 우하
    };

    static GLfloat color[] = {
        1,1,1,                  // 중앙
        0,0,1,                  // 우상
        1,0,0,                  // 좌상
        1,1,0,                  // 좌하
```

```
            0,1,0,                  // 우하
        };

        static GLubyte index[] = {
            0, 1, 2,                //12시
            0, 2, 3,                // 9시
            0, 3, 4,                // 6시
            0, 4, 1,                // 3시
        };

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glEnable(GL_DEPTH_TEST);
        glShadeModel(GL_FLAT);

        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glRotatef(xAngle, 1.0f, 0.0f, 0.0f);
        glRotatef(yAngle, 0.0f, 1.0f, 0.0f);
        glRotatef(zAngle, 0.0f, 0.0f, 1.0f);

        glColor3f(1,1,1);
        glRectf(-0.5, 0.5, 0.5, -0.5);

        glEnableClientState(GL_VERTEX_ARRAY);
        glVertexPointer(3, GL_FLOAT, 0, vert);
        glEnableClientState(GL_COLOR_ARRAY);
        glColorPointer(3, GL_FLOAT, 0, color);

        glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_BYTE, index);

        glPopMatrix();
        glFlush();
    }
```

color 배열에 각 정점에 대응되는 색상을 지정하되 정점 배열과 일대일로 대응되므로 순서를 동일하게 유지해야 한다. GL_COLOR_ARRAY 기능을 켜고 glColorPointer 함수로 색상 배열의 위치와 구조를 알려 주었다. 그리는 함수는 똑같이 glDrawElements이다. 이 함수는 인덱스 배열로부터 각 정점의 좌표를 꺼내면서 동시에 각 색상의 좌표도 꺼내 정점에 적용한다.

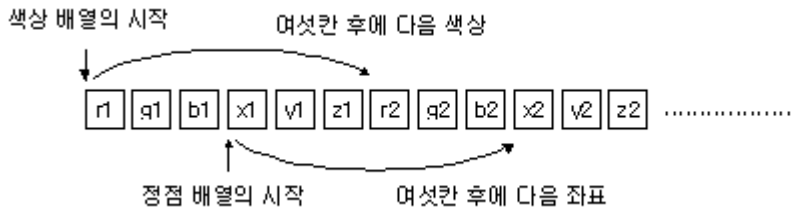


실행 결과는 애초에 만들었던 피라미드 예제와 동일하다. 정점 배열과 색상 배열은 하나의 정점에 대한 정보이되 하나는 좌표이고 하나는 색상이어서 두 개의 배열로 따로 나누어 저장했다. 요소의 타입만 일치한다면 이 두 배열을 하나로 합칠 수도 있다. 이때 사용되는 인수가 stride이다. 같은 배열에 순서대로 정의해 놓아도 stride 인수로 건너뛸 바이트 수를 지정할 수 있기 때문이다.

```
void DoDisplay()
{
	static GLfloat vertcolor[] = {
		1,1,1,	0, 0, -0.8,		// 중앙
		0,0,1,	0.5, 0.5, 0,	// 우상
		1,0,0,	-0.5, 0.5, 0,	// 좌상
		1,1,0,	-0.5, -0.5, 0,	// 좌하
		0,1,0,	0.5, -0.5, 0,	// 우하
	};

	static GLubyte index[] = {
		0, 1, 2,		//12시
		0, 2, 3,		// 9시
		0, 3, 4,		// 6시
		0, 4, 1,		// 3시
	};

	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	glEnable(GL_DEPTH_TEST);
	glShadeModel(GL_FLAT);

	glMatrixMode(GL_MODELVIEW);
	glPushMatrix();
	glRotatef(xAngle, 1.0f, 0.0f, 0.0f);
	glRotatef(yAngle, 0.0f, 1.0f, 0.0f);
	glRotatef(zAngle, 0.0f, 0.0f, 1.0f);

	glColor3f(1,1,1);
	glRectf(-0.5, 0.5, 0.5, -0.5);

	glEnableClientState(GL_VERTEX_ARRAY);
	glVertexPointer(3, GL_FLOAT, sizeof(GLfloat)*6, &vertcolor[3]);
	glEnableClientState(GL_COLOR_ARRAY);
	glColorPointer(3, GL_FLOAT, sizeof(GLfloat)*6, vertcolor);

	glDrawElements(GL_TRIANGLES, 12, GL_UNSIGNED_BYTE, index);

	glPopMatrix();
	glFlush();
}
```

vertcolor 배열은 정점의 좌표와 색상을 같이 저장한다. 배열의 0,1,2 번째 요소는 색상값이고 3,4,5번째 요소는 정점의 좌표값이며 두 값이 번갈아 나타난다. 정점 배열의 시작 위치를 3번째 요소의 번지에 맞추어 두고 6칸씩 건너뛰도록 지정하면 순서대로 꺼내서 사용할 것이다. glVertexPointer 함수의 인수들이 이런 정보를 제공한다. 배열의 시작위치는 &vertcolor[3]이며 stride는 GLfloat 요소 6개분이라는 것을 알려주고 있다.

마찬가지로 색상 배열은 선두에서 시작하되 마찬가지로 6칸씩 건너뛰도록 했다. stride는 동일하며 배열의 시작 번지는 vertcolor이다. 이 번지는 곧 첫번째 요소의 번지인 &vertcolor[0]와 같으며 배열의 선두 번지이다. 두 정보는 한 배열에 다음과 같이 오손 도손 섞여 있다.



두 정보를 한 배열에 섞어서 저장하되 각 정보의 크기가 일정하므로 크기만큼씩 건너뛰면서 읽으면 원하는 정보만 정확하게 빼내 사용할 수 있다. 한 배열에 두 정보가 모여 있어 허벌나게 헷갈리는 모양새지만 그건 컴퓨터가 알아서 할 일이므로 걱정하지 않아도 된다. 소스에서는 한 행에 한 정점의 정보를 기록하는 식으로 정리하고 탭으로 띄어쓰기만 잘하면 왼쪽은 색상, 오른쪽은 좌표이므로 사람은 오히려 더 편리하다.

좌표와 색상 뿐만 아니라 법선이나 텍스처 좌표 등도 동일한 방법으로 한 배열에 저장할 수 있다. 여러 배열로 나누어 저장하는 것보다 한 배열에 섞어서 저장하되 주기적인 거리만큼만 잘 배치하면 논리적으로 아무 문제가 없다. 이 방법을 좀 더 공식화한 것을 인터리브 배열이라고 한다. 말 그대로 이놈 저놈 막 섞여 있고 건너 뛰어 가며 읽는 배열이라는 뜻이다.

void glInterleavedArrays(GLenum format, GLsizei stride, const GLvoid * pointer);

format은 배열에 어떤 정보가 같이 들어 있는지를 지정한다. 레퍼런스를 보면 여러 가지 가능한 조합들에 대해 상수가 정의되어 있다. 대표적인 몇 가지만 소개하자면 다음과 같다.

GL_C3F_V3F : 색상값 3개, 좌표값 3개가 교대로 들어 있다

GL_C4F_N3F_V3F : 색상값 4개, 법선 3개, 좌표값 3개가 교대로 들어 있다.

GL_T2F_C4F_N3F_V3F : 텍스처, 색상, 법선, 좌표값이 들어 있다.

해당 정보 배열은 자동으로 활성화되므로 glEnableClientState 함수는 호출하지 않아도 상관없다. stride는 인터리브 배열의 건너뛸 거리인데 이 인수를 활용하면 그 외의 추가 정보도 더 넣을 수 있다. pointer는 물론 인터리브 배열의 선두 위치이다. 위 예제에서는 색상 배열과 좌표 배열을 활용하기 위해 다음 4줄의 함수를 호출했다.

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(GLfloat)*6, &vertcolor[3]);
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(3, GL_FLOAT, sizeof(GLfloat)*6, vertcolor);
```

각 배열을 사용하겠다는 의사 표시를 하고 또 각 배열에 대한 모양과 위치를 가르쳐 주어야 한다. 인터리브 배열을 사용하면 다음 한줄로 간단하게 이 호출을 대신할 수 있다.

```
glInterleavedArrays(GL_C3F_V3F, 0, vertcolor);
```

색상값 3개, 좌표값 3개가 교대로 들어 있다는 뜻이다. 이 선언을 한 후 glDrawElements 함수를 호출하면 인터리브 배열에서 필요한 값을 알아서 꺼내 사용할 것이다. 소스가 훨씬 더 짧아지고 배열을 관리하기도 더 수월하다.

7-4.출력 목록

glBegin과 glEnd 블록에서 그리기 명령을 직접 실행하는 것을 즉시 모드(immediate mode)라고 한다. 이 블록에 포함된 명령은 서버로 즉시 전송되어 바로 실행된다. 이에 비해 출력 목록(display list)은 그리기 명령의 집합을 일단 정의한 후 한꺼번에 실행하는 방법이다. 미리 컴파일해 놓음으로써 출력 속도가 향상되고 동일한 명령을 여러 번 반복할 때 유리하다.

반복을 최소화하고 재사용성을 높인다는 면에서 프로그래밍에서 함수를 정의하는 것과 유사하다. 출력 목록은 이름 대신 정수 ID로 구분한다. 아무 정수나 쓸 수 없고 출력 목록끼리 구분되어야 한다. 다음 함수로 출

력 목록의 ID를 생성한다.

GLuint glGenLists(GLsizei range);

필요한 개수를 전달하면 이 개수만큼 빈 영역을 찾아 시작 ID를 리턴한다. 이 ID 이후 range -1번까지의 ID를 사용할 수 있다. 하나만 필요하다면 glGenLists(1)을 호출하여 리턴된 값을 바로 사용하면 된다. 두 개가 필요하다면 glGenLists(2)을 호출하고 리턴값을 dl 변수로 받은 후 dl과 dl + 1을 사용하면 된다. 출력 목록을 시작할 때는 다음 함수를 호출한다.

void glNewList(GLuint list, GLenum mode);

list는 출력목록의 이름이다. mode는 출력 목록을 작성만 할 것인지(GL_COMPILE) 아니면 작성과 동시에 실행할 것인지(GL_COMPILE_AND_EXECUTE)를 지정한다. 다음 함수는 출력 목록 작성을 종료한다.

void glEndList(void);

glNewList와 glEndList 사이의 명령들이 출력 목록에 등록된다. 실행할 때는 다음 함수를 호출한다. 실행할 출력목록의 ID를 전달한다.

void glCallList(GLuint list);

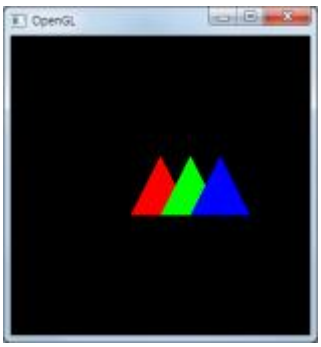
이때 목록에 저장된 그리기 명령이 실행된다. 다음 예제는 삼각형을 그리는 명령을 출력 목록에 저장해 두고 3번 호출한다.

DisplayList

```
#nclude <windows.h>
#nclude <gl/glut.h>

void DoDisplay();
int dl;

int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
        ,LPSTR lpszCmdParam,int nCmdShow)
{
      glutCreateWindow("OpenGL");

      dl = glGenLists(1);
      glNewList(dl, GL_COMPILE);
      glBegin(GL_TRIANGLES);
      glVertex2f(0.0, 0.2);
      glVertex2f(-0.2, -0.2);
      glVertex2f(0.2, -0.2);
      glEnd();
      glEndList();

      glutDisplayFunc(DoDisplay);
      glutMainLoop();
      return 0;
}
```

```
void DoDisplay()
{
	glClear(GL_COLOR_BUFFER_BIT);

	glColor3f(1,0,0);
	glCallList(dl);

	glTranslatef(0.2, 0.0, 0.0);
	glColor3f(0,1,0);
	glCallList(dl);

	glTranslatef(0.2, 0.0, 0.0);
	glColor3f(0,0,1);
	glCallList(dl);

	glFlush();
}
```

메인 함수에서 출력 목록을 미리 작성해 놓는다. 컴파일만 해 놓는 것이므로 출력 함수에서 작성하지 않아도 상관없다. 1개의 출력 목록 ID를 dl이라는 변수에 저장하고 이 목록에 삼각형을 그리는 명령들을 저장해 두었다. DoDisplay에서는 색상과 좌표를 조금씩 바꿔 가며 출력 목록을 세번 호출한다.



세 개의 삼각형이 나란히 출력된다. 출력 목록이 아니라면 같은 명령을 일일이 3번 나열해야 할 것이다. 함수가 함수를 호출하는 것과 마찬가지로 출력 목록끼리 서로 호출하는 것도 가능하다. 복잡한 도형을 여러 개의 서브 도형으로 나눈 후 출력 목록을 조합하여 여러 번 재사용할 수 있다.

출력 목록은 서버에 저장된다. 그래서 실행할 때마다 명령을 다시 전달하지 않음으로써 속도가 직접 모드보다 빠르다. 하지만 모든 명령은 서버상에서 실행되므로 클라이언트 상태를 변경하거나 참조하는 명령은 포함될 수 없다.