

Computer Graphics

Prof. Jibum Kim

Department of Computer Science & Engineering

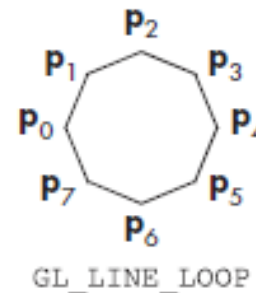
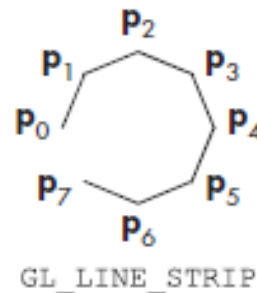
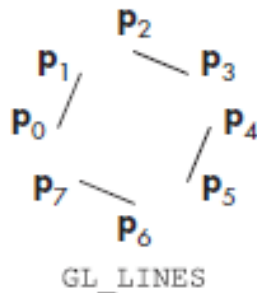
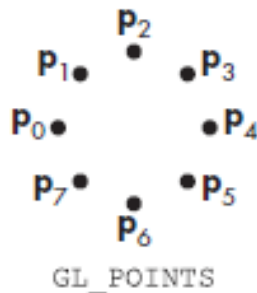
Incheon National University

■ Geometric primitives

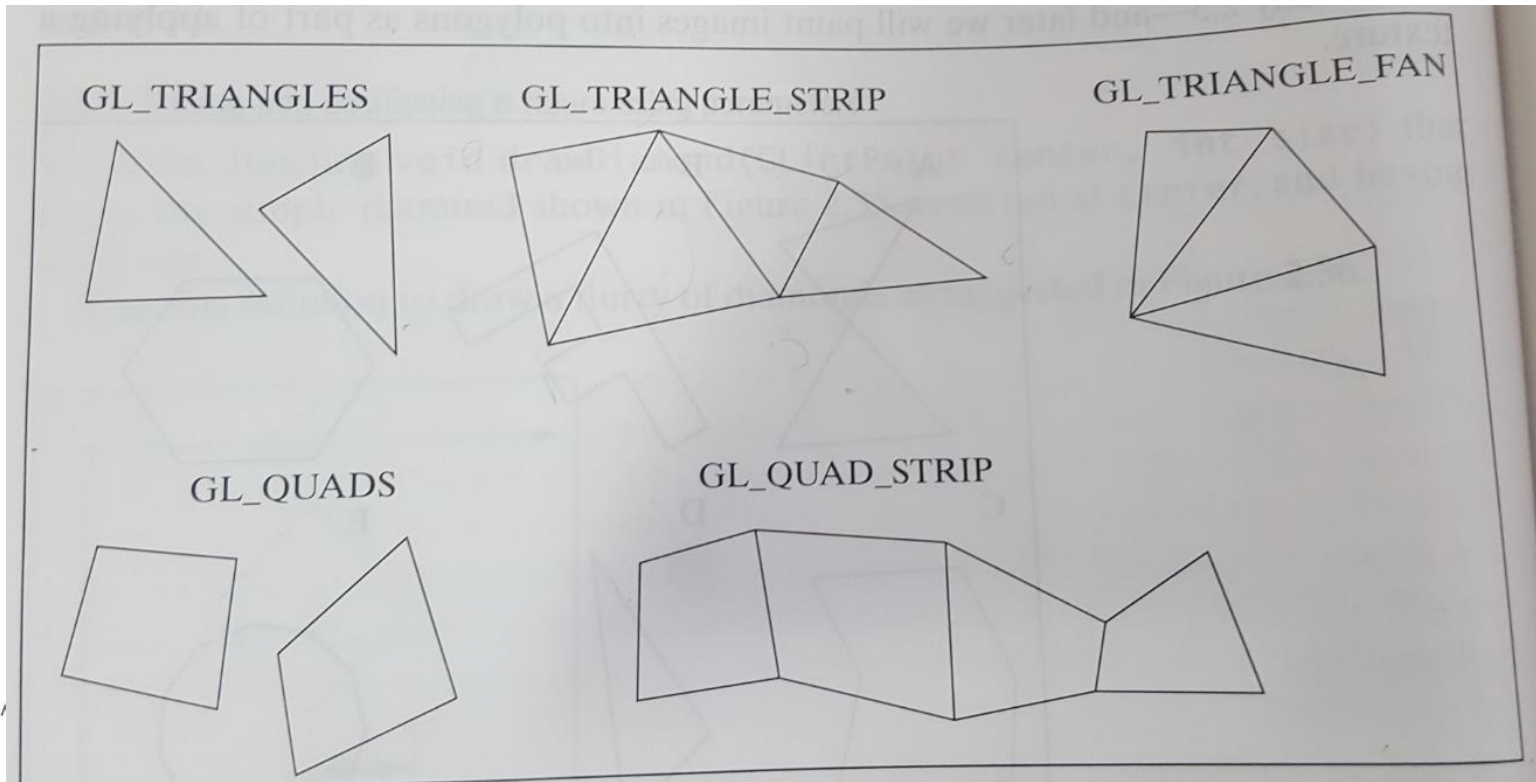
-
- **Geometric primitives** (also, called drawing primitives, graphics primitive) of OpenGL are the parts that programmers use in Lego-like manner to create objects
 - **Simplest geometric objects that the system can draw**
 - All OpenGL geometric primitives are variants of points, line segments and triangular polygons

-
- We use the terms vertex and point in a somewhat different manner. A vertex is a position in space
 - We use **vertices to specify the atomic geometric primitives that are recognized by our graphics system**
 - The simplest geometric primitive is a point in space, which is usually specified by a single vertex
 - Two vertices can specify a line segment
 - Three vertices can specify three points or two connected line segment

- **OpenGL geometric primitives**
- 1. Points (GL_POINTS)
- 2. Lines (GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP)
- 3. Polygons (GL_POLYGON)
- 4. Rectangles (GL_QUADS, GL_QUAD_STRIP)
- 5. Triangles (GL_TRIANGLES, GL_TRIANGLE_STRIP)
- ...



■ OpenGL geometric primitives



-
- **Points in OpenGL**
 - **The simplest geometric primitive is a point in space, which is usually specified by a single vertex**
 - **Each vertex is displayed at a size of at least one pixel**

■ Line drawings in OpenGL

-
- OpenGL에서의 line 그리기
 - 예)
 - `glBegin(GL_LINES);`
 - `glVertex2i(40, 100);`
 - `glVertex2i(202, 96);`
 - `glEnd();`

-
- 만일 glBegin(GL_LINES)와 glEnd() 사이에 2개 이상의 vertex가 있다면?
 - 예)
 - **glBegin(GL_LINES);**
 - **glVertex2i(10, 20);**
 - **glVertex2i(40, 20);**
 - **glVertex2i(20, 10);**
 - **glVertex2i(20, 40);**
 - **glEnd();**


-
- Line의 두께 (thickness) 조절 방법
 - glBegin(GL_LINES) 위에 아래 코드 추가
 - glLineWidth(4.0);

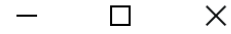
- 이전 square 예제 코드에서
- drawScene 함수를 다음과 같이 바꿈

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

    glLineWidth(5.0);
    glBegin(GL_LINES);
    glColor3f(1.0, 0.0, 0.0);
    glVertex3f(20.0, 20.0, 0.0);
    glColor3f(0.0, 1.0, 0.0);
    glVertex3f(80.0, 20.0, 0.0);
    glEnd();

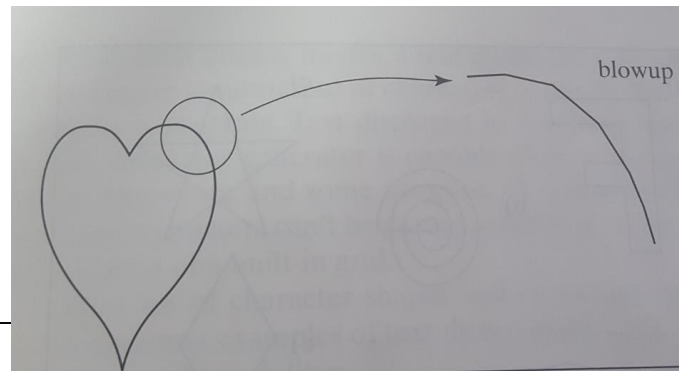
    glFlush();
}
```

 square.cpp



■ Polyline, Line strip , Line loop

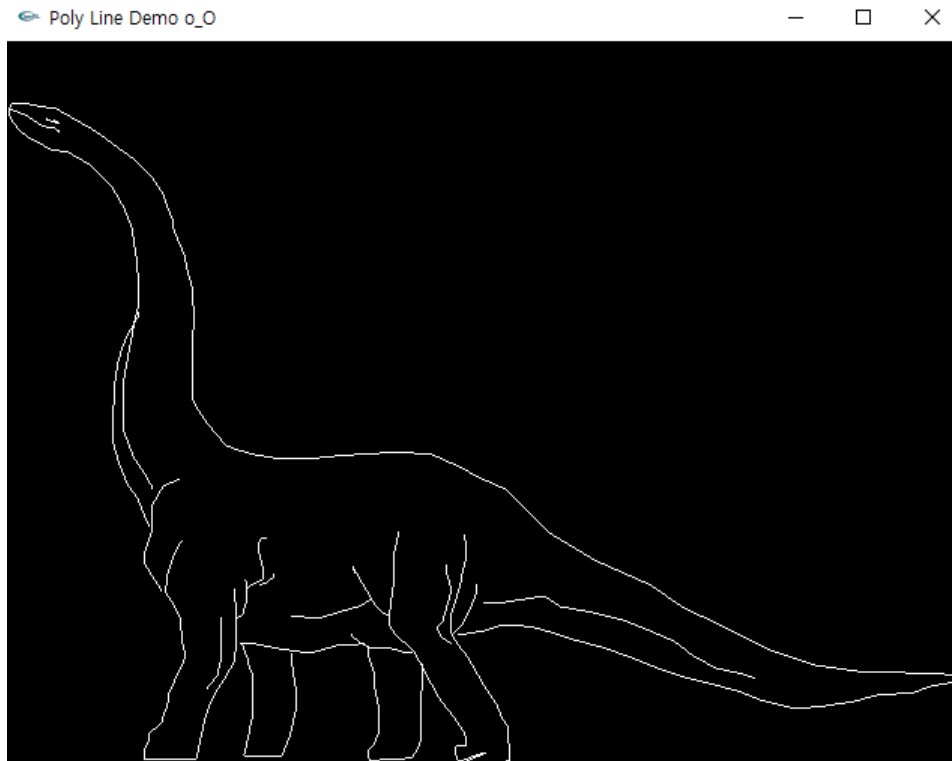
- polyline 이란?
- A polyline is a **connected sequence of straight lines**
- **Many curves can be approximated via suitable polyline**
- polyline은 직선 형태이지만 어떤 경우에는 부드러운 curve (곡선) 형태를 보일 때도 있다



-
- If we wish the polyline to be closed, we can locate the final vertex in the same place as the first, or we can use the `GL_LINE_LOOP` type, which will draw a line segment from the final vertex to the first, thus creating a closed path.

-
- 외부의 저장된 파일과 OpenGL의 line strip을 이용하여 보다 실제적인 polyline 그리기

■ poly line을 이용한 dinosaurs



외부 데이터 파일 dino.dat 를 열어보면 다음과 같이 구성되어 있다

■ <https://www.dropbox.com/s/q69t0yj7vy27jux/dino.dat?dl=0>

- 총 21개의 polyline
- 각 polyline의 (x, y) coordinate
- glVertex2i 로 vertex 위치 정함
- 정해진 vertex들을 이용해
- line strip으로 polyline 그림

```
1 21
2 29
3 32 435
4 10 439
5 4 438
6 2 433
7 4 428
8 6 425
9 10 420
10 15 416
11 21 413
12 30 408
13 42 406
14 47 403
15 56 398
16 63 391
17 71 383
18 79 369
19 84 356
20 87 337
21 89 316
22 88 302
23 86 294
24 83 278
25 79 256
26 78 235
27 79 220
28 85 204
29 94 190
```

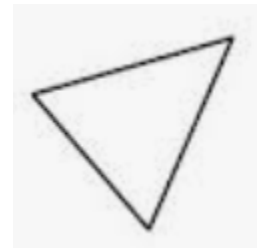
-
- 실행 방법: 현재 실행 폴더에 dino.dat 파일을 옮기고 아래 코드를 실행해 보자
 - <https://www.dropbox.com/s/26wz6qx5q6b9x6m/dinosaur.txt?dl=0>

■ Drawing Rectangles

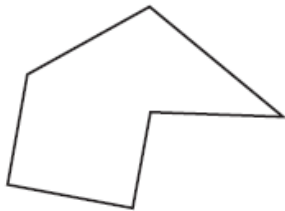
-
- Rectangle 그리기
 - Rectangle을 그리기 위해서는 두 점이 필요하다
 - 이 두 점 (x_1, y_1) , (x_2, y_2) 은 rectangle의 양 코너점을 의미한다
 - `glRecti(GLint x1, GLint y1, GLint x2, GLint y2);`

■ Polygon basics

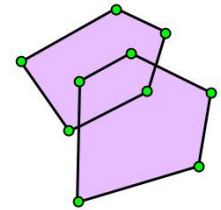
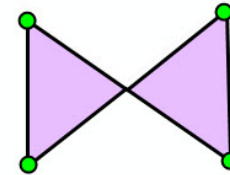
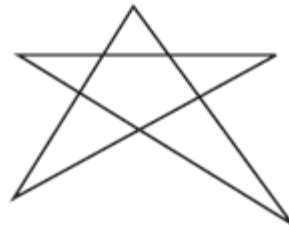
- **Polygon: 다각형**
- We reserve the name polygon for **an object that has a border that can be described by a line loop but also has a well-defined interior**
- The performance of graphics systems is characterized **by the number of polygons per second that can be rendered**
- Three properties will ensure that a polygon will be displayed correctly: **simple, convex, flat (or planar)**



- **Simple polygon (단순 다각형): a polygon is simple if no two of its edges cross each other**



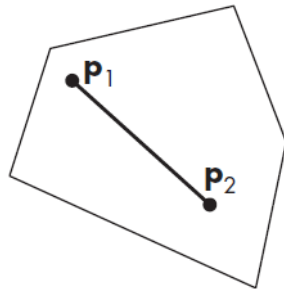
simple polygon 예



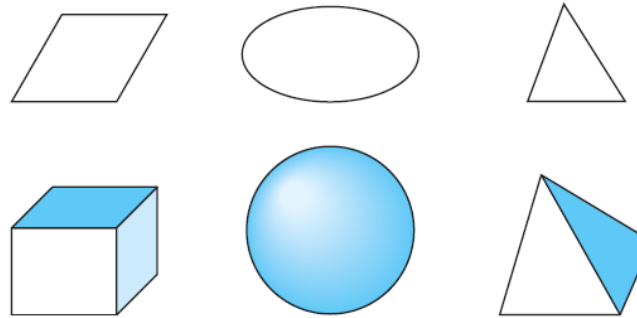
non-simple polygon 예

- **The cost of testing for a simple polygon is sufficiently high**

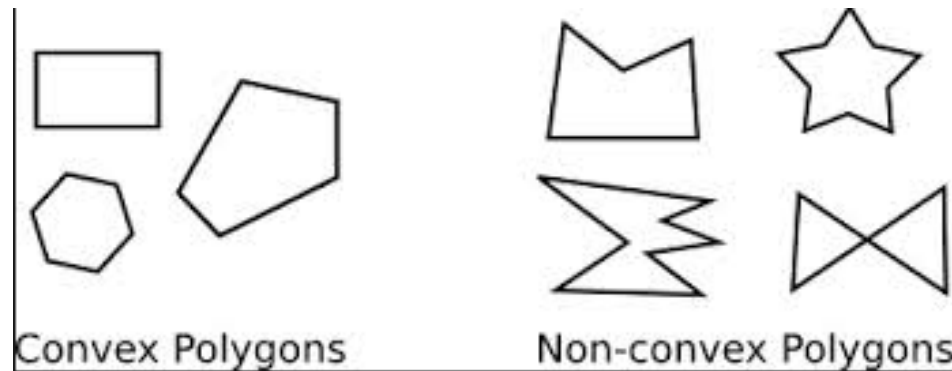
- **Convex:** An object is **convex** if all points on the line segment between any two points inside the object, or on its boundary, are inside the object
- 아래 예: p_1 and p_2 are arbitrary points inside a polygon and the entire line segment connecting them is inside the polygon



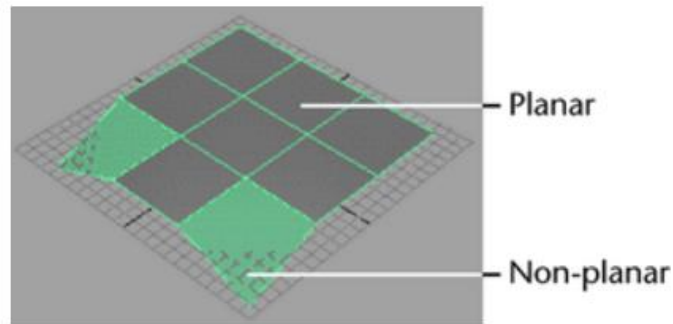
- Convex objects include triangles, tetrahedral, rectangles, circles, spheres



- Like simplicity testing, convexity testing is expensive



- In 3D, unlike 2D objects, all the vertices that specify the polygon need not lie in the same plane
- **A polygon is planar when all of its vertices in a certain plane. A triangle is always planar (flat)**



- Often, we are almost forced to use **triangles** because typical rendering algorithms are guaranteed to be correct only if the vertices form a flat convex polygon

■ OpenGL에서의 POLYGON

-
- **GL_POLYGON** draws a polygon with the vertex sequence
 - v_0, v_1, \dots, v_{n-1}
 - (n must be at least 3)
 - The programmer should ensure that when using “GL_QUADS, GL_QUAD_STRIP, GL_POLYGON” that each individual quadrilateral, or the polygon is a **plane convex** figure. Otherwise, rendering is unpredictable

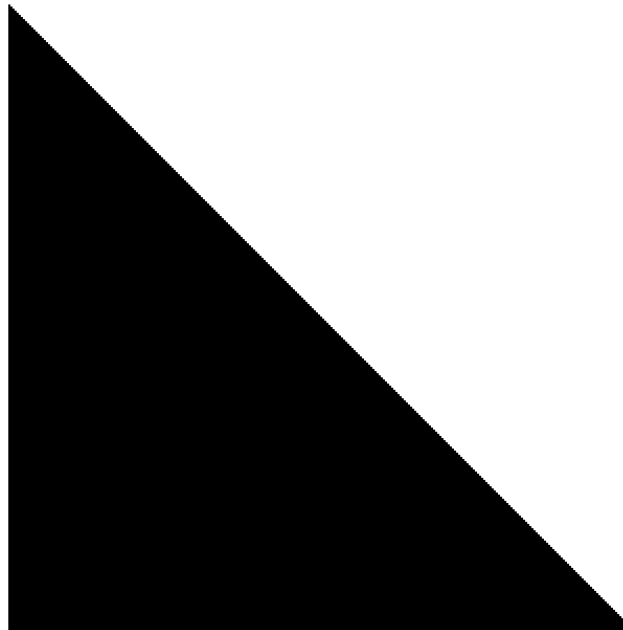
- 이전 square 예제 코드에서
- drawScene 함수를 다음과 같이 바꿈

```
glBegin(GL_POLYGON);  
glVertex3f(20.0, 20.0, 0.0);  
glVertex3f(50.0, 20.0, 0.0);  
glVertex3f(80.0, 50.0, 0.0);  
glVertex3f(80.0, 80.0, 0.0);  
glVertex3f(20.0, 80.0, 0.0);  
glEnd();
```

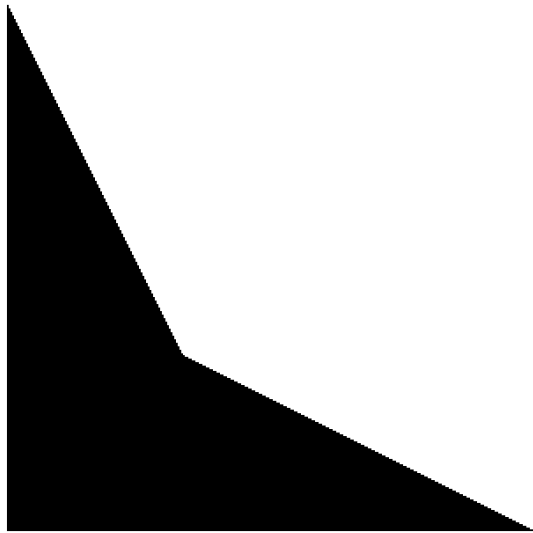


■ Non-convex polygon (quad)의 예

```
glBegin(GL_POLYGON);  
glVertex3f(80.0, 20.0, 0.0);  
glVertex3f(40.0, 40.0, 0.0);  
glVertex3f(20.0, 80.0, 0.0);  
glVertex3f(20.0, 20.0, 0.0);  
glEnd();
```



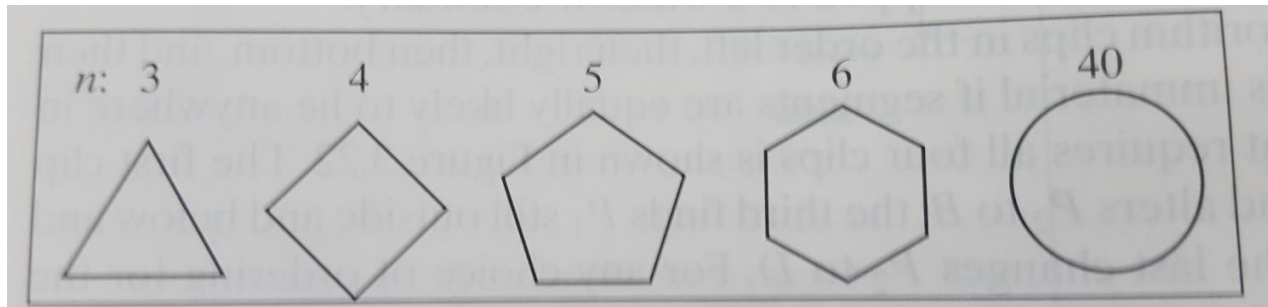
- Q: 이번에는 동일한 cycle이지만 최초 vertex만 바꾸어 봤다. 어떤 결과가 나오는가? 왜 그럴까?



```
glBegin(GL_POLYGON);  
glVertex3f(20.0, 20.0, 0.0);  
glVertex3f(80.0, 20.0, 0.0);  
glVertex3f(40.0, 40.0, 0.0);  
glVertex3f(20.0, 80.0, 0.0);  
glEnd();
```

■ Regular polygon

- **Regular polygon:** a polygon is regular if it is simple and equiangular (모든 각이 같음) and equilateral (모든 변의 길이가 같음)
- **n-gon:** 변의 길이가 n개인 regular polygon



Observation: n-gon에서 n이 커지면 원에 가까워 진다

원을 바로 그리지 않고 n을 증가시키면서 n-gon의 형태로 근사화 가능하다

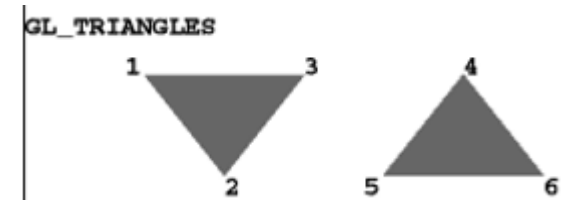
■ OpenGL에서의 TRIANGLES

■ GL_TRIANGLES

- Draws a sequence of triangles using three vertices at a time

If there are n vertices

$V_0V_1V_2, V_3V_4V_5, \dots, V_{n-3}V_{n-2}V_{n-1}$



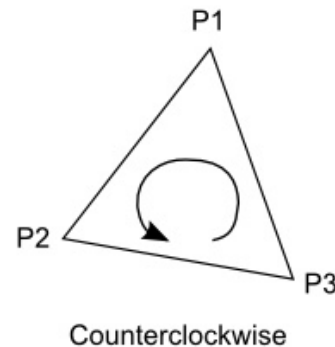
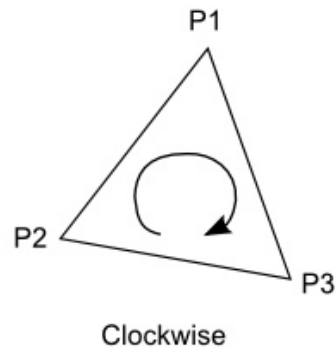
Note: if n is not a multiple of 3, the last one is ignored

By default, triangles are filled

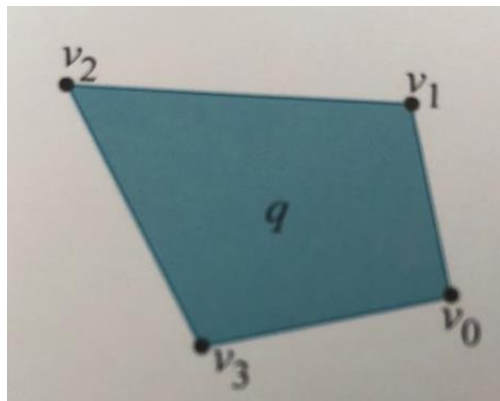
- 앞의 square 예제를 다음과 같이 바꾸어 보자

```
glBegin(GL_TRIANGLES);  
glVertex3f(10.0, 90.0, 0.0);  
glVertex3f(10.0, 10.0, 0.0);  
glVertex3f(35.0, 75.0, 0.0);  
glVertex3f(90.0, 90.0, 0.0);  
glVertex3f(80.0, 40.0, 0.0);  
glEnd();
```

- You will be curious whether the given order of vertices will affect the drawing result. 이를 **orientation** 이라 한다
- Orientation means whether a cycle goes around **clock wise (CW, 시계 방향)** or **counter-clockwise (CCW, 반시계)**



- 정의: Two orders of the vertices of a polygon are said to be **equivalent** (동일) if one can **be cyclically rotated** into the order

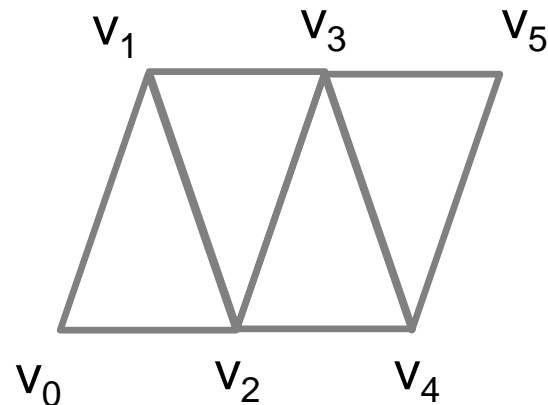


- $v_0v_1v_2v_3$, $v_1v_2v_3v_0$, $v_2v_3v_0v_1$, $v_3v_0v_1v_2$ (동일)
- $v_0v_3v_2v_1$, $v_3v_2v_1v_0$, $v_2v_1v_0v_3$, $v_1v_0v_3v_2$ (동일)

- Orientation은 왜 중요할까?
- Viewer (camera) 입장에서 특정 polygon (triangle)이 viewer에게 **앞면 (front-face)인지 후면 (back-face)인지 판단**하는 기준이 된다
- OpenGL에서는 triangle의 orientation이 viewer가 보기에 CCW (반시계 방향)이면 앞면, CW (시계 방향)이면 후면으로 판단한다
- 보다 자세한 내용은 back-face culling 부분에서 배울 예정

- **GL_TRIANGLE_STRIP**

- Draws a sequence of triangles called a triangle strip
- If there are n vertices ($v_0 v_1 v_2 v_3 v_4 v_5, \dots, v_{n-1}$)
- Draws a series of using vertices v_0, v_1, v_2 , then v_2, v_1, v_3 (note the order), then v_2, v_3, v_4 , and so on. The ordering is to ensure that the triangles are all drawn with the **same orientation**
- E.g., Say we have 6 vertices

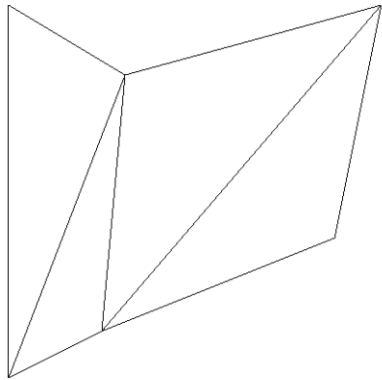


front face와 back face 모두

Boundary edges of the polygon are drawn as line segments.

시각화시에 좋음

[glPolygonMode - OpenGL 4 Reference Pages \(khronos.org\)](http://www.khronos.org/OpenGL/4/ReferencePages/glPolygonMode)



```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  
glBegin(GL_TRIANGLE_STRIP);  
glVertex3f(10.0, 90.0, 0.0);  
glVertex3f(10.0, 10.0, 0.0);  
glVertex3f(35.0, 75.0, 0.0);  
glVertex3f(30.0, 20.0, 0.0);  
glVertex3f(90.0, 90.0, 0.0);  
glVertex3f(80.0, 40.0, 0.0);  
glEnd();
```

-
- Triangle strip을 만들 때 주의해야할 점은 없을까?
 - 1. orientation (각 삼각형의 orientation)
 - 2. triangulation (삼각형 모양)

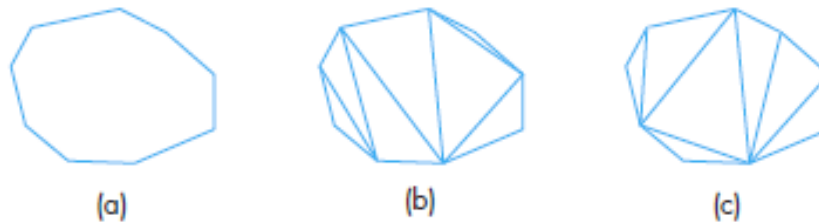
-
- **GL_TRIANGLE_FAN: draws a series of connected triangles based on triplets of vertices**
 - **$v_0v_1v_2, v_0v_2v_3, v_0v_3v_4, \dots$**

이전 square 예제 코드에서

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  
glBegin(GL_TRIANGLE_FAN);  
glVertex3f(10.0, 10.0, 0.0);  
glVertex3f(15.0, 90.0, 0.0);  
glVertex3f(55.0, 75.0, 0.0);  
glVertex3f(80.0, 30.0, 0.0);  
glVertex3f(90.0, 10.0, 0.0);  
glEnd();  
glFlush();
```

■ Triangulation (삼각 분할)

- In practice, we need to deal with more general polygons. The usual strategy is to start with a list of vertices and **generate a set of triangles consistent with the polygon** defined by the list, a process known as **triangulation**
- Quadrilateral에 대한 triangulation 예 Triangulation은 unique하지 않다



(a) Polygon (b) A triangulation (c) another triangulation

- **Long thin triangles are bad!**
- 정삼각형에 가까운 삼각형들보다 long thin 삼각형들은 일반적으로 rendering 시에 더 안좋은 영향을 미친다고 알려져 있다
- 그렇다면 앞의 (b)와 (c) 중에 더 바람직한 triangulation은?



(a)



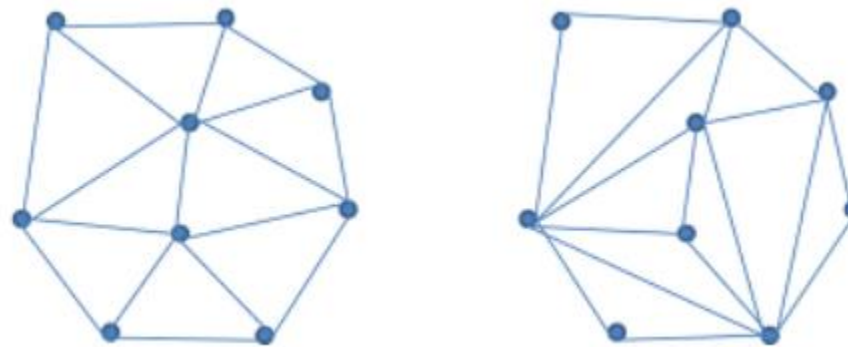
(b)



(c)

- 가장 유명한 triangulation 방법중의 하나는 Delaunay triangulation이다

- Delaunay triangulation algorithm finds a **best triangulation** in the sense that when we consider the circle determined by any triangle, **no other vertex lies in this circle**



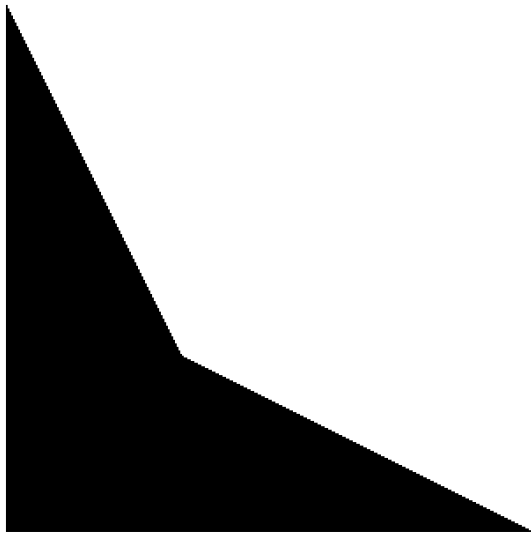
(Left) Delaunay triangulation (right) Non-Delaunay triangulation

- **We can avoid long thin triangles!**

-
- **Triangulation is a special case of the more general problem of **tessellation**, which divides a polygon into a polygonal mesh, not all of which need be triangles**
 - **General tessellation algorithms are complex, especially when the initial polygon may contain holes**

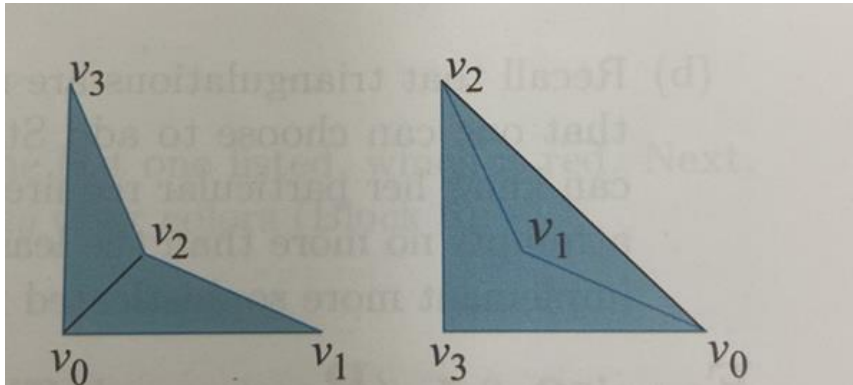
■ Revisit mystery case

■ Revisit mystery case of non-convex polygon



```
glBegin(GL_POLYGON);  
glVertex3f(20.0, 20.0, 0.0);  
glVertex3f(80.0, 20.0, 0.0);  
glVertex3f(40.0, 40.0, 0.0);  
glVertex3f(20.0, 80.0, 0.0);  
glEnd();
```

- When OpenGL is asked to draw a filled polygon P with n vertices, it renders a fan of $n-2$ triangles around the first vertex (same as `GL_TRIANGLE_FAN`)
- OpenGL renders a fan of $n-2$ triangles around the first vertex



```
glBegin(GL_POLYGON);  
glVertex3f(20.0, 20.0, 0.0);  
glVertex3f(80.0, 20.0, 0.0);  
glVertex3f(40.0, 40.0, 0.0);  
glVertex3f(20.0, 80.0, 0.0);  
glEnd();
```

```
glBegin(GL_POLYGON);  
glVertex3f(80.0, 20.0, 0.0);  
glVertex3f(40.0, 40.0, 0.0);  
glVertex3f(20.0, 80.0, 0.0);  
glVertex3f(20.0, 20.0, 0.0);  
glEnd();
```