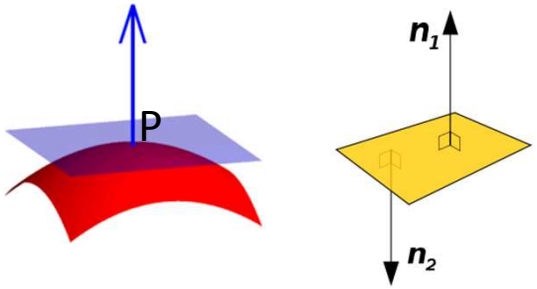
Normal vector

- The surface normal at a point P is a vector that is perpendicular to the tangent plane to that surface at P.

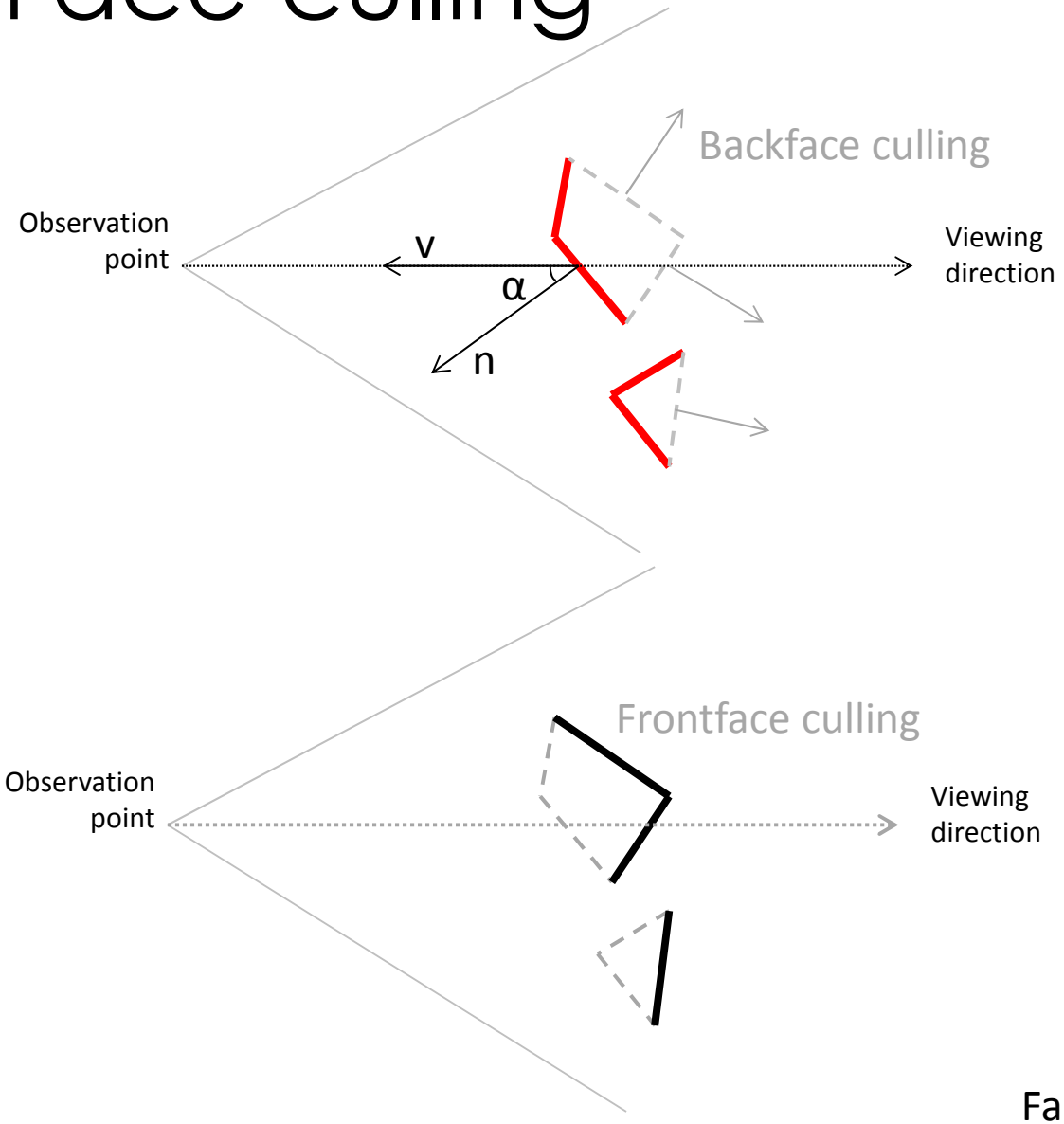


- This vector should be normalized so that its length is 1.

- Normal vectors are used:
  - For face visibility (face culling)
  - For illumination / shading

Face culling

To "cull" (verb): (a) to *select from a large quantity;* (b) to choose from



$\overline{N}.\overline{V} = |n|.|v|.\cos(\alpha)$

For backface culling:
> 0 ? ($\alpha < \pi/2$) (visible)
< 0 ? ($\alpha > \pi/2$) (not visible)

Face culling accelerates rendering!

Face culling (OpenGL)

- The culling process is performed automatically.
- However some instructions are required to configure and activate culling.

- `void glCullFace(GLenum mode);`
  - mode: GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK
  - Specifies the cull face mode.

- `void glFrontFace(GLenum mode);`
  - mode: GL_CW, GL_CCW
  - On a freshly created OpenGL Context, the default front face is GL_CCW
  - Defines which side is considered the "front" side.

- `glEnable (GL_CULL_FACE);`
  - Actives face culling

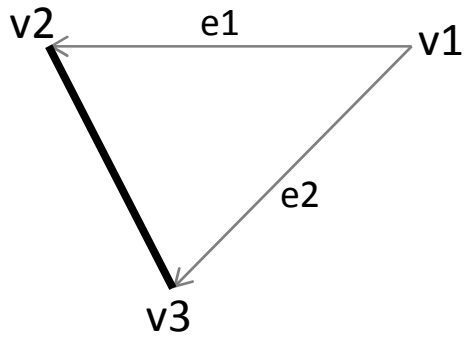
- `glDisable (GL_CULL_FACE);`
  - Deactivates face culling

Face culling (OpenGL)

The following code is equivalent to consider the polygon is facing the same direction.

```
glFrontFace(GL_CCW);
glBegin(GL_POLYGON);
        glVertex3fv(v1);
        glVertex3fv(v2);
        glVertex3fv(v3);
glEnd();
```

```
glFrontFace(GL_CW);
glBegin(GL_POLYGON);
        glVertex3fv(v1);
        glVertex3fv(v3);
        glVertex3fv(v2);
glEnd();
```




```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
```

Normals (for illumination)

Diffuse reflection for a single light source:

$$\boldsymbol{Rd} = \boldsymbol{I}.\boldsymbol{Kd}.\boldsymbol{\cos}(\theta)$$

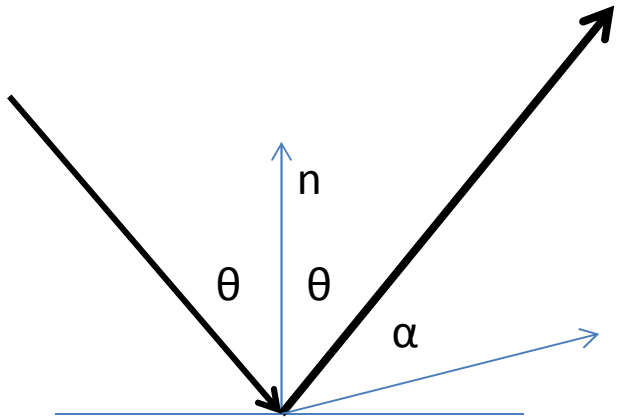
Rd : Difuselly reflected light
I : Light source intensity
Kd {0..1} : Diffuse reflectivity (depends on the material nature)
θ : is the angle between the **surface normal** and a line from the surface point to the light source.

Diffuse reflection for multiple light sources:

$$\boldsymbol{Rd} = \boldsymbol{Kd} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{I_i}\, \boldsymbol{cos}(\theta_i)$$

Ii : Light *i* source intensity
θ_i: is the angle between the **surface normal** and a line from the surface point to the *i* light source.
n : number of lights

Normals (for illumination)



Reflection (ambient, difuse, specular) for a single light source:

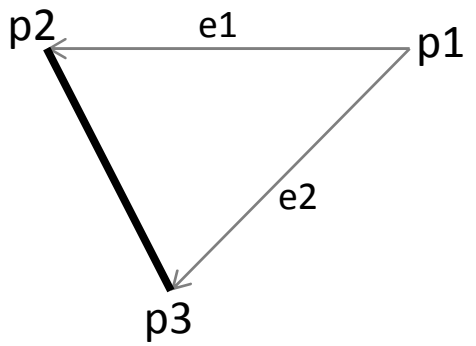
$$\mathbf{R} = K_a I_a + (K_d . \cos\theta + K_s . cos^n(\alpha)).I$$

R : Reflected light
ka, kd, ks {0..1} : ambient, difuse and specular reflectivity
Ia : global ambient intensity
I : light source intensity
Cos teta :
Cos alpha: r escalar v

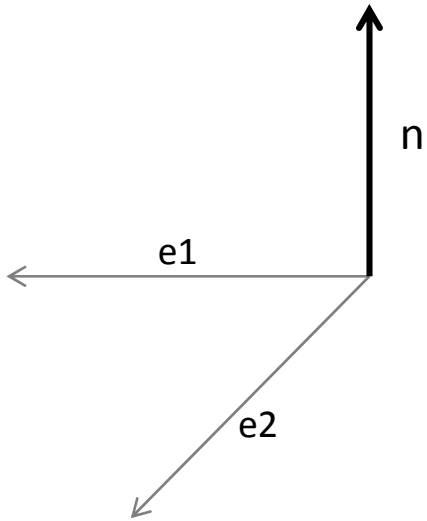
In OpenGL illumination is evaluated for each vertex -> one normal per vertex

Normal evaluation (triangle)

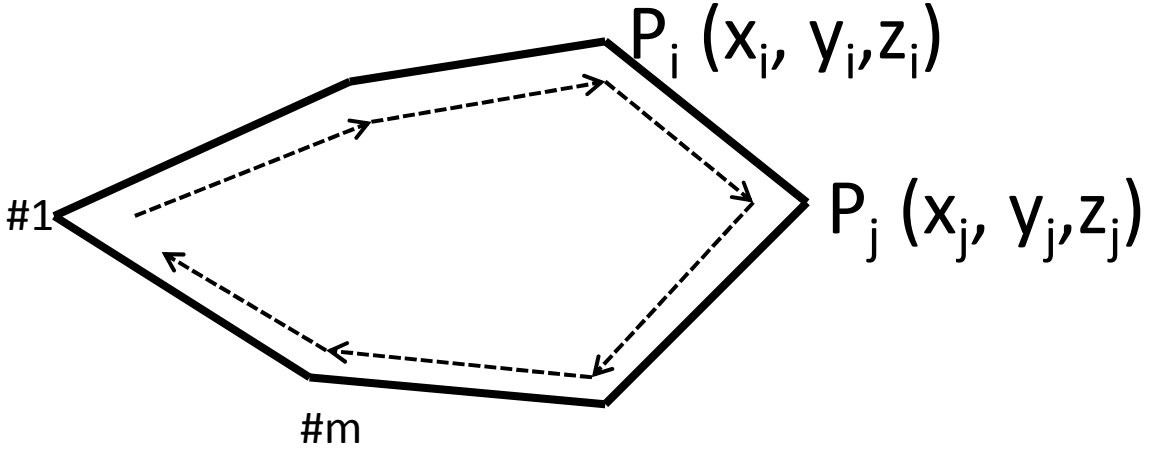
```
// p1, p2, p3: triangle vertexes
e1 = p2-p1
e2 = p3-p1
n = cross(edge1, edge2).normalize()
```




```
// vector supports: x,y,z or type float
struct vector e1, e2, n;
float l;

e1.x = p2.x - p1.x;
e1.y = p2.y - p1.y;
e1.z = p2.z - p1.z;

e2.x = p3.x - p1.x;
e2.y = p3.y - p1.y;
e3.z = p3.z - p1.z;

n.x = (e1.y * e2.z) - (e1.z * e2.y);
n.y = (e1.z * e2.x) - (e1.x * e2.z);
n.z = (e1.x * e2.y) - (e1.y * e2.x);

// Normalize (divide by root of dot product)
l = sqrt(n.x * n.x + n.y * n.y + n.z * n.z);
n.x /= l;
n.y /= l;
n.z /= l;
```



Newell's method (normal evaluation)

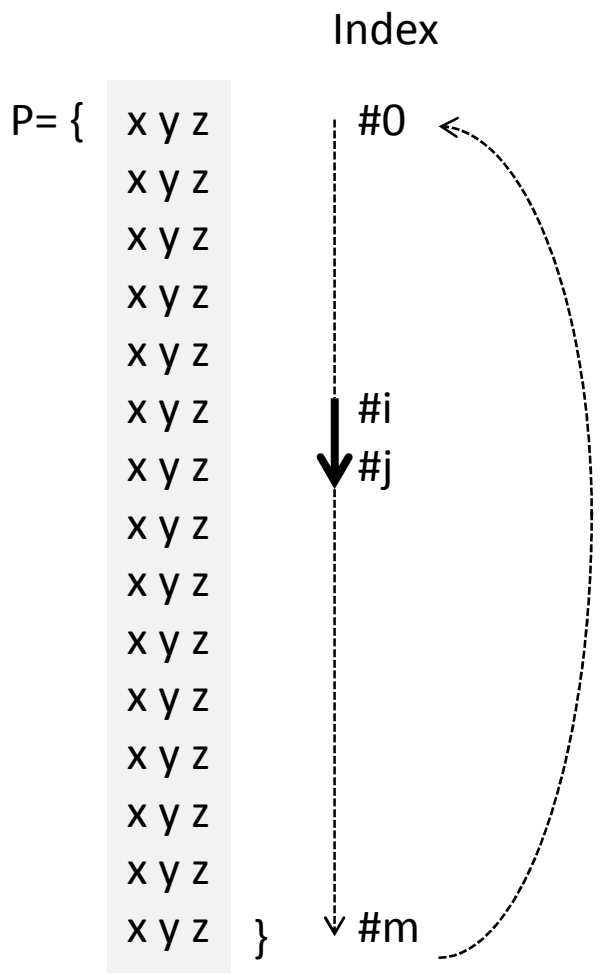


$$n_x = \sum_{\substack{i=1 \\ j=i+1 \\ i=m \\ j=1}}^{\substack{m \\ i<m}} (y_i - y_j) \cdot (z_i + z_j)$$

$$n_y = \sum_{\substack{i=1 \\ j=i+1 \\ i=m \\ j=1}}^{\substack{m \\ i<m}} (z_i - z_j) \cdot (x_i + x_j)$$

$$n_z = \sum_{\substack{i=1 \\ j=i+1 \\ i=m \\ j=1}}^{\substack{m \\ i<m}} (x_i - x_j) \cdot (y_i + y_j)$$

Newell's method (normal evaluation)



Example for n_x:

$$n_x = (y0-y1).(z0+z1) + (y1-y2).(z1+z2) + (y2-y3).(z2+z3) + \ldots + (yi-yj).(zi+zj) + \ldots + (ym-y0).(zm+z0)$$

Newell's method (normal evaluation)

```
//pseudo-code

Vector getSurfaceNormal (Input polygon) {

	Vertex normal = (0, 0, 0);

	for i = 0; i < polygon.size; i++) {

		Vertex current = polygon.verts[i] // { 0… m }
		Vertex next = polygon.verts[(i+1) % polygon.size] // { 1 … m,1 }

		normal.x += ((current.y - next.y) * (current.z + next.z));
		normal.y += ((current.z - next.z) * (current.x + next.x));
		normal.z += ((current.x - next.x) * (current.y + next.y));

	}

	return normalize(normal);
}
```

Setting normals in OpenGL

```
// Create a triangle and provide the normal vector
// shared by all vertexes
glBegin(GL_TRIANGLES);
      glNormal3f(n.x, n.y, n.z);
      glVertex3f(p1.x, p1.y, p1.z);
      glVertex3f(p2.x, p2.y, p2.z);
      glVertex3f(p3.x, p3.y, p3.z);
glEnd();
```

```
// Create a polygon. Each vertex has its own normal
glBegin(GL_POLYGON);
      glNormal3fv(n1);
      glVertex3fv(v1);
      glNormal3fv(n2);
      glVertex3fv(v2);
      glNormal3fv(n3);
      glVertex3fv(v3);
      glNormal3fv(n4);
      glVertex3fv(v4);
      glNormal3fv(n5);
      glVertex3fv(v5);
glEnd();
```

Used In
smooth/gouraud shading!