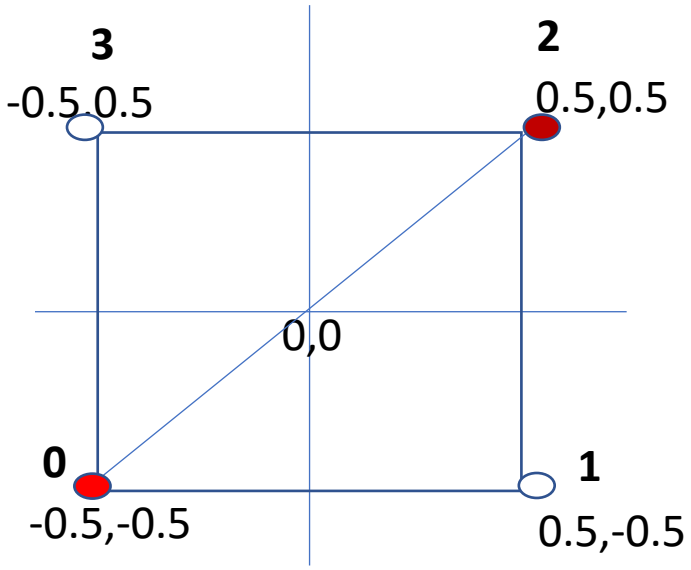


Si considera per i vertici di un triangolo un ordinamento **in senso antiorario** (importante per la visibilità delle facce)

```
float vertices[] = {
// posizioni
//I Triangolo
-0.5f, -0.5f, 0.0f,  //Vertice in basso a sinistra
  0.5f, -0.5f, 0.0f, // Vertice in basso a destra
 0.5f,  0.5f, 0.0f , // Vertice in alto a destra
//II Triangolo
 0.5f,0.5f,0.0f,      //Vertice in alto a destra
-0.5f,0.5f,0.0f,      // Vertice in alto a sinistra
-0.5f,-0.5f,0.0f      // Vertice in basso a sinistra
};
```

```
float colori[] = {
// colori
 1.0f,0.0f,0.0f,1.0f,
 1.0f,1.0f,1.0f,1.0f,
 0.5f,0.0f,0.0f,1.0f,
 0.5f,0.0f,0.0f,1.0f,
 1.0f,1.0f,1.0f,1.0f,
 1.0f,0.0f,0.0f,1.0f
};
```

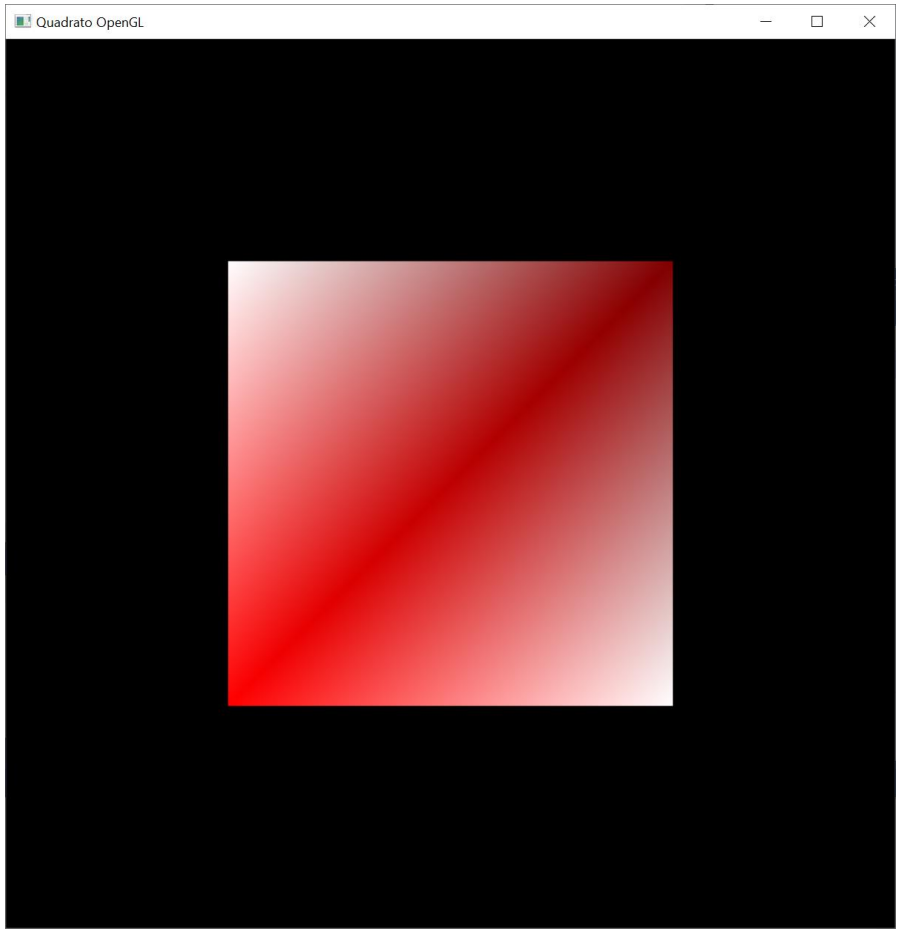
```
//Genero un VAO
glGenVertexArrays(1, &VAO);
//Ne faccio il bind (lo collego, lo attivo)
glBindVertexArray(VAO);
//AL suo interno genero un VBO per l'attributo posizione dei vertici
glGenBuffers(1, &VBO);
//Ne faccio il bind (lo collego, lo attivo, assegnandogli il tipo GL_ARRAY_BUFFER)
glBindBuffer(GL_ARRAY_BUFFER, VBO);
//Carico i dati vertices sulla GPU
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// Configurazione dell'attributo posizione: informo il vertex shader su: dove trova le informazioni sulle posizioni e
come le deve leggere
//dal buffer caricato sulla GPU
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
//All'interno del VAO genero un VBO per i colori
glGenBuffers(1, &VBO_COLORI);
//Ne faccio il bind (lo collego, lo attivo, assegnandogli il tipo GL_ARRAY_BUFFER)
glBindBuffer(GL_ARRAY_BUFFER, VBO_COLORI);
//Carico i dati vertices sulla GPU
glBufferData(GL_ARRAY_BUFFER, sizeof(colori), colori, GL_STATIC_DRAW);
// Configurazione dell'attributo Colore: informo il vertex shader su: dove trova le informazioni sulle posizioni e come
le deve leggere dal buffer caricato sulla GPU
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)0);
glEnableVertexAttribArray(1);
glBindVertexArray(0);  //SCOLLEGO IL VAO
```

```
#version 420 core


layout(location = 0) in vec3 aPos;
layout(location =1 ) in vec4 Color;

out vec4 colore_frag;


void main(){

   gl_Position = vec4(aPos.x,aPos.y,aPos.z,1.0);
   colore_frag= Color;
}
```

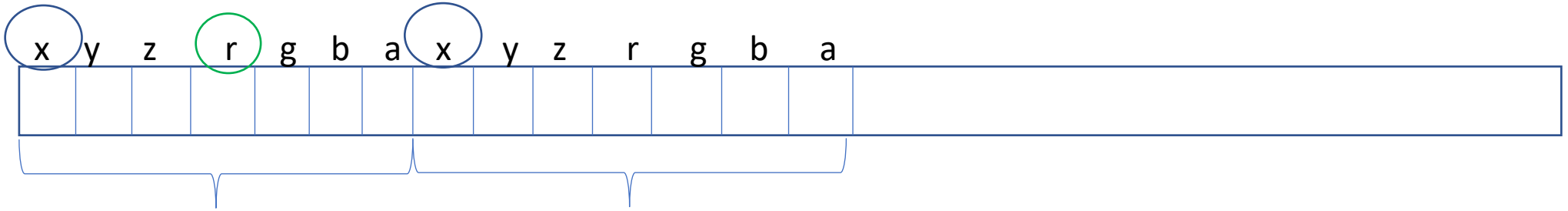
```
#version 420 core

// Viene preso in input il colore interpolato da assegnare al
//frammento e viene passato in output inalterato


in vec4 colore_frag;
out vec4 color;

void main(){
//Viene assegnato lo stesso colore ad ogni pixel
            color = colore_frag;

}
```

Nella callback function all'evento disegno (nel nostro caso Drawscene), bisogna scrivere il codice di renderizzazione

```
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 6);

glutSwapBuffers();
```



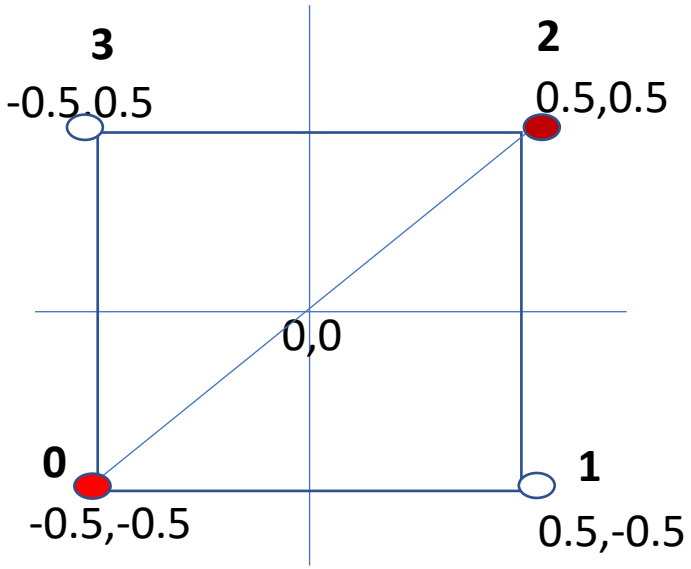
```
float vertices[] = {
// posizioni          COLORI
//I Triangolo
-0.5f, -0.5f, 0.0f, 1.0f,0.0f,0.0f,1.0f, //Vertice in basso a sinistra
0.5f, -0.5f, 0.0f, 1.0f,1.0f,1.0f,1.0f,  // Vertice in basso a destra
0.5f,  0.5f, 0.0f , 0.5f,0.0f,0.0f,1.0f, // Vertice in alto a destra
//II Triangolo
0.5f,0.5f,0.0f, 0.5f,0.0f,0.0f,1.0f,    //Vertice in alto a destra
-0.5f,0.5f,0.0f, 1.0f,1.0f,1.0f,1.0f,   // Vertice in alto a sinistra
-0.5f,-0.5f,0.0f, 1.0f,0.0f,0.0f,1.0f    // Vertice in basso a sinistra
};
```




```
glGenVertexArrays(1, &VAO);
//Ne faccio il bind (lo collego, lo attivo)
glBindVertexArray(VAO);
//Al suo interno genero un VBO
glGenBuffers(1, &VBO);
//Ne faccio il bind (lo collego, lo attivo, assegnandogli il tipo GL_ARRAY_BUFFER)
glBindBuffer(GL_ARRAY_BUFFER, VBO);
//Carico i dati vertices sulla GPU
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
//Le coordinate dei vertici (attributo 0) si trovano a stride 7 * sizeof(float) a partire dalla posizione 0 del buffer e sono float di 3 com
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 7 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
//I colori dei vert.(attributo 1)si trovano a stride 7 * sizeof(float) a partire dalla posizione 3*sizeof(float) del buffer,sono float 4 com
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 7 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
//Scollego il VAO (utilizzando argomento 0 nella chiamata glBindVertexArray.)
glBindVertexArray(0);
```

Ridondanza di vertici

```
float vertices[] = {
// posizioni
//I Triangolo
-0.5f, -0.5f, 0.0f,  //Vertice in basso a sinistra
  0.5f, -0.5f, 0.0f, // Vertice in basso a destra
 0.5f,  0.5f, 0.0f , // Vertice in alto a destra
//II Triangolo
 0.5f,0.5f,0.0f,      //Vertice in alto a destra
-0.5f,0.5f,0.0f,      // Vertice in alto a sinistra
-0.5f,-0.5f,0.0f      // Vertice in basso a sinistra
};
```




```
float vertices[] = {
// posizioni
-0.5f, -0.5f, 0.0f,  // vertice in basso a sinistra
 -0.5f, 0.5f, 0.0f  // vertice in basso a destra
  0.5f,0.5,0.0f,    // vertice in alto a destra
 0.5f, -0.5f, 0.0f,  // vertice in alto a sinistra
};

unsigned int indices[]=
{ 0,1,2, //primo triangolo
   2,3,0, // secondo triangolo
};
```

EBO (ELEMENT BUFFER OBJECT)

```
glGenVertexArrays(1, &VAO);
//Ne faccio il bind (lo collego, lo attivo)
glBindVertexArray(VAO);

//AL suo interno genero un VBO per l'attributo posizione
glGenBuffers(1, &VBO);
//Ne faccio il bind (lo collego, lo attivo, assegnandogli il tipo GL_ARRAY_BUFFER)
glBindBuffer(GL_ARRAY_BUFFER, VBO);
//Carico i dati vertices sulla GPU
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// Configurazione dell'attributo posizione: informo il vertex shader su: dove trova le informazioni sulle posizioni e come le
deve leggere
//dal buffer caricato sulla GPU
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

//All'interno del VAO  genero un VBO_COLORI per l'attributo colore
glGenBuffers(1, &VBO_COLORI);
//Ne faccio il bind (lo collego, lo attivo, assegnandogli il tipo GL_ARRAY_BUFFER)
glBindBuffer(GL_ARRAY_BUFFER, VBO_COLORI);
//Carico i dati vertices sulla GPU
glBufferData(GL_ARRAY_BUFFER, sizeof(colori), colori, GL_STATIC_DRAW);

// Configurazione dell'attributo posizione: informo il vertex shader su: dove trova le informazioni sulle posizioni e come le
deve leggere //dal buffer caricato sulla GPU
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)0);
glEnableVertexAttribArray(1);

//All'interno del VAO  genero un EBO (ELEMENT BUFFER OBJECT), buffer di tipo GL_ELEMENT_ARRAY_BUFFER, dove vengono caricate le
informazioni su come i vertici devono essere collegati tra loro per ottenere la geometria desiderata.
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

glBindVertexArray(0);
```

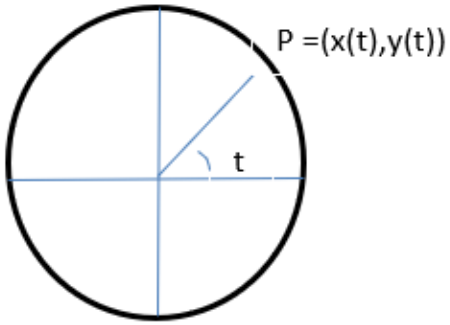
Nella callback function all'evento disegno (nel nostro caso Drawscene) sostituire la chiamata per il rendering glDrawArrays con

```
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
glutSwapBuffers();
```

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

Disegnare una circonferenza:

Circonferenza di centro l'origine e raggio r



$$C(t) = \begin{cases} x(t) = r\,cos(\,t) \\ y(t) = r\,sin(\,t) \end{cases} \qquad t \in [0,2\pi]$$

Circonferenza di centro C=(cx,cy) e raggio r

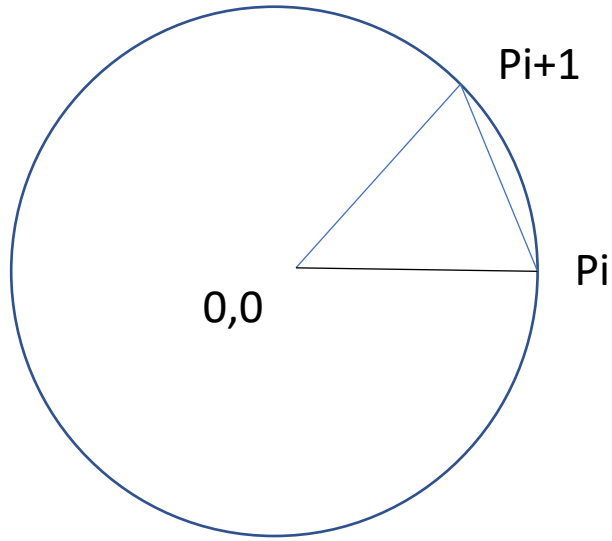
$$C(t) = \begin{cases} x(t) = r\,cos(t) + cx \\ y(t) = r\,\sin(t) + cy \end{cases} \qquad t \in [0, 2\,\pi]$$

```
nPoints = Numero dei punti sul cerchio

float stepA = (2 * PI) / nPoints;
```

Coordinate dell'i-esimo punto sulla circonferenza con centro l'origine e raggio r

t= (double)i * stepA

```
x(i)= cos(t) * raggiox;

y(i)= sin(t) * raggiox
```



Luna

$$C(t)=\begin{cases} x(t)= 3*\sin(t); \\ y(t)=0.5-\cos(2t)-\cos(t); \end{cases} \qquad t \in [0,2\pi]$$

Cuore

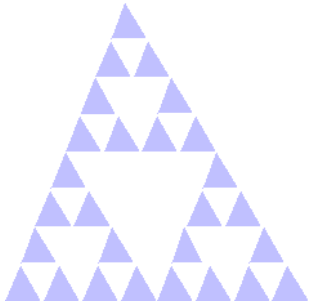
$$C(t)= \begin{cases} x(t)=16*(\sin^3 t); \\ y(t)=13*\cos(t)-5*\cos(2t)-2*\cos(3t)-\cos(4t); \end{cases} \qquad t \in [0,2\pi]$$

Farfalla

$$C(t)=\begin{cases} x(t)=\sin(t)*(\exp(\cos(t)-2*\cos(4t))+\sin^5(t/12); \\ y(t)=\cos(t)*(\exp(\cos(t)-2*\cos(4t))-\sin^5(t/12); \end{cases} \qquad t \in [0,2\pi]$$

Sierpinski gasket.

- E’ una costellazione di punti.
- Ogni punto viene definito in maniera procedurale dal precedente.
- La regola procedurale è molto semplice:
- La configurazione finale è un frattale.



le posizioni dei punti, (x0 , y0) ,(x1 , y1) ,...,
vengono determinate da un semplice algoritmo.

1. Scegliamo tre punti del piano che individuano un certo triangolo

$T0=(x0 , y0) , T1 = (x1 , y1), T2 = (x2 , y2)$

2. Scegliamo il punto iniziale p0 , scegliendolo a caso da uno dei tre vertici T0 ,T1 ,T2 .

Adesso iteriamo i seguenti passi finchè la nostra configurazione si riempie in maniera soddisfacente:

3. Scegliamo a caso uno dei tre punti T_0 ,T_1 ,T_2 e chiamiamolo T .

4. Costruiamo il prossimo punto p_k come il punto medio tra T e il precedente punto trovato p_{k-1} . Cioè:

p_k = punto medio tra p_{k-1} e T.

5. Disegna p_k .

glPolygonMode(face, mode): controlla la rasterizzazione dei poligoni.

Face descrive a quali facce del poligono deve essere applicato la modalità di rasterizzazione, individuata da mode: GL_FRONT_AND_BACK per applicare la modalità di rasterizzazione sia alle facce davanti che dietro.

Mode può assumere i seguenti valori:

GL_POINT

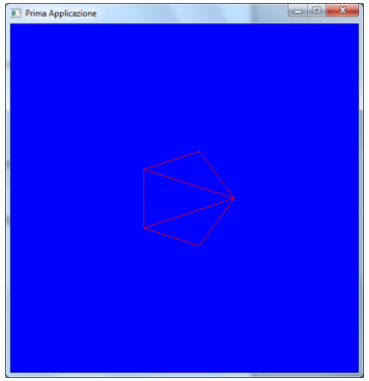
I vertici del poligono che sono contrassegnati come l'inizio di un bordo di confine sono disegnati come punti.

GL_LINE :

Vengono disegnati gli spigoli di contorno del poligono mediante dei segmenti.

GL_FILL

Vengono colorati i punti interni del poligono.



glDrawArrays – disegna primitive a partire da un buffer di dati:

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

Argomenti di input:

mode

Specifica il tipo di primitive che si vogliono renderizzare:
GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES,

first

Specifica l'indice del buffer da cui partire per fare il disegno

count

Specifica il numero di indici da renderizzare.

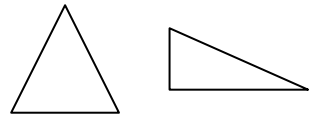
GL_POINTS: Visualizzazione per punti

GL_LINES: se sono specificati più di due vertici, essi sono considerati a coppia e viene disegnata una linea separata per ogni coppia.

GL_LINES_STRIP: i punti vengono come vertici di una polyline (collezione di segmenti, tali che l'estremo finale di un segmento è congiunto con l'estremo iniziale del segmento che segue)

Se si vuole collegare l'ultimo punto della polyline con il primo punto, per trasformare la polyline in un poligono, basta sostituire **GL_LINE_STRIP con GL_LINE_LOOP**

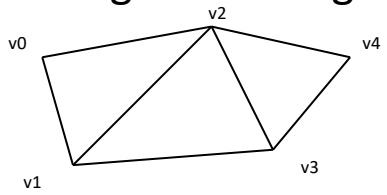
GL_TRIANGLES: prende i vertici nella lista tre alla volta e disegna triangoli separati per ogni terna di vertici riempiti del colore corrente.



GL_TRIANGLE_STRIP: disegna una serie di triangoli basati su terne di vertici:

(v0,v1,v2), (v2,v1,v3), (v2,v3,v4), ETC

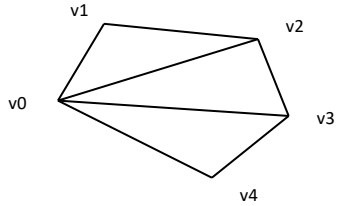
Ogni vertice aggiuntivo determina un nuovo triangolo. I triangoli sono riempiti del colore corrente.



GL_TRIANGLE_FAN: disegna una serie di triangoli connessi basati su terne di vertici:

(v0,v1,v2), (v0,v2,v3), (v0,v3,v4), ETC.

I triangoli sono riempiti del colore corrente.



Interazioni con il mouse e la tastiera

- La grafica interattiva porta l'utente a controllare il flusso del programma puntando e facendo click con il mouse, premendo diversi tasti sulla tastiera.
- La posizione del mouse al momento del click o l'identificazione del tasto che è stato premuto vengono rese disponibili all'applicazione grafica e appropriatamente elaborate.
- Quando l'utente preme o rilascia un bottone del mouse o preme un tasto della tastiera, avviene un evento.

Usando l'OpenGL Utility Toolkit (GLUT) il programmatore può registrare una funzione di risposta associata a ciascuno di questi eventi usando i seguenti comandi:

glutMouseFunc(myMouse) che registra myMouse( ) come la funzione di risposta che viene eseguita quando il bottone del mouse viene premuto o rilasciato.

glutMotionFunction(myMovedMouse), che registra myMovedMouse( ) come la funzione di risposta che viene eseguita quando il mouse viene mosso mentre uno dei bottoni è premuto.

glutKeyboardFunc(myKeyboard) che registra myKeyboard( ) come la funzione di risposta che viene eseguita quando un tasto della tastiera viene premuto.

Nelle call-back function è necessario il comando

glutPostRedisplay();

che forza l'evento disegno, in questo modo la funzione drawScene viene ridisegnata con il parametri aggiornati.

Interazione con il mouse

Come vengono spediti all'applicazione i dati che riguardano il mouse?

E' necessario costruire la funzione di risposta myMouse( ) che prende quattro parametri in input

```
void myMouse(int button, int state, int x, int y);
```

Allora, quando avviene un evento del mouse, il sistema chiama la funzione di risposta fornendole i valori per questi parametri.

Il valore di button sarà uno dei seguenti:

GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON.

Il valore di state sarà uno dei seguenti:

GLUT_UP
GLUT_DOWN.

I valori di x ed y individuano la posizione del mouse al momento dell'evento.

Attenzione: il valore di x è il numero di pixel dalla sinistra della finestra, ma y è il numero di pixel dal top della finestra verso il basso.

Rotellina del mouse

La rotella del mouse non è contemplata nelle specifiche di GLUT.

Freeglut include la possibilità di registrare una funzione di callback per la rotellina del mouse.

La funzione di registrazione è:

glutMouseWheelFunc (myMouseWheel)

dove la funzione myMouseWheel è definita come:

void myMouseWheel (int wheel, int direction, int x, int y)

Parametri:

wheel: il numero di rotelline, se il mouse ha solo una rotellina questo sarà pari a zero.
Direzione: un valore di +/- 1 che indica la direzione di movimento della rotellina
•X, Y: le coordinate del mouse sulla finestra

Questa funzione viene chiamata quando la rotellina del mouse si muove per ogni tacca. Nota: la rotella del mouse funziona ancora come il tasto centrale, ma questi eventi sono trattati come eventi di pressione regolare e rilascio del pulsante del mouse.

Gestione del mouse in movimento con il tasto sinistro premuto

Nel main, viene registrata la funzione di risposta “ mouseMotion” (scritta dall’utente) all’evento mouse in movimento

```
glutMotionFunc(mouseMotion);
```

La funzione mouseMotion ha la seguente forma:

```
void mouseMotion(int x, int y)
```

dove x, y sono le coordinate occupate dal mouse durante il suo movimento con il tasto sinistro premuto.

Interazione con la tastiera

La pressione di un tasto sulla tastiera determina un evento.

La funzione di risposta viene associata a questo evento nel seguente modo:

glutKeyboardFunc(myKeyboard);

myKeyboard è una funzione scritta in C dall'utente dove vengono descritte le azioni che dovranno essere eseguite quando viene premuto un tasto.

La funzione myKeyboard deve esser di questo tipo:

void myKeyboard(unsigned char key, int x, int y)

Il valore di key è il valore ASCII del tasto premuto.

I valori di x ed y rappresentano la posizione del mouse al momento in cui è avvenuto l'evento,

Utilizzo Tasti speciali sulla Tastiera

Per poter usare eventi legati alla pressione di tasti speciali sulla funzione si usa l'evento

glutSpecialFunc(keyspecial):

dove la funzione keyspecial è la call-back function con cui il programmatore gestisce le azioni da svolgere nel caso in cui l'utente interagisce con i tasti speciali:

La funzione keyspecial è del tipo:

void keyspecial(int key, int x, int y)

dove key può assumere I seguenti valori:

```
GLUT_KEY_F1 GLUT_KEY_F2 GLUT_KEY_F3 GLUT_KEY_F4

GLUT_KEY_F5 GLUT_KEY_F6 GLUT_KEY_F7 GLUT_KEY_F8

GLUT_KEY_F9 GLUT_KEY_F10 GLUT_KEY_F11

GLUT_KEY_F12

GLUT_KEY_LEFT
```

GLUT_KEY_UP Up directional key.

GLUT_KEY_RIGHT Right directional key.

GLUT_KEY_DOWN Down directional key.

GLUT_KEY_PAGE_UP Page up directional key.

GLUT_KEY_PAGE_DOWN Page down directional key.

GLUT_KEY_HOME Home directional key.

GLUT_KEY_END End directional key.

GLUT_KEY_INSERT Inset directional key.

```
void keyspecial(int key, int x , int y)
{
    switch(key)
 {
        case GLUT_KEY_F1:
       glClearColor (1.0,0.0,0.0,0.0);
       glutPostRedisplay();
       break;
       case GLUT_KEY_F2:
       glClearColor (1.0,1.0,0.0,0.0);
       glutPostRedisplay();
       break;
       case GLUT_KEY_F3:
       glClearColor (0.0,0.0,0.0,0.0);
       glutPostRedisplay();
       break;
       case GLUT_KEY_F4:
       glClearColor (1.0,0.0,1.0,0.0);
       glutPostRedisplay();
       break;
 }
}
```

Evento Idle.

È l'evento di "oziosità" e si verifica quando non avvengono altri eventi durante l'esecuzione del main loop.

Un modo per gestire l'evento idle è l'utilizzo della funzione
glutTimerFunc (unsigned int msecs , void (*func)(int value), value);

che esegue la funzione specificata nel secondo parametro dopo almeno msecs millisecondi, passando il valore value.

Nella funzione di risposta alla all'evento disegno, generalmente si inserisce l'istruzione

glClearColor(GL_BUFFER_BIT);

che inizializza il frame buffer al colore definito mediante l'istruzione

```
glClearColor(r,g,b,a);
```

dove (r,g,b,a) sono le componenti del colore con cui inizializziamo il frame buffer che rappresenta il colore della finestra grafica.

Blending in Opengl

Il Blending in OpenGL è una tecnica per implementare la trasparenza degli oggetti.

La trasparenza riguarda gli oggetti (o parti di essi) che non hanno un colore solido, ma si presentano con un colore dato dalla combinazione di colori dell'oggetto stesso e di qualsiasi altro oggetto dietro di esso con intensità variabile.

Una finestra di vetro colorato è un oggetto trasparente; il vetro ha un colore tutto suo, ma il colore risultante con cui si presenta mescola i colori di tutti gli oggetti dietro il vetro.

Gli oggetti trasparenti possono essere

completamente trasparenti (lasciano passare solamente il colore degli oggetti che gli stanno dietro) o

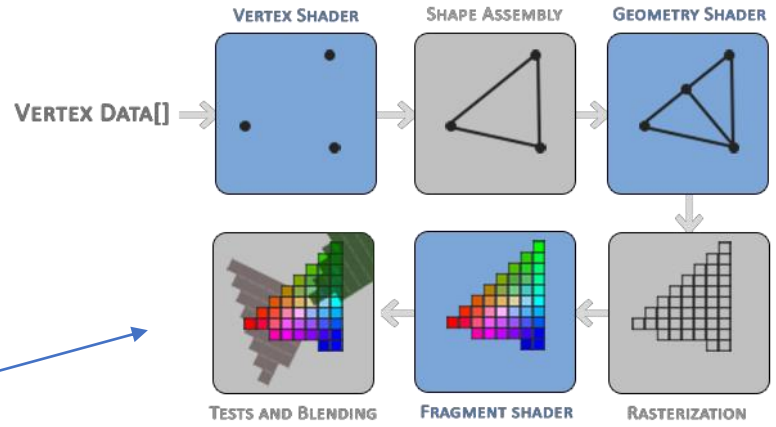
parzialmente trasparenti (lasciano passare il loro colore mescolato ai colori dell'oggetto che gli sta dietro.

La quantità di trasparenza di un oggetto è definita dal valore alfa del suo colore, quarta componente del colore.

ad alfa=1 corrisponde trasparenza nulla.

Un valore alfa di 0.0 comporta la trasparenza completa dell'oggetto.

Un valore alfa di 0.5 indica che il colore dell'oggetto è costituito dal 50% del suo colore e dal 50% dei colori dietro l'oggetto.



- Nel sottosistema raster della pipeline grafica, dopo che è stato eseguito il Fragment Shader che assegna un colore a tutti i frammenti, l'oggetto finale passerà attraverso un ulteriore stadio che chiamiamo *alfa-test* e *blending*.
- Questa fase controlla anche i valori di alfa (i valori alfa definiscono l'opacità di un oggetto) e miscela gli oggetti di conseguenza.

Per rendere le immagini con diversi livelli di trasparenza dobbiamo abilitare il Blending, abilitando GL_BLEND:

glEnable(GL_BLEND);

Una volta abilitato il blending, bisogna specificare come deve avvenire il blending.

Il Blending in OpenGL avviene con la seguente equazione:

$$C_{result} = C_{source} * F_{source} + C_{destination} * F_{destination}$$

C_{source} = colore di output del fragment shader.

$C_{destination}$ = colore attualmente memorizzato nel frame color buffer

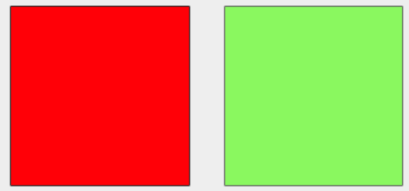
F_{source} = valore alfa del colore di output del fragment shader

$F_{destination}$ = valore alfa del colore attualmente memorizzato nel frame color buffer.

Dopo che il fragment shader è stato eseguito, l' equazione di blending viene eseguita tra l'output del colore del frammento generato dal fragmente shader e con tutto ciò che è attualmente memorizzato nel buffer dei colori.

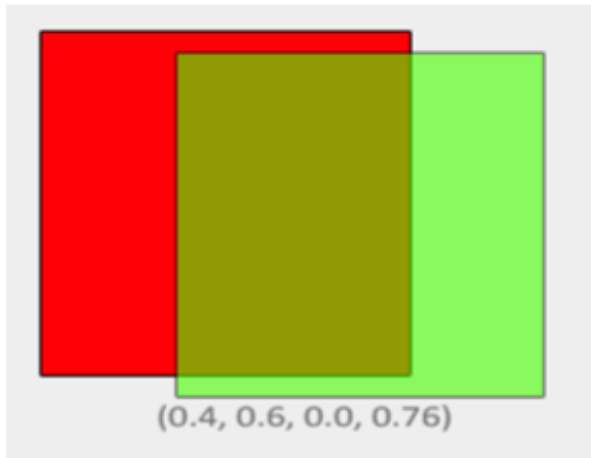
F_{source} ed $F_{destination}$ possono essere impostati su un valore a scelta. Facciamo un semplice esempio:

Abbiamo due quadrati: vogliamo disegnare il quadrato verde semitrasparente sopra il quadrato rosso. Il quadrato rosso sarà il colore di destinazione (e quindi già memorizzato nel frame color buffer) e ora disegneremo il quadrato verde (alfa=0.6) sopra il quadrato rosso (alfa=1.0)



Possiamo moltiplicare il quadrato verde con il suo valore alfa, che in questo caso è 0.6, quindi impostiamo F_{source}= 0.6, ed il valore di $F_{destination}$ ad (1-0.6), così se il quadrato verde contribuisce per il 60% al colore finale, vogliamo che il quadrato rosso contribuisca per il 40% al colore finale, ad es. 1.0 – 0.6.

$$\bar{C}_{result} = \begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.6 \end{pmatrix} * 0.6 + \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{pmatrix} * (1 - 0.6)$$



Il colore risultante viene quindi memorizzato nel buffer colore, sostituendo il colore precedente.

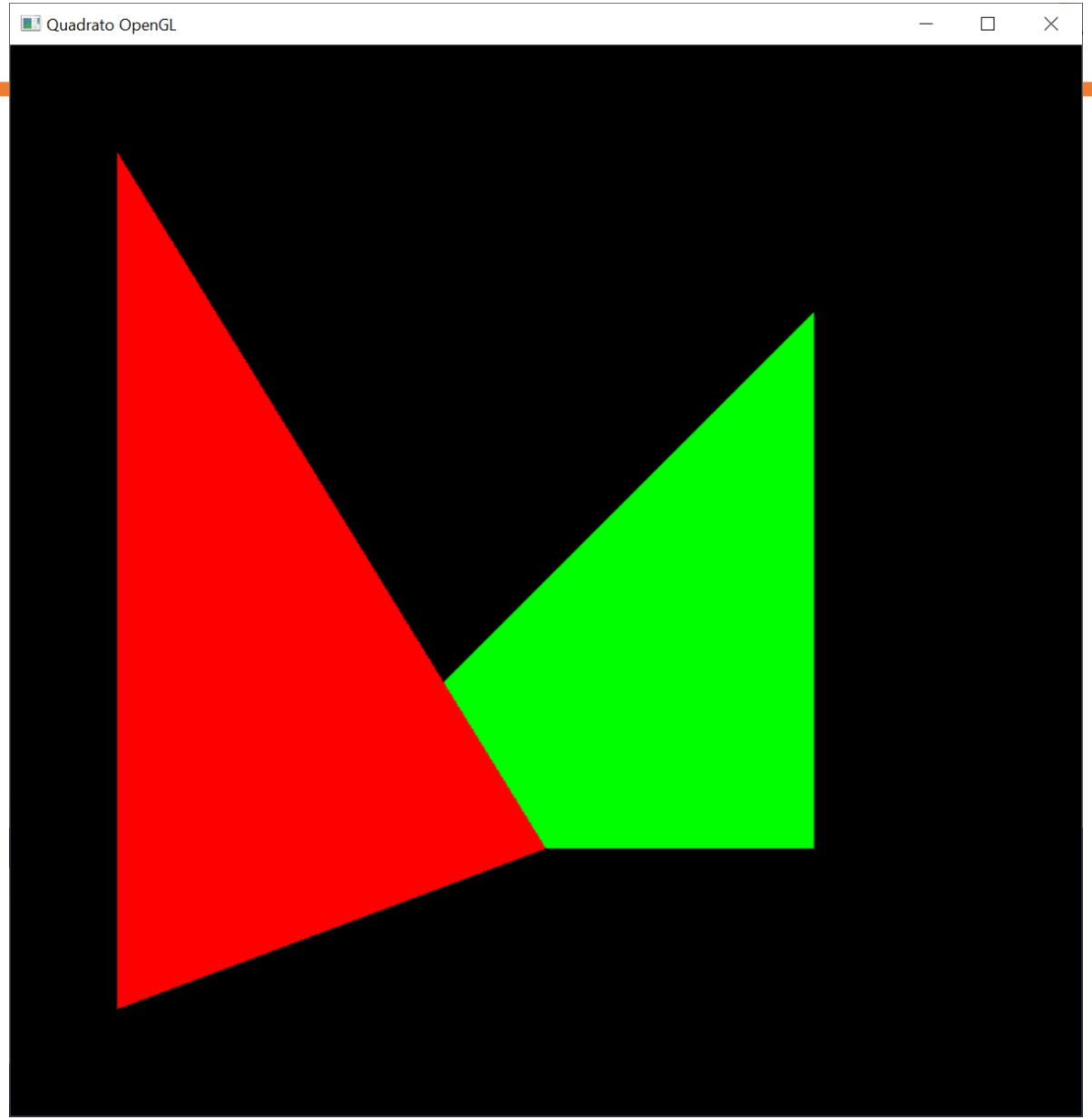
La funzione **glBlendFunc(Glenum sfactor, Glenum dfactor)** specifica come vengono usati nell'equazione di blending i valori di F_{source} (sfactor) e di $F_{destination}$ (dfactor)

Option	Value
GL_ZERO	Factor is equal to 0.
GL_ONE	Factor is equal to 1.
GL_SRC_COLOR	Factor is equal to the source color vector $\bar{C}_{source}$.
GL_ONE_MINUS_SRC_COLOR	Factor is equal to 1 minus the source color vector: $1 - \bar{C}_{source}$.
GL_DST_COLOR	Factor is equal to the destination color vector $\bar{C}_{destination}$
GL_ONE_MINUS_DST_COLOR	Factor is equal to 1 minus the destination color vector: $1 - \bar{C}_{destination}$.
GL_SRC_ALPHA	Factor is equal to the $alpha$ component of the source color vector $\bar{C}_{source}$.
GL_ONE_MINUS_SRC_ALPHA	Factor is equal to $1 - alpha$ of the source color vector $\bar{C}_{source}$.
GL_DST_ALPHA	Factor is equal to the $alpha$ component of the destination color vector $\bar{C}_{destination}$.
GL_ONE_MINUS_DST_ALPHA	Factor is equal to $1 - alpha$ of the destination color vector $\bar{C}_{destination}$.
GL_CONSTANT_COLOR	Factor is equal to the constant color vector $\bar{C}_{constant}$.
GL_ONE_MINUS_CONSTANT_COLOR	Factor is equal to 1 - the constant color vector $\bar{C}_{constant}$.
GL_CONSTANT_ALPHA	Factor is equal to the $alpha$ component of the constant color vector $\bar{C}_{constant}$.
GL_ONE_MINUS_CONSTANT_ALPHA	Factor is equal to $1 - alpha$ of the constant color vector $\bar{C}_{constant}$.

Per ottenere il risultato dell'esempio dei due quadrati fatto precedentemente dobbiamo usare

glBlendFunc(GL_SRC_ALPHA, GL_ONES_MINUS_SRC_ALPHA)

```
float vertices[] = {
// posizioni
 -0.5f, -0.5f, 0.0f,
  0.5f, -0.5f, 0.0f,
 0.5f,  0.5f, 0.0f ,
 0.0f,-0.5f,0.0f,
-0.8f,0.8f,0.0f,
-0.8f,-0.8f,0.0f
};

float colori[] = {
// colori
 0.0f,1.0f,0.0f,1.0f,
 0.0f,1.0f,0.0f,1.0f,
 0.0f,1.0f,0.0f,1.0f,
 1.0f,0.0f,0.0f,1.0f,
 1.0f,0.0f,0.0f,1.0f,
 1.0f,0.0f,0.0f,1.0f
};
```



```
float colori[] = {
        // colori
         0.0f,1.0f,0.0f,1.0f,
         0.0f,1.0f,0.0f,1.0f,
         0.0f,1.0f,0.0f,1.0f,
         1.0f,0.0f,0.0f,0.6f,
         1.0f,0.0f,0.0f,0.6f,
         1.0f,0.0f,0.0f,0.6f
};
```

Nel main:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```