CWindow White Paper

Paolo Medici
Dipartimento di Ingegneria dell'Informazione dell'Università di Parma

10 dicembre 2010

`draft-0.2 per gold-4.3 e gold-4.4`
È possibile trovare l'ultima versione di questo documento a `http://vislab.it/medici`.
Evita se possibile di stampare questo documento per risparmiare carta.

1 Introduzione

Il sistema delle finestre di GOLD è composto da un client (`CWindow`) e un server (`CWindowCore`). Solitamente il server è asincrono rispetto al client e in generale va reputato tale.

Il client possiede una coda di disegno locale che viene cancellata attraverso il metodo `Clear` e trasferita al server con il metodo `Refresh`. Tutte le chiamate ai metodi di disegno di `CWindow` provocano aggiunte di widget alla coda locale e non un effettivo disegno.

Per usare le finestre è necessario includere l'header:

```
#include <CWindow.h>
```

e verificare che nella CMakeLists.txt sia presente la libreria `libgold_cwindows.so`.

2 Operazioni Base

In questa sezione vengono presentate le operazioni base sulle `CWindow`.

2.1 Creazione

Le `CWindow` possono essere allocate sia come variabili membro della classe, sia allocate in maniera dinamica con `new`. Se inserite nella classe le `CWindow` con il costruttore di default non avranno titolo ne dimensioni: tali parametri dovranno essere impostate dall'utente nella fase di inizializzazione.

Il costruttore più diffuso della classe CWindow è:

```
CWindow::CWindow(const std::string & Title, unsigned int Width, unsigned int Height
                    [,int Flags])
```

che permette di impostare il titolo `Title` della finestra e le sue dimensioni fisiche `Width` e `Height` (e automaticamente quelle logiche). L'ultimo parametro opzionale, `Flags`, vale di default `CWF_DEFAULT`: impostato diversamente permette di creare finestre con proprietà particolari.

Negli esempi si fa sempre riferimento a `CWindow` come puntatore e allocata con una `new`. È chiaro che tali esempi valgono anche con una CWindow inserita direttamente come variabile membro.

Esempio di `CWindow` creata come variabile membro:

```
class MyClass {
CWindow m_myWin;
...
void On_Initialization() {
 m_myWin.SetSize(m_width, m_height);
 m_myWin.SetVirtualView(m_width, m_height);
```

```
 m_myWin.SetTitle("Title");
 ...
 m_myWin.Show();
}
```

Esempio di `CWindow` puntatore creata con `new`:

```
class MyClass {
CWindow *m_myWin;
...
void On_Initialization() {
 m_myWin = new CWindow("Title", m_width, m_height);
 ...
 m_myWin->Show();
}

void On_ShutDown() {
 delete m_myWin;
}
```

Le finestre possono essere mostrate in ogni momento con il metodo `CWindow::Show` e nascoste con il metodo `CWindow::Hide`. Chiamare una `CWindow::Show` su una finestra già visibile, o `CWindow::Hide` su una finestra nascosta, non provoca nessun effetto ma sarebbe da evitare per ridurre inutili controlli.

2.2 ViewPort e Coordinate Logiche

Il ViewPort rappresenta il sistema di coordinate logiche della finestra. La finestra può essere ridimensionata ma il Viewport rappresenterà sempre lo stesso sistema di coordinate.

Per selezionare un sistema di coordinate schermo (X crescente verso destra, Y crescente verso il basso) si usa la chiamata `CWindow::SetVirtualView`.

È possibile specificare una semplice estensione del viewport con la chiamata

```
win->SetVirtualView(width, height);
```

In questo modo la finestra avrà come coordinata `(0,0)` in alto a sinistra e `(width,height)` in basso a destra.

È possibile anche indicare la coordinata del punto in alto a sinistra con la versione estesa

```
win->SetVirtualView(x0,y0,x1,y1);
```

Questo metodo permette di creare una finestra che ha `(x0,y0)` come coordinata in alto a sinistra e `(x1,y1)` in basso a destra. Tale funzione permette di esegure flip verticale e orizzontale delle coordinate (non rotazione però).

2.2.1 Camera e Viewport tridimensionali

Se si vuole usare un ViewPort in coordinante cartesiane, uno tridimensionale o in generale usare una `CWindowCamera` diversa da quella di default, bisogna includere prima di tutto

```
#include <CWindowCamera.h>
```

Per allocare una CWindowCamera bisogna dichiarare un oggetto come `boost::shared_ptr<CWindowCamera>` per lasciare la deallocazione alla finestra. Quando si vuole assegnare la camera così creata bisogna chiamare il metodo `CWindow::SetCamera`:

```
boost::shared_ptr<CWindowCamera> camera ( costruttore da puntatore ) ;
// ...
win->SetCamera(camera);
```

Il viewport cartesiano è un particolare tipo di viewport tridimensionale. Ha l'asse X crescente verso destra, Y crescente verso l'alto e l'asse Z diretto verso la camera.

Un classico view cartesiamo diventa pertanto:

```
boost::shared_ptr<CWindowCamera3D> camera(
           new CWindowCamera3D( XMIN, YMIN, XMAX, YMAX, -M_PI/2.0 ) );
win->SetCamera(camera);
```

dove sono da indicare gli intervalli di X e Y.

Le camere possono accettare parametri di una CCamera.

Esempio:

```
m_camera = boost::shared_ptr<CWindowCamera3D>(
           new CWindowCamera3D(hfov, vfov, 0.0, 0.0, yaw, pitch, roll, X, Y, Z) );
win->SetCamera(m_camera);
```

2.2.2 Coordinate Schermo

È possibile usare invece che le coordinate logiche, le coordinate schermo fisiche, attraverso il comando:

```
win->SetViewportMode(WCVM_SCREEN);
```

In coordinate fisiche l'unità di misura delle coordinate è in pixel, con $(0, 0)$ l'angolo in alto a sinistra del canvas della finestra.

Per ritornare in coordinate logiche bisogna poi chiamare

```
win->SetViewportMode(WCVM_CAMERA);
```

2.3 Colore di disegno

Il colore di disegno può essere la classica tripletta `RGB`, come un tono di grigio, come un colore più trasparenza `RGBA`:

```
win->SetColor(127);     // tono di grigio
win->SetColor(192,0,0); // Red, Green, Blue
win->SetColor(0,255,0, 127); // Red, Green, Blue, Alpha
```

Il metodo `SetColor` accetta anche oggetti `cimage::RGBA8` e `cimage::RGB8`:

```
cimage::RGB8 color;
color.R = ...
color.G = ...
color.B = ...
win->SetColor(color);
```

Il colore viene applicato a tutte le successive chiamate di disegno.

Dopo una `CWindow::Clear` il colore di disegno è da considerare indefinito e perciò va sempre impostato.

2.4 Trasparenza

Le CWindow supportano il disegno con trasparenza. Di default tuttavia tale elaborazione è disabilitata. Per abilitare la trasparenza chiamare il metodo `CWindow::EnableBlend`:

```
win->EnableBlend();
```

per disabilitare la trasparenza bisogna invece usare il metodo `CWindow::DisableBlend`:

```
win->DisableBlend();
```

Quando la trasparenza è abilitata il parametro `alpha` di `SetColor` viene effettivamente utilizzato. Con `alpha=0` verranno disegnati oggetti totalmente invisibili, mentre con `alpha=255` oggetti totalmente opachi.

2.5 Layer

Le CWindow supportano molteplici Layer di disegno in modo da poter aggiungere primitive in maniera non sequenziale. Siccome l'ordine di disegno è strettamente quello di inserimento è normale che le nuove primitive di disegno si sovrappongano a quelle precedentemente disegnate. Con i layer una parte del programma potrebbe per esempio aggiungere primitive prima di una parte di programma eseguita in precedenza. Il numero di layer è limitato solo dalla disponibilità di memoria, ma è consigliabile usare strettamente il numero di layer necessari.

Per creare e cambiare Layer si usa il metodo `CWindow::SetLayer`.

Esempio:

```
win->SetLayer(1);
win->Clear();
// ... Draw Something
win->SetLayer(0);
win->Clear();
// ... Draw Something
```

È da notare che il metodo `CWindow::Clear` cancella solo il Layer corrente: per cancellare tutti i Layer bisogna usare il metodo `CWindow::ClearAllLayers`. È possibile anche cancellare un layer diverso dal corrente con il metodo `CWindow::ClearLayer(int layer_no)`.

2.6 Primitive di Disegno

È possibile disegnare punti, linee, triangoli, quadrilateri nativamente. A ogni primitiva di disegno esiste la versione che disegna un array degli stessi elementi: dal punto di vista delle performance è sempre preferibile ridurre le chiamate di disegno e l'utilizzare i metodi di disegno di array di elementi risulta vantaggioso.

Ognuna delle primitive può essere disegnata con il colore corrente impostato da `CWindow::SetColor`, ma esistono anche versioni delle chiamate che permettono di disegnare array di punti, rette, triangoli e quadrilateri esprimendo un colore differente per ogni vertice. Allo stesso modo è possibile disegnare array di primitive usando coordinate di texture.

2.6.1 DrawPixel: Disegno di punti

Disegno di un punto usando le coordinate X,Y:

```
win->DrawPixel(x,y);

win->DrawPixel(50,100);
```

Disegno di un punto tridimensionale usando le coordinate X,Y,Z:

```
win->DrawPixel(x,y,z);

win->DrawPixel(10.0,0.0,1.0);
```

Esempio di disegno di un punto usando un `Point2<T>`:

```
Point2f a;
a.x = ...
a.y = ...
win->DrawPixel(a);
```

Esempio di disegno di un punto tridimensionale usando un `Point3<T>`:

```
Point3d b;
b.x = ...
b.y = ...
b.z = ...
win->DrawPixel(b);
```

Esempio di disegno un punto tridimensionale colorato usando un `C4UB_V3F_t` (in tal caso il colore di disegno impostato da `SetColor` viene ignorato):

```
C4UB_V3F_t c;
c.x = ...
c.y = ...
c.z = ...
c.r = ...
c.g = ...
c.b = ...
c.a = ...
win->DrawPixel(c);
```

Per migliorare le prestazioni in presenza di un numero elevato di punti è possibile disegnare con una sola chiamata un array o un vettore di punti. `CWindow::DrawPixels` disegna un elenco di punti bidimensionali, `CWindow::DrawPixels3` disegna un elenco di punti tridimensionali, `CWindow::DrawColorPixels3` disegna un elenco di `C4UB_V3F_t` e `CWindow::DrawTexturePixels3` un elenco di `T2F_V3F_t`.

Esempio di disegno bidimensionale con un array di 4 elementi con sintassi `DrawPixels(puntatore al primo elemento, numero di punti)`:

```
Point2f pts[4];
// ... riempire l'array pts di 4 elementi
win->DrawPixels(&pts[0], 4);
```

Esempio di disegno tridimensionale con un array di `n` elementi:

```
Point3f *pts = new Point2f[n];
// ... riempire l'array pts
win->DrawPixels3(&pts[0], n);
```

Esempio di disegno bidimensionale con un `std::vector`:

```
std::vector<Point2f> pts;
pts.push_back( ... );
pts.push_back( ... );
// ...
win->DrawPixels(pts);
```

Esempio di disegno tridimensionale con un array di `n` elementi colorati:

```
Point3f *pts = new C4UB_V3F_t[n];
// ... riempire l'array pts
win->DrawColorPixels3(&pts[0], n);
```

Da `gold 4.4` in poi è disponibile anche `CWindow::DrawTexturePixels3` per disegnare punti `T2F_V3F_t` con indice in una texture.

2.6.2 Disegno di rette

`CWindow::DrawLine` permette di disegnare il segmento (retta) che congiunge due punti (sia bidimensionali che tridimensionali).

Per disegnare una retta dal punto di coordinate (x_0, y_0) al punto di coordinate (x_1, y_1):

```
win->DrawLine(x0,y0, x1,y1);
// esempio:
win->DrawLine(10,20, 100,100);
```

`DrawLine` accetta anche oggetti `Point2<T>` e `Point3<T>`:

```
Point2d a,b;
// ... do something ...
// for example a = Point2d(10,20); b = Point2d(100,100);
win->DrawLine(a,b);
```

È possibile disegnare anche oggetti di tipo `Line3<T>` fornendo un rettangolo di clipping dove limitare il rendering.

In questo esempio limita il rendering della linea `l` all'area `(0,0) - (640,480)`:

```
Line3d l;
// fill line l with something
win->DrawLine(l, Rect2(0,0, 640,480));
```

È possibile disegnare in maniera atomica array e vector di segmenti con il metodo `CWindow::DrawLineStream` (`CWindow::DrawLineStream3` per segmenti tridimensionali). Ogni due punti del vettore o dell'array rappresentano una retta. La sintassi è

```
win->DrawLineStream(puntatore_al_primo_elemento, numero_di_segmenti);
```

Esempio del disegno di 5 segmenti (10 punti):

```
Point2d lines[10]; // 10 points
// do something
win->DrawLineStream(&l[0], 5); // 5 segments
```

Il metodo con vettori di segmenti è presente da `gold-4.4` in poi.

2.6.3 Disegno di rettangoli

`CWindow::DrawBox` permette di disegnare rettangoli pieni usando 2 punti, mentre `CWindow::DrawRectangle` permette di disegnare il contorno di un rettangolo.

Questo esempio permette di disegnare un rettangolo dal punto di coordinata (x_0, y_0) al punto di coordinate (x_1, y_1):

```
win->DrawBox(x0,y0, x1,y1);
// esempio:
win->DrawBox(20,20, 100,100);
```

Allo stesso modo

```
win->DrawRectangle(x0,y0, x1,y1);
// esempio:
win->DrawRectangle(20,20, 100,100);
```

I rettangoli possono essere disegnati usando sia i `Rect2_t<T>` che i `Rect2<T>`.

`CWindow::DrawBoxes` e `CWindow::DrawRectangles` sono le funzioni che permettono di disegnare in maniera atomica un array o vector di rettangoli.

```
Rect2d list[4];
// do something
win->DrawBoxes(&list[0], 4);

std::vector<Rect2d> list;
// do some push_back in list
win->DrawBoxes(list);
```

2.6.4 Disegno di cerchi ed ellissi

`CWindow::DrawCircle` permette il rendering di circonferenze (contorno) e cerchi (pieno).

Alcuni esempi di disegno di cerchi e circonferenze.

Cerchio pieno con centro in (x, y) e raggio r:

```
win->DrawCircle(x,y,r,true);
// esempio:
win->DrawCircle(100,70,10,true);
```

Circonferenza (vuota) con centro in (x, y) e raggio r:

```
win->DrawCircle(x,y,r,false);
```

`CWindow::DrawCircle` accetta come parametro anche i `Point2<T>`:

```
Point2d c;
// do something with c
win->DrawCircle(c,r,true_or_false);
```

`CWindow::DrawEllipse` permette invece di disegnare ellissi. In tal caso bisogna fornire come parametri gli estremi del rettangolo che iscrive l'ellisse:

```
win->DrawEllipse(x0,y0,x1,y1, true_or_false);
// esempio: ellisse vuoto tra 10,10 e 210,110
win->DrawEllipse(10,10, 210, 110, false);
```

tale ellisse avrà centro in `(110,60)`, semiasse maggiore `100` e semiasse minore `50`.

`CWindow::DrawEllipse` accetta anche `Rect_2<T>` e `Rect2<T>`

È possibile anche disegnare fette di ellisse e di cerchio con le funzioni `CWindow::DrawSlicedEllipse` e `CWindow::DrawSlicedCircle` che accettano angoli di partenza e fine (espressi in radianti).

2.6.5 Disegno di poligoni

`CWindow::DrawPolygon` permette di disegnare poligoni (pieni o solo contorno, aperti o chiusi) formati da una sequenza ordinata di punti.

Per disegnare un array o un vettore di punti si usa la sintassi:

```
win->DrawPolygon(pointer_to_first_element, NumberOfVertex, IsClosed, IsFilled);
win->DrawPolygon(vector, IsClosed, IsFilled);
```

Esempio: disegnare un array di 4 elementi, aperto e solo contorno:

```
Point2d v[4];
// do something on v
// draw an open outline
win->DrawPolygon(&v[0], 4, false, false);
```

Disegno di un poligono usando `std::vector`:

```
std::vector<Point2d> v;
// do some push_back in v
win->DrawPolygon(v, false, false);
```

Esiste anche un metodo che permette di unire due array o vettori separati per le coordinate `x` e `y`:

```
float *x = new float [n]
float *y = new float [n]
// do something to fill x and y
// draw a closed and filled polygon:
win->DrawPolygon(x,y,n,true,false);
```

Allo stesso modo è possibile disegnare poligoni in 3 dimensioni usando `CWindow::DrawPolygon3`.

Non è possibile disegnare correttamente poligoni pieni concavi in trasparenza: la scheda video scompone il poligono in triangoli che tenderanno a sovrapporsi.

2.6.6 Disegno di istogrammi

In `CWindow` esistono dei wrapper al disegno dei poligoni per semplificare il disegno di dati quali grafici a istogramma.

Per disegnare un istogramma dove l'asse delle categorie è quello orizzontale (asse `x`) si usa `CWindow::DrawHHistogram` mentre `CWindow::DrawVHistogram` per l'asse delle categorie in verticale. Per disegnare un istogramma è necessario fornire il punto di inizio e la direzione e il fattore di scala con cui eseguire il disegno, l'array con i dati da graficare e la sua dimensione e se si vuole disegnare pieno o vuoto.

Esempio: istogramma a base orizzontale (tradizionale), 256 valori (esempio scala di grigio) allineato con il fondo pagina:

```
unsigned int histo[256];
// do something... fill histo with something
win->DrawHHistogram(0,480,1.0,-1.0, histo,256, false);
```

In questo caso l'istogramma parte da `(0,480)`, l'asse delle categorie è crescente lungo le `x`, ogni categoria è larga `1` pixel. i valori dell'istogramma vengono moltiplicati per `-1` perciò crescenti nel verso opposto delle `y`. L'istogramma infine è solo contorno e non pieno.

Esiste anche la versione che usa gli `std::vector`:

```
std::vector<int> histo;
// do some push_back on histo
win->DrawHHistogram(0,480,1.0,-1.0, histo, false);
```

2.6.7 Disegno di triangoli e quadrilateri

Nel disegno di triangoli e quadrilateri esistono solo le funzioni che accettano un elenco di triangoli. I triangoli e i quadrilateri possono essere sia vuoti che pieni (che è il default).

La sintassi per disegnare un array di triangoli e quadrilateri è:

```
win->DrawTriangles(&v[0], number_of_triangles, is_filled_or_not);
win->DrawQuads(&v[0], number_of_quads, is_filled_or_not);
```

ricordando che sono richiesti 3 punti nell'array per ogni triangolo e 4 punti per ogni quadrilatero.

per disegnare usando `std::vector`:

```
win->DrawTriangles(v, is_filled_or_not);
win->DrawQuads(v, is_filled_or_not);
```

È possibile disegnare anche array di Point3¡T¿ con i metodi `CWindow::DrawTriangles3` e `CWindow::DrawQuads3`.

È anche possibile disegnare triangoli con ogni vertice un colore differente. In questo caso è necessario preparare un array o vector di `C4UB_V3F_t` e usare i metodi `CWindow::DrawColorTriangles3` o `CWindow::DrawColorQuads3`.

Esempio di disegno di triangoli colorati:

```
std::vector<C4UB_V3F_t> p;
// ... push_back C4UB_V3F_t vertexes, 3 per triangle ...
win->DrawTriangles(p);
```

Da `gold 4.4` in poi sono disponibili anche i metodi per il disegno di triangoli e quadrilateri con texture attraverso i metodi `CWindow::DrawTextureTriangles3` e `CWindow::DrawTextureQuads3`.

2.6.8 Disegno di crocette e vettori

Per semplificare diversi compiti sono disponibili metodi per disegnare crocette `CWindow::DrawCross` e frecce `CWindow::DrawVector`.

In particolare è possibile disegnare anche vettori (o array) di crocette usando il metodo `CWindow::DrawCrossArray`.

2.6.9 Proprietà di disegno

È possibile attivare delle proprietà che agiscono sulle successive operazioni di disegno. Prima della `CWindow::Refresh` chi modifica una di queste caratteristiche deve riportarle al valore di default.

`CWindow::SetPointSize` permette di selezionare la dimensione di come viene renderizzato il singolo punto nelle `CWindow::DrawPixel` e `CWindow::DrawPixels` successive. L'unità di misura è pixels e il default è `1`.

`CWindow::SetPointSmoothing` abilita o disabilita l'antialiasing nel rendering dei punti. Di default è `false`.

`CWindow::SetLineWidth` permette di selezionare la dimensioni di come vengono renderizzate le rette, polilinee e quadrati. L'unità di misura è pixels e il default è `1`.

`CWindow::SetLineSmoothing` abilita o disabilita l'antialiasing nel rendering delle rette. Di default è `false`.

2.7 Testo

La sintassi del comando base per il disegno del testo è il seguente:

```
bool CWindow::DrawText(x, y, text);
```

dove è necessario indicare le coordinate logiche `x,y` dove far cominciare il testo e il suo contenuto `text`.

Per decidere il significato del punto `x,y` si usa il metodo `CWindow::SetTextAlign`:

```
m_win->SetTextAlign(TA_CENTER_, TA_CENTER_);
```

Per default il punto `x,y` coincide con il punto sinistro basso del testo.

Per impostare la dimensione del font si usa il metodo

```
bool SetFontSize(double fontSize, bool absolute = false);
```

dove `fontSize` è una dimensione in unità logiche della finestra. Il parametro opzionale `absolute` se impostato a `true` fa in modo di usare dimensioni fisiche in pixel e non logiche in modo che in caso di zoom la dimensione del carattere non cambi.

Per cambiare il font di default si usa il metodo `SetFontName` in `gold 4.3` bisogna fornire un font tra quelli a scelta della cartella `Font` di gold:

```
bool CWindow::SetFontName(const char *file_ttf_nella_cartella_font_di_gold);
```

`gold 4.4` o superiore permettono di sfruttare i font del sistema. Con questa versione per cambiare il font di default si usa sempre il metodo `SetFontName` che però ha la sintassi:

```
bool CWindow::SetFontName(const char *fontFamilyName);
```

il quale accetta una stringa rappresentante il nome di una famiglia di font come ritornato dal programma da linea di comando *fc-list*.

È possibile utilizzare anche modifiche ai font sfruttando la sintassi `chiave=valore`. Le chiavi e i possibili valori sono:

- `family` una famiglia ritornata da `fc-list`
- `slant` inclinazione. Possibili valori sono `roman|italic|oblique`
- `weight` livello di grassetto: `thin|light|book|normal|medium|bold|black`
- `spacing` spaziatura: `mono|proportional`

Esempi:

```
win->SetFontName("Liberation Sans");
win->SetFontName("family=Liberation Sans");
win->SetFontName("family=Liberation Sans,slant=italic");
win->SetFontName("family=Liberation Sans,weight=bold");
```

2.8 Disegno di Immagini

È possibile disegnare sia immagini come array di byte sia `cimage::CImage`.

Per disegnare una `cimage::CImage` che riempie tutta la finestra

```
win->DrawImage(img);
```

Per disegnare una array di `unsigned char` che riempie tutta la finestra (la sintassi è `CWindow::DrawImage(buf, img_width, img_height)`):

```
unsigned char *buf = new unsigned char [320 * 240];
// ... do something ...
win->DrawImage(buf, 320, 240);
```

Per disegnare una `cimage::CImage` con l'angolo in alto a sinistra in `(x,y)` e grande come la `width` e la `height` dell'immagine stessa:

```
win->DrawImage(x,y,img);
```

o per gli array:

```
win->DrawImage(x,y, buf, img_width, img_height);
```

Per disegnare una immagine disegnata con l'angolo in alto a sinistra in `(x,y)` ed estesa `(w,h)` si usa la sintassi:

```
win->DrawImage(x,y,w,h,img);
```

in questo caso l'immagine verrà riscalata alla dimensione impostata. Nel caso di array la sintassi sarà:

```
win->DrawImage(x,y, w,h, buf, img_width, img_height);
```

2.9 Note

Non tutte le immagini possono essere disegnate. Attualmente è implementato il disegno di immagini `MONO8`, `RGB8`, `RGBA8` e le `MONO8` con pattern di Bayer.

Le immagini `MONO8s` vengono disegnate convertite in `MONO8` aggiungendo 128 al valore del pixel.

Le immagini `MONO16` attualmente non possono essere disegnate.

3 Comandi base

Nella tabella seguente sono presentati i tasti base sulle CWindow con driver OpenGL:

H Mostra l'help con questi e altri comandi

D Salva su disco il fotogramma corrente (in `windows.ini` è indicato il formato con cui l'immagine viene salvata)

C Salva su disco tutti i fotogrammi disegnati. Compare una `[C]` nel titolo della finestra. (in `windows.ini` è indicato il formato con cui la sequenza di immagini viene salvata)

Se la finestra non è in modalità `KEY_HANDLE` (compare una `[M]` sul titolo) è necessario premere `CTRL` più il tasto sopraindicato per eseguire la stessa azione.

4 Argomenti Avanzati

In questa sezione sono presentati argomenti avanzati sull'uso delle `CWindow`.

4.1 ZBuffer

Nella visuale tridimensionale capita spesso che non si vogliano vedere gli oggetti nell'ordine esatto in cui sono stati disegnati, ma piuttosto che gli oggetti vicini sovrascrivino quelli lontani. In tal caso bisogna abilitare una modalità che fa uso di un buffer extra nella scheda video dove la distanza di ogni pixel disegnato viene salvata e confrontata con quella dei nuovi pixel che vogliono essere disegnati. Il metodo `CWindow::EnableZBuffer(near, far)` abilita questa modalità ma richiede due parametri che servono per dimensionare correttamente la quantizzazione tra distanza e valore nel buffer.

Esempio, per abilitare lo ZBuffer ma limitare il redering tra 1.0 metri e 200 metri:

```
win->EnableZBuffer(1.0, 200.0);
```

Per disabilitare lo ZBuffer e tornare al rendering ordinato chiamare il metodo:

```
win->DisableZBuffer();
```

È da notare come sia importante dimensionare correttamente i parametri passati a `CWindow::EnableZBuffer` in modo da separare punti tra loro vicini.

4.2 Trasformazioni di Immagini

In alcuni casi si vuole che il l'immagine disegnata venga processata prima di essere disegnata. Per semplificare le operazioni di disegno sono introdotti i metodi `CWindow::DrawXImage`, uguali ai corrispettivi `CWindow::DrawImage` che accettano un parametro in più per specificare quale processing applicare su una immagine prima che questa venga disegnata.

Per poterli usare è necessario includere l'header

```
#include <XImgProc.h>
```

Le possibili operazioni sono

- `XConversion` che converte `RGB8/MONO8` l'immagine in `RGBA8` aggiungendo trasparenza a tutta l'immagine;
- `XColorKey` che converte `RGB8/MONO8` l'immagine in `RGBA8` cambiando solo un colore in trasparente;
- `XOpacity` che converte `RGB8/MONO8` in `RGBA8` usando il tono di grigio dell'immagine sorgente come canale alpha;
- `XFilter` che converte `RGB8/MONO8` in `RGB8` modulando il tono con un `RGB8`;

4.3 Pulsanti

Le CWindow presentano dei controlli per semplificare la creazione di HMI evolute. Per usare i controlli di default includere

```
#include <CWindowControls.h>
```

o se si desiderano le versioni che usano invece CImage come sfondo dei pulsanti:

```
#include <CSkinWidgets.h>
```

Esempio di pulsante con testo che passa tra due stati che chiama a ogni cambiamento di stato una callback passando il valore del nuovo stato:

```
m_win << window::CTwoStateButtonWidget(x0,y0,x1,y1, "Text", boost::bind(&MyClass::OnChangeStatus, t
```

Il pulsante si estenderà da (x_0, y_0) fino a (x_1, y_1).

Esempio di pulsante sempre con due stati con bitmap:

```
m_win << window::CSkinCheckBox(x0,y0, boost::bind(&MyClass::OnChangeStatus, this, _1), image_on_off,
```

Il pulsante si estenderà da (x_0, y_0) per la dimensione della bitmap in coordinate logiche.

È consigliabile con i pulsanti lavorare in coordinate schermo.

4.4 Creazione di Widget

Nel caso i widget a disposizione non siano sufficienti è possibile creare dei propri widget. Chiaramente la creazione di nuovi Widget è sconsigliata e va verificato che non esistano in libreria widget che possono soddisfare i requisiti.

In questo caso creare una classe che discenda da `CWidget` (vedere l'header `CDrawingObject.h`) ed implementare i metodi virtuali puri `Draw` e `Interact`.

`Draw` si occupa del rendering del widget vero e proprio. Riceve come parametro il `CWindowCore` su cui eseguire il rendering. Il rendering-context ha funzioni simili a CWindow, ma non proprio uguali, perchè orientate più che altro all'efficienza piuttosto che alla flessibilità. `Draw` deve ritoranre il numero di primitive grafiche effettivamente disegnate, per motivi di statistica.

`Interact` invece riceve un oggetto `CWindowEvent` che contiene la destrizione di un evento. Se il widget ha processato questo evento e non vole che questo evento venga processato da altri widget deve ritornare `true`. In tutti gli altri casi deve ritornare invece `false` e lasciare il controllo dell'evento agli altri widget. Da dentro `Interact` è possibile chiamare il metodo `CWindowCoreManager::Refresh` che chiama in maniera asincrona un refresh sulla finestra.