Progettazione e sviluppo di un ambiente a finestre in un sistema multiprogrammato

Università di Pisa

Antonio Le Caldare

23 Febbraio 2017

Introduzione

Questo progetto di tesi nasce come contributo per il Nucleo di Calcolatori Elettronici, realizzato dal prof. Giuseppe Lettieri, ma soprattutto anche come occasione per soddisfare delle personali curiosità, e per affrontare le interessanti difficoltà del realizzare un ambiente grafico di un Sistema Operativo.

L'obiettivo posto è quello di creare un desktop grafico nella quale i processi utente possano creare finestre, popolarle di componenti grafici (es. bottoni, aree di testo, etichette) e creare delle interazioni con gli utenti tramite il modello degli eventi. L'utente deve poter interagire con le finestre create tramite un cursore, con la quale potremo avere operazioni di resize, trascinamento e chiusura, indipendenti dall'implementazione delle applicazioni.

Ogni Sistema Operativo risolve questo problema in una maniera differente, tuttavia l'obiettivo finale resta quello di realizzare un'interazione tra l'utente e i processi che si servono dell'interfaccia grafica. Per perseguire lo scopo della tesi, dovremo progettare una nuova architettura grafica e definire tutte le componenti richieste per renderla funzionale. La fase di progettazione, tuttavia, richiede una conoscenza almeno basilare di tutti i problemi da risolvere per quanto riguarda la programmazione grafica, in generale. Per tal motivo il primo capitolo di questo documento discute la fase iniziale di prototipazione del progetto. Seguono la fase di progettazione delle librerie grafiche e del processo gestore delle finestre, per poi concludere con la descrizione dell'API utente.

Ringraziamenti

Un caloroso ringraziamento va a tutti i colleghi che mi hanno sostenuto e mi hanno incoraggiato a portare avanti questo progetto. Tuttavia il ringraziamento più sentito va al Prof. Lettieri, che, oltre a permettermi di sviluppare quest'idea, mi ha fornito un grande aiuto e ha permesso che sviluppassi innumerevoli competenze nel campo dei sistemi multiprogrammati, degli strumenti di collaborazione e della progettazione software.

Indice

Parte I

Sottosistema grafico del Sistema Operativo

Capitolo 1

Prototipazione

1.1 Algoritmo del Pittore

Supponiamo di dover realizzare un disegno 2D composto da più elementi: un semplice esempio è un paesaggio composto da un tramonto sullo sfondo, un gruppo di colline e un albero in rilievo. L'algoritmo del Pittore descrive come e quando questi elementi vanno disegnati per raggiungere il risultato finale. Ogni oggetto è caratterizzato da coordinate (posx e posy), che ne rappresentano la posizione (es. angolo alto sinistro), e una dimensione (sizex e sizey). Per comporre il disegno finale, tuttavia, manca un'ulteriore informazione: se due elementi condividono una parte del piano, va stabilito quale oggetto deve avere più rilievo rispetto all'altro. Per tale motivo definiamo lo z-index: in caso di sovrapposizione, l'elemento che avrà lo z-index più alto avrà la meglio. Nell'esempio iniziale potremmo assegnare i seguenti z-index: tramonto 0, gruppo di colline 1, albero in rilievo 2. Per realizzare il disegno, intuitivamente, potremmo disegnare prima lo sfondo, poi il gruppo di colline e in seguito l'albero. Ne deriva che, per creare un'immagine che rispetti le priorità fra gli elementi, basti semplicemente disegnare ogni elemento partendo dallo z-index più basso, da cui deriva l'algoritmo del Pittore.

1.1.1 Semplificazioni

L'algoritmo appena mostrato non risulta efficace nel caso in cui un componente si sovrapponga in maniera non uniforme rispetto ad altri componenti (es: tre poligoni sovrapposti triangolarmente, stile domino) per via dell'introduzione dello z-index: in tal caso sarebbe necessario definire una depth map, cioè una struttura in cui si può definire uno z-index per ogni pixel. Considerando che lo scopo di questo lavoro è quello di realizzare un desktop grafico bidimensionale, non abbiamo necessità di risolvere questo problema.

1.2 Double Buffering

Uno dei problemi derivanti dall'uso dell'algoritmo del Pittore (puro), nel caso in cui siano presenti elementi sullo schermo in movimento, è la creazione di una serie di artefatti grafici abbastanza rilevanti. Questi sono causati dal fatto che generare il disegno finale comporta realizzare molteplici operazioni di *drawing* sullo schermo: questo genera una serie di risultati parziali visibili prima di arrivare al risultato finale. La soluzione a questo problema è un classico della programazione degli algoritmi grafici e si risolve con la tecnica del *Double Buffering*: realizziamo tutte le operazioni parziali su un buffer grafico separato e, alla fine delle operazioni, copiamo il contenuto del buffer grafico su quello dello schermo. In tal modo si eliminano tutti gli artefatti grafici dovuti al drawing di oggetti complessi e oggetti in movimento. Tuttavia è da considerare che questa operazione aggiuntiva impatta l'efficienza dell'algoritmo, sia in termini di CPU che di memoria.

1.3 Rendering Ottimizzato

Dobbiamo considerare che l'algoritmo del Pittore, per quanto sia efficace, ha notevoli problemi dal punto di vista della complessità computazionale: ogni qual volta un oggetto è modificato, questo costringe a ridisegnare completamente l'immagine finale. Inoltre se un elemento è coperto per intero visivamente da un altro, questo viene comunque ridisegnato. Inoltre anche la tecnica del Double Buffering può essere ottimizzata, in quanto non è sempre necessario ricopiare tutto il buffer temporaneo sullo schermo, che è una delle operazioni più onerose da realizzare. Quindi l'algoritmo del Pittore sarà solo la base per realizzare un algoritmo ottimizzato, che dovrà sfruttare tecniche più avanzate per risolvere i problemi mostrati.

Capitolo 2

Architettura del sistema grafico

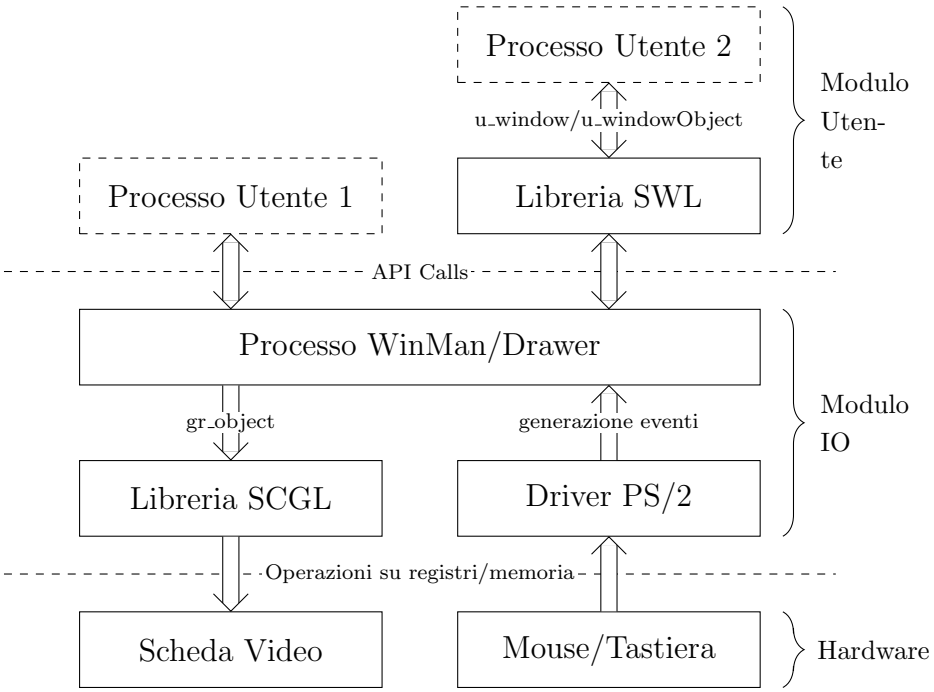


Figura 2.1: Architettura del sistema grafico

2.1 Scheda Video (Hardware)

La scheda video, nel progetto, definisce come le immagini (bitmap) vanno rappresentate e come renderizzarle sullo schermo. Ogni scheda video può essere configurata in varie modalità di funzionamento e, tramite uno dei set di API Grafiche (es. Open-

GL, DirectX, Vulkan, etc..), è possibile accedere alle funzioni di accelerazione della GPU. La scrittura di un driver video è resa complessa a causa della mancanza di specifiche per l'Hardware, e questo è uno dei motivi per cui non è stato possibile sfruttare pienamente le funzionalità della GPU. In realtà, anche in presenza delle specifiche, il lavoro sarebbe stato comunque complesso da svolgere e argomento di una nuova tesi. Nel nostro caso utilizzeremo la modalità Standard VGA che è implementata, ormai, da quasi tutte le schede video in commercio: è ancora richiesto un driver, ma molto banale. Questa modalità di funzionamento prevede che la memoria video sia usata come ***framebuffer***, cioè come una matrice linearizzata di una serie di byte rappresentanti i pixel. La quantità di bit usata per rappresentare un pixel delinea la ***profondità*** del colore, in genere ci troviamo a lavorare con 8, 16, 24 o 32 BPP (Bit Per Pixel). La libreria SCGL, che vedremo in seguito, supporta tutte le modalità, e per la sola 32BPP supporta anche la trasparenza. Ogni modalità di rappresentazione ha le sue specifiche: ad esempio la modalità 32BPP prevede di suddividere i bit in 4 byte, nella quale il primo rappresenta il livello di trasparenza (α), il secondo la quantità di colore rosso (R), il terzo il colore verde (G) e il quarto il colore blu (B). Abbreviatamente è chiamata anche modalità ARGB. Per attivare la modalità Standard VGA dovremmo, in via teorica, eseguire alcune routine del BIOS in modalità reale. Questo, tuttavia, non è realizzabile nel nucleo del corso di Calcolatori Elettronici perchè il bootloader viene eseguito già in modalità protetta, sfruttando alcune funzionalità del virtualizzatore Qemu. Per ovviare a questo problema possiamo sfruttare una estensione della modalità di configurazione della scheda video *std* di Qemu, chiamata *Bochs VBE Extensions*: la scheda video è vista come un dispositivo PCI con deviceId 1234 e vendorId 1111 e funziona sfruttando il Bus Mastering. Nello spazio di configurazione *BAR0* corrisponde ad un indirizzo di memoria nella quale viene mappato il framebuffer della scheda video, mentre in *BAR2* è presente uno spazio grande 4KiB nella quale sono mappati una parte dei registri del controller VGA e una serie di registri che servono per specificare modalità di funzionamento, risoluzione e profondità di colore dello schermo.

2.2 Libreria SCGL (Requisiti)

Nel primo capitolo sulla Prototipazione abbiamo visto che l'algoritmo del Pittore è una delle possibili soluzioni al problema della costruzione di immagini modellizzate come la composizione di uno o più elementi. Nel nostro caso ci siamo posti come obiettivo la realizzazione di un desktop completo basato su finestre, dove ogni finestra può contenere degli oggetti, semplici o complessi. Un oggetto complesso può essere composto da più oggetti (es: bottone composto da un'immagine di sfondo e un testo). E' chiaro che l'immagine da costruire, usando l'algoritmo, sia il desktop finale, ma in realtà l'astrazione dello z-index non è sufficiente ad inglobare la gerar-

chia che si viene a comporre tra finestre, oggetti semplici e complessi. Una soluzione può essere quella di considerare il desktop, ogni singola finestra e oggetto complesso come una serie di immagini da costruire e poi assemblare successivamente, in base alla gerarchia definita. A questo punto, visto che l'algoritmo del Pittore va utilizzato per costruire ognuna di queste immagini, ci viene utile creare una libreria che implenti l'algoritmo e realizzi anche l'astrazione gerarchica. La progettazione della libreria prende come riferimento un classico problema di design pattern, il ***Component***, con alcune modifiche: l'interfaccia Component non è un'interfaccia ma una classe instanziabile che si comporta come un contenitore per altri oggetti.

2.3 Processo WinMan/Drawer

A questo punto va definita quale componente software avrà il compito di creare nuovi oggetti e gestirli, ovviamente tramite l'astrazione offerta dalla libreria SCGL. Questo compito può essere affidato ad un processo che si occuperà sia di gestire tutti gli oggetti e sia di imporsi come un server per i processi utente. La mediazione tra quest'ultimi e il nuovo processo, che chiameremo *WinMan/Drawer*, è realizzata tramite un meccanismo di API Call (che tratteremo nell'apposito capitolo) e una coda di richieste/task. I task rappresentano le operazioni richieste dai processi utente e che devono essere svolte dal drawer: la coda è prevista per realizzare un meccanismo di sincronizzazione più avanzato e per ridurre il numero di cambi di contesto tra il processo server e quelli utente.

2.4 Libreria SWL (Requisiti)

La API presentata ai processi utente risulta essere molto di basso livello e poco object-oriented. Dobbiamo considerare, inoltre, che alcune funzionalità richieste dai programmi con interfaccia grafica risultino essere molto comuni: ad esempio la gestione degli eventi di resize, la gestione degli eventi di click su oggetti comuni (bottoni, label, testo, etc.), etc. Per questi motivi può essere molto utile progettare una nuova libreria che, lavorando col la API di basso livello, possa fornire un set di funzioni e oggetti più semplici da gestire, in aggiunta di una serie di funzionalità comuni già implementate. Quindi si tratterebbe di una libreria di *Middleware*.

Capitolo 3

Progettazione della libreria grafica SCGL

3.1 Introduzione

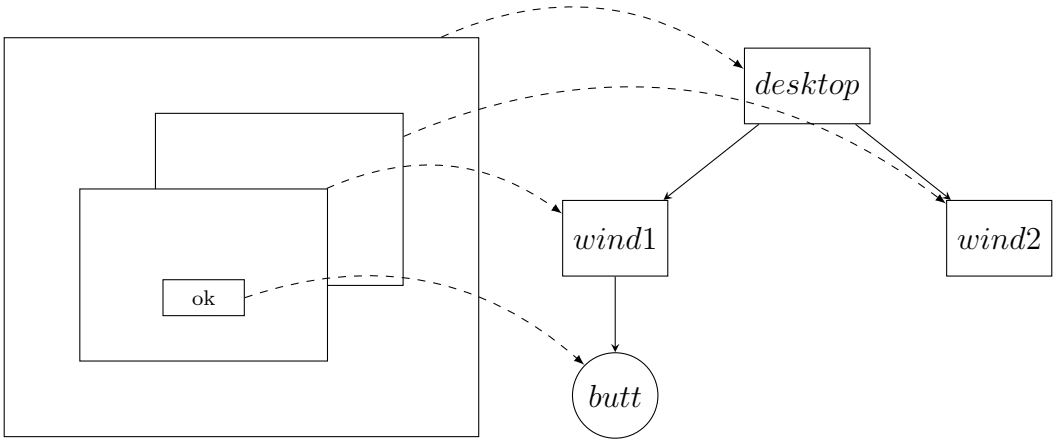
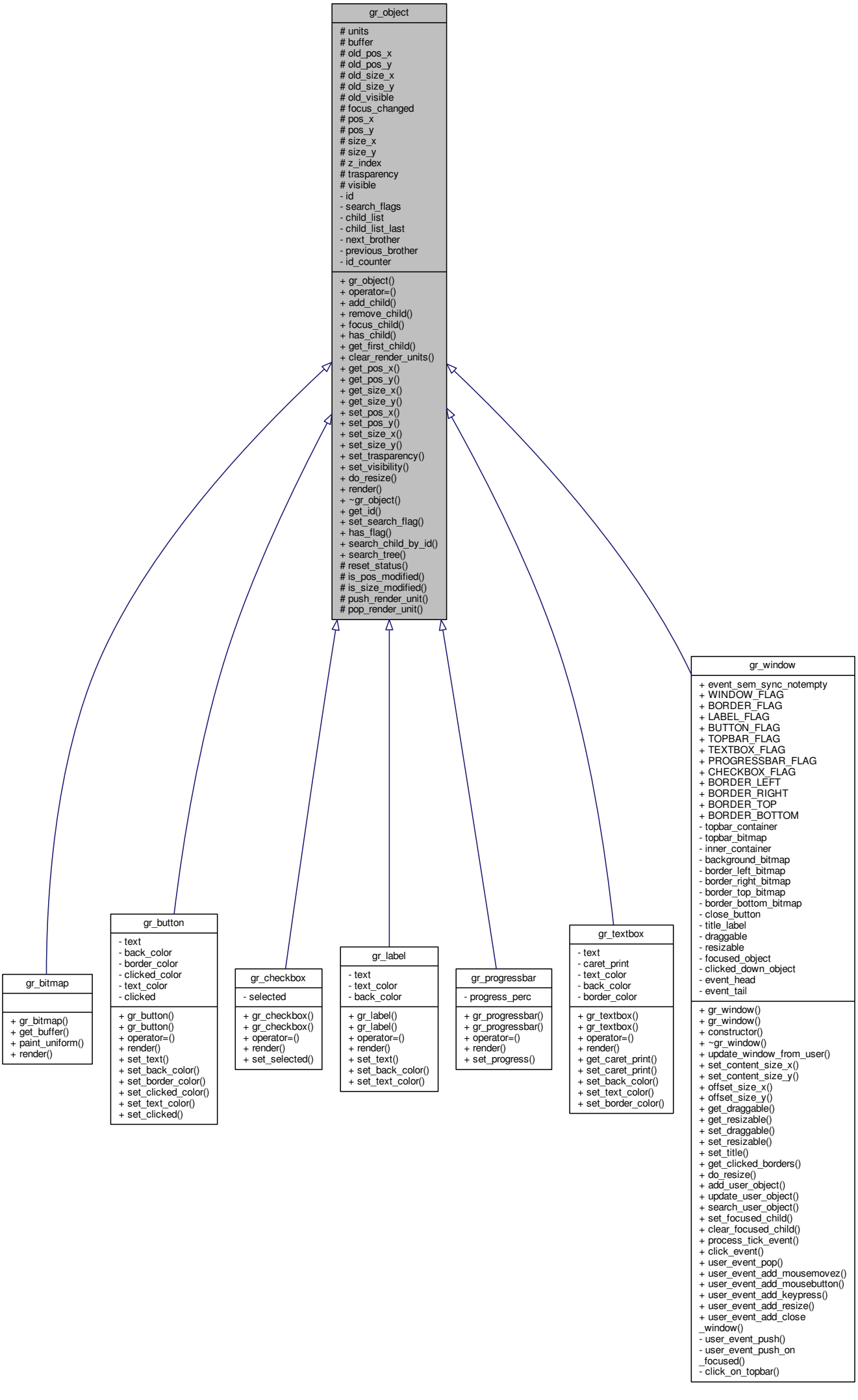


Figura 3.1: Rappresentazione gerarchica degli oggetti di un desktop

Come visto nel capitolo della Prototipazione, la libreria SCGL (Simple Container Graphic Library) deve essere realizzata con lo scopo di creare l'astrazione dei ***Container*** e implementare un algoritmo di rendering (o drawing) sufficientemente ottimizzato. Il primo requisito definisce come un programmatore può descrivere le immagini e gli elementi di un'immagine, mentre il secondo specifica come va realizzato l'algoritmo, anche se è comunque strettamente legato al primo requisito.

3.2 Astrazione

La relazione presente tra gli elementi grafici di un comune desktop è gerarchica. Come visto precendentemente, una soluzione per adattare l'algoritmo del Pittore a questa gerarchia è quella di considerare ogni oggetto complesso come un'immagine da costruire usando l'algoritmo, partendo dalle foglie fino a risalire alla radice. L'operazione di costruzione dell'immagine verrà realizzata da un metodo *render*, implementato da una classe *gr_object*, la quale conterrà anche altri membri dato/-funzione per completare l'astrazione. Tra questi possiamo evidenziare dimensione e posizione rispetto all'oggetto parente, una lista di figli di tipo *gr_object*, e un buffer video, nella quale viene mantenuto il risultato dell'ultima operazione di rendering. Il metodo *render()* è virtuale non puro: l'implementazione di default è una versione ottimizzata dell'algoritmo del Pittore e adattata alla gerarchia degli oggetti, quindi adatta ai ***Container***, ma gli oggetti semplici possono ridefinirlo affinche si comporti come un semplice metodo di drawing (es. *render()* bottone, definisco le operazioni per disegnare un bottone, quindi stampo lo sfondo e poi il testo della label associata). La classe *gr_object* è quindi, spesso, ereditata da altre classi, ad esempio: *gr_button*, *gr_label*, *gr_window*.


gr_object
units
buffer
old_pos_x
old_pos_y
old_size_x
old_size_y
old_visible
focus_changed
pos_x
pos_y
size_x
size_y
z_index
trasparency
visible
- id
- search_flags
- child_list
- child_list_last
- next_brother
- previous_brother
- id_counter
+ gr_object()
+ operator=()
+ add_child()
+ remove_child()
+ focus_child()
+ has_child()
+ get_first_child()
+ clear_render_units()
+ get_pos_x()
+ get_pos_y()
+ get_size_x()
+ get_size_y()
+ set_pos_x()
+ set_pos_y()
+ set_size_x()
+ set_size_y()
+ set_trasparency()
+ set_visibility()
+ do_resize()
+ render()
+ ~gr_object()
+ get_id()
+ set_search_flag()
+ has_flag()
+ search_child_by_id()
+ search_tree()
# reset_status()
# is_pos_modified()
# is_size_modified()
# push_render_unit()
# pop_render_unit()
gr_bitmap
+ gr_bitmap()
+ get_buffer()
+ paint_uniform()
+ render()
gr_button
- text
- back_color
- border_color
- clicked_color
- text_color
- clicked
+ gr_button()
+ gr_button()
+ operator=()
+ render()
+ set_text()
+ set_back_color()
+ set_border_color()
+ set_clicked_color()
+ set_text_color()
+ set_clicked()
gr_checkbox
- selected
+ gr_checkbox()
+ gr_checkbox()
+ operator=()
+ render()
+ set_selected()
gr_label
- text
- text_color
- back_color
+ gr_label()
+ gr_label()
+ operator=()
+ render()
+ set_text()
+ set_back_color()
+ set_text_color()
gr_progressbar
- progress_perc
+ gr_progressbar()
+ gr_progressbar()
+ operator=()
+ render()
+ set_progress()
gr_textbox
- text
- caret_print
- text_color
- back_color
- border_color
+ gr_textbox()
+ gr_textbox()
+ operator=()
+ render()
+ get_caret_print()
+ set_caret_print()
+ set_back_color()
+ set_text_color()
+ set_border_color()
gr_window
+ event_sem_sync_notempty
+ WINDOW_FLAG
+ BORDER_FLAG
+ LABEL_FLAG
+ BUTTON_FLAG
+ TOPBAR_FLAG
+ TEXTBOX_FLAG
+ PROGRESSBAR_FLAG
+ CHECKBOX_FLAG
+ BORDER_LEFT
+ BORDER_RIGHT
+ BORDER_TOP
+ BORDER_BOTTOM
- topbar_container
- topbar_bitmap
- inner_container
- background_bitmap
- border_left_bitmap
- border_right_bitmap
- border_top_bitmap
- border_bottom_bitmap
- close_button
- title_label
- draggable
- resizable
- focused_object
- clicked_down_object
- event_head
- event_tail
+ gr_window()
+ gr_window()
+ constructor()
+ ~gr_window()
+ update_window_from_user()
+ set_content_size_x()
+ set_content_size_y()
+ offset_size_x()
+ offset_size_y()
+ get_draggable()
+ get_resizable()
+ set_draggable()
+ set_resizable()
+ set_title()
+ get_clicked_borders()
+ do_resize()
+ add_user_object()
+ update_user_object()
+ search_user_object()
+ set_focused_child()
+ clear_focused_child()
+ process_tick_event()
+ click_event()
+ user_event_pop()
+ user_event_add_mousemovez()
+ user_event_add_mousebutton()
+ user_event_add_keypress()
+ user_event_add_resize()
+ user_event_add_close
_window()
- user_event_push()
- user_event_push_on
_focused()
- click_on_topbar()

3.3 Ottimizzazioni sull'Algoritmo del Pittore

Un'analisi della complessità computazionale dell'algoritmo del Pittore ci mostra che le operazioni per costruire l'immagine (operazione di rendering) sono onerose e, in molti casi, eliminabili. Supponiamo di avere un desktop con una finestra e uno sfondo. La prima volta che svolgiamo un'operazione di rendering del desktop finale, siamo costretti a utilizzare l'algoritmo del Pittore puro, e le potenziali ottimizzazioni sono poche (es. non stampare una finestra coperta da un'altra finestra, nel caso ce ne fosse più di una). Alla successiva operazione di rendering, invece, le cose cambiano. Supponiamo che venga modificata una piccola parte della finestra: seguendo l'algoritmo del Pittore saremmo costretti a ridisegnare prima lo sfondo e poi la finestra (perchè più in rilievo). Una potenziale operazione di ridisegno superflua è la parte di sfondo che non è coperta da alcuna delle finestre, in quanto non modificata. Lo stesso ragionamento è applicabile anche alla stessa finestra, ma sulla base della modifica effettuata. Quindi un primo criterio di ottimizzazione potrebbe essere quello di basarsi solo sulle parti di immagine effettivamente modificate e lasciare inalterate le restanti. Una seconda possibile ottimizzazione è quella di non stampare parti di elementi coperti da altri (parzialmente o totalmente). Tuttavia non la terremo in considerazione perchè porterebbe ad avere una serie di problemi con gli oggetti semitrasparenti o trasparenti.

3.3.1 Render Area

Una Render Area rappresenta una sezione rettangolare dell'immagine che ha subito modifiche dall'ultima operazione di rendering. Usare un oggetto rettangolare, però, non può descrivere, precisamente, l'area di tutti i tipi di modifiche effettuabili (es: disegnamo un cerchio su di una finestra). Tuttavia è la scelta che rappresenta il miglior *tradeoff* tra la granularità dell'area rappresentabile e la velocità di drawing dell'algoritmo (più è complesso il bordo dell'area, maggiore è la complessità dell'algoritmo di drawing). Le Render Area, quindi, fanno riferimento al *gr_object* alla quale sono associate le modifiche: ne deriva che ogni istanza di *gr_object* dovrà avere anche una lista di Render Area. E' da notare che che queste aree possono essere portate a livelli superiori della gerarchia. Chiariamo con un esempio: abbiamo un desktop e una finestra che contiene un bottone. Se viene modificato il bottone questo avrà una nuova Render Area e andrà effettuata un'operazione di rendering prima sull'oggetto finestra e poi sul desktop. E' da notare che entrambe le operazioni di rendering hanno come obiettivo la stessa Render Area, ma solo su oggetti più alti nella gerarchia (bottone, finestra e infine il desktop). Questo perchè la modifica si propaga allo stesso modo ai livelli superiori. Questa è la potenzialità più grande introdotta dalle Render Area.

3.3.2 Ottimizzazioni di basso livello

Prima di passare all'algoritmo effettivo è bene considerare che alcune migliorie possono essere effettuate anche su operazioni più di basso livello, compiute dall'algoritmo. Quella più frequente è quella di copiare parte di un buffer video di un oggetto figlio sul buffer video del padre. Questa operazione implica, sostanzialmente, copie di porzioni di memoria, per cui è possibile utilizzare la funzione *memcpy()* piuttosto che copiare i pixel singolarmente con un ciclo for. Tuttavia la versione della *memcpy()* fornita dal nucleo originale non era del tutto ottimizzata. Viene qui mostrata la versione migliorata, con la quale si è avuto un grosso miglioramento delle prestazioni dell'algoritmo:

```
1 void *gr_memcpy(void *__restrict dest, const void *__restrict src,
      unsigned long int n)
2 {
3     PIXEL_UNIT *s1 = static_cast<PIXEL_UNIT*>(dest);
4     const PIXEL_UNIT *s2 = static_cast<const PIXEL_UNIT*>(src);
5     for(; 0<n; --n)*s1++ = *s2++;
6     return dest;
7 }
```

Similmente per la *memset()*:

```
1 void *gr_memset(void *__restrict dest, const PIXEL_UNIT value, unsigned
      long int n)
2 {
3     PIXEL_UNIT *s1 = static_cast<PIXEL_UNIT*>(dest);
4     for(; 0<n; --n)*s1++ = value;
5     return dest;
6 }
```

Le migliorie sono dovute alla presenza della keyword *__restrict* e all'uso dei soli puntatori (senza indici) nei cicli for di copia/scrittura.

3.3.3 Algoritmo semplificato (solo buffering)

Per semplicità verrà mostrata la prima versione dell'algoritmo, per poi introdurre quella più complessa. Come abbiamo visto nei paragrafi precedenti, un possibile algoritmo può essere suddivisto in due fasi:

1) individuare tutte le Render Area da ridisegnare;

2) disegnare solo le aree modificate.

Una nota sul primo punto: la responsabilità di creare le Render Area è del metodo *render()*. Questo metodo, come abbiamo detto, può essere definito dalle classi che ereditano *gr_object*, le quali dovranno occuparsi di generare le Render Area quando

necessario. Tuttavia la generazione delle Render Area tra oggetti semplici e container è in parte comune, facciamo riferimento ai casi di:

- ridimensione;
- spostamento;
- cambio di visibilità;
- cambio di profondità (z-index).

Questo può essere imputato alla presenza di una serie di membri acquisiti dalla classe *gr_object*, quali la dimensione e la posizione. Per tal motivo, la creazione di queste Render Area è affidata al metodo *render()* del container padre, quindi all'implementazione di default del metodo. E' possibile mostrare il metodo completo sotto forma di pseudo codice:

```
1  void gr_object::render()
2  {
3      // 1) Acquisizione delle Render Area dai figli di this
4
5      for(ogni oggetto obj del gr_object this)
6      {
7          // devo capire se l'oggetto e' stato spostato di posizione o
               cambiato di dimensione: in tal caso devo creare una
               render_unit che corrisponda esattamente alla dimensione e
               posizione della finestra in quanto l'oggetto deve essere
               ridisegnato sulla nuova area. E' inclusa un'ulteriore
               ottimizzazione in questo caso: se creo una render_unit
               grande quanto l'oggetto obj, e' inutile scorrere le
               ulteriori render_unit dell'oggetto in quanto sono tutte
               contenuta nella render_unit appena creata. Fa esclusione la
                render_unit "oldareaunit" (vedi dopo) perche' puo' anche
               sforare i bound dell'oggetto (unica eccezione ammessa in
               generale).
8
9          render_subset_unit *newareaunit = 0;
10         render_subset_unit *oldareaunit = 0;
11         bool modified = obj->is_pos_modified() || obj->is_size_modified
               ();
12
13         if(modified || now_visible || focus_changed)
14         {
15             newareaunit = new render_subset_unit(0, 0, obj->size_x, obj
                   ->size_y);
16
17             // cancello tutte le render_unit dell'oggetto (
                   OTTIMIZZAZIONE)
```

```
18              obj->clear_render_units();
19          }
20
21          // controllo di presenza della VECCHIA POSIZIONE/DIMENSIONE (
                AREA PRECEDENTEMENTE OCCUPATA):
22          if(modified || now_not_visible)
23          {
24              oldareaunit = new render_subset_unit(obj->old_pos_x, obj->
                    old_pos_y, obj->old_size_x, obj->old_size_y);
25
26              // visto che le coordinate old fanno gia' riferimento al
                    genitore corrente, faccio una offset al contrario, cosi
                    '
27              // che venga annullata quella all'interno del successivo
                    ciclo for (e' un trucco per risparmiare codice
                    ridondante)
28              oldareaunit->offset_position(obj->pos_x*-1, obj->pos_y*-1);
29          }
30
31          // resettiamo lo stato delle coordinate old e assegnamogli
                quelle correnti. anche la visibilita' old e focus sono
                coinvolti
32          obj->reset_status();
33
34          // la vecchia area potrebbe intersecarsi con la nuova, quindi
                magari e' meglio farne una che contiene entrambi in tal
                caso
35          if(newareaunit && oldareaunit && newareaunit->intersects(
                oldareaunit))
36          {
37              newareaunit->expand(oldareaunit);
38              obj->push_render_unit(newareaunit);
39              delete oldareaunit;
40          }
41          else // se, invece, le due aree non si intersecano, allora le
                tengo separate
42          {
43              if(oldareaunit)
44                  obj->push_render_unit(oldareaunit);
45              if(newareaunit)
46                  obj->push_render_unit(newareaunit);
47          }
48
49          // estraggo tutte le Render Area di obj
50          for(per ogni Render Area objunit di obj)
51          {
52              // se l'oggetto non e' visibile, allora nessuna Render Area
                    dovra' essere applicata
53              if(!obj->visible)
```

```
54                  continue;
55
56              // sistemo i riferimenti della Render Area, perche' le
                    coordinate sono relative a this
57              objunit->offset_position(obj->pos_x, obj->pos_y);
58
59              // la unit non deve sforare i bound dell'oggetto parente,
                    quindi dopo aver cambiato i riferimenti lo restringo,
                    se necessario
60              objunit->apply_bounds(this->size_x, this->size_y);
61
62              this->push_render_unit(objunit);
63          }
64      }
65
66      // 2) Inizio fase di stampa
67
68      for(per ogni oggetto figli obj di this)
69      {
70          // se l'oggetto non e' visibile, questo non deve essere
                visualizzato
71          if(!obj->visible)
72              continue;
73
74          for(per ogni Render Area r di obj)
75          {
76              <interseco r con l'area dell'oggetto obj, per non sforarne
                    i limiti>
77              if(!obj->trasparency)
78                  <copia porzione intersecata di obj su this>
79              else
80                  <copia porzione intersecata di obj su this usando con
                        alpha blending>
81          }
82      }
83  }
```

L'esempio mostra una possibile configurazione dell'albero degli oggetti grafici. E' possibile vedere come avvengono i trasferimenti di immagine tra i vari oggetti della gerarchia. Si parte con l'operazione di rendering su gr4 e gr5, necessari per poi renderizzare l'oggetto gr2, che usa l'implementazione già fornita della *render()*, in quanto container. Ricorsivamente, la stessa operazione può essere svolta anche sull'oggetto gr1.

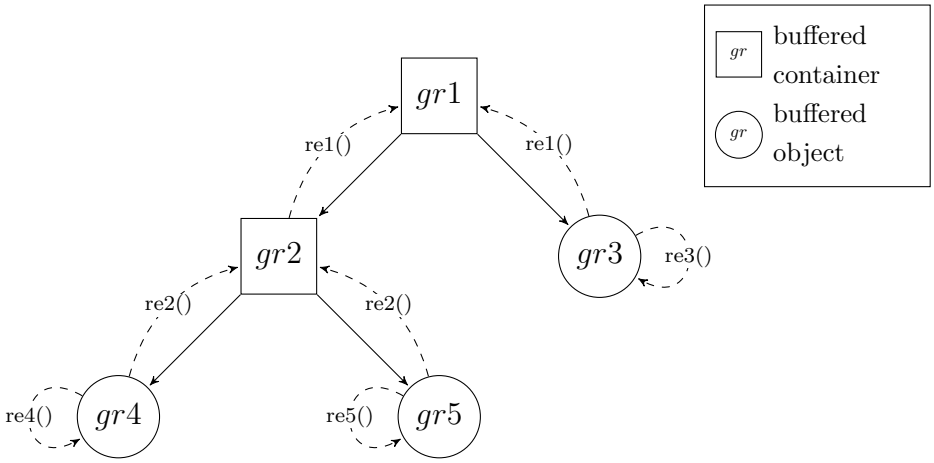


Figura 3.2: Esempio albero degli oggetti grafici

3.3.4 Algoritmo reale (con unbuffering)

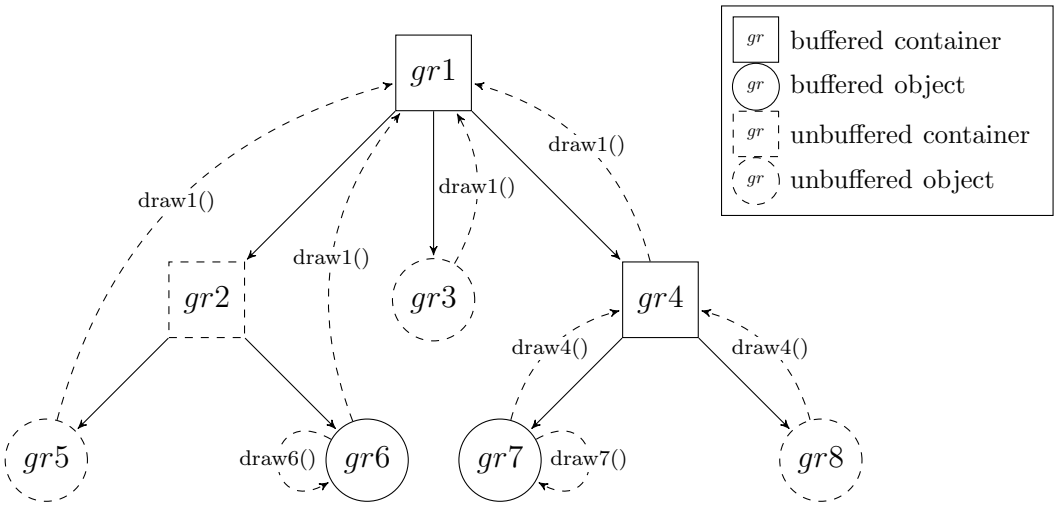


Figura 3.3: Esempio albero degli oggetti grafici (con oggetti unbuffered)

La descrizione dell'algoritmo reale è abbastanza complessa, per questo motivo si rimanda al sorgente del branch *feat-unbuffer*. I problemi che riguardano il primo algoritmo sono principalmente due: le operazioni di trasferimento del buffer video e la quantità di memoria allocata. Il primo problema riguarda il fatto che le operazioni possono solo partire dal figlio per giungere al padre: questo implica che in un albero sufficientemente esteso verticalmente, una modifica ad un oggetto foglia implichi una catena di chiamate al metodo *render()*, per ogni antenato. Il secondo problema

è causato dal fatto che ogni oggetto deve avere un buffer video per memorizzare il risultato dell'esecuzione del metodo *render()*, e questo spesso causa esagerati consumi di memoria Heap. Per risolvere questo problema introduciamo gli oggetti *unbuffered*, oggetti che non hanno un buffer per il metodo *render()*. Questa nuova astrazione permette di bypassare gli oggetti padre ed effettuare le operazioni di copia/costruzione dell'immagine direttamente sul buffer di un antenato. Definiamo il nuovo metodo virtuale *draw()* che si occuperà delle operazioni di copia per i container (implementazione di default) e di disegno/ridisegno per gli oggetti specializzati. Il metodo *render()* sfrutterà un altro metodo *recursive_render()*, appunto ricorsivo, che si occuperà di dirigere e generare tutte le chiamate alle *draw()* sugli oggetti discendenti, ma che abbiano come buffer di destinazione quello dell'oggetto sulla quale è stata chiamata la *render()*. Sfrutterà anche il metodo *build_render_areas()* per acquisire, ricorsivamente, tutte le Render Area dai primi discendenti *buffered* (container o non). Ci fermiamo alla descrizione perchè il codice è reso complesso dai molteplici controlli di dimensione e trasferimenti delle Render Area, dai discendenti all'oggetto da disegnare, si rimanda ai sorgenti ufficiali.

3.3.5 Profondità e Trasparenza

La libreria è compilabile con supporto agli 8 o 32 BPP di profondità. Tuttavia solo nella seconda modalità è possibile utilizzare le funzionalità di trasparenza: nella classica modalità a 32BPP uno dei quattro canali è l'*alpha channel*, che permette avere un livello di trasparenza variabile per ogni pixel dell'immagine. La trasparenza è, purtroppo, solamente utilizzabile per oggetti senza buffer, in quanto l'operazione di *draw()* deve operare sulla porzione di immagine al disotto dell'oggetto, per miscelare i colori.

Capitolo 4

Librerie di supporto

4.1 LibGr

Questa libreria grafica mette ad disposizione quattro funzioni per abbassare la complessità dell'algoritmo di rendering e dei metodi *draw()* ridefiniti.

```
1 void inline set_pixel(PIXEL_UNIT * buffer, int x, int y, int MAX_X, int
      MAX_Y, PIXEL_UNIT col);
```

Stampa un pixel di colore col sulla cordinata (x,y) del buffer, tenendo conto dei limiti di dimensione MAX_X e MAX_Y. Se il punto è fuori dall'immagine stampa un errore per segnare l'evitato buffer overflow.

```
1 // solo per 32BPP !!
2 void inline set_pixel_alpha_blend(PIXEL_UNIT * buffer, int x, int y,
     int MAX_X, int MAX_Y, PIXEL_UNIT src_pixel);
```

E' una versione modificata della precedente funzione, con la differenza che il pixel stampato è il risultato di un'operazione di *alpha blending* tra il pixel (x,y) del buffer e src_pixel.

```
1 void *gr_memcpy(void *__restrict dest, const void *__restrict src,
     unsigned long int n);
2 void *gr_memset(void *__restrict dest, const PIXEL_UNIT value, unsigned
      long int n);
```

Queste due funzioni sono già state mostrate nel sottocapitolo dei Possibili Miglioramenti nel capitolo sulla libreria SCGL. Si occupano di stampare o copiare parti di buffer video. Sono state ottimizzate perchè fanno parte delle funzioni più computazionalmente rilevanti dell'algoritmo.

4.2 LibTGA

```
1  class TgaParser {
2      unsigned char *src_file;
3      tga_header* header;
4      unsigned char *img_data;
5
6  public:
7      TgaParser(void *src_file);
8      bool is_valid();
9      int get_width();
10     int get_height();
11     int get_depth();
12     void to_bitmap(void *dest);
13 };
```

L'implementazione del caricamento da file di una bitmap può essere molto utile per acquisire immagini create esternamente al nucleo. Tuttavia bisogna allinearsi ad un formato diffuso: tra questi scegliamo il formato *(*TarGA) perchè supporta le bitmap senza compressione. Questo porterà a problemi di dimensioni dell'immagine, ma renderà più semplice l'algoritmo da sviluppare. Il formato *TarGA* prevede una serie di campi contenenti informazioni sull'immagine, ma tra questi possiamo evidenziare *img_descriptor*, i quali bit 4 e 5 indicano in che verso è rappresentata la bitmap all'interno del file, e la profondità del colore. Attualmente sono supportare solo due direzioni. Inoltre la classe non può sfruttare alcun metodo di lettura sul file perchè il Nucleo non supporta nessun tipo di file system. Per aggirare questo problema le immagini vengono incluse nel progetto utilizzando grossi file header nella quale sono presenti gli array dei byte dei file *TarGA*. L'array viene dato in pasto al costruttore della classe *TgaParser*, mostrata all'inizio. Il metodo *to_bitmap()* permette di estrarre l'immagine nel classico formato 0xARGB.

4.3 LibFont

Fino ad ora non abbiamo discusso di come poter effettuare il rendering di un testo, spesso conosciuto come operazione di *rastering*. Esistono varie tecniche a tale scopo, ma, per semplificare il più possibile, scegliamo quella della font bitmap: ci basiamo su un'immagine nella quale è presente una tabella di caratteri (16x16 bit ognuno) ordinata secondo quella dei caratteri ASCII.
Di seguito una lista dei metodi forniti:

```
1  void _libfont_init();
```

Chiamata nella *window_init()*, serve per caricare la bitmap da un file TarGA. Va eseguita prima di usare qualsiasi altra funzione di questa libreria.

```
1 int set_fontchar(PIXEL_UNIT *buffer, int x, int y, int MAX_X, int MAX_Y
    , int nchar, PIXEL_UNIT text_color, PIXEL_UNIT backColor);
```

Stampa un carattere a partire da (x,y) sul buffer. Vanno rispettati tutti i limiti MAX_X e MAX_Y per ogni pixel del carattere da stampare. Va specificato colore del testo e colore di sfondo.

```
1 int get_fontstring_width(const char * str);
```

Ottiene la lunghezza totale dei caratteri della stringa passata come parametro.

```
1 int get_charfont_width(char c);
```

Ottiene la larghezza del carattere specificato.

```
1 void set_fontstring(PIXEL_UNIT *buffer, int MAX_X, int MAX_Y, int x,
    int y, int bound_x, int bound_y, const char * str, PIXEL_UNIT
    text_color, PIXEL_UNIT backColor, bool print_caret=false);
```

Rasterizza una stringa sul buffer video, contenendo il testo nel box con coordinate (x,y) e dimensione (bound_x, bound_y). Vanno rispettati i limiti MAX_X e MAX_Y per ogni pixel stampato. E' possibile concatenare un *caret* (cursore del testo) usando il parametro di default *print_caret*.

Capitolo 5

Processo WinMan/Drawer

5.1 Introduzione

Il processo WinMan/Drawer ha due competenze:

- rapportarsi con i processi utente attraverso la API;
- acquisire gli eventi da mouse e tastiera, creando un rapporto con i relativi processi esterni.

5.2 Coda dei Task, API Utente e Interazione con dispositivi di I/O

Le API per l'utente complete sono mostrate nel capitolo successivo, ma in sintesi permettono di creare il rapporto tra i processi utente e il processo qui descritto. Il meccanismo implementato per la gestione dei task (cioè delle operazioni che il processo deve svolgere) è quasi del tutto simile alla soluzione del problema Produttori/Consumatore, comunissimo nella letteratura. In questo caso il Consumatore è rappresentato dal processo WinMan/Drawer, mentre i Produttori possono essere due:

i processi utente le quali chiamate alle API causano un inserimento di un'operazione nella coda;

i processi esterni di mouse e tastiera i quali possono inoltrare gli eventi (chiamati *IO* per distinguerli da quelli utente) attraverso la coda dei task.

5.2.1 Eventi IO

```
1 enum {MOUSE_UPDATE_EVENT, MOUSE_Z_UPDATE_EVENT, MOUSE_MOUSEUP_EVENT,
    MOUSE_MOUSEDOWN_EVENT, KEYBOARD_KEYPRESS_EVENT}
```

Per semplificazione non sono stati implementati gli eventi di KEY_UP e KEY_DOWN della tastiera. L'implementazione degli eventi del mouse, invece, è completa.

5.2.2 Eventi inoltrabili ai processi utente

```
1 enum user_event_type {NOEVENT, USER_EVENT_MOUSEZ, USER_EVENT_MOUSEUP,
    USER_EVENT_MOUSEDOWN, USER_EVENT_KEYBOARDPRESS, USER_EVENT_RESIZE,
    USER_EVENT_CLOSE_WINDOW};
```

Ogni evento può portare con sè diverse informazioni, dipendentemente dal tipo di evento. E' possibile consultare tutti i campi nella struttura *des_user_event* o consultare il capitolo sulle API Utente.

5.3 Funzionalità di gestione finestre

Il processo, come abbiamo visto, può filtrare alcuni eventi della quale è chiamato ad agire: in specifico parliamo degli eventi di click sui bordi e sulle topbar degli oggetti gr_window. Questo perchè il processo svolge delle funzioni di gestione delle finestre: trascinamento, ridimensionamento e chiusura (tasto x rosso). Entrambi le funzioni sono collegate agli eventi di click e spostamento del cursore. Nulla toglie, però, che questa categoria di eventi IO possa causare l'inoltro di uno o più eventi ai processi utente. Possiamo specificare i vari casi:

trascinamento nessun evento utente generato;

ridimensione è generato un evento globale a livello di finestra, per permettere al processo utente di ridimensionare i componenti in maniera personalizzata;

chiusura è generato un evento di chiusura destinato alla finestra;

5.4 Drawing

La seconda competenza del processo è il Drawing: l'obiettivo è costruire, complessivamente, il desktop, utilizzando l'astrazione dei *gr_object* introdotta dalla Libreria SCGL. Questi oggetti vengono costruiti in parte all'avvio (sfondo e label memoria) e in parte a seguito dello svolgimento dei task di creazione/aggiornamento di finestre/oggetti. Bisogna, inoltre, gestire il rapporto fra la Libreria SCGL e la Scheda

Video, e implementare il metodo del Double Buffering, per risolvere i problemi visti nel capitolo sull'analisi.
Come integrare il *framebuffer* della scheda video nella gerarchia degli oggetti grafici? Il costruttore di *gr_object* permette di specificare il buffer video dell'istanza che si sta creando, sostituendolo a quello che lo stesso costruttore dovrebbe allocare: questo ci permette di creare un nuovo *buffered gr_object* il cui metodo *render()* opera direttamente sul *framebuffer*.
Come integrare il metodo del Double Buffering nella libreria? Questo metodo, come visto all'inizio, consistere nel realizzare tutte le operazioni parziali di ridisegno su un buffer temporaneo, per poi copiare tutto, alla fine, sul *framebuffer*: se pensiamo all'implementazione della libreria possiamo notare che ogni *buffered gr_object* fa più o meno la stessa cosa, cioè memorizza il risultato del metodo *render()* sul proprio buffer. Da questo si ha che basta istanziare un altro oggetto *gr_object* bufferato, chiamato *doubled_framebuffer_container*, figlio di *framebuffer_container*, con le medesime dimensioni e con posizione (0,0). Si tratta di un contenitore che "ricopre" esattamente quello del *framebuffer*. Tutti gli oggetti del desktop dovranno discendere da questo nuovo contenitore: in tal modo il metodo *doubled_framebuffer_container.render()* costruirà il double buffer e il metodo *framebuffer_container.render()* copierà il double buffer sul *framebuffer*. Il fatto che l'astrazione realizzata dalla libreria SCGL possa risolvere anche questi due problemi implica che, in entrambi i casi, saranno mantenute tutte le ottimizzazioni implementate nell'algoritmo: il beneficio che si ottiene in termini di prestazioni è molto alto. Nella figura è rappresentata l'implementazione completa del desktop.

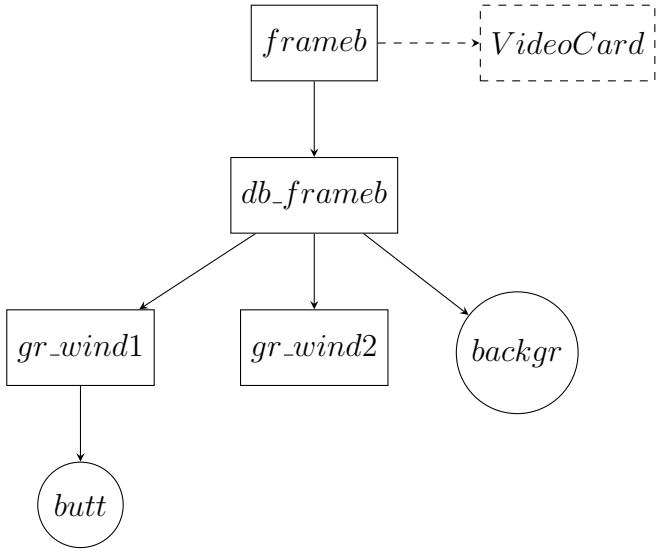


Figura 5.1: Implementazione del desktop completo usando la libreria SCGL

5.5 gr_window

L'oggetto più complesso da disegnare e gestire è la finestra, perchè deve essere popolabile con i componenti scelti dal processo utente, deve essere sempre composta da alcuni elementi (es. topbar e bordi) ed è il contenitore per gli eventi utente che vengono generati. Una finestra può essere vista come un container specializzato, cioè un container in cui la gestione dei figli è realizzata attraverso metodi non elementari. Per specializzare un contenitore è sufficiente creare una nuova classe che erediti *gr_object* ma che non ridefinisca il metodo *render()* e *draw()*: in tal modo la classe si comporterà sempre da contenitore, ma con la possibilità di ampliarla con i metodi specializzati. Ogni finestra è composta sempre da:

- una *topbar* (barra superiore) sulla quale è presente un pulsante di chiusura e una label titolo;
- quattro bordi per ogni lato;
- un *inner_container*, nella quale verranno inseriti i componenti utente.

Le competenze della classe, come visto prima, sono:

- costruire l'oggetto finestra con tutti gli elementi visti nella precedente lista;
- gestire gli eventi IO destinati a questi elementi;
- permettere l'aggiunta e la gestione dei componenti utente (*u_windowObject*);
- gestire gli eventi utente inoltrati dal processo WinMan/Drawer.

Di seguito la dichiarazione della classe, in parte semplificata:

```
1  class gr_window : public gr_object
2  {
3  private:
4      // componenti della finestra
5      gr_object * topbar_container;
6      gr_bitmap * topbar_bitmap;
7      gr_object * inner_container;
8      gr_bitmap * background_bitmap;
9
10     gr_bitmap * border_left_bitmap;
11     gr_bitmap * border_right_bitmap;
12     gr_bitmap * border_top_bitmap;
13     gr_bitmap * border_bottom_bitmap;
14
15     gr_button * close_button;
16     gr_label * title_label;
```

```
17
18 public:
19     gr_window(int pos_x, int pos_y, int size_x, int size_y, int z_index
           , const char* title);
20
21     // costruttore usato con l'oggetto utente
22     gr_window(u_basicwindow* u_wind);
23
24     // construttore comune
25     void constructor(const char * title);
26
27     // distruttore
28     ~gr_window();
29
30     // funzione per aggiornare lo stato della finetra attraverso un
           oggetto finestra utente
31     void update_window_from_user(u_basicwindow * u_wind);
32
33     // funzioni per la gestione delle coordinate
34     void set_content_size_x(int newval);
35     void set_content_size_y(int newval);
36     int offset_size_x(int offset);
37     int offset_size_y(int offset);
38     bool get_draggable();
39     bool get_resizable();
40     void set_draggable(bool draggable);
41     void set_resizable(bool resizable);
42
43     void set_title(const char *str);
44
45     // metodo per capire se ho cliccato su bordi o angoli della
           finestra
46     border_flags get_clicked_borders(int rel_pos_x, int rel_pos_y);
47
48     // metodo da invocare ogni qual volta viene fatto un resize della
           finestra (ridimensiona gli oggetti interni)
49     void do_resize();
50
51     // funzioni per gli oggetti passati dall'utente al nucleo
52     gr_object *add_user_object(u_windowObject * u_obj);
53     bool update_user_object(u_windowObject * u_obj);
54     gr_object *search_user_object(u_windowObject * u_obj);
55
56 private:
57     bool draggable;
58     bool resizable;
59
60     // funzioni per la gestione degli eventi
61     gr_object *focused_object;
```

```
62      gr_object *clicked_down_object;
63      des_user_event *event_head;
64      des_user_event *event_tail;
65
66      void user_event_push(des_user_event * event);
67      void user_event_push_on_focused(int abs_x, int abs_y,
            des_user_event * event);
68
69      // gestisce gli eventi (interni) destinati alla topbar
70      window_action click_on_topbar(gr_object * dest_obj, mouse_button
            butt, bool mouse_down);
71
72  public:
73      // gestione del focus interno degli oggetti utente
74      bool set_focused_child(gr_object *obj);
75      void clear_focused_child();
76      void process_tick_event();
77
78      // processa tutti gli eventi destinati alla finestra (topbar
            compresa, ad eccezione dei bordi)
79      window_action click_event(gr_object *dest_obj, int abs_pos_x, int
            abs_pos_y, mouse_button butt, bool mouse_down);
80
81      // metodi specifici per ogni tipo di evento utente da inoltrare
82      natl event_sem_sync_notempty;
83      des_user_event user_event_pop();
84      void user_event_add_mousemovez(int delta_z, int abs_x, int abs_y);
85      void user_event_add_mousebutton(user_event_type event_type,
            mouse_button butt, int abs_x, int abs_y);
86      void user_event_add_keypress(char key);
87      void user_event_add_resize(int delta_pos_x, int delta_pos_y, int
            delta_size_x, int delta_size_y);
88      void user_event_add_close_window();
89  };
```

5.6 Pseudo codice del processo

```
1   struct des_windows_man
2   {
3       // Variabili di stato
4       <Variabili di gestione del focus su finestre e oggetti>;
5       <Variabili per gestire lo stato di resize o trascinamento>;
6
7       // Coda richieste primitive
8       des_window_req req_queue[MAX_REQ_QUEUE];
9       natb top;
```

```
10      natb rear;
11
12      // Semafori di sincronizzazione
13      natb mutex;
14      natb sync_notempty;
15      natb sync_notfull;
16  };
17
18
19  void main_windows_manager(int n)
20  {
21      des_cursor main_cursor = {0,0,0,0};
22      win_man.is_dragging=false;
23
24      while(true)
25      {
26          sem_wait(win_man.sync_notempty);
27          sem_wait(win_man.mutex);
28
29          des_window_req newreq;
30          windows_queue_extract(win_man, newreq);
31
32          switch(newreq.act)
33          {
34              case task_API:
35                  <svolgi task API>;
36              case task_evento_IO:
37                  if(evento destinato a bordi o topbar)
38                  {
39                      <svolgi routine evento di gestione della finestra>;
40                      <crea e inoltra eventi utente se previsto>;
41                  }
42                  else if(evento destinato a finestra)
43                      <crea e inoltra evento utente alla finestra di
                            destinazione>;
44
45                  if(evento mouse)
46                      <aggiorna stato del cursore>;
47          }
48
49          sem_signal(win_man.mutex);
50          sem_signal(win_man.sync_notfull);
51      }
52  }
```

Parte II

Interfaccia Utente

Capitolo 6

Interfaccia Utente

6.1 API Grafiche per l'Utente

Un processo può creare una o più finestre e gestirne il contenuto attraverso un'Application Programming Interface. Le funzioni messe a disposizione del programmatore sono le seguenti:

```
1  class u_basicwindow {
2  public:
3      int w_id;
4
5      int pos_x;
6      int pos_y;
7      int size_x;
8      int size_y;
9      char title[150];
10
11     bool visible;
12     bool draggable;
13     bool resizable;
14 };
```

```
1  extern "C" void crea_finestra(u_basicwindow * u_wind);
```

Questa funzione crea una finestra, le cui proprietà sono descritte nel parametro di tipo *u_basicwindow*. E' possibile specificare se la finestra è visibile, trascinabile e ridimensionabile.

```
1  extern "C" int chiudi_finestra(int w_id);
```

Chiude una finestra, deallocando tutti i componenti interni. In questo caso non è necessario passare l'indirizzo della struttura *u_basicwindow*, ma è sufficiente indicare

solo l'id della finestra.

```
1 extern "C" void aggiorna_finestra(u_basicwindow * u_wind, bool sync);
```

Nel caso in cui il processo utente voglia aggiornare posizione, dimensione, titolo e tutte le altre proprietà viste nella struttura *u_basicwindow*, è possibile usare questa funzione, alla quale è sufficiente passare l'istanza di *u_basicwindow* associata alla prima crea_finestra.

```
1 extern "C" int crea_oggetto(int w_id, void * obj);
```

Ogni processo può popolare la finestra che ha creato attraverso l'aggiunta di componenti. Ogni componente è rappresentato da una classe che eredita la classe astratta u_windowObject. *w_id* è l'id della finestra sulla quale l'operazione deve essere effettuata. Gli oggetti *u_windowObject* già implementati in questo progetto sono: *u_label*, *u_textbox*, *u_button*, *u_progressbar*, *u_checkbox*, le cui dichiarazioni possono essere trovate all'interno dei sorgenti del progetto, in particolare nel file *include/-windows/sys.h*.

```
1 extern "C" void aggiorna_oggetto(int w_id, u_windowObject * u_obj, bool
      sync);
```

Similmente alla funzione aggiorna_finestra, abbiamo una equivalente aggiorna_oggetto per i componenti già creati nella finestra con id *w_id*. Dobbiamo provvedere a fornire anche l'istanza del u_windowObject associata alla prima crea_oggetto.

```
1 extern "C" des_user_event preleva_evento(int w_id);
```

Questa funzione serve per acquisire gli eventi associati alla finestra con id *w_id*. L'evento è rappresentato da un'istanza della struttura des_user_event, della quale segue la dichiarazione:

```
1  enum user_event_type {NOEVENT, USER_EVENT_MOUSEZ, USER_EVENT_MOUSEUP,
       USER_EVENT_MOUSEDOWN, USER_EVENT_KEYBOARDPRESS, USER_EVENT_RESIZE,
       USER_EVENT_CLOSE_WINDOW};
2  enum mouse_button {LEFT,MIDDLE,RIGHT};
3
4  struct des_user_event
5  {
6      int obj_id;
7      user_event_type type;
8      union
9      {
10         mouse_button button;     //mousebutton
```

```
11          int delta_z;          //mousez
12          natb k_char;          //tastiera
13          int delta_pos_x;      //resize
14      };
15      union
16      {
17          int rel_x;        //mousebutton
18          natb k_flag;          //tastiera
19          int delta_pos_y;      //resize
20      };
21      union
22      {
23          int rel_y;        //mousebutton
24          int delta_size_x;     //resize
25      };
26      union
27      {
28          int delta_size_y;     //resize
29      };
30  };
```

La presenza delle union è dovuta al fatto che ogni tipo di evento porta con sè differenti informazioni, le quali differiscono sia in numero che tipologia. L'uso della union è una semplificazione, ma si sarebbe potuta definire una classe astratta *evento* dalla quale far derivare tutte le classi specializzate.

Tutte le funzioni e le strutture qui viste sono messe a disposizione dei programmi del Modulo Utente attraverso il file header *sys.h*. La struttura per la gestione degli eventi, invece, è fornita dal file *user_event.h*. Entrambi i file sono presenti nella cartella *include/windows/* dei sorgenti del nucleo.

6.2 Libreria SWL

La libreria **S**imple **W**indows **L**ibrary è una libreria che fa parte del *Middleware* del progetto. Serve per semplificare, a beneficio dei programmi utente, la gestione delle finestre e dei componenti. Inoltre mette a disposizione una serie di metodi per la gestione degli eventi, già implementati, il cui comportamento può essere alterato tramite un'altra serie di metodi.

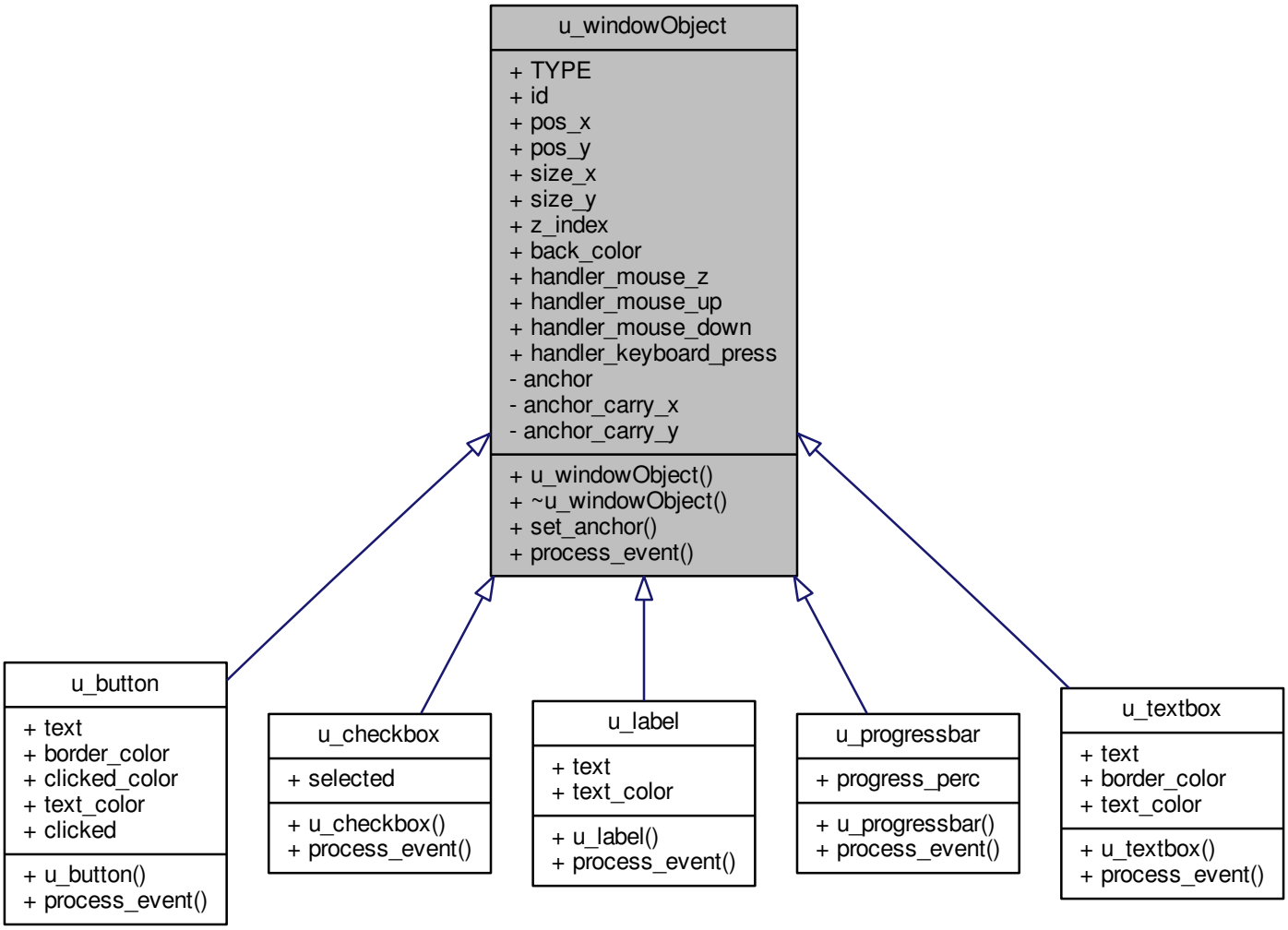
La semplificazione della gestione degli oggetti è realizzato tramite la classe *u_window* qui mostrata:

```
1  class u_window
2  {
3  private:
4      u_basicwindow sysprop;
5
```

```
6       // lista oggetti finestra (id e puntatori a struct)
7       static const int MAX_USER_OBJECTS=100;
8       int objs_ident[MAX_USER_OBJECTS];
9       u_windowObject * objs[MAX_USER_OBJECTS];
10      int objs_count;
11
12  public:
13      // handler dell'utente relativi a eventi della finestra
14      void(*handler_closing_window)(des_user_event event);
15      void(*handler_closed_window)(des_user_event event);
16
17      u_window(int size_x, int size_y, int pos_x, int pos_y, const char *
            title) : objs_count(0), handler_closing_window(0),
            handler_closed_window(0);
18
19      // metodi per la modifica delle proprieta' della finestra
20      void resizable(bool newval);
21      void draggable(bool newval);
22      void apply_changes(bool sync=true);
23
24      void show(bool sync=true);
25
26      // gestione dei componenti
27      bool add_object(u_windowObject* obj);
28      bool update_object(u_windowObject* obj, bool async=false);
29      u_windowObject * get_object(int id);
30      void next_event();
31  };
```

Qui, invece, è presente la gerarchia delle classi *u_windowObject* e quelle che la ereditano, nella quale è possibile notare una serie di membri *handler_** con la quale viene semplificata la gestione degli eventi utente. Queste classi, come visto precedentemente, sono usate anche per le *API Call* di gestione dei componenti (creazione, aggiornamento ed eliminazione). Il metodo *process_event()* è virtuale non puro, e fornisce un'implementazione per la gestione degli eventi per ogni tipo di oggetto. L'implementazione della classe base è quella che permette di unificare il codice di gestione di eventi comuni ad ogni componente grafico (es. eventi del mouse/tastiera). Le implementazioni delle classi derivate realizzano funzionalità specifiche dell'oggetto.

E' possibile visionare una serie di programmi utente di esempio, realizzati sfruttando la libreria SWL, nella cartella *utente/prog/* dei sorgenti del progetto.


u_windowObject
+ TYPE
+ id
+ pos_x
+ pos_y
+ size_x
+ size_y
+ z_index
+ back_color
+ handler_mouse_z
+ handler_mouse_up
+ handler_mouse_down
+ handler_keyboard_press
- anchor
- anchor_carry_x
- anchor_carry_y
+ u_windowObject()
+ ~u_windowObject()
+ set_anchor()
+ process_event()
u_button
+ text
+ border_color
+ clicked_color
+ text_color
+ clicked
+ u_button()
+ process_event()
u_checkbox
+ selected
+ u_checkbox()
+ process_event()
u_label
+ text
+ text_color
+ u_label()
+ process_event()
u_progressbar
+ progress_perc
+ u_progressbar()
+ process_event()
u_textbox
+ text
+ border_color
+ text_color
+ u_textbox()
+ process_event()

Parte III

Conclusioni

Capitolo 7

Possibili miglioramenti

Il lavoro realizzato è abbastanza esteso e in certe parti incompleto. Questa è una lista dei problemi e delle possibili estensioni del progetto:

7.1 Libreria SCGL

7.1.1 Astrazione

- Importante: implementare interamente il Design Pattern *Component*. Attualmente la classe *gr_object* non è astratta, ma di default si comporta come un container. Ulteriori annotazioni sono incluse nei sorgenti di *gr_object.cpp* del branch.
- Non è possibile utilizzare il design pattern *Decorator* per progettare delle classi decorative, per via del buffer. Sarebbe molto utile nel caso della classe *gr_window*.

7.1.2 Algoritmo

- Sostituire le restrizioni da semplici coordinate a rettangoli (si possono usare le Render Area) per ridurre il numero di parametri da passare al metodo *draw()*.
- Realizzare la v3 dell'algoritmo, nella quale ogni Render Area si scompone in più Render Area nella quale ridisegnare solo un sottoinsieme degli oggetti, tenendo conto delle sovrapposizioni. Valutare se quest'algoritmo può migliorare la complessità nel caso medio.
- Delegare le funzioni di stampa/blending alla LibGr.

- Realizzare un layer tra le funzioni new/delete e l'allocazione/deallocazione degli oggetti e dei buffer video della libreria.

7.2 Librerie di Supporto

7.2.1 LibTGA

- Attualmente sono supportate solo due direzioni di rappresentazione della matrice linearizzata dei pixel. Completare le altre due direzioni e rendere il codice più leggibile

- Supporto alla compressione? Le immagini compresse non sono supportate.

7.2.2 LibFont

- La versione dell'Algoritmo con oggetti unbuffered ha costretto ad aggiornare le funzioni di rastering del testo per supportare il rendering parziale. Tuttavia la soluzione implementata non è efficiente perchè le restrizioni sono valutate pixel per pixer, quando in realtà è meno complesso farlo carattere per carattere. Inoltre sussiste lo stesso problema della draw: la restrizione è rappresentata in termini di coordinate e dimensioni, le funzioni hanno troppi parametri.

- Creare un rasterizer serio senza che si basi un una font bitmap.

7.3 Processo WinMan/Drawer

7.3.1 Coda delle richieste/eventi

- La API Utente prevede metodi asincroni: l'asincronicità non è completa perchè se la coda è piena il processo è comunque costretto ad aspettare che almeno una richiesta sia prelevata. Sarebbe meglio restituire un errore in questo caso.

7.3.2 gr_window

- Gestire in modo migliore l'allocazione degli oggetti figli della finestra. Quando viene effettuata un'operazione di resize molto spesso capita che la memoria Heap venga saturata e che l'operazione non possa essere portata a termine. Attualmente questa situazione porta ad un crash di tutto il sistema.

- La classe ha troppe competenze, sarebbe meglio creare una classe dedicata alla gestione degli eventi separandola dal resto.

- Progettare delle classi decorative usando il pattern *Decorator*. Servirebbero per creare finestre con bordi, senza bordi, con topbar o senza. Tuttavia manca il supporto da parte della libreria gr_object.

7.3.3 API utente

- Implementare il design pattern *Factory* per omogeneizzare il codice adibito alla conversione dalle classi che ereditano *u_windowObject* alle rispettive derivate da *gr_object*.

Capitolo 8

Risultati

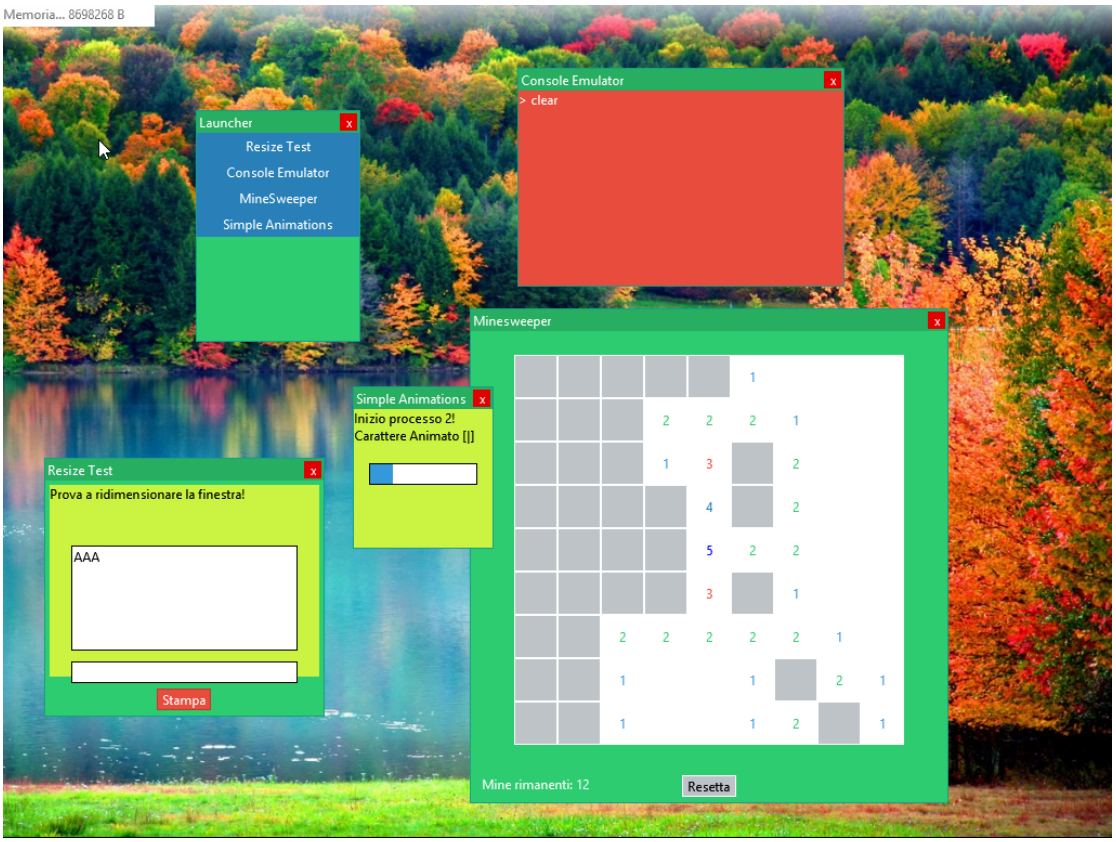


Figura 8.1: Desktop completo

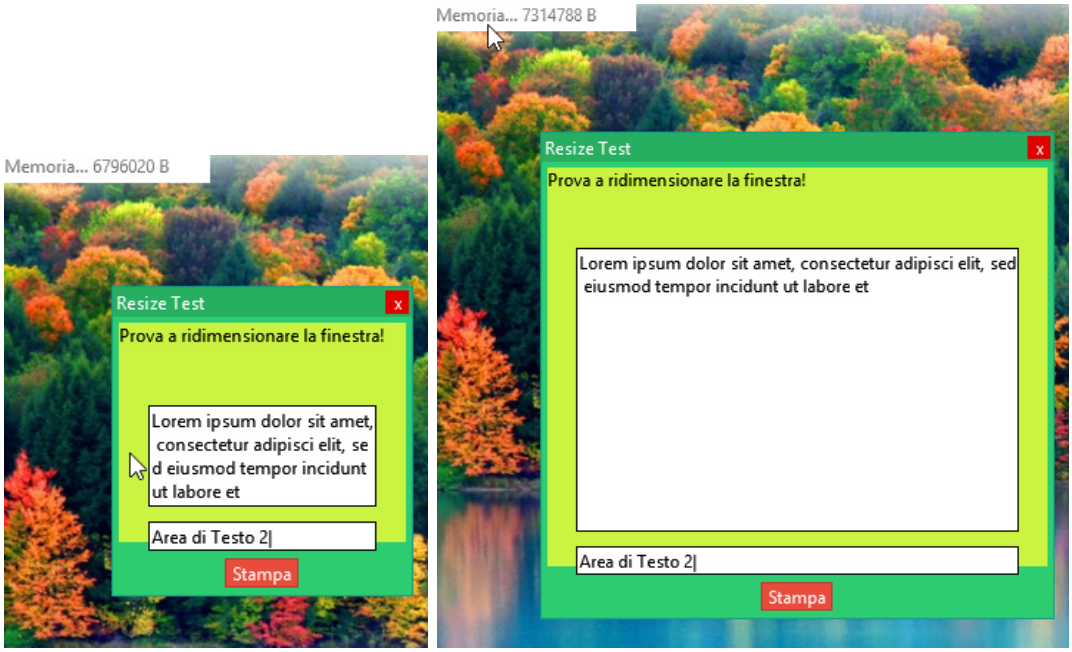


Figura 8.2: Esempio di Resize e gestione dell'evento

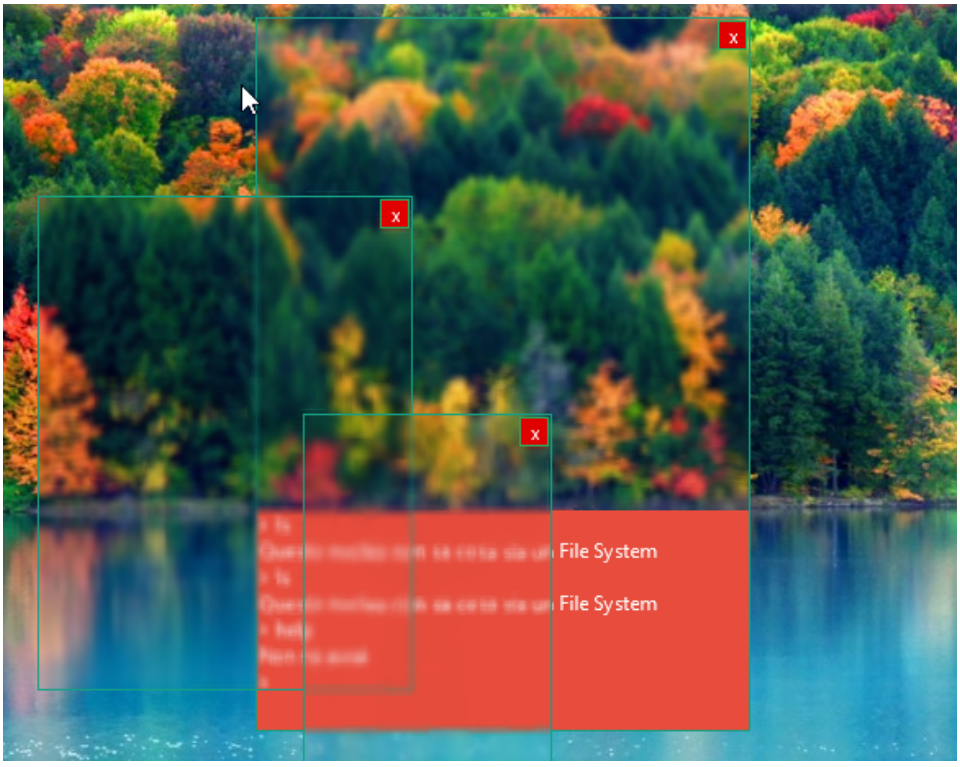


Figura 8.3: Esempio avanzato Blur Gaussiano