YX OS

Volume 3: The Buddy GUI

VERSION 0.5 • 21-Aug-14

Document Identification

OBJECTIVE		
To document Buddy GUI for YX OS.		
STAKEHOLDERS		
Name	**Role**	**Contact**
Tomaz Stih	Software Engineer	tstih@yahoo.com

DOCUMENT HISTORY				
Date	**Version**	**Description**	**Created**	**Approved**
Sep 23rd 2013	0.1	Initial draft.	Tomaz Stih	Tomaz Stih
Aug 19th 2014	0.2	Added windows messages.	Tomaz Stih	Tomaz Stih
Aug 20th 2014	0.3	Added message dispatching	Tomaz Stih	Tomaz Stih
Aug 21st 2014	0.4	Added graphics	Tomaz Stih	Tomaz Stih
Aug 21st 2014	0.5	Added the Architecture chapter	Tomaz Stih	Tomaz Stih

Table of Contents

Introduction

This is the technical documentation for Buddy GUI. It deals with the internals of Buddy – a simple but complete modern windows GUI written in the C programming language to be used in limited 8 environments (i.e. slow computers, low memory, etc.).

It is assumed that you already are familiar with modern GUI environments and understand the basics: such as a window, a rectangle, an event, a message, an event-loop and a window procedure. It will not deal with what these are - but how they are implemented.

You should also be fairly familiar with the C programming language including advanced concepts such as pointers to functions.

Although this document is for YX OS the concepts are described in an OS independent manner.

Rectangles

This chapter describes core Buddy algorithms for handling rectangles.

Definitions

All Buddy windows are rectangles. Basic windows operations such as containment or intersection are operations on rectangles. Therefore we start a chapter on algorithms by defining a rectangle and describing some basic algorithms that operate on a group of rectangles.

Rectangle Coordinates

A rectangle has four coordinates:

- **x** … the left coordinate,
- **y** … the top coordinate,
- **w** … width, and
- **h** … height.

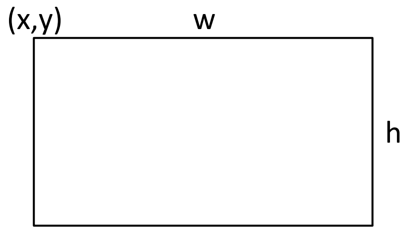

Figure 1. Rectangle coordinates

Coordinates x0, y0, x1, and y1 are defined as:

- **x0** … x,
- **y0** … y,
- **x1** … x0 + w – 1, and
- **y1** … y0 + h - 1.

If w and h are both 1 then x0 = x1 in y0 = y1. If w and h are both 0 then rectangle is called the **zero rectangle**.

Rectangle **vertices** are points A(x0,y0), B(x1,y0), C(x0,y1), and D(x1,y1). Rectangle **edges** are lines connecting following vertices: AB, AC, BD, and AD. Point T(x,y) i.e. point A is rectangle's **origin**.

The Z-Order

If we draw a rectangle one on top of another, we create an order. A general definition says that Z-order is an ordering of overlapping two-dimensional objects, such as windows in a graphical user interface (GUI), or shapes in a vector graphics editor.

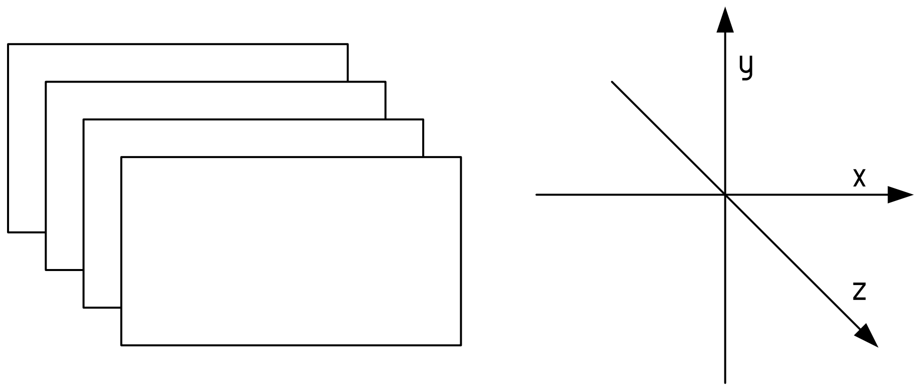

Figure 2. The Z-Order

Algorithms

Rectangle Intersection

Two rectangles intersect if we can place any vertex of any rectangle inside another. A brute force method requires checking all eight vertices.

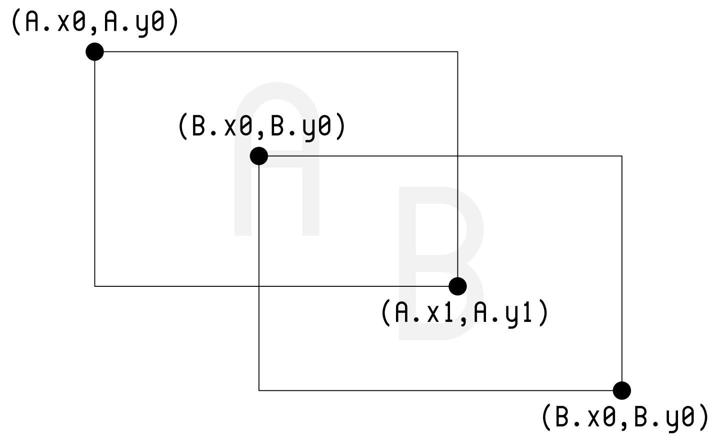

Figure 3. Rectangle intersection

A more elegant approach is to negate the success criteria. Two rectangles do not intersect when one is above, bellow, left or right of another. Let us write down these conditions for rectangles A and B:

- **A.y1 < B.y0** … A is above B
- **A.y0 > B.y1** … A is bellow B
- **A.x1 < B.x0** … A is left from B
- **A.x0 > B.x1** … A is right from B

Now we can derive our intersection algorithm:

```
algorithm RectsOverlap
input:      Rectangle A
            Rectangle B
output:     Answer yes/no
begin
      return !(A.y1 < B.y0 || A.y0 > B.y1 || A.x1 < B.x0 || A.x0 > B.x1)
end
```

This algorithm helps us quickly determine if two rectangles intersect. The algorithm also helps us determine coordinates of the actual intersection: if two rectangles intersect then the coordinates are larger of x0 and y0; and smaller of x1 and y1 of both rectangles. Let us write this:

```
algorithm GetRectsIntersect
input:      Rectangle A
            Rectangle B
output:     Answer Intersect
            Answer yes/no
begin
      if call RectsOverlap(A,B)
            Intersect.x0 ← max(A.x0,B.x0), Intersect.y0 ← max(A.y0,B.y0)
            Intersect.x1 ← min(A.x1,B.x1), Intersect.y1 ← min(A.y1,B.y1)
            return yes
      else return no
end
```

Rectangle Decomposition / Subtraction

Imagine cutting out a small rectangle B out of larger rectangle A. We make a hole into rectangle A and create something that looks like a picture frame. Now we want to decompose this “frame” of rectangle A into smaller rectangles so that they cover the result of subtraction. Because the smaller rectangle is in the middle of the larger one: the remains of rectangle A are decomposed to rectangles: A1, A2, A3, and A4. This figure shows the operation.

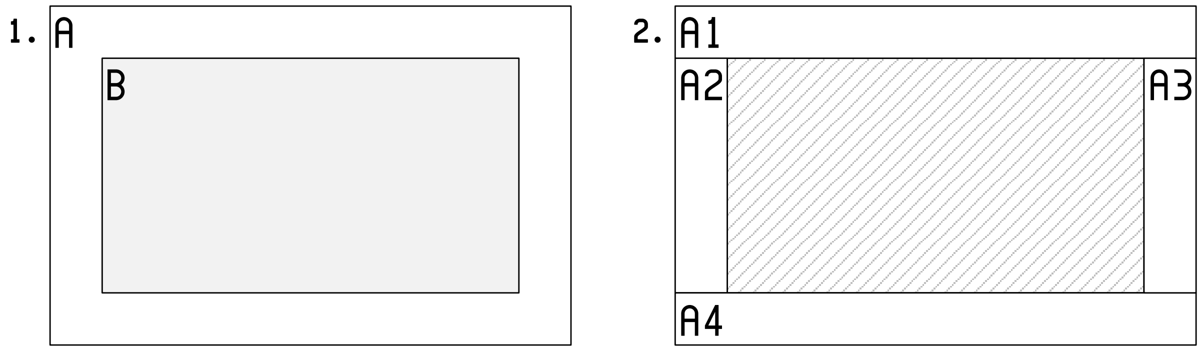

Figure 4. Decomposing A by subtracting B from it

Let us observe two more cases of cutting small rectangle from large one and decomposing it.

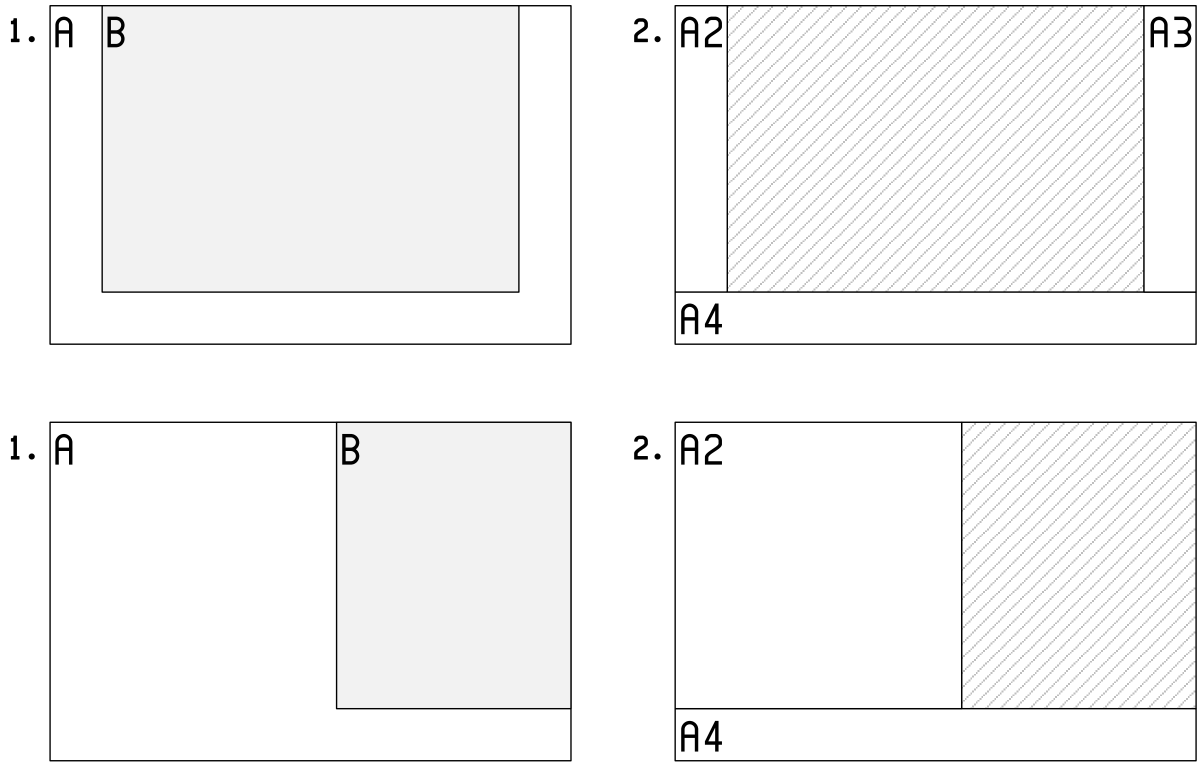

Figure 5. Cutting small rectangle from larger one and decomposing it

Can you see the pattern? The case when the small rectangle is in the middle of larger rectangle is the worst case and four(4) the maximum number of rectangles the large rectangle is decomposed into. All other cases are just subcases of the worst case.

This simple geometric rule can be exploited to create decomposition algorithm. We are making an assumption that rectangle A starts at (0,0) to simplify calculations.

If we remove rectangle B: then the coordinates of newly formed rectangles are:

- **Rectangle A1** … (0, 0, A.x1, B.y0). But if A.y0 is B.y0 then its height is zero and A1 does not exist.
- **Rectangle A4** … (0, B.y1, A.x1, A.y1). But if A.y1 is B.y1 then A4 does not exist.
- Same pattern is used to generate left and right rectangles: A2 and A3.

Here's the algorithm:

```
algorithm SubtractRects
input:      Outer rectangle A
            Inner rectangle B
output:     List of rectangles An
begin
      if A.y1 < B.y0
            A1 ← (A.x0, A.y0, A.x1, B.y0)
            add A1 to An
      if B.y1 < A.y1
            A4 ← (A.x0, B.y1, A.x1, A.y1)
            add A1 to An
      if A.x0 < B.x0
            A2 ← (A.x0, B.y0, B.x0, B.y1)
            add A2 to An
      if B.x1 < A.x1
            A3 ← (B.x1, B.y0, A.x1, B.y1)
            add A3 to An
      return An
end
```

Windows

Definitions

Window

What is it?

The Desktop Window

Desktop window is a special window that covers the entire screen; but otherwise behaves just like any other window. It is also known as the screen. It is “the father of all windows” as all top windows are children of the Desktop window. We will discuss more about parent/child relationships between windows in later chapters. The figure bellow shows desktop window A and child windows B, C, D, E and F on top of it.

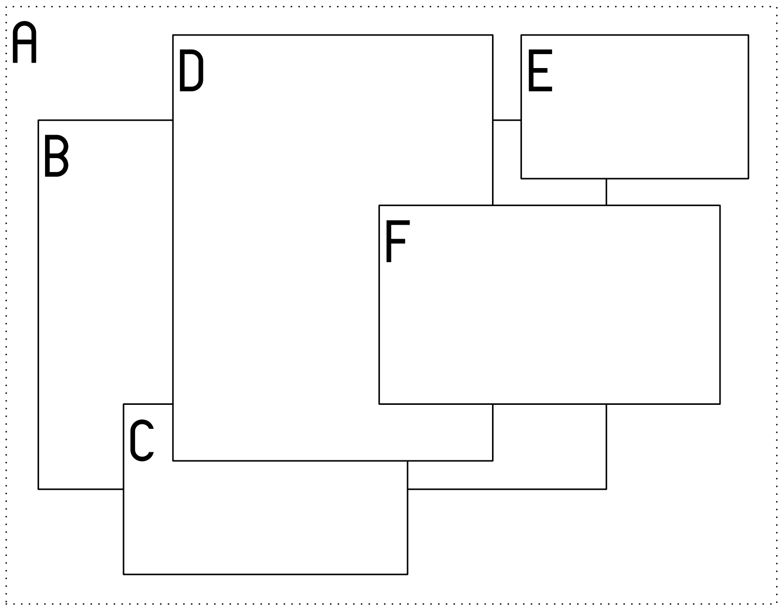

Figure 6. The desktop window A and its children

Parent/Child Windows

A parent-child relationship can be established between two windows. For example - all top windows are children of the Desktop window. There are two types of windows and consequently two types of parent-child relationship. First type of child window is **a control**. An example of this is a button or a label. Control's coordinates are relative to its parent, not to screen. Control is clipped inside windows bounds. The desired consequence of relative coordinates is that moving parent window automatically moves its children too.

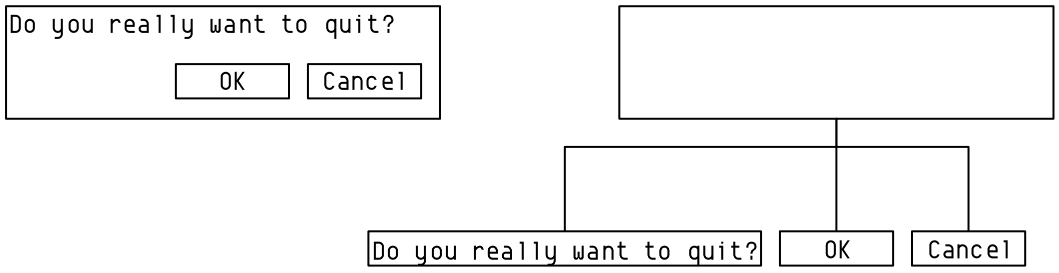

Figure 7. Parent window and its children

This picture shows parent dialog window hosting four child controls: the label "Do you really want to quit" and two buttons "Ok" and "Cancel". All children have coordinates relative to parent so when parent window is moved all contents is moved automatically.

Second type of child window is **a form**. Form window has absolute coordinates. An example of a form window is dialog or a modeless toolbar child window.

Data Structures

window_t data structure is defined in file *window.h*. It contains fields:

- **string title** … window title
- **rectangle_t* window_rectangle** … rectangle, relative to parent
- **window_t* first_form** … first form window child
- **window_t* first_control** … first control window child
- **window_t* next_sibling** … next sibling window
- **window_t* parent** … parent window
- **(word))(*wndproc)(window_t *me, word msg, word p1, word p2))** … window procedure.
- **word attributes** … windows flags, for example WATTR_TABSTOP,etc.

Window Is a Rectangle

Every window is a rectangle. Hence the window data structure contains a pointer to *window_rectangle*. Previously described rectangle algorithms are used to handle windows operations.

Window Procedure

Window procedure is the procedure that receives and handles all windows events. Each class of window has its own windows procedure. Well known procedures can be reused to replicate window class. For example the button_wndproc is window procedure handling all button events and the dialog_wndproc is the procedure handling the dialog window.

In Microsoft Windows same behavior is achieved by setting window class. Window class is just a way to tell Microsoft Windows which window procedure should be used as default for that window.

Input to window procedure are: message code and parameters p1 and p2 which contain context information for the message. Window procedure returns 0 if message was not handled and any other value of message was handled.

Messages

send_message

To send window a message directly (i.e. without queuing) you use the *send_message* function. This function has a high performance fee and should only be used when you need immediate reaction from the window procedure. For example when you want window to immediately repaint itself. Under normal circumstances you can use queued *post_message* function, which is explained in the next chapter. Here is code for send_message:

```
word send_message(window_t *window, word msg, word p1, word p2) {
	if (is_window(window)) {
		return window->wndproc(window, msg, p1, p2);
	}
}
```

post_message and the Message Queue

Besides sending messages to windows directly you can also queue them. All events that do not need immediate response (i.e. can wait several milliseconds) such as keyboard events, are candidates for queuing. These events are processed inside main window loop which every application implements. This is standard implementation:

```
/* message loop */
while (get_message(&m) && m.code!=MSG_QUIT)
	dispatch_message(m);
```

No suprises here. We get message from the queue and we dispatch it. How message is dispatched down the windows hierarchy will be discussed later.

MSG_MOVE

This message is received by the window when it needs to move. The user can grab window by the title and moves it to a new location. This causes change of window origin T(x,y). After move we need to calculate which windows need repainting.

The way to do this is to first change window origin and send the paint message for the entire window. The window which was moved will repaint itself first which makes sense since it is most likely the window user is most interested in.

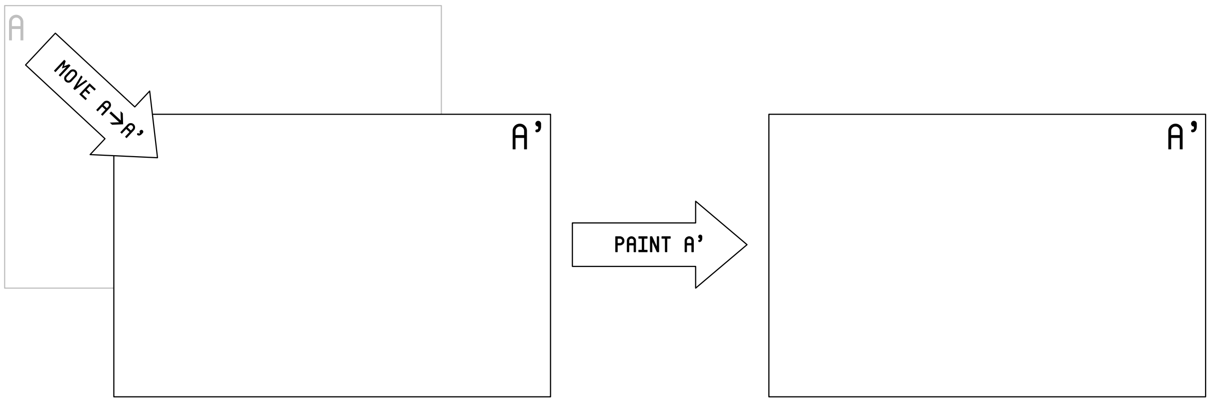

Figure 8. Move window step 1 - repaint moved window

Afterwards we need to calculate which windows were uncovered by the move. The uncovered rectangles can only be where window used to be. They can be calculated by first calculating the intersection of old rectangle A and new rectangle A'. If there is no intersection then the affected area is entire old area A. So by using *FindAffectedWindowsRectangles* we can find out which windows are affected in this area. If there is an intersection (lets call it B) then we need to exclude the intersection B from the old rectangle A and decompose A to smaller rectangles.

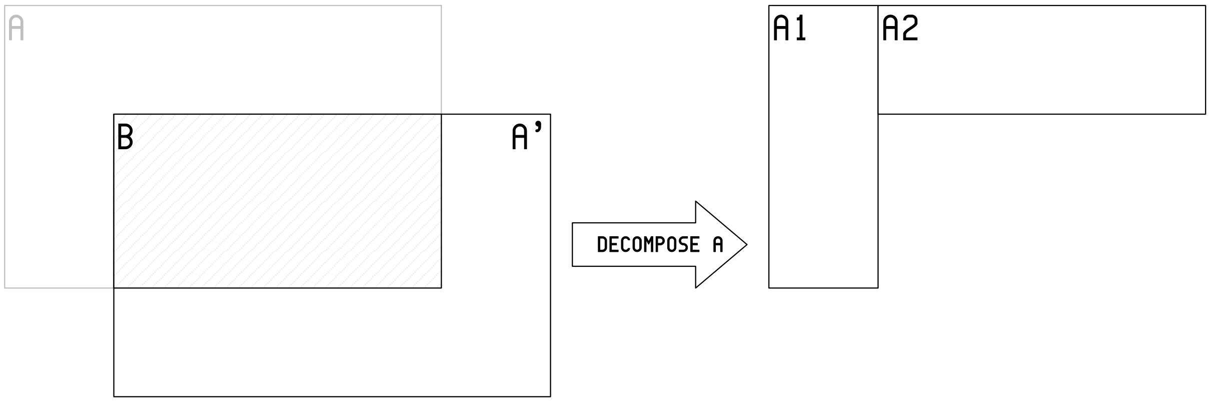

Figure 9. Move window step 2 - find affected rectangles

Smaller rectangles A1 and A2 are affected areas. So we can use *FindAffectedWindowsRectangles* on these ones too.

MSG_SIZE

This message is received by the window when it resizes. The user can use the mouse to change window size. He or she can grab right or bottom edge of window (or the right- bottom corner) and drag it.

We only allow dragging right and bottom edge to make implementation easier. This change affects max. Two rectangles. The left right rectangle and the bottom rectangle. What needs to be repainted depends on whether window has increased or decreased size in given direction.

If window size increased then we send message to the window being resized to paint its new (extended) area. If window size decreased then we send windows bellow this window messages to repaint their own areas using the *FindAffectedWindowsRectangles* algorithm.

Here is an example where Window A is resized. Width is increased but height is decreased. We call the resized Window A'.

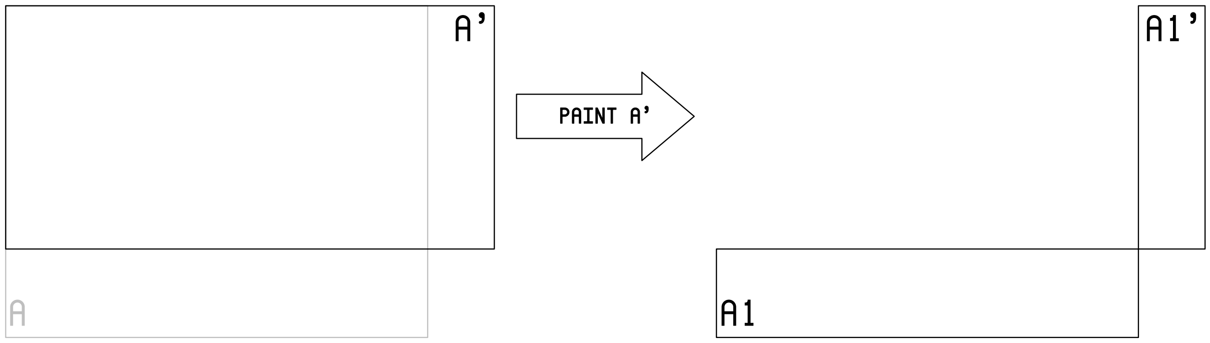

Figure 10. Resize window

A' receives MSG_PAINT to repaint area A1'. And the newly uncovered area A1 is used as input to FindAffectedWindowsRectangles algorithm to find out which windows should receive MSG_PAINT and for what area.

When changing window size it is not always desireable that the system sends automatic MSG_PAINT message. Imagine a situation where window contents would be adjusted to the size. The window procedure would catch event MSG_SIZE, calculate new contents size and send MSG_PAINT. And then the system would send MSG_PAINT again.

To prevent this two attributes can be added to the window: WATTR_HREDRAW and WATTR_VREDRAW. If these are set in windows attributes bitmask then when width and height change the window is send MSG_PAINT. Otherwise it is not.

MSG_PAINT

This message is received by the window when it needs to repaint part of it. Parameter p1 is either NULL which means that entire window area needs repaint or contains a clip rectangle meaning that only contents that is visible inside that rectangle needs to be redrawn. Parameter p2 holds initialized graphics contents which can be used to draw inside window and handles relative coordinates and clipping for you.

MSG_TIMER

A window can register a timer. One timer tick is 1/10th of a second. Upon registration the user can choose number of timer ticks to expire before triggering timer event. Upon timer event this message is received by the window which registered a timer.

MSG_MOUSE_MOVE

This message is received by the window which captured mouse when mouse is moved. Parameter p1 holds new x and y position of the mouse and parameter p2 holds old x and y position.

MSG_MOUSE_BUTTON_DOWN

This message is received by the window which captured mouse when mouse button is pressed. Parameter p1 holds x and y position, parameter p2 holds mouse button codes (left, right).

MSG_MOUSE_BUTTON_UP

This message is received by the window which captured mouse when mouse button is released. Parameter p1 holds x and y position, parameter p2 holds mouse button codes (left, right).

MSG_KEY_DOWN

This message is received by the window which has focus when keyboard button is pressed. Parameter p1 holds keyboard code and parameter p2 holds flags.

MSG_KEY_UP

This message is received by the window which has focus when keyboard button is released. Parameter p1 holds keyboard code and parameter p2 holds flags.

MSG_QUIT

This message is received by the application when it needs to quit. This message should be checked in the main message loop and the loop should exit when received.

Dispatching Messages

Dispatching messages is small science. Finding correct routes for a message and sending it to the correct window is key to intuitive behavior of user interface. To demonstrate this let us first create a sample window setup that we are going to use to demonstrate various dispatching techniques.

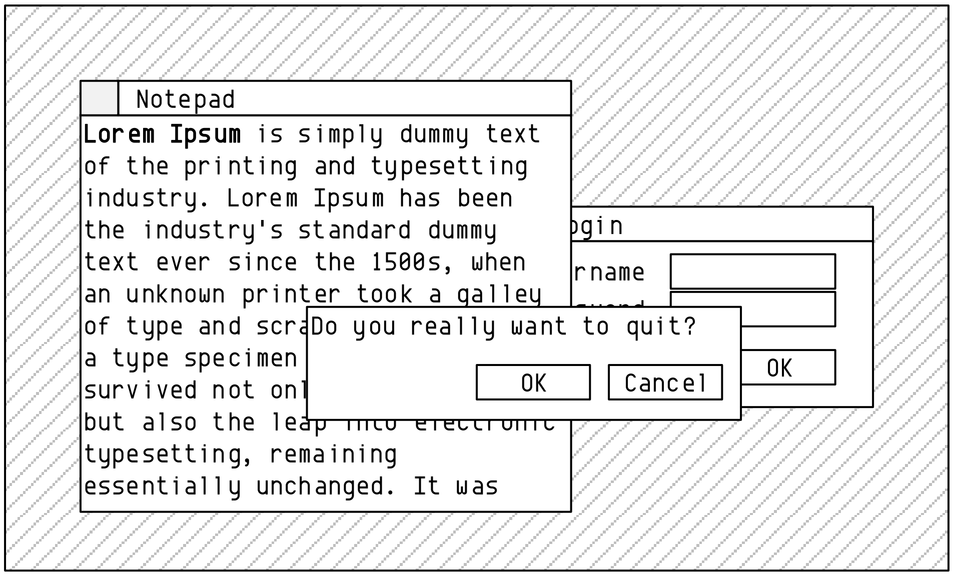
Notepad
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scra
a type specimen
survived not onl
but also the leap into electronic typesetting, remaining essentially unchanged. It was
gin
rname
OK
Do you really want to quit?
OK
Cancel

Figure 11. Dispatching messages scenario

Form Windows Hierarchy

On the figure above we have Notepad. In front there is a message box asking user to confirm closing the Notepad. In the background there is IRC application that demands user login. And both of these applications are on desktop. On the next page you can see the data structure of windows in memory.

The root window is -of course- the desktop. There are two top windows. Notepad application window and IRC Login application window. It is very important to understand that the order of windows in the list is not random. It reflects the z-order. To traverse the tree you first go to the bottom of the tree using first child pointer as far as it goes. And you then move via z-order using next sibling pointer. When there are no more siblings you move up one level and repeat the procedure.

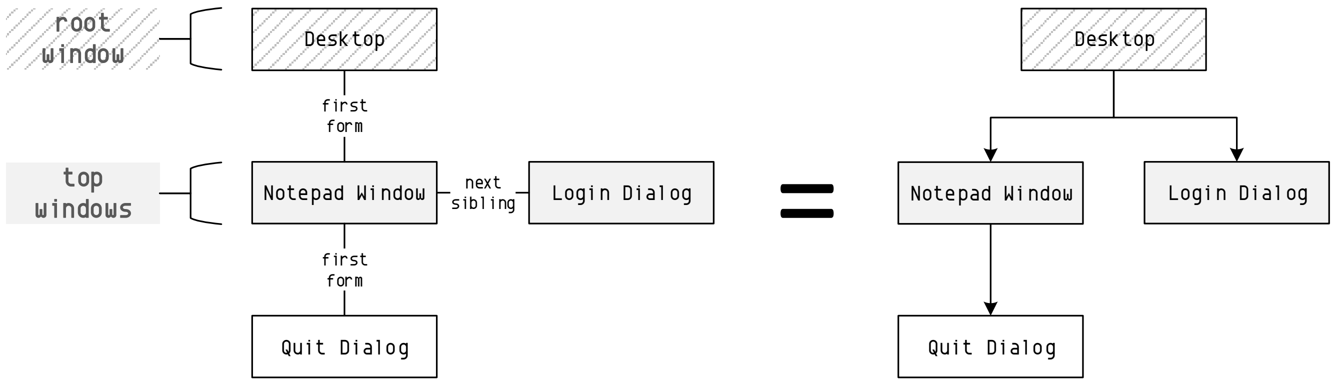

Figure 12. Tree struture of window from previous figure

Note that window structure can be seen as a tree structure as shown on the right. Let us traverse this tree in z-order. First window in the z-order is the Quit Dialog. Next is the Notepad Window, followed by Login Dialog. Last one in the z-order is the Desktop. If any child form is modal then all parent forms above it are event sinks i.e. do not respond to mouse or keyboard events. But other top windows do. This allows freezing application with the modal window but switching in another application which is the intuitive behavior we'd expect.

If we want to move window up z-order we simply rewire it as the first child of parent window and let its next sibling be previous first child of the same parent. For example if you would click on the IRC Login window the first child of Desktop would become Login Dialog and the next sibling of Login dialog would become Notepad Window. So all events would first be resolved in Login dialog window.

Controls Hierarchy

There is another hierarchy in the Notepad/Login window sample that we presented earlier. It is the hierarchy of controls or windows that are children of form windows and hosted inside its boundaries. Lets display these hierarchies.

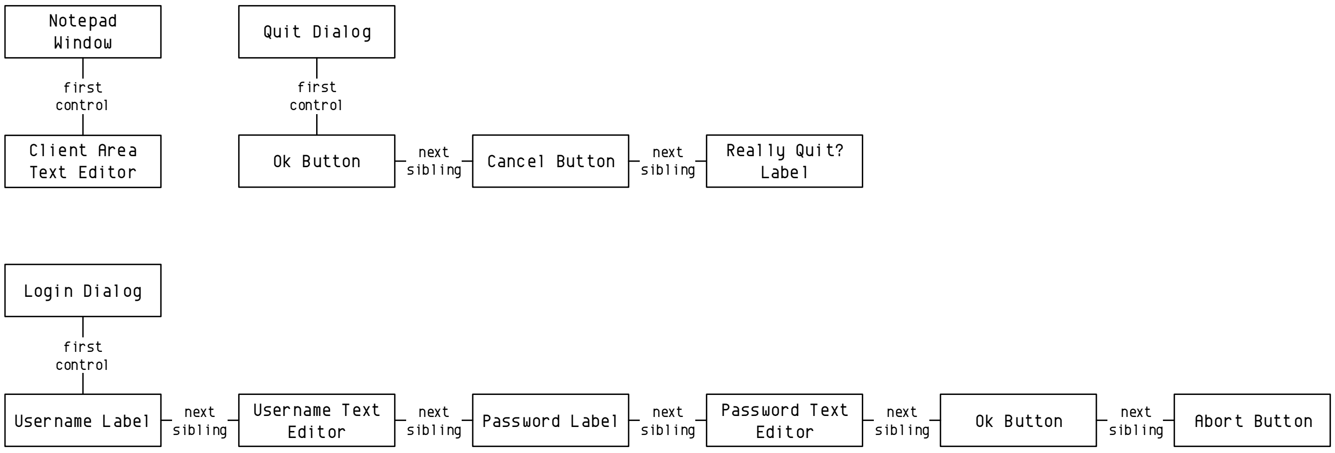

Figure 13. Control hierarchies

These hierarchies are handled a bit differently then form hierarchies. Basically – mouse and keyboard events are checked against form hierarchies. And only if top conditions are fulfilled (i.e. no modal form is blocking form hosting controls accepting events) the event is dispatched to controls.

Keyboard Events

Keyboard events are easiest to dispatch. Only one control or form window has **a focus**. Always. It can be set by your application or automatically when showing new form window or bringing it to the front. The window that has a focus receives keyboard events and is sent MSG_KEYDOWN and MSG_KEYUP messages. Period.
If zero is returned by the windows procedure then next window in z-order receives the same message. Controls always have priority over forms. If windows procedure returns non-zero value it is assumed that it processed the message and no further processing takes place.

One of the windows flags is WATTR_TABSTOP. Windows that have this flag set (for example: for button and edit box) have window procedures that do not process tab key press. If it is pressed the wndproc returns 0. Tab keyboard event is then forwarded to parent. Window procedure for dialog and for window on the other side process tab events. If it is pressed it sets input focus to next sibling control. If there are no more sibling control it sets focus to the first control (again). If there is none it forwards event to parent. And so on. This way you can use tab key to move between edit controls and buttons. You can also impelement general purpose shortcuts such as close window or close application which will simply be handled by parent and not handled by controls.

User can change control that receives keyboard focus by calling function *set_focus*.

Mouse Events

Mouse movement and clicking is generally reported to window (control or form) over which the mouse cursor moves or resides. One exception to that is if certain window has captured the mouse by calling *mouse_capture* function. In this case all mouse events will be sent to this window directly until it call *mouse_release* function.

Mouse events sent to window that has captured the mouse are in coordinates relative to the window and can be negative i.e. if window is on screen position (10,10) then mouse move to (-10,-10) means screen position (0,0).

If no window has captured the mouse then mouse move or click event is (as all mouse events) first filtered using form window tree z-order. The system tries to find a form rectangle inside which x and y mouse coordinates reside. When it is found it checks that this form has no child modal form shown. If it has one then the event is trashed. If not then the event is forwarded to controls inside this form using z-order (again). If no control is found under mouse cursor then mouse move message is sent to form. Otherwise it is sent to this control. If event is not handled i.e. wndproc does not return value different then 0 then event is delegated to parent window. All the way to desktop.

Paint events

The general rule is that the system detects a need for a repaint only when form window is removed, shown, moved or resized. Because all control windows are part of form window (and the affected area anyways). So when a paint event is detected by the system then it sends paint event to forms using the affected rect algoritm. These forms in turn detect which of their controls need to be repainted and forward paint events to them using the same logic (of affected windows) only inside form window.

Algorithms

The Affected Windows Algorithm

One of the most important windows operations is calculating which windows are affected by certain rectangle on the screen. Discovering which windows in an optimal way (specifically which rectangle inside which window so that we don’t paint parts of window that we don’t need to) is the essence of this algorithm.

Simple window operations such as moving window, closing window or resizing window always uncover parts of screen that were previously covered. Commonly the uncovered part can be decomposed into multiple rectangles that need to be repainted. This is done by sending paint message to each affected window for each affected area.

Here is how you can find out which windows were affected by uncovering Area. As you can see this algorithm uses previous algorithms *SubtractRects*.

```
algorithm FindAffectedWindowsRectangles
input:        Z-ordered WindowsList
              Rectangle Area
output:       Messages sent to all affected windows
begin
       if Area is null return
       Window ← top window from WindowsList
       IntersectRect ← intersection between Area and Window
       if exists IntersectRect
              send repaint message to Window for IntersectRect
              remove Window from WindowsList
       if WindowsList is empty return
       SmallerAreas ← call SubtractRects(IntersectRect, Area)
       foreach SmallerArea in SmallerAreas
              call FindAffectedWindowsRectangles(WindowsList, SmallerArea)
end
```

This algorithm is not such a monster. Let us walk through an example (figure is on the next page) to see what is happening.

In the example we are removing window F. Because there is no intersection between E and F, E is removed from the WindowList. Intersection between D and F is the rectangle, that window D must refresh, before being removed from the WindowList. Remaining F' does not intersect with C and therefore C is merely removed from WindowList. Interesection betwen B and F' is rectangle that B must refresh. Then B is removed from WindowList. Last remainder is F''. This is compared to A (the desktop) and because F'' is completely inside A, A repaints entire F'' area and concludes work.

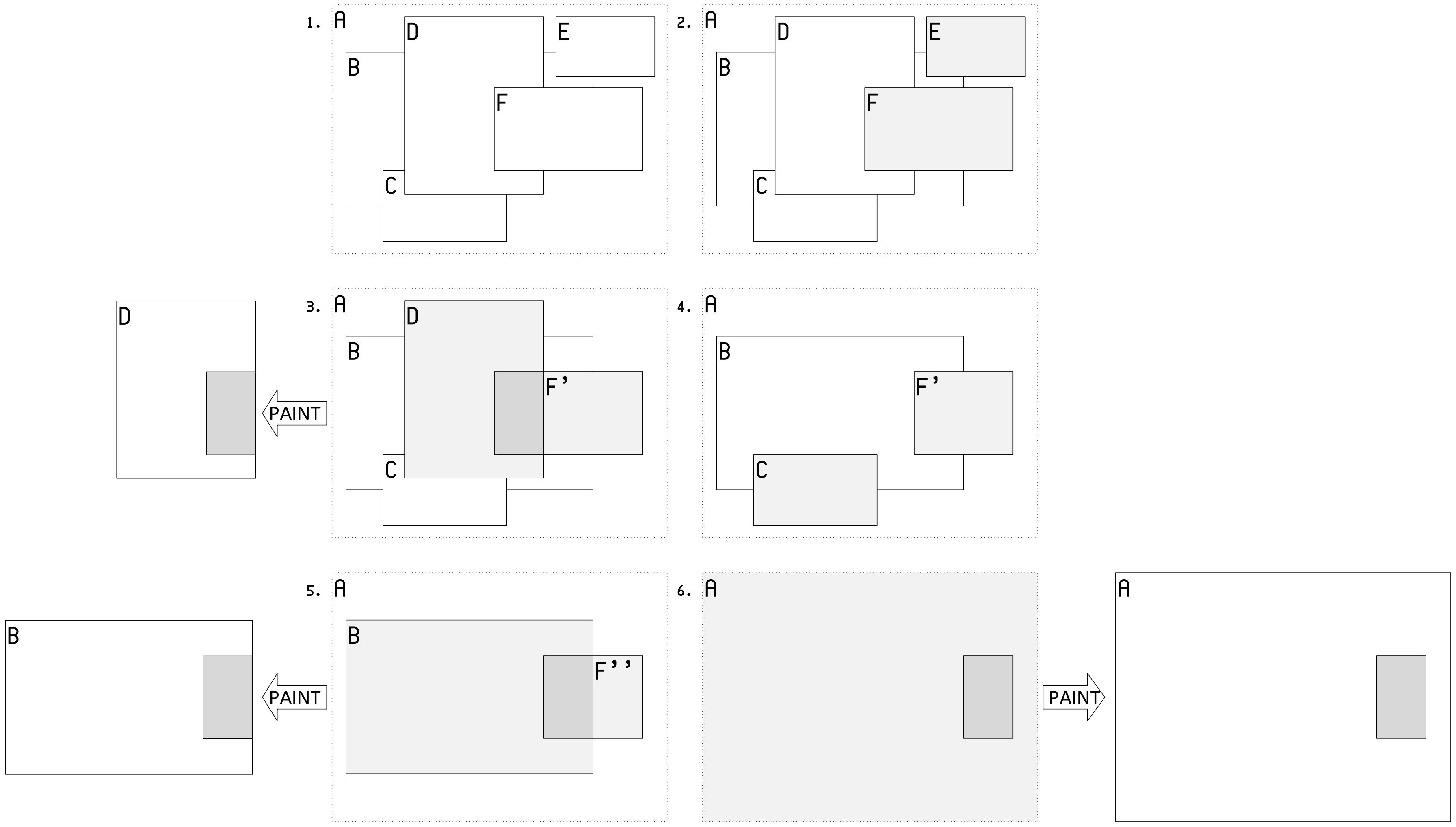

Figure 14. Finding affected windows rectangles

Graphics

This chapter discusses the graphics API of Buddy. It's scope is very limited: it has functions for drawing and filling rectangles, drawing lines, drawing pixels, drawing text using fonts and drawing bitmaps. Each function respects window relative coordinates and clipping.

Data Structures

graphics_t

Graphics data structure is graphics_t and can be found in the graphics.h file. It contains following members:

- **window_t* window** … pointer to window
- **rectangle_t* clip_rectangle** … clipping rectangle, relative to parent if null then it uses window's rectangle
- **pen_t* pen** … current selected pen for lines
- **brush_t* brush** … current selected brush for brush operations
- **font_t* font** … current selected font for font operations
- **byte attr** … drawing attributes for example GATTR_XOR to XOR all pixels

pen_t

About the pen_t. (single byte line pattern)

brush_t

About the brush_t (8x8 brush)

font_t

About the font_t (a font is a set of glyphs with header specifying metrics)

glyph_t

About the glyph_t (a glyph is an optimal sprite structure, max size is screen size to allow saving and resoring it)

Relative Coordinates

All coordinates for graphical operations are relative to window. When drawing the system simply adds window's x0 and y0 to each x and y.

Due to ZX Spectrum's resolution of 256 x 192 only one byte coordinates are used. This causes two problems. First is that some drawing algorithms test exit condtitions with a smaller then comparision and fail at value 255 (which is still on screen) because a value of 256 does not exist i.e. it is 0. Second problem is using negative coordinates with mouse capture events. Negative coordinates are rarely used and coded by using point of origin and screen resolution as references. If your origin's x is 250 then your valid values are from 0 to 5. A value of 6 is in fact -250, a value of 7 is -249 and a value of 255 is -1.

Clipping

A single clipping rectangle is supported by all drawing operations. If it si not defined in clip_rectangle member then window's area is used for clipping. By defining it you can limit drawing inside window into even smaller rectangle. If it is defined it must be in relative coordinates and is clipped itself by window area if too large.

Functions

Following functions are defined by graphics

- **draw_pixel(graphics_t*,x,y)** … draw a pixel to the window
- **draw_hline(graphics_t*,x0,x1,y0)** … draw very fast horizontal line to the window
- **draw_vline(graphics_t*,x0,y0,y1)** … draw very fast vertical line to the window
- **draw_line(graphics_t*,x0,y0,x1,y1)** … draw line
- **set_brush(graphics_t*,brush_t*)** … set current selected brush for fill operations
- **set_pen(graphics_t*,pen_t*)** … set current selected pen for line operations
- **set_clip(graphics_t*, rect_t*)** … set current clipping rectangle (NULL to clear it)
- **draw_rect(graphics_t*,x0,y0,x1,y1)** … draw very fast rectangle using pen
- **fill_rect(graphics_t*,x0,y0,x1,y1)** … fill area with brush very fast
- **draw_glyph(graphics_t*,glyph_t*,x0,y0)** … draw a glyph (a raster bitmap)
- **draw_msk_glyph(graphics_t*,glyph_t* glyph, glyph_t *mask, x0,y0)** … draw masked glyph
- **draw_char(graphics_t*, byte,x0,y0)** … draw char using current selected font
- **draw_string(graphics_t*, string, x0,y0)** … draws an entire strin
- **measure_char(graphics_t*,byte)** … returns character's size
- **measure_string(graphics_t*, string)** … returns string's size
- **fill_attr(graphics_t*, attr_t, x0, y0, x1, y1)** … sets color attributes for particular region

Functions are optimized for speed and size. For example, all font drawing functions and mouse cursor drawing functions use same data structures as glyphs and use *draw_glyph* or *draw_msk_glyph* underneath. If memory allows, perhaps, one day we'll extend the functions.

Architecture

Buddy GUI strives to fulfil following architectural objectives:

- Run on multiple ZX Spectrum operating systems i.e. Sinclair ROM, efxDOS, yx os, etc…
- Modular and portable

Layered Architecture

Following figure shows the layered architecture of Buddy.

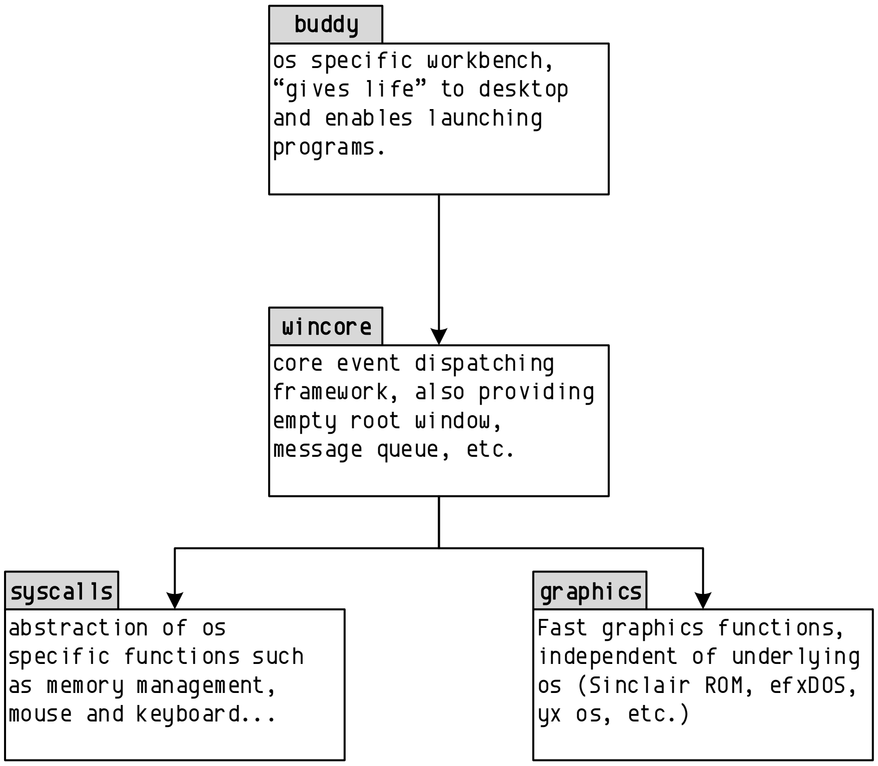
buddy
os specific workbench, “gives life” to desktop and enables launching programs.
wincore
core event dispatching framework, also providing empty root window, message queue, etc.
syscalls
abstraction of os specific functions such as memory management, mouse and keyboard...
graphics
Fast graphics functions, independent of underlying os (Sinclair ROM, efxDOS, yx os, etc.)

Figure 15. Layered architecture of Buddy

Syscalls Component

At the bottom of the architecture is module named **syscall**. All modules above this use it. This module abstracts most common system services such as reading mouse and keyboard, allocating and freeing memory blocks, etc.

By replacing this module with your own it is possible to make buddy run on top of any operating system. Buddy GUI runs in single thread and implements so called cooperative multitasking making it possible to run on top of multi - threaded yx as well as single threaded efxDOS or Sinclair ROM. The event system gives the impression that multiple applications are running at the same time.

Graphics Component

Second bottom level component is the **graphics**. The contents of this component is described in the Graphics section of this document - its purpose is to provide OS independent drawing routines to buddy.

WinCore

Third component is **wincode**. This is the heart of the GUI. It implements all window and event gathering logic and provides an empty root window to applications running in the Buddy environment. It is also responsible for drawing all the

Buddy Component

The last component is **buddy** or the Buddy Workbench. It runs on top of wincore and “gives life to buddy” by providing basic desktop management functions such as: menus, icons, starting applications, etc. The Workbench is OS dependent. You can implement it as a simple file manager, resource manager, etc… Note that it is not a window manager. wincode takes care for drawing the window title, border and some other elements. However, Windows does allow component to replace standard drawing logic with its own so Buddy can skin all windows by providing simple drawing procedures.