YX OS

Volume 3: The Buddy GUI

VERSION 0.1 • September 23, 2013

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	Updated document.	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.

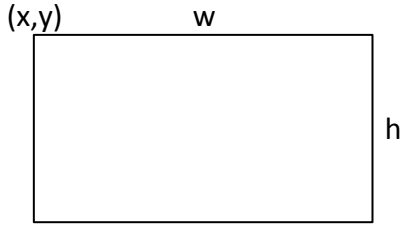

Figure 1. Rectangle coordinates

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.

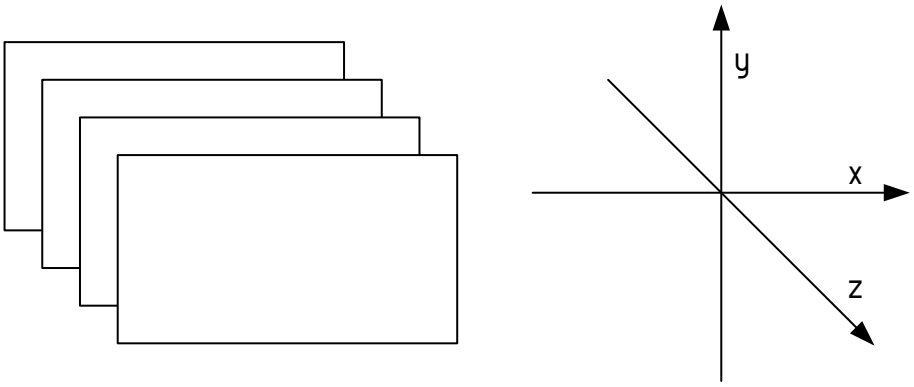

Figure 2. The Z-Order

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.

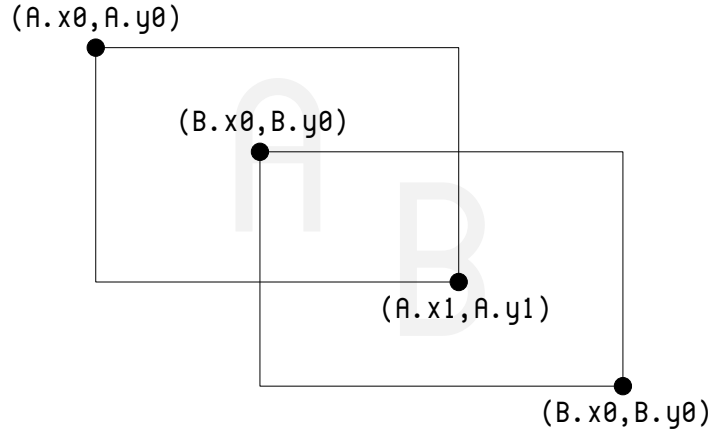

Figure 3. Rectangle intersection

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.

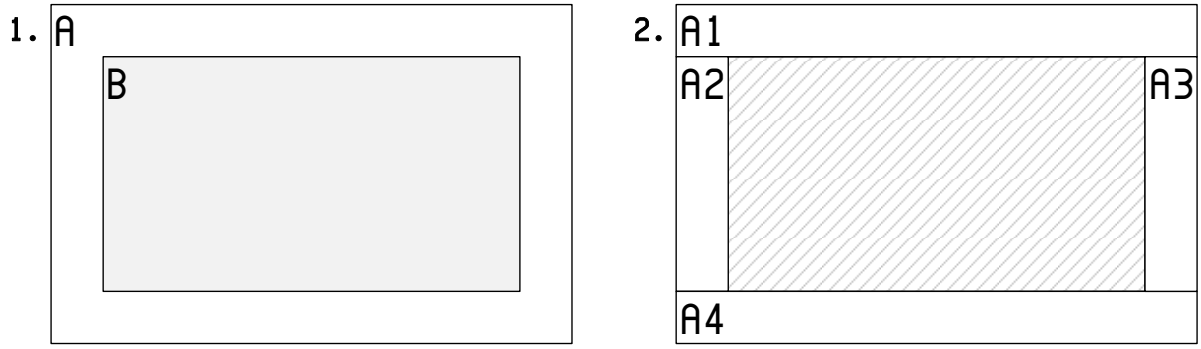

Figure 4. Decomposing A by subtracting B from it

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

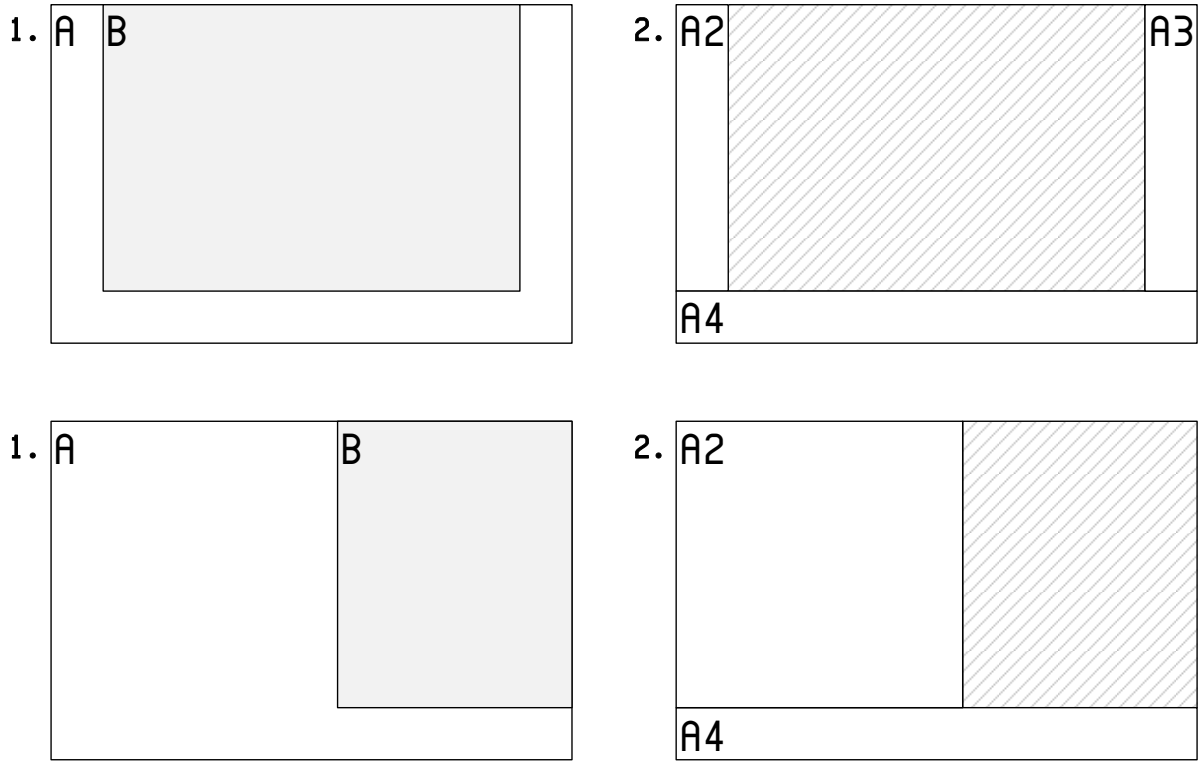

Figure 5. Cutting small rectangle from larger one and decomposing it

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.

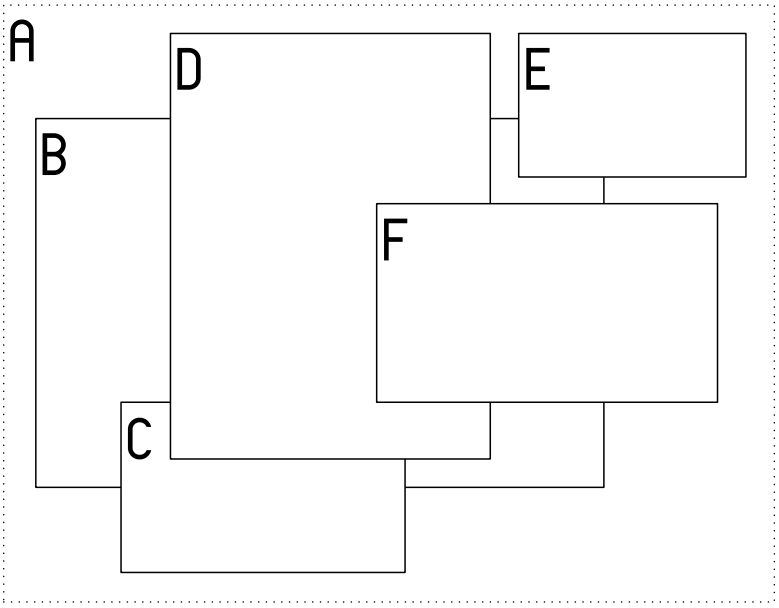

Figure 6. The desktop window A and its children

Parent/Child Windows

A parent-child relationship can be established between two windows. For example - all top windows are children of the Desktop window. Window coordinates are commonly relative to its parent, not to screen. They are relative to screen only if windows direct parent is the Desktop window. The desired consequence of relative coordinates is that moving parent window automatically moves its children too.

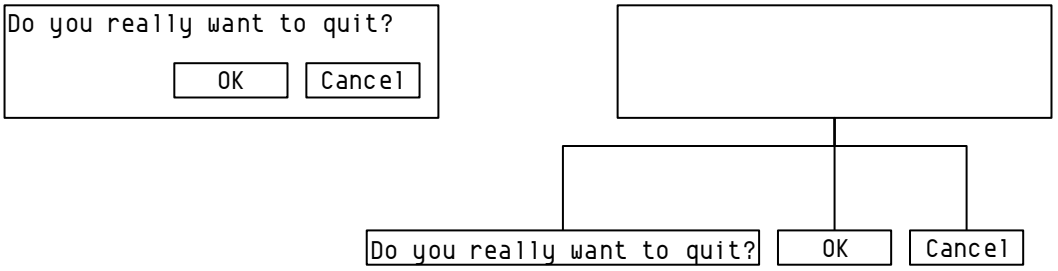

Figure 7. Parent window and its children

This picture shows parent dialog window hosting four child windows: 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.

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_child** … first child window
- **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.

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. Now let us look at some standard window messages.

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.

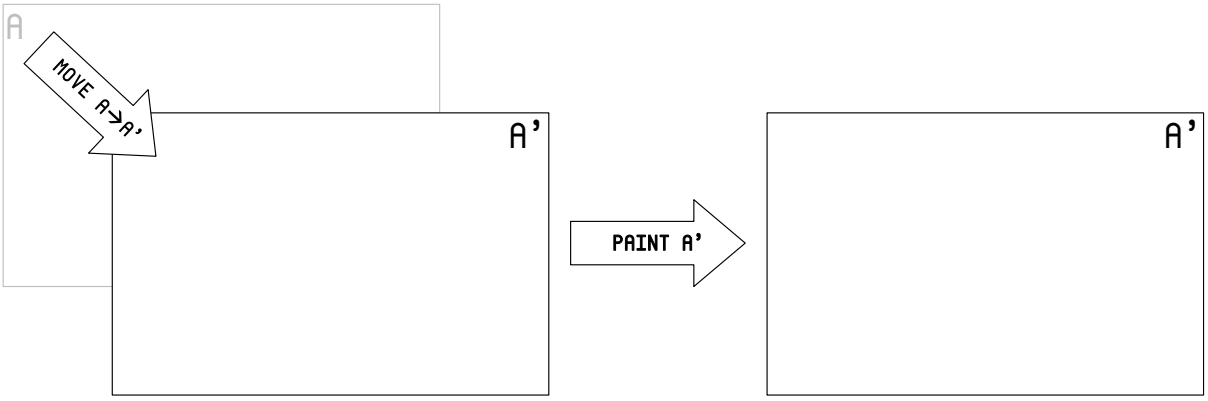

Figure 8. Move window step 1 - repaint moved window

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.

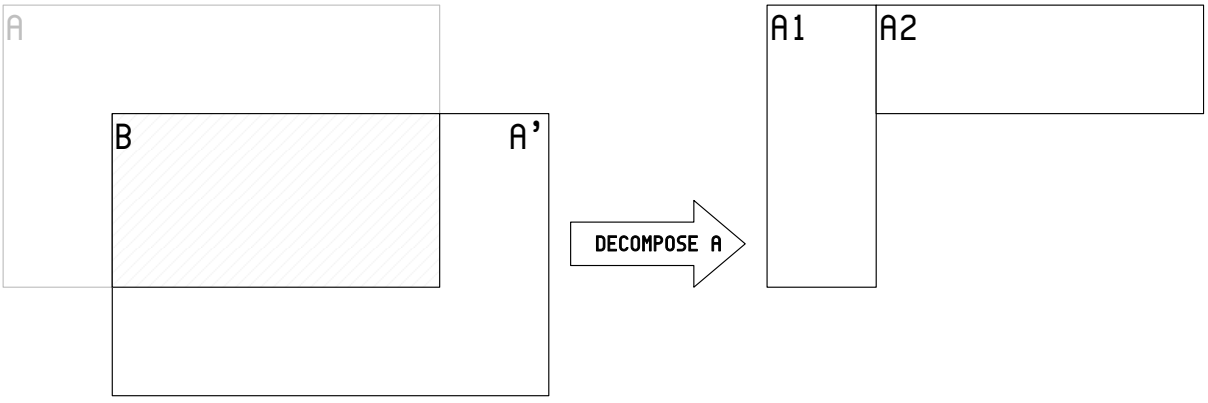

Figure 9. Move window step 2 - find affected rectangles

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'.

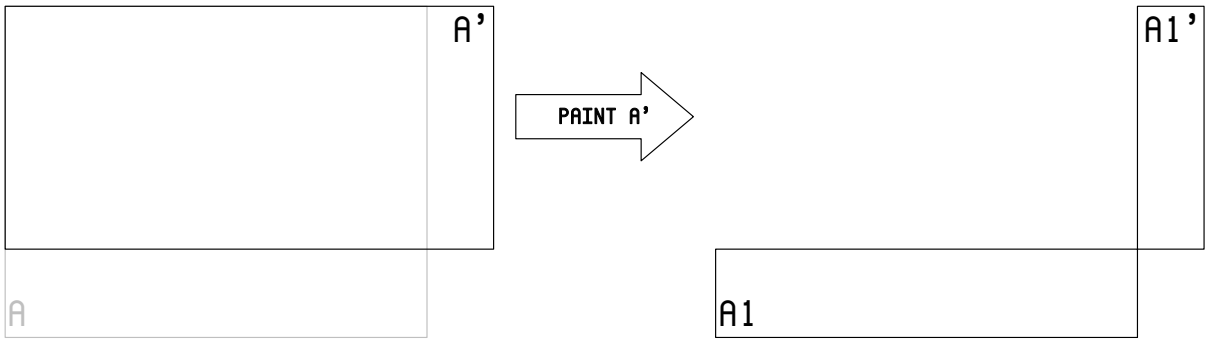

Figure 10. Resize window

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.

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.

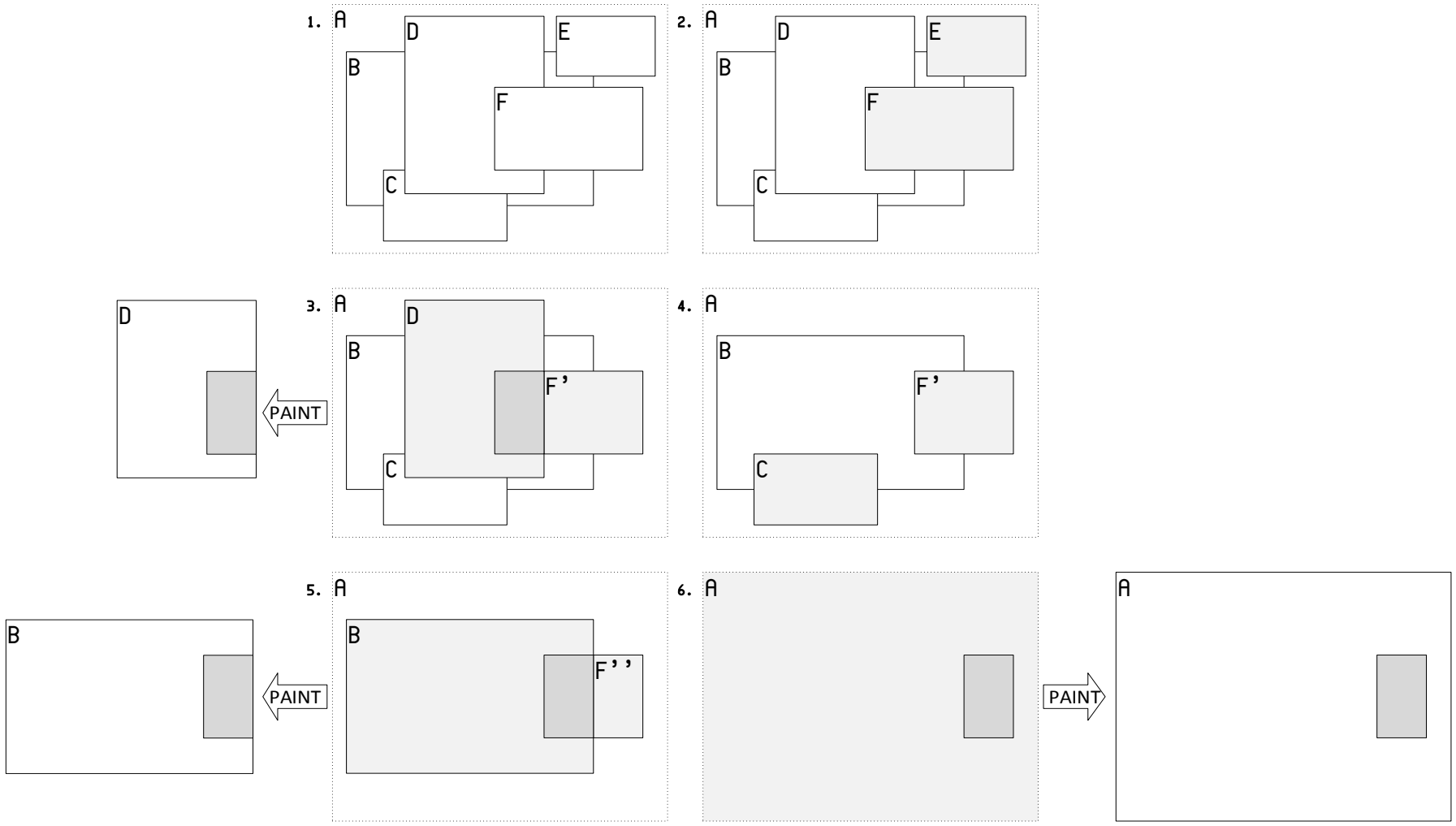

Figure 11. Finding affected windows rectangles

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.

To be continued…