Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 3619

Compositing Concepts for the Presentation of Graphical Application Windows on Embedded Systems

Riccardo Cecolin

Course of Study:	INFOTECH
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dipl.-Inf. Simon Gansel
	Dipl.-Inf. Stephan Schnitzer
Started:	7. Oct 2013
Completed:	8. Apr 2014
CR-Classification:	D.4.9,H.5.2,I.3.6

Abstract

Modern automotive systems feature multiple displays used to render 2D and 3D graphical applications to provide functionalities like driving assistance and entertainment. The performance of the hardware used in automotive environments differs greatly from what is commonly available on desktop workstations, since automotive embedded systems are limited by power consumption and size.

The windowing system supporting the graphical applications displayed on board must therefore achieve a greater level of efficiency compared to those used on desktop systems. The compositor, the component of the windowing system that is responsible to draw the content of the applications on screen, must therefore be efficient in the bitblitting operations, especially by minimizing the overdraw that occurs in case of overlapping windows.

A concept for a compositor was developed, which features a data structure designed for storing overlapping windows and a set of algorithms to optimize the bitblitting operations. The compositor, using a prediction model that represents the time requirements of the bitblitting commands, is able to reduce the bitblitting time by choosing at runtime the best strategy to draw windows on screen.

Contents

1 Introduction

Automotive software improved radically the interactivity of the vehicle user interfaces in recent years. The amount of information that is presented to the driver is steadily increasing with every product iteration, directly influencing the size and number of displays mounted in modern cars used to render 2D and 3D graphical applications. Many models on the market already feature a full sized display as instrument cluster (IC) to present speed information, notification icons and warnings to the driver. The head unit (HU), the unit controlling the infotainment system, displays the navigation system and media for entertainment on a large screen. Rear seat displays are used to present interactive applications or play videos for the passengers and head-up displays provide driving assistance and augmented reality functionality directly on the windshield. Prototypes and concept cars [1] (Fig. 1.1) show that the trend keeps moving in this direction, increasing resolution and size of the screens on board. While the number of screens available in a car is increasing, their usage patterns are also recently becoming more flexible. The IC screen is used to play videos when the car is standing, reducing the display area dedicated to the speedometer and tachometer in favor of the video player. Third party applications are given access to the HU, sharing the display area flexibly and seamlessly integrating with the rest of the system [2].

Figure 1.1: Mercedes concept car F125 [3]

To support these features a graphical software system is required. Embedded systems used in automotive environments are not as powerful as desktop systems: power consumption and form factor limit the available computing power. A typical desktop workstation draws around 300W, while an embedded system requires power in the range of 5W. For this reason graphical software systems on embedded platforms face a distinct set of challenges, having to meet higher efficiency requirements in order to achieve the desired performance.

The commonly used abstraction in graphical user interfaces is the *windowing* abstraction [4], where the content of every application belongs to a window. In an automotive windowing system safety issues arise due to the overlapping of windows. Critical components like the speedometer are not allowed to be covered by other windows while the car is in motion, for example. Videos should not be played on the HU when the car is not parked, to prevent driver distraction [5] [6]. Therefore the user has not full control on the windows' arrangement: the OEM specifies which positions, sizes and animations of the windows are allowed.

In such a system the OEM is free to define any combination of overlapping windows as long as they don't go against the safety restrictions, therefore requiring the compositor, the software component that is responsible for drawing the content of the applications on the visible screen, to support any possible layout of overlapping windows. Large screen surfaces require high performance, hence the compositing software has to be efficient in the execution of the drawing operations.

In a windowing system a window is a rectangle with defined size and position on the screen. The size of a rectangle is measured in pixels, the fundamental unit on displays. Windows can typically overlap as long as the system restrictions are not voided, therefore resulting in a wide range of overlapping combinations. Using this abstraction windows are handled as rectangles, and the task of a compositor can be described as analyzing the set of rectangles that has to be displayed on screen and efficiently bitblitting them. Systems with graphical capabilities have a bitblit command, which is used by a compositor to draw the rectangular application windows on the screen according to their position and size. The execution of a bitblit command takes an amount of time proportional to the size of the area to draw, requiring more time to complete operations on larger areas. Since the advent of the first VGAs an effort in minimizing the amount of pixels written in video memory was shown [7], mostly because of the limited speed of the memory and the waste of bandwidth due to overdraw [8]. This is still valid in today's GPUs, which typically advertise in their specification the amount of pixels/second that they are able to bitblit [9], and it is even more relevant in *system on chip (SoC)* embedded systems, where the GPU in not a dedicated unit on the system board but rather a component of a single chip. The approaches commonly used to bitblit a set of application windows on screen are *full compositing* and *tiled compositing* strategies. Full compositing consists of drawing the content of each window in z order on screen, while tiled compositing is a more sophisticated approach that keeps track of the visible areas of windows, clipping them in rectangles called tiles and bitblitting them, avoiding any overdraw [10]. An improvement in performance of the graphical stack on such systems can therefore be achieved by reducing the time spent bitblitting, optimizing the number of bitblitting operations as well as minimizing the screen area to be bitblitted. This is not fully addressed by compositors aimed to embedded systems like Wayland [11] or Android's compositor SurfaceFlinger [12], which resort to the use of plain *tiled* or *full* compositing strategies and do not further optimize.

In an automotive environment there are components of the system that have to be handled with customized solutions. There are applications that need low latency and dedicated access to the screen. An example is the rear view camera, whose frames should be shown to the driver on the vehicle display with minimal latency. In this case a compositor aimed to automotive systems is aware of such regions and draws the rest of the screen avoiding to bitblit in the areas dedicated to the low latency video sources. To achieve this the compositor needs to store knowledge on the way windows overlap and be able to partially draw windows, which overlap with the reserved regions. Compositors currently available for use in embedded systems are missing this feature.

In this work a concept for a compositor to minimize the bitblitting time is presented. Analyzing the operations that a compositor typically supports and the usage patterns involved, suitable data structures have been devised to efficiently store and access windows. Algorithms dedicated to minimize the bitblitting time by reducing the number and entity of drawing operations have been designed on top of such data structures to achieve the performance required by a compositor for automotive embedded systems.

Outline

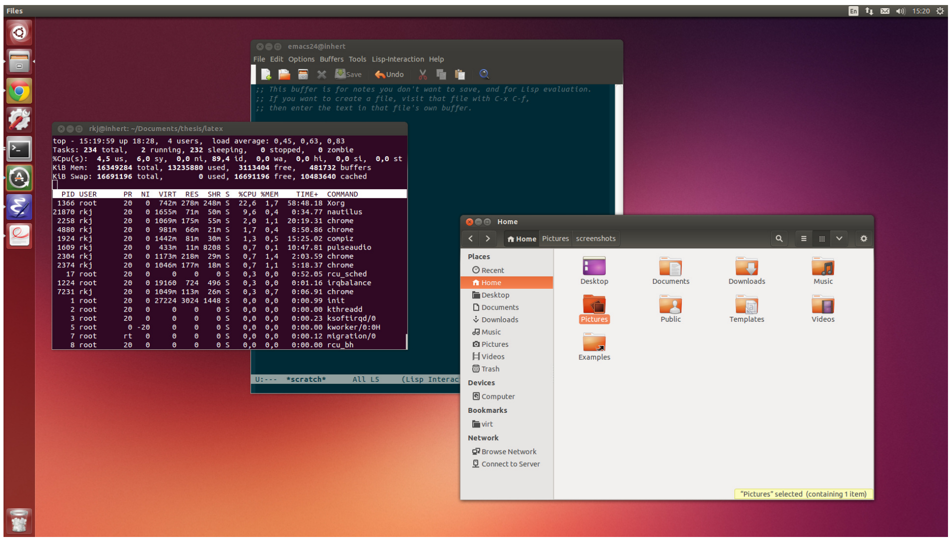
Chapter 2 gives an overview on the background technologies and terminology is given: EGL, OpenGLES, bitblitting. In Chapter 3 the system model the compositor builds upon and the assumptions taken are described. Chapter 4 presents the related work concerning window managers and compositors. In Chapter 5 the methods used to store and manage windows are outlined, together with the operations that the interface of a compositor should offer. The data structures and the algorithms developed to optimize the bitblitting are then thoroughly described, along with examples. Chapter 6 describes the details on the implementation of the compositor featuring the newly introduced algorithms. Chapter 7 presents the tests necessary to generate a calibration model. Then performance analysis and comparison of the novelty algorithms with the state of the art is done in a wide range of tests.

2 Background

The development of a compositor builds on top of many technologies already available, relying on graphical libraries and drivers to perform its tasks. A compositor is just part of the graphical stack of a computer system, the interactions between it and the rest of the system must also be outlined. This chapter aims to provide a general background on the components necessary for a compositor to work.

2.1 Windowing systems

A windowing system includes the software components that create a graphical user interface on computer displays, implementing the *window* paradigm, where applications are displayed usually in rectangles of different sizes positioned on the screen surface [4].



```
;; This buffer is for notes you don't want to save, and for Lisp evaluation.
;; If you want to create a file, visit that file with C-x C-f,
;; then enter the text in that file's own buffer.
```

```
U:---  *scratch*      All L5     (Lisp Interac
```

```
top - 15:19:59 up 18:28,  4 users,  load average: 0,45, 0,63, 0,83
Tasks: 234 total,   2 running, 232 sleeping,   0 stopped,   0 zombie
%Cpu(s):  4,5 us,  6,0 sy,  0,0 ni, 89,4 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:  16349284 total, 13235880 used,  3113404 free,   481732 buffers
KiB Swap: 16691196 total,        0 used, 16691196 free, 10483640 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
 1366 root      20   0  742m 278m 248m S  22,6  1,7  58:48.18 Xorg
21870 rkj       20   0 1655m  71m  50m S   9,6  0,4   0:34.77 nautilus
 2258 rkj       20   0 1069m 175m  55m S   2,0  1,1  20:19.31 chrome
 4880 rkj       20   0  981m  66m  21m S   1,7  0,4   8:50.86 chrome
 1924 rkj       20   0 1442m  81m  30m S   1,3  0,5  15:25.02 compiz
 1609 rkj       20   0  433m  11m 8208 S   0,7  0,1  10:47.81 pulseaudio
 2304 rkj       20   0 1173m 218m  29m S   0,7  1,4   2:03.59 chrome
 2374 rkj       20   0 1046m 177m  18m S   0,7  1,1   5:18.37 chrome
   17 root      20   0     0    0    0 S   0,3  0,0   0:52.05 rcu_sched
 1224 root      20   0 19160  724  496 S   0,3  0,0   0:01.16 irqbalance
 7231 rkj       20   0 1049m 113m  26m S   0,3  0,7   0:06.91 chrome
    1 root      20   0 27224 3024 1448 S   0,0  0,0   0:00.99 init
    2 root      20   0     0    0    0 S   0,0  0,0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S   0,0  0,0   0:00.23 ksoftirqd/0
    5 root       0 -20     0    0    0 S   0,0  0,0   0:00.00 kworker/0:0H
    7 root      rt   0     0    0    0 S   0,0  0,0   0:00.12 migration/0
    8 root      20   0     0    0    0 S   0,0  0,0   0:00.00 rcu_bh
```

Figure 2.1: Windowing graphical user interface on Ubuntu Linux [13]

A windowing system supports the underlying graphic hardware, providing an abstraction layer on top of the graphics primitives of the compatible hardware processors and also taking advantage of it [14]. In this way client applications are not dependent on the graphic platform directly, but rather can be programmed targeting the chosen windowing system interface. Another feature of windowing systems is input handling: data coming from peripherals like mouse, keyboard and touchscreen displays is collected and then processed or forwarded to the application the data was originated for.

The main purpose of a windowing system is to allow multiple graphical programs to coexist on the same display screen. Each program runs in its own window, which is at minimum defined by its size and position on screen. The components of a windowing system can provide additional functionality: window borders and buttons to close or minimize them are also displayed to increase the usability of the system. Moving, resizing and closing windows are operations typically supported. Some windowing systems have also network capabilities, to display in a local window an application running on a remote system [15].

Some windowing systems are designed in a modular way, splitting their functionality in different processes to achieve greater flexibility and code reusability. When not, the common components are still present in the system, although combined in a single process or library.

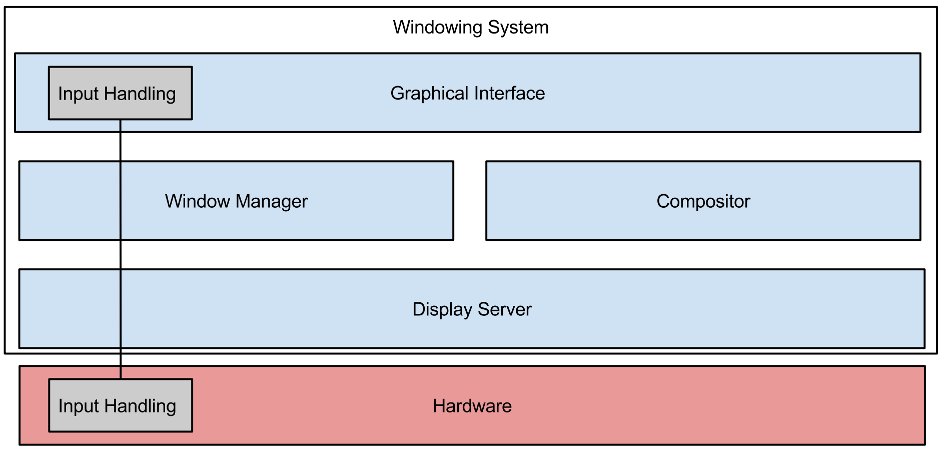


Figure 2.2: Components of a windowing system

While the most popular windowing systems vary greatly in functionality, all are built on the same basic components, depicted in fig. 2.2.

The *display server* is the core of a windowing system. It is responsible to coordinate the client applications with the rest of the components. The display server typically communicates with the graphic hardware, therefore implementing the functionality necessary to interact with the graphic drivers.

The *window manager* controls the placement of windows on the display screen. Additional metadata is also stored (minimized, full screen windows...). It provides an API to applications to create, modify and destroy windows. It also gives access to the compositor to metadata of the windows: their position and size, for example.

The *compositor* is the component responsible to grab the content of the application windows and paint it to the main screen on the rectangles associated with them. May be separated or included in the window manager depending on the windowing system architecture. The content of the single windows is stored in *offscreen buffers*, which are chunks of video memory not displayed on screen. The compositor draws the content of the offscreen buffers to a *screen buffer*, whose content is displayed on the screen visible to the end user. On and off screen buffers are called also *framebuffers*.

Also referred as desktop environment, the *graphical interface* is usually a set of graphical applications that provide widgets and decorations (icons, toolbars, panels, buttons) that improve the usability of the window paradigm adding the typical desktop functionality.

Input events like mouse clicks and key presses are forwarded to the window which they are intended for. This is done using the coordinates of the mouse event or the window currently in focus, for example.

Popular windowing systems distributed with source code available include Xserver[16], which is by far the most used graphical stack on unix-like systems, and Wayland [11], which is designed as a lightweight successor of Xserver.

2.2 OpenGL ES 2.0

OpenGL [17] is an Application Programming Interface (API) used to render 2D and 3D graphics. It is designed to communicate with graphics hardware in order to benefit from hardware acceleration in a device independent way. OpenGL itself defines just a standard, an API that vendors implement to be compatible with graphical applications. Multiple versions are available, up to OpenGL 4.4 [18], as well as versions designed for embedded systems called OpenGL ES [19]. OpenGL ES provides a subset of the OpenGL functionality, adapting the interface to run more efficiently on low powered embedded devices. The latest version of it, which is still relatively new, is OpenGLES 3.0, but most of the software currently in use still builds on GLES 2.0 (GLESv2), since drivers must first support the new platform before software can be developed/ported to it. OpenGL ES 2.0 and following differ greatly in functionality with version 1.1 and are not backwards compatible with it, therefore the analysis of GLES 2.0 that follows is not always valid for older versions.

OpenGL ES 2.0 offers a simpler interface compared to standard OpenGL, allowing the implementation to be optimized to run efficiently on embedded devices. The most important changes compared to it [20] are that there is no support for the fixed-function pipeline, everything has to be rendered through custom vertex and fragment shader programs, which are written in a c-like language, compiled and then executed on the GPU. There are also not `glBegin` and `glEnd` anymore: meshes are stored in vertex arrays and vertex buffers objects instead. Additionally, in OpenGLES only points, line and triangles can be rasterized (no quads).

OpenGL operates in a *context* based way. A context behaves like a container for a rendering state. Executing GL commands modifies only the state of the context which is currently active. In this way it is possible to draw different scenes on different context in the same application, and later combine them or show them independently.

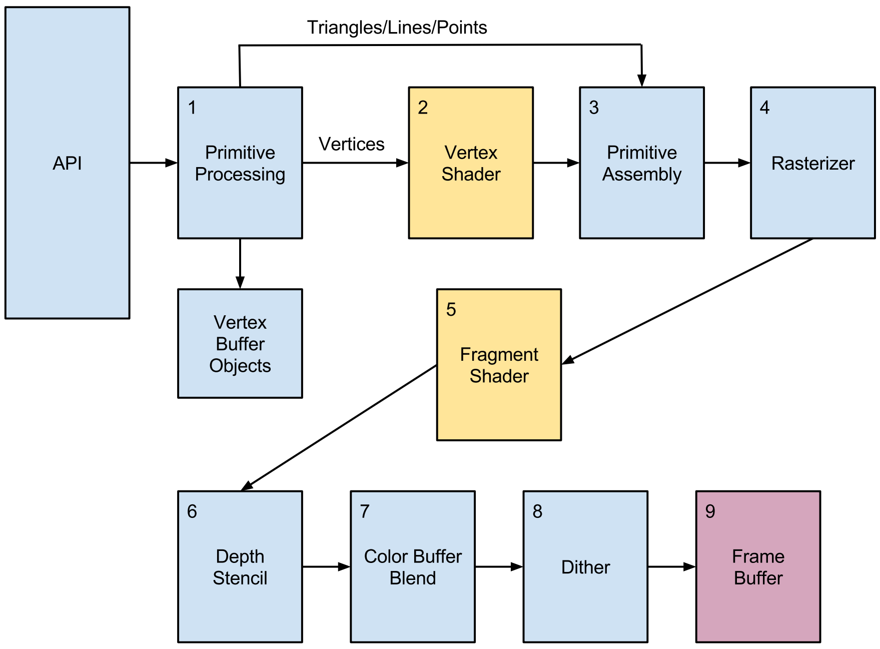


Figure 2.3: GLES 2.0 Pipeline (khronos.org)

OpenGL ES 2.0 features 2 drawing commands: `glDrawElements` and `glDrawArrays`. They are the entry points of the rendering pipeline depicted in fig. 2.3. They both specify (although in different ways) a set of vertices and a primitive to render them (points, lines, triangles). `glDrawArrays` sends the vertices in linear order, while with `glDrawElements` they can be submitted in a custom order, allowing for vertex reuse and therefore saving some memory in specific cases.

Since the pipeline in GLES 2.0 is programmable, transformations like rotations and translations are handled in the vertex shaders and executed in the pipeline in step 2 of fig. 2.3. Lightning and coloring have to be programmed in the fragment shaders instead, exectued in step 5. The final rendered result is drawn in a buffer provided by EGL.

2.3 EGL

A component of the OpenGL ES specification is the platform interface layer, EGL. The implementations of EGL provide bindings to the platform of choice, namely windowing systems and/or framebuffers. EGL is primarily responsible to provide surfaces to render on. A surface is a chunk of memory used in the last step of the GL pipeline, as seen in fig. 2.3. In case of an EGL backed up by a windowing system, the surface will appear within a window on screen. In case of a raw framebuffer backend, the surface will be rendered on an onscreen buffer directly or on an offscreen one, to later be accessed by a compositor.

A critical functionality offered by EGL is double buffering. Every drawable surface requested is allocated twice, referred as front and back buffers. The drawing commands are rendered on the backbuffer, while keeping the front one intact. When the command `eglSwapbuffers` is issued the command pipeline is flushed executing an implicit `glFlush` and then front and back buffers are exchanged. In this way there is always a frame, a graphical representation of a fully rendered application window at one point in time, rendered consistently that can be displayed on screen if necessary. This method makes it also possible for a compositor to run in parallel to the applications, fetching the surface content without delaying further drawing operations.

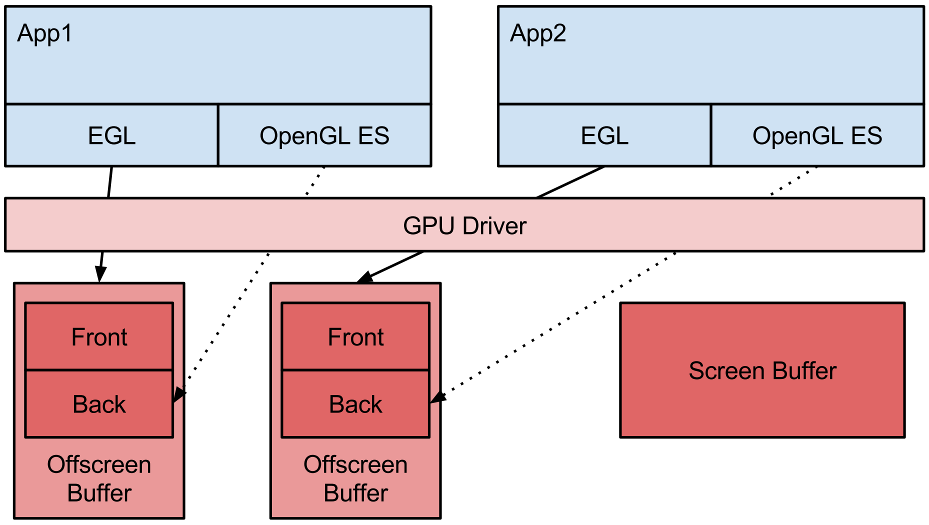


Figure 2.4: EGL usage with OpenGLES

In fig. 2.4 the usage of EGL in combination with OpenGL ES is presented. EGL is used in the client applications to communicate with the graphics driver to allocate offscreen buffers, in this way getting access to a drawable surface. OpenGLES API functions render the window content in the given surface, as seen in the last step of fig. 2.3. OpenGL ES takes advantage of the double buffering feature of EGL, drawing always in the backbuffer and completing a frame calling `eglSwapbuffers`.

The compositor is notified when `eglSwapbuffers` is called that the application has updated its content and therefore the corresponding region on screen must be updated accessing the updated offscreen buffer and drawing it in on screen. A compositor can also use GLES drawing functions to compose the windows on screen. This a practice adopted by many compositors on desktop systems [21]. Another option is to use drawing functions on framebuffers directly, which is often done on embedded systems with dedicated 2D libraries [22].

2.4 Bitblitting

Bitblitting is a computer graphics operation used to combine bitmaps according to the properties of a raster operator [23]. In this case, *bitmaps* are defined as rectangular images composed by pixels. Every pixels carries *RGB* values: red, green, blue and eventually an alpha channel (*RGBA* in that case). In the simplest case of bitblitting, only source and destination bitmaps must be specified. A third bitmap can act as *mask*. A raster operation, *ROP*, must also be specified: the ROP defines how the pixels of source, destination and mask bitmaps are combined. Comparing pixel by pixel huge bitmaps is highly inefficient to be done in software, therefore bitblitting support is typically integrated in GPUs.

In fig. 2.5, a bitblitting operation using a mask is shown. The first three panels represent source, mask and destination bitmaps. The fourth one shows the result of applying the operation: `result = (source AND mask) OR destination`.

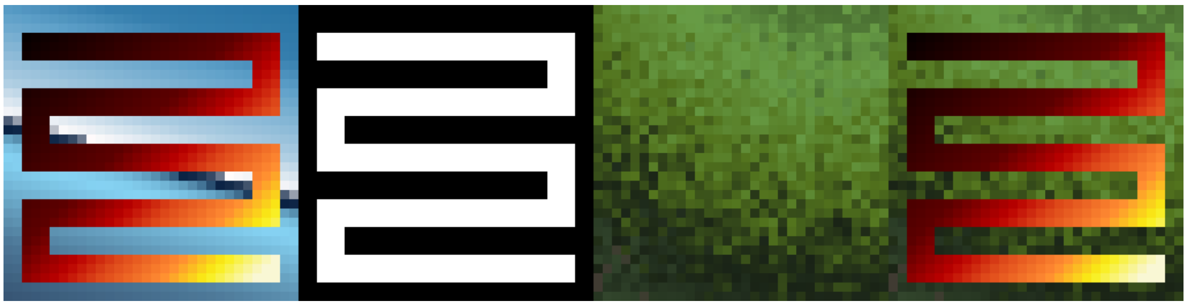


Figure 2.5: Bitblitting operation with source, mask, destination, result.

In the following chapters the term *bitblitting* will be generically used to define the operation of *copy-over* for rectangles: `result = source`. In absence of alpha channel (no transparency), copy-over just writes the source rectangle at the given location in the destination buffer. In case of alpha compositing, the *over* operator [24] is applied: $C_{out} = C_a\alpha_a + C_b\alpha_b(1 - \alpha_a)$. Alpha compositing is used when bitmaps also have an alpha channel.

3 System Model

A few assumptions on the system concerning the architecture and the components that interact with the compositor were made.

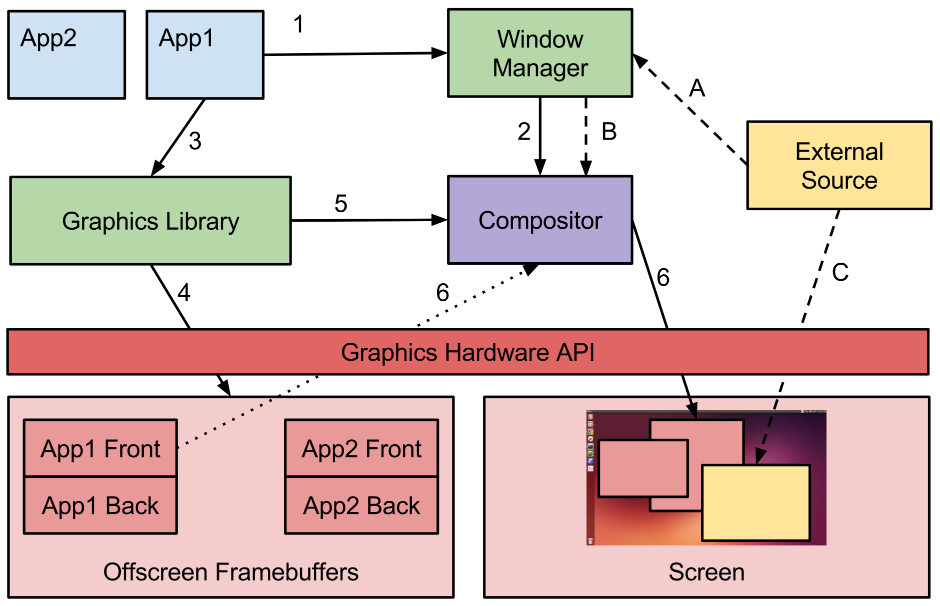


Figure 3.1: System model

Fig. 3.1 depicts the system model considered. In step 1, an application creates a new window through a window manager, which associates to every window size and position on the visible screen. The window manager restricts access and enforces rules on the requests for new windows. When a new window is created, the window manager informs the compositor (step 2) with id, size and position on screen of the new window. The window manager also communicates to the compositor the z placement of the window (how it appears in relation to other windows: above or below).

Graphical applications use a graphic library to allocate drawing surfaces in *offscreen buffers* (steps 3 and 4). Offscreen buffers are chunks of video memory which is not directly displayed on screen, but rather stored in memory only. Applications make use of the graphics library to render the graphical content on the offscreen buffers. A buffer is able to store one frame, a snapshot of a rendered window.

Additionally, the graphic library in use supports double buffering: for every desired application window two offscreen buffers are allocated, commonly named *frontbuffer* and *backbuffer*. The graphic library executes the rendering commands drawing on the backbuffer, swapping the pointers to front and backbuffers when the rendering of one frame is complete. In this way the frontbuffer contains always a consistent frame.

When a frame of a window is rendered completely the graphics library notifies the compositor with the window id (step 5). Periodically, the compositor accesses the frontbuffers of the applications which have updated their content (step 6) and bitblits them at the correct location on screen. The compositor and the graphics library both access the graphics hardware through a dedicated API.

The compositor is also aware that external sources can have exclusive access to regions of the visible screen, and therefore those regions must not be bitblitted at all. Those external sources are considered high priority applications that have full control on their region of screen, bypassing completely the compositing step. In fig. 3.1 an external source has been given full access to a rectangle on screen: in step A the application created through the window manager a new reserved window. In step B the window manager notifies the compositor that the reserved region must not be bitblitted. In step C the external application draws on the screen completely disjointed from the compositor's task, even if the window dedicated to the external source is overlapping with another window.

It is also assumed that the bitblitting hardware shows a dependency between amount of pixels to bitblit and time required to complete the operation. Specifically, bigger areas require more time to be bitblitted, depending on the fillrate of the GPU. The fillrate is defined as the number of pixels that the GPU can write to video memory in a second [25], and the peak value for specific GPUs can typically be found in their specification sheets.

4 Related Work

Compositors can be classified using two metrics: the interaction with the other components of the graphical system and the way the bitblitting is internally done. In this section three of the most popular windowing systems on desktops or embedded platforms are briefly examined and their compositors analyzed according to their system integration and bitblitting components. The three subsystems were chosen for different reasons: *X11* is historically the most popular window manager on unix systems, *Wayland* has been designed to replace it adopting EGL/GLES as standard platform and focusing on performance, *Surface Manager* is the window-manager/compositor with the deepest market penetration on embedded systems, being built-in in the Android mobile operating system.

4.1 Bitblitting

To bitblit a set of overlapping windows on screen there are two commonly used strategies, *full* and *tiled* compositing. Full compositing is the simplest algorithm for bitblitting, since involves applying the painter's algorithm [26] checking for overlapping windows. The painter's algorithm works by starting from the back, checking every window. If its content has been updated it will be bitblitted on screen along with all the windows overlapping it. This drawing method is very simple to implement but generates a lot of overdraw, since depending on the overlaps many pixels could be drawn two times or more. Time would be wasted bitblitting pixels that will be immediately overwritten.

Tiled compositing requires *window clipping* [27], which consists in analyzing how the windows appear on screen and cutting them into *tiles*, which are rectangles that combined together form the visible part of a window. The implementations of tiled compositing differ greatly. The algorithms used to decide how to clip the windows and how to store and access them have a significant impact on the final performance result. Pixman [28], for example, is a tiling library used in Wayland that does window tiling using horizontal stripes of rectangles and linked lists as data structure to store the generated tiles.

The left side of fig. 4.1 shows how a scenario with 3 windows is bitblitted using the *full* strategy. Bitblitting A, for example, requires redrawing the complete scenario: since both B and C overlap it, bitblitting A will partially overwrite them, therefore requiring their redraw.

On the right side of fig. 4.1 the tiling algorithm of the Pixman library is applied to the same scenario of 3 windows. In this case bitblitting A, B or C does not affect the other windows because only the visible parts of the windows are bitblitted. In this case drawing A in tiled mode requires bitblitting 3 rectangles for a total area smaller than the original area of *A*. The same is valid for *B*.

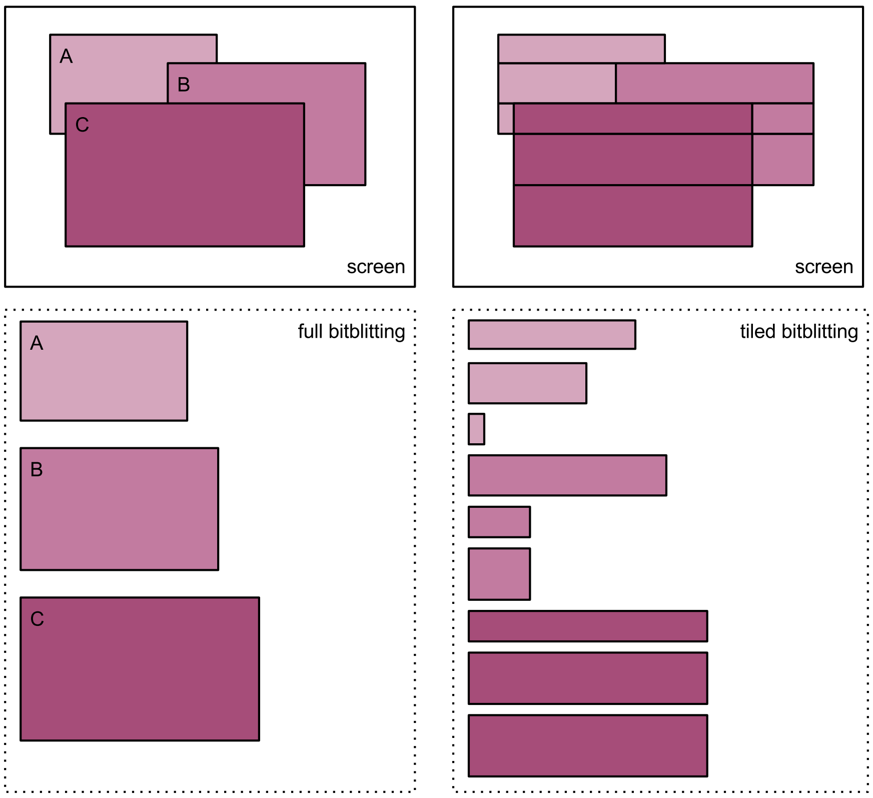


Figure 4.1: Full (on the left) and tiled (on the right) compositing strategies

The *full* and *tiled* strategies both present advantages and disadvantages, none of them is strictly better than the other one. Applying the *full* strategy usually results in a lot of overdraw, where regions on the screen are drawn multiple times effectively wasting bitblitting time. The *tiled* strategy prevents overdraw but introduces additional overhead due to the higher number of drawing operations (one for every tile instead of one for every window), which have typically a non-zero overhead.

4.2 Xserver

The X windowing system (also called X11) is a popular windowing system that targets desktop computers [29]. It is built following the client/server architecture, with Xserver being the server part. An overview of how the Xserver interacts with the rest of the system is first of all given. The focus is on the graphical subsystem since more relevant to the topic, input handling is not analyzed. Fig. 4.2 shows the layers of a typical X11 graphics stack. At the top of the stack X11 client applications are found, which can create windows and also draw their content using the X API.

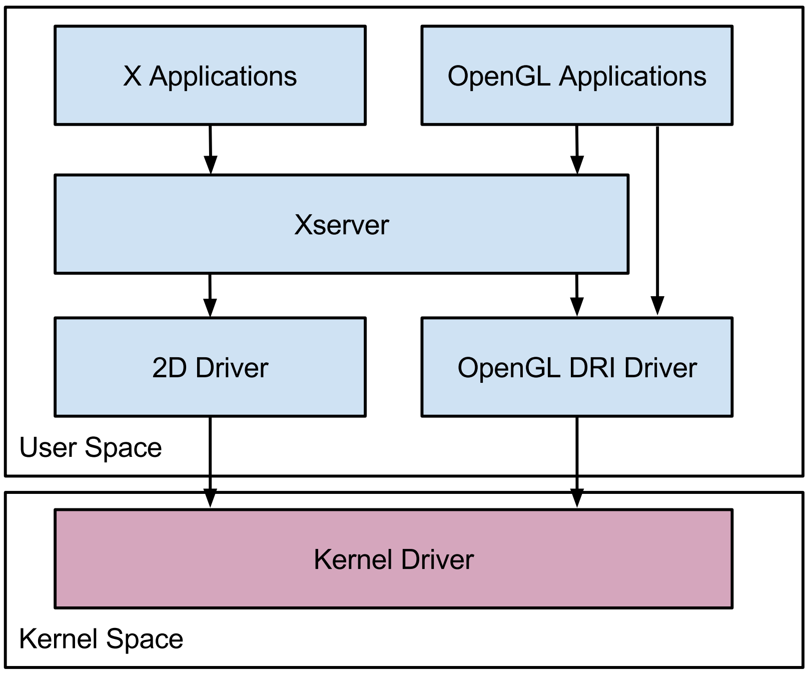


Figure 4.2: Architecture of a typical X windowing system

OpenGL applications behave differently: they interact with the Xserver to create a window in the system and to obtain a drawing surface identifier, but the actual drawing is done in the application using OpenGL primitives, which in turn can be (and often are) GPU accelerated. 3D graphical applications are supported in this way. A more detailed explanation of how OpenGL rendering works is described in section 2.2.

The X11 system in order to support the underlying hardware requires user space libraries that follow the X11 standard, providing a hardware abstraction layer on top of the wide range of GPU drivers that X supports. In this way the code of X applications remains portable among different platforms, requiring only the development of such libraries to provide X with access to the hardware dependent graphic API in a standardized way. As fig. 4.2 shows, X11 has typically access to 2D and 3D (OpenGL DRI) backends through those libraries, which act as user space drivers: they are responsible to communicate with the kernel space and send/receive graphical commands and data.

Internally X is designed to be extremely modular, only the basic framework to handle windows is built in. The specific graphical appearance is dictated by other programs, which behave as compositors and window decorators. The advantage of this architecture is that X, although offering a basic compositing functionality if nothing else is provided, must not be directly involved in the rendering of the user interfaces, allowing the window manager and the compositor to run as separate processes.

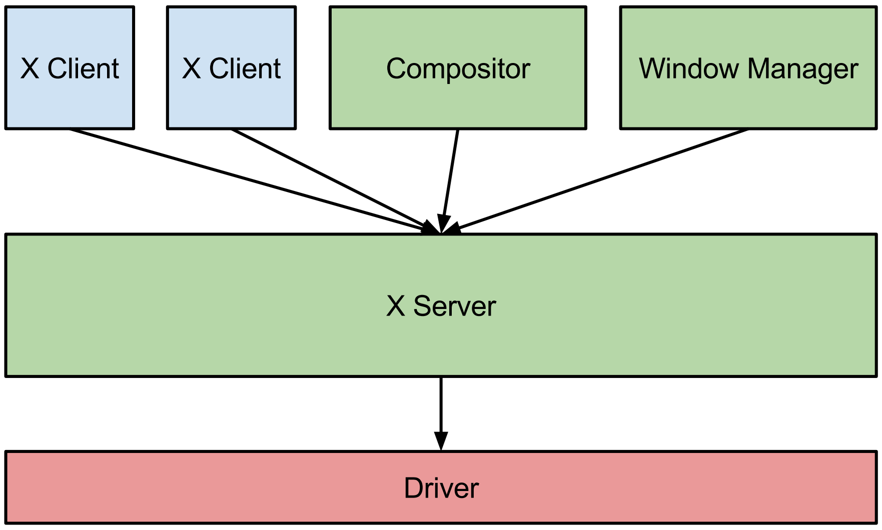


Figure 4.3: Compositors in X

Fig. 4.3 depicts the relationship between Xserver and Xclients. Compositors and window managers are also client of the Xserver, using the X API to manage and bitblit windows.

If the X server is configured with the *composite* [16] extension enabled, the content of all windows can be redirected to offscreen buffers with the function `XCompositeRedirectSubwindows` [30]. In this way a compositing process can access their content and have full control on the visible screen buffer. The applications themselves are unaware that they are drawing offscreen, therefore introducing a custom compositor is completely transparent from the applications point of view and no additional programming is necessary. The information that the compositor is given about the windows consists of their position on the screen (x and y coordinates), their size (width and height) and a pointer to the offscreen buffer (which also contains data about color format and alpha channel). When the content of one window is updated the compositor is notified and can then choose itself how and when to update the screen. This is the same behavior that the system model described in chapter 3 expects.

xcompmgr [31] is a simple although fully functional compositor for X, providing transparency effects, window decorations and shadows to the managed windows. *xcompmgr* supports alpha compositing, following the principles of Port-Duff transparency [24], enabling windows to have fully or partially transparent pixels. It uses the *xdamage* [16] extension to track the regions of the screen that have been modified and must be updated. X uses an event system to track window modifications, which are parsed by the damage extension to mark the windows that require a redraw. When a window appearing below another one has been drawn to the screen buffer, the windows on top of it must also be drawn again on the screen to keep the order consistent. This is known as painter's algorithm [26], which consists in keeping the windows in a z-ordered list and painting them back to front. If only the updated windows and those laying directly above of them are painted, some drawing time can be saved.

```
for (w : windows) {
  if (! is_damaged(w))
    continue;

  draw(w);
  damage(region(w));
}
```

Listing 4.1: xcompmgr rendering loop

The core rendering loop of *xcompmgr* can be represented by the pseudocode in listing 4.1. Every window is checked for updates (using the xdamage extension). If the window did not update the buffer and no overlapping window below it was drawn, the window is not drawn. If drawing is necessary, the windows is drawn and the region marked as *damaged*. In this way overlapping windows on top of it will be also drawn in the next cycles of the loop. The algorithm implements the *full* compositing strategy described in section 4.1, exhibiting significant overdraw in case of multiple windows overlapping each other.

4.3 Wayland

Wayland has been developed as a replacement for X, a lightweight solution that does not carry all the functionality that X accumulated over the years and focuses on simplicity and performance [11]. In a Wayland windowing system there is no separation between display server, window manager and compositor, everything is implemented in the same Wayland server. Wayland is just a protocol that client applications use to communicate with a Wayland compositor, and the actual rendering of the windows' content happens always in the client applications, like in the case of 3D OpenGL applications on X server. This simplifies a lot the server that can be seen now as a bare compositor, delegating also the window decoration to the client applications or to the client Wayland library implementation (toolkit).

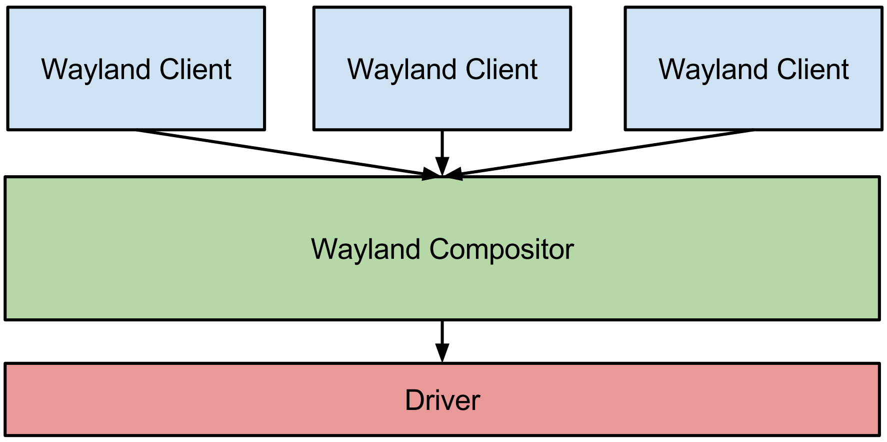


Figure 4.4: Compositors in Wayland

Wayland accesses the hardware drivers through EGL and GLESv2, the GL version designed for embedded systems, avoiding in this way to pull in any dependency to X or other windowing systems. The most obvious candidates, GL and GLX, would require linking to the X client libraries, therefore making Wayland dependent on X rather than an alternative solution.

Since Wayland just specifies a communication protocol, the compositing operations are not standardized among the Wayland window manager implementations. The most popular of these is *Weston*, which could be considered as the reference implementation. In Wayland compositor and window manager coincide, as shown in fig. 4.4.

Weston internally offers different compositors for different backends, making also use of the *Pixman* library [28]. The library provides *tiling* functionality, taking into account how windows overlap and building lists of the resulting visible *tiles*, rectangular slices of windows. In this way the compositor can minimize the surface that has to be blitblitted, avoiding unnecessary redraws of windows that are immediately going to be covered by another draw command. By using Pixman, the compositor in Weston is affected by advantages and disadvantages of using a *tiling* compositing strategy, as described in section 4.1.

4.4 Android's Surface Manager

Android names his window manager *surface manager*, which is actually managing surfaces (widgets or views) on screen rather than traditional windows, since windows typically extend to the full screen on Android. The surfaces in Android behave like windows on desktop systems, since they can be placed in any overlapping combination. Fig. 4.6 colors in green, red and blue the surfaces in the Android calculator and calendar applications.

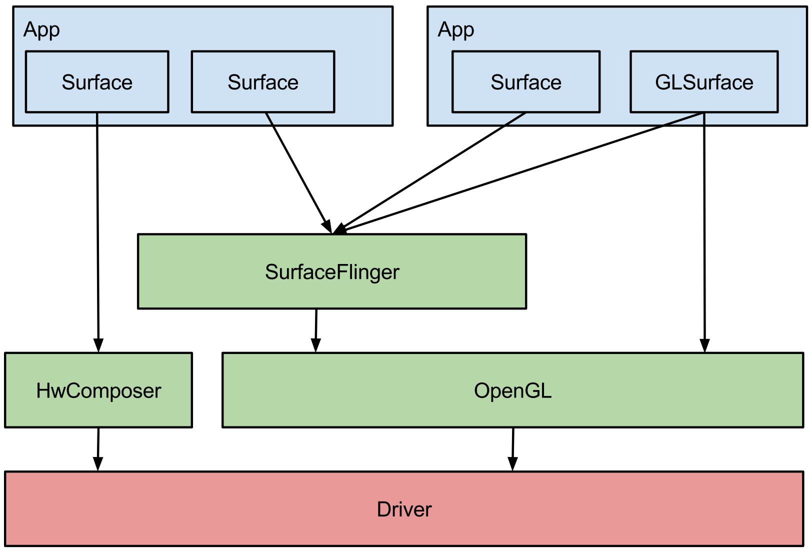


Figure 4.5: Compositor in Android

The drawing in Android is done with OpenGLES, applications can use the API directly on GL surfaces as well as using the Android UI API to draw widgets, forms, buttons. Internally the compositing is done by *SurfaceFlinger* [32], which also uses OpenGLES to compose the surfaces together on display. Being the compositor a performance critical service running on an embedded system, a further optimization is present on some embedded hardware: SurfaceFlinger can offload to a *hardware composer* the job of drawing one or more surfaces directly on the corresponding regions of the screen, as seen in fig. 4.5. SurfaceFlinger performs the compositing in a way similar to X, comparing rectangles at runtime to detect what regions have been updated to redraw the surfaces which have to appear on top of them [12]. This corresponds to the *full* compositing strategy described in section 4.1, not avoiding overdraw in every way. This can be seen in fig. 4.6 too, where the overlapping surfaces are colored by the Android GPU debugger, exhibiting overdraw.

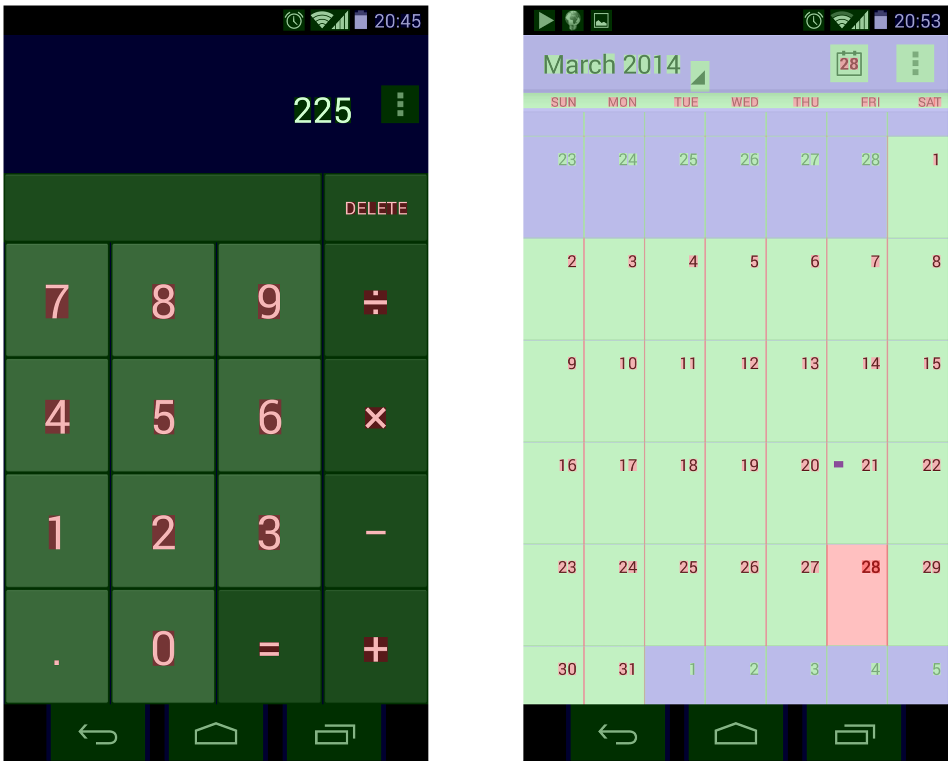


Figure 4.6: Surface overdraw in Android [33]

5 Concepts

In this chapter a concept for an efficient compositor is presented, which makes use of informations about the duration of single bitblitting operations to optimize the number and size of drawing commands sent to the GPU. When two overlapping windows are bitblitted completely they cause *overdraw*. In this case the window below is fully or partially not visible because covered by the topmost one, and the time spent bitblitting the region not visible is wasted. In case of a complex scenario involving multiple overlapping windows a lot of overdraw could happen. The bitblitting time of every scenario of windows where two or more of them overlap could be therefore optimized by analyzing the window layout and devising a strategy to avoid overdraw. The compositing concept presented in this chapter addresses the problem of handling and drawing sets of overlapping windows.

The abstraction used to represent application windows on screen is first of all explained and the relationships that could manifest between windows classified. Then the design of a solution to store and retrieve windows metadata is described, along with a novel compositing strategy, which is able to effectively operate on the data structure to find the optimal set of bitblitting commands to draw the application windows on screen.

5.1 Architecture

In fig. 5.1 the architecture of the compositor is presented. Every graphical *application* creates windows using the API of a window manager, receiving a unique window id. A graphics library is used to render the graphical content on offscreen framebuffers stored in video memory and to communicate with the compositor when the content of a window has been updated. The *window manager* handles the creation and destruction of windows, enforcing also access restriction according to the system configuration. It forwards the window data when a new window is created or destroyed to the compositor using the compositor's API.

The *compositor* is responsible to draw the content of the applications in the corresponding regions on screen. The compositor receives information on the disposition of windows on screen in the form of positioned rectangles. Since the content of the windows does not affect their layout on screen, the compositor needs only the coordinates of one corner as well as width and height of the displayed window to draw it correctly. Because of this property, storing and drawing windows is handled using rectangles to represent them. The window manager and the graphics library use an API to communicate with the compositor, which supports through it a set of *operations* to be executed on rectangles. The rectangles sent by the window manager are processed then and stored in an internal *data structure* suited to efficiently access and further process them. The compositor implements *drawing algorithms* which operate on the rectangles and decide how the bitblitting for the current frame must be done to minimize the bitblitting time. The decisions are taken using the *prediction* module, which estimates the time requirements of single bitblitting commands. The set of rectangles generated by the drawing algorithms is then passed to the *bitblitting functions*, which access the applications' framebuffers and execute the platform dependent drawing commands on the screen's framebuffers.

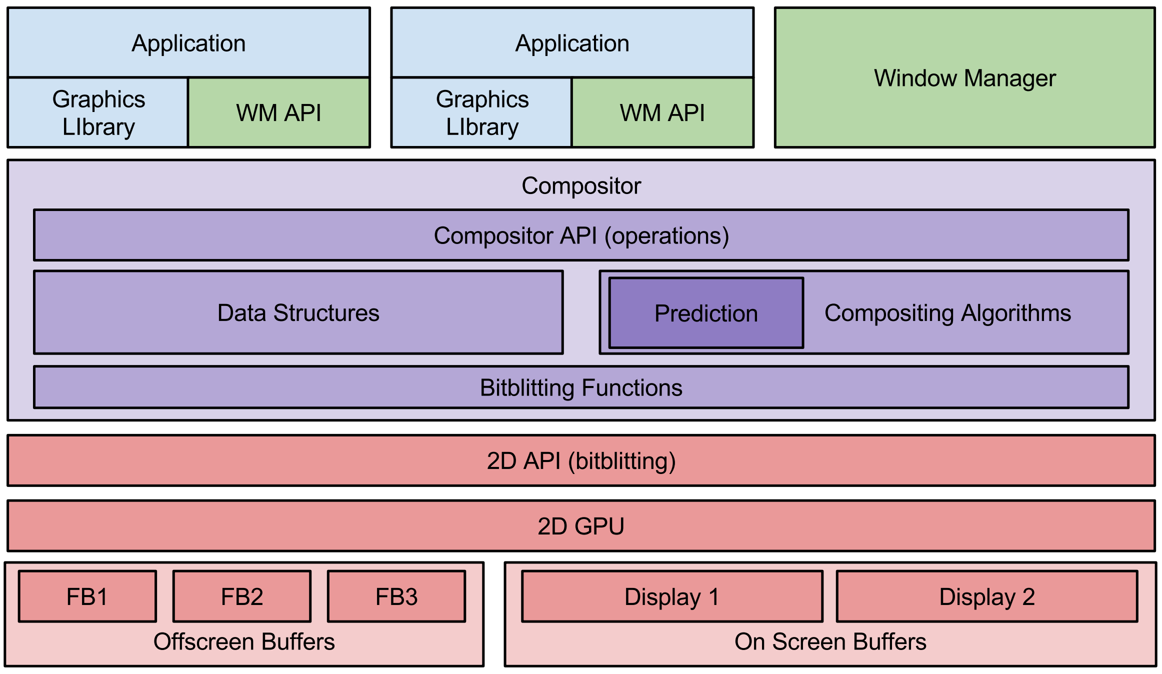


Figure 5.1: Compositor's Architecture

A *2D API* is used by the compositor for bitblitting, backed by a *2D GPU* for hardware accelerated bitblitting operations. The bitblitting functions implemented in the compositor require the 2D API to support transparency, using an alpha channel or color masks.

Offscreen framebuffers are used by the graphics library to store the content of the rendered frames. They are stored in viedo memory and not visible to the user until the compositor bitblits their content on the *screen buffers* which are displayed on the screen visible to the end user. Every screen buffers represents one connected display.

5.2 Operations

The functionality of the compositor involves operations on rectangles. An API is provided to the window manager and to the graphical libraries to interact with the compositor, defined by a set of operations that cover the core functionality that defines its behavior.

To define the rectangles a model was chosen, which identifies them with the tuple (x, y, w, h, z): x coordinate of the top-left corner, y coordinate of the top-left corner, width, height and z value. *Z* defines how rectangles are ordered on the z-axis: using this parameter new rectangles can be sent to the compositor not only in foreground or background, but also between other stacked ones.

The basic operations on rectangles of a set are: **insert** a new rectangle, **remove** an existing one, **modify** the properties of one, **mark** a new rectangle when the content of the window that generated it is updated, **compose** the screen by drawing the rectangles in the set which have been updated. Other operations can be implemented on top of the basic ones.

For every operation the frequency of its execution must be considered in order to devise an optimal data structure to hold the rectangles. The core operations that the compositor will support are now analyzed focusing on this aspect.

Insert

`insert(wid, rect)`: The **insert** operation adds a rectangle to the set which is not yet part of it keeping also track of the *window id* (*wid*). In this way it can later be referenced by other operations. The rectangle can overlap in any way the other members of the set, and can appear on top, on bottom or at any z-value without any restriction on the order of the insertions. Insertion is a basic operation on the set but is seldom used compared to other ones, since it is called by the window manger only when a new window is created.

Remove

`remove(wid)`: The **remove** operation subtracts a rectangle belonging to the set from it. Any rectangle of the set can be removed without any restriction. Removal is not expected to be an operation executed frequently since it occurs only when a new window is destroyed by the window manager.

Modify

`modify(wid, rect)`: The **modify** operation changes the properties of a rectangle belonging to the set. It is possible that windows are being modified rarely (in case of resize) as well as very often (in case of animations), therefore **modify** can be considered an operation executed more often than the previously described ones. It is called by the window manager.

Mark

`mark(wid)`: The **mark** operation notifies that the content of a rectangle of the set has been updated. This is a fundamental information necessary to minimize the number of windows to redraw. This operation is considered to be a time critical one, since in every rendered frame multiple windows will update their content and therefore execute **mark**. This operation is used in the graphics library.

Compose

`compose()`: The **compose** operation yields a set of rectangles that have to be redrawn on screen, using the inserted and the marked rectangles as input. This operation is typically run at a fixed frequency to achieve a fixed framerate, and it is by design the most expensive operation of the core ones because it requires to parse the data structure containing the windows to retrieve those one selected to be displayed. When the execution of the operation is completed, previous **marks** are reset. It is executed in the compositor's own process.

5.3 Classification of overlapping rectangles

The compositor aims to improve the bitblitting of windows sets presenting overlapping rectangles. To do this, the ways rectangles can overlap are analyzed and rules to classify their interactions outlined.

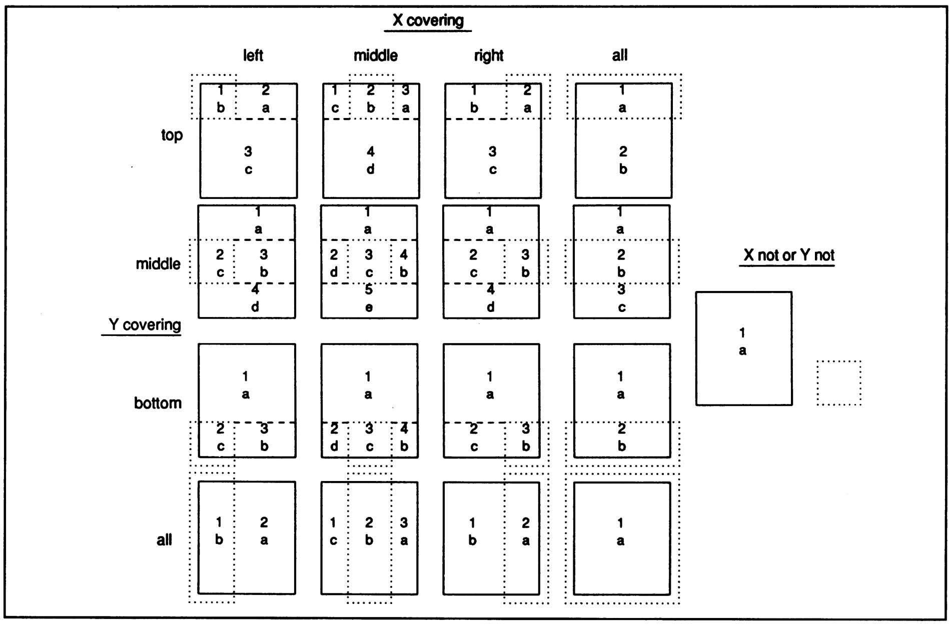


Figure 5.2: Myers [10] classification in 17 cases

Related work on rectangle overlaps is available, in [10] the relationships between two rectangles are classified in 17 cases, as shown in fig. 5.2. The classification rule takes into account how two rectangles overlap on the X and Y axis independently. The dotted rectangle can cover the solid one one the left or right side, on the middle or cover it completely. This is considered for both X and Y axis, yielding 16 cases. The 17th one matches rectangles that do not overlap at all.

The compositor aims to reduce overdraw, therefore a classification method focusing more on the overlap type rather than its orientation is defined. The classification is performed taking two rectangles and applying two rules:

1. If the topmost rectangle is not completely contained into the bottom one, only the overlapping part has to be considered for the classification.
2. The classification depends on the number of matching sides between the top and the bottom rectangles.

When applying the two rules defined above those 16 are reduced to 5. Any possible intersection between two rectangles can therefore be categorized into one of those 5 basic cases.

The cases differentiate in the number of sides that at least partially match between the two rectangles, and in the z-value. For two rectangles A and B, each defined by the tuple (x, y, w, h), the expression

$$N = (A_x == B_x) + (A_y == B_y) + ((A_x + A_w) == (B_x + B_w)) + ((A_y + A_h) == (B_y + B_h))$$

yields the number of matching sides, in the range $[0, 4]$. In the expression, the symbol $==$ results in 1 when the two operands are equal, zero otherwise. Summing the output of the four comparison operations results in the matching case.

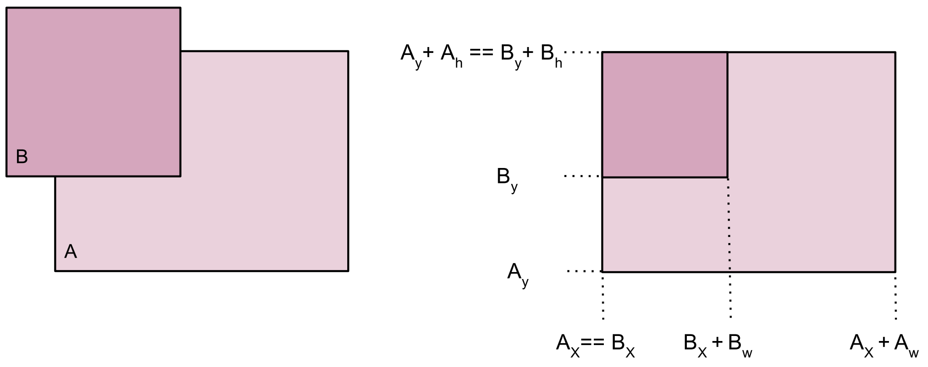


Figure 5.3: Matching sides

In fig. 5.3 an example on how matching sides are calculated is shown. First of all the rectangle lying above, B, is cut to stay within A, according to rule 1. Then rule *2* is applied to the two rectangles, which have

$$A_x == B_x$$

and

$$(A_y + A_h) == (B_y + B_h)$$

resulting in two matching sides.

In fig. 5.4 the cases M_0, M_1, M_2 and M_3 are displayed. Case M_4 is handled in a special way: the bottom rectangle is completely covered, see fig. 5.5. As long as the bottom rectangle is completely covered even if all 4 sides don't match the pair of rectangles is classified as belonging to case M_4.

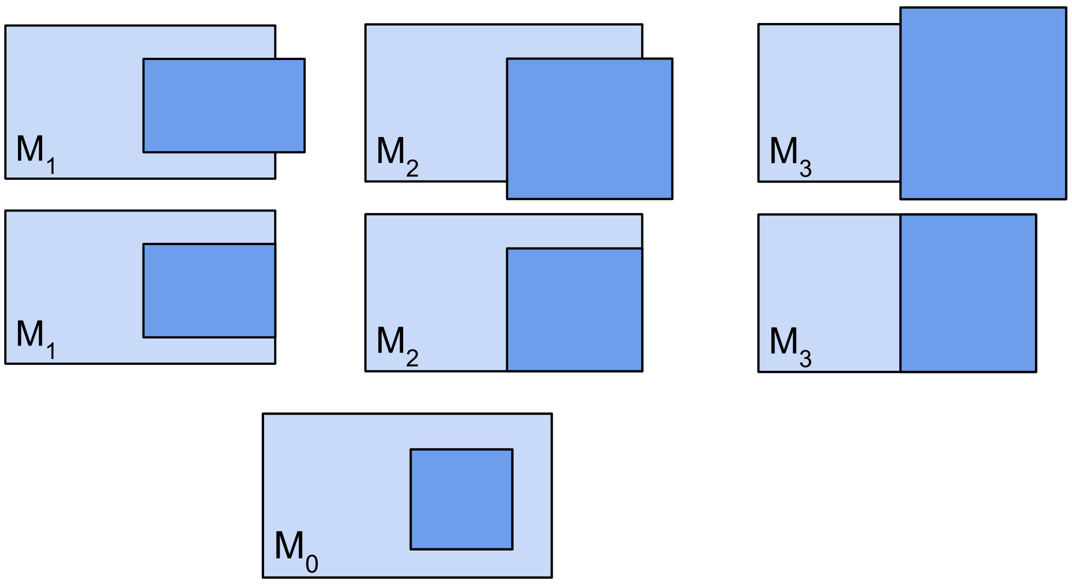


Figure 5.4: Classification cases M_0, M_1, M_2, M_3

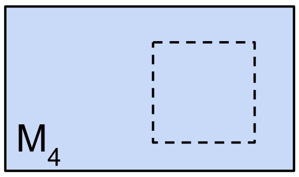


Figure 5.5: Special case M_4

5.4 Tiling of overlapping rectangles

To minimize overdraw, rectangles representing overlapping windows should not be drawn completely. The bitblitting functions implemented in the 2D graphics library support although only complete rectangular regions: it is not possible to specify that, within a rectangle to be bitblitted, part of it must not be drawn. For this reason rectangles representing windows must be split into smaller ones representing the visible parts of them and then used in the bitblitting functions. This process is called *tiling*, and the smaller rectangles belonging to the original one are called *tiles*.

Using the classification cases described in the section 5.3, a tiling algorithm has been developed to generate the visible tiles of a rectangle lying below another one, while also minimizing the number of tiles generated in this way. The tiling algorithm maps the possible outcomes directly to the 5 classification cases: the amount of generated tiles depends on the number of matching sides.

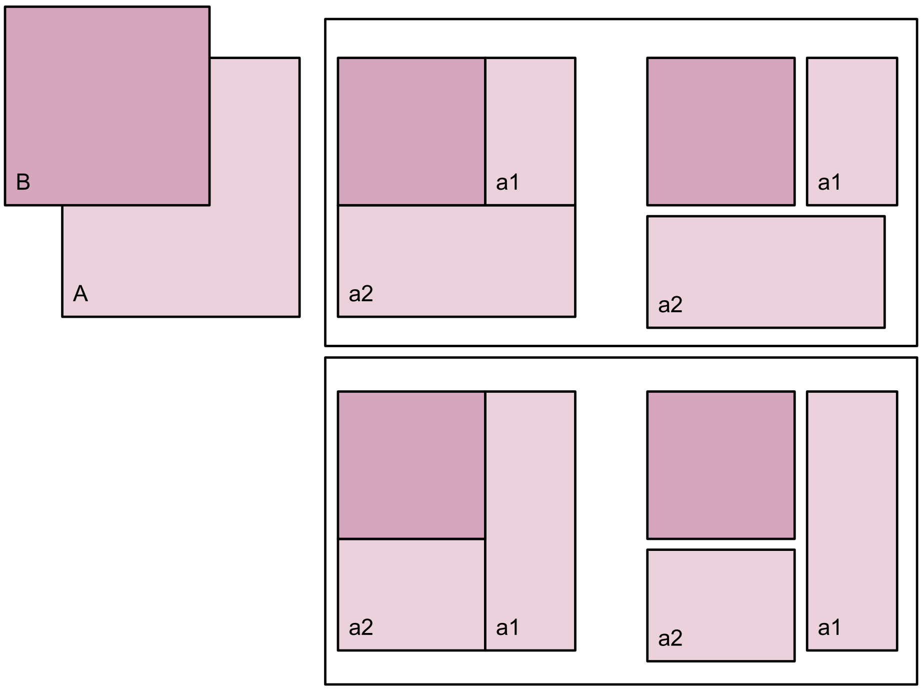


Figure 5.6: Tiling example

Tiling is done by cutting the bottom rectangle on the extension of the sides of the top one. In fig. 5.6, an example is presented. On the left side, the original two rectangles are shown. On the right, two ways of cutting the bottom rectangle A are proposed, both leading to the same total surface. The choice in the shape of the tiles (to cut vertically or horizontally in this case) does not affect the final area and is therefore be left to be implementation dependent.

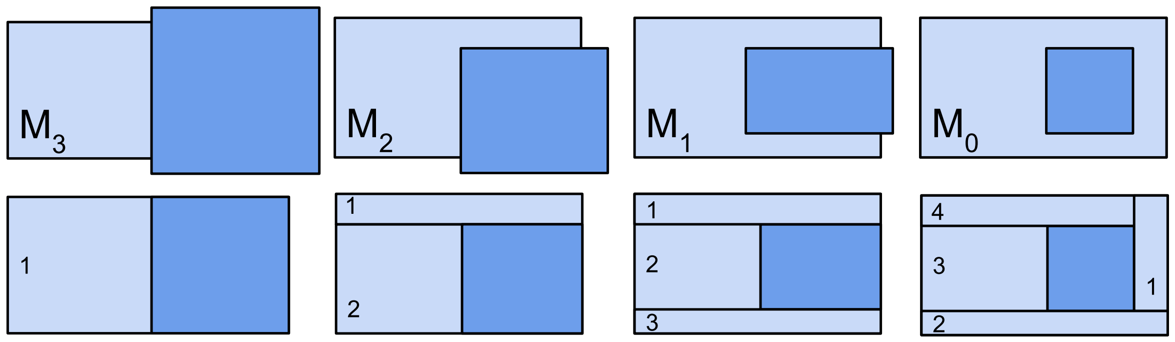


Figure 5.7: Tiling for classification cases M_0, M_1, M_2, M_3

The 5 classification cases are depicted in fig. 5.7. M_3 occurs when the top and bottom rectangles have 3 matching sides. Subtracting from the bottom rectangle the topmost one results in a single tile, labeled *1*. M_2 occurs when the top and bottom rectangles have 2 matching sides. Subtracting from the bottom rectangle the topmost one results in a shape which can be cut on the extensions of one of the non-matching sides of the topmost rectangle yielding 2 tiles, labeled *1* and *2*. M_1 occurs when the top and bottom rectangles have 1 matching side. Cutting the visible part of the bottom rectangle on the extensions of the topmost rectangle's sides yields 3 tiles, labeled *1*, *2*, *3*. M_0 occurs when the top and bottom rectangles have no matching sides and the topmost rectangles is contained in the bottom one. Cutting the visible part of the bottom rectangle on the extensions of the topmost one's sides yields 4 tiles, labeled *1*, *2*, *3* and *4*. In case M_4 there are no tiles, since only one rectangle is visible.

```
function do_tiling(a,b) {
  if (not a.overlap(b))
    return a
  else
    return cut_new_tiles(a,b) // according to the classification
}

rectangles = sort_by_z(rectangles)
r = rectangles.shift

tiles = array()
tiles.push(r)

for (v : rectangles) {
    new_tiles = array()
    for(t : tiles) {
      new_tiles.push(do_tiling(t, v))
    }
    tiles = new_tiles
  }
}
```

Listing 5.1: Tiling algorithm

The tiling algorithms described is valid for 2 rectangles. A scenario of more than 2 rectangles is processed recursively, starting with the bottom rectangle. The pseudocode in listing 5.1 describes how the tiling algorithm is applied to find the `tiles` of rectangle `r`. In the pseudocode, the rectangles are stored into an array sorted by z value `rectangles`. `r`, the rectangle lying on the bottom, is taken out of the array to find its tiles. At first, there are no tiles in the resulting array `tiles`. The original rectangle is pushed into it. A loop is run on every remaining rectangle, and every tile is checked against the current one, slicing the tile according to the rules defined above.

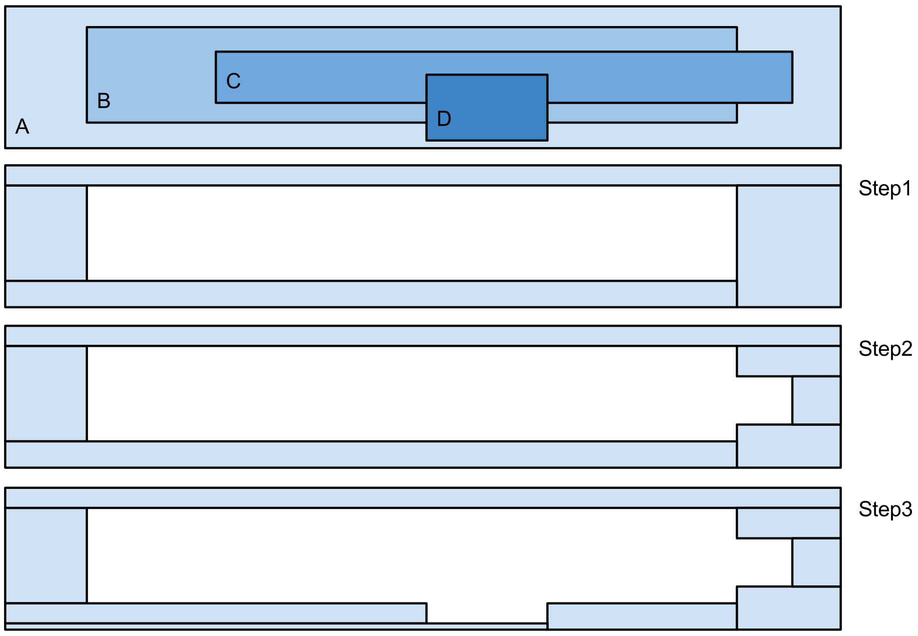


Figure 5.8: Application of the tiling rules with more than 2 overlapping rectangles.

In fig. 5.8 a practical example is shown, depicting the steps necessary to find the visible tiles of the bottom rectangle A of a set $S = \{A, B, C, D\}$. The rectangle lying directly above A is the first one involved in the tiling process, rectangle B. A is cut along the extension of B, resulting in 4 tiles (step 1). Now C is considered, being the rectangle immediately above B. The overlap of C with every tile of A until now generated is checked: one is matching. It will be further cut into 3 new tiles (step 2). The 6 tiles created will be then compared with D to find the final set of tiles of A not covered by any of the rectangles on top of it (step 3).

5.5 Data structure

In the previous section it was shown how tiling can avoid the problems of overdraw in case of overlapping rectangles. On the other hand, running the tiling algorithm described on every frame in the **compose** operation, does not look feasible for performance reasons, being the algorithm fairly demanding with higher number of rectangles in the set. In contrast, **insert** and **remove** operations happen only rarely. For this reason a data structure has been designed to store rectangles and tiles, in order to facilitate their handling and retrieval running the **compose** operation, which is expected to be the most computationally intensive operation.

The factors that influence the design of a data structure are CPU and memory usage. The latency of access and manipulation of the data must be compared to the amount of memory necessary to store it. In this specific case, the memory impact of the data to be stored is minimal: a structure containing the tuple (x, y, w, h, z) can be stored in a few bytes. On the other hand, many of the operations described in 5.2 are run many times every second requiring minimal latency.

Being the minimization of bitblitting time the main focus, simple bitblitting strategies as those described in 4.1 don't suffice. A straightforward example consists in a linked list containing the rectangles: in case of a complicated scenario with overlapping windows a lot of time would be wasted redrawing the same areas of the screen again and again. Thus a tiling approach is considered: rectangles should be split into tiles according to the overlap relations and then stored to avoid having to recalculate all the tiles at **compose** time. In this way overdrawing is avoided to the expense of the number of rectangles to bitblit: a pure tiling approach can lead to a very high number of tiles, each of them requiring to be bitblitted with a non-zero overhead for every bitblitting function call.

5.5.1 Graph

A *graph* structure has been chosen to store the tiles generated with the algorithm in 5.4 as well as the original rectangles, to be flexible enough to be used in a more advanced bitblitting strategy.

The graph has two different types of *nodes*, each node stores a rectangle or a tile. The entry point of the graph is a z-node which contains the rectangle of the bottom window in the windows set.

z-nodes are top-level nodes which contain information on the original window size, and point to the next window in z-order, using the z value of the rectangle contained in them to do the ordering.

t-nodes are tile nodes, pointed by window nodes or other tile nodes. They contain the rectangles still visible after overlapping the rectangle in the parent z-node with another rectangle in the z-nodes.

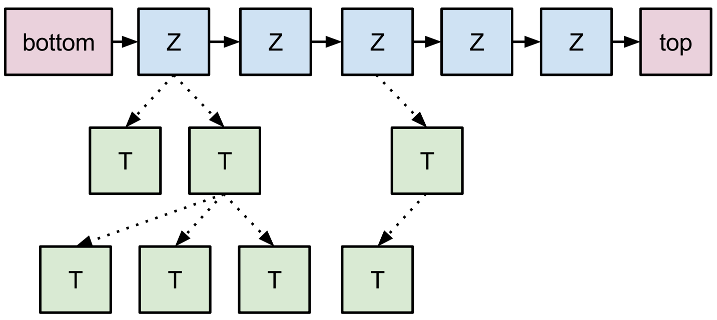


Figure 5.9: z-nodes and t-nodes

There are three kinds of *links* connecting the nodes, each link is unidirectional and connects two nodes of same or different kind, depending on the type of the link. In fig. 5.10: straight solid lines are z-links, straight dotted lines are t-links, curved solid lines are dep-links.

z-links connect a z-node to the z-node with the next z-value above it.

t-links connect a z-node or a t-node to a t-node generated from it.

dep-links are dependency links: they connect a t-node to the z-node that, overlapping the window the t-node belongs to, originated the t-node.

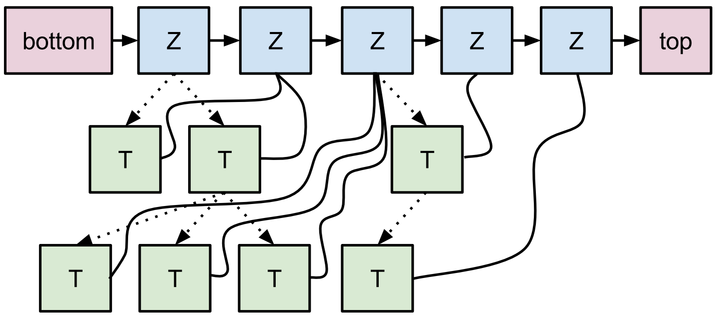


Figure 5.10: Graph structure with links

The graph is built into the **insert** operation. The pseudocode in listing 5.2 describes how the graph is generated.

```
insert(wid, new_rect) {

  create a new z-node with wid and new_rect
  insert the new z-node where it belongs in the zlinked list

  for every z-node: {
    if z-node has a z-value below new_rect:
       free t-nodes of this z-node which have as dependency: z-nodes with z-values above new_rect
       regenerate t-nodes of this z-node using: new_rect and all rectangles with z-values above
             new_rect
```

```
   if z-node is new_rect:
      regenerate t-nodes of this z-node using: new_rect and all rectangles with z-values above
          new_rect

   if z-node is above new_rect:
      free all t-nodes of this z-node
      regenerate the tiles of this z-node with: all rectangle with zvalues above this z-node
 }
}
```

Listing 5.2: Graph insert

The pseudocode in listing 5.2 roughly defines what happens when a new rectangle in inserted in the graph: new rectangles coming from the window manager are stored into z-nodes ordered by z-value. For each of them, all the tiles generated by tiling them with every other rectangle on top of them are stored in t-nodes, in a tree linked from the parent z-node.

Fig. 5.11 shows a scenario on left and the tiles generated from it on the right. This scenario is used then to present an example of the algorithm that builds the graph.

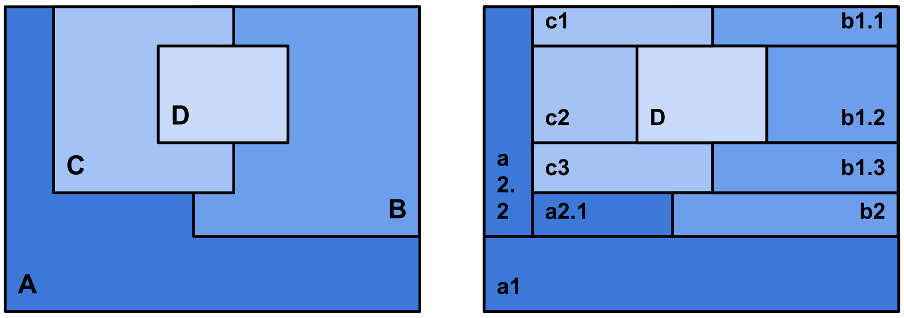


Figure 5.11: Example scenario

The graph is built from the scenario of 4 windows (A, B, C, D) seen in fig. 5.11. The graph itself is shown in fig. 5.12. The graph is generated from the bottom to the top, comparing every pair of overlapping windows according to the rules in 5.3. S is the entry point of the graph, and always points to the bottom z-node. The first window to be inserted is A: a z-node containing A is created and S updated to point to it. Then B is considered and compared with A: they have two matching sides, therefore this is case M_2: 2 tiles will be generated from A, *a1* and *a2*. They will point with a dep-link to B. A z-link is now added between A and B, and t-links between A and *a1*, A and *a2*.

C is then added: it will be first checked for overlaps with A: it matches, so the t-nodes belonging to A have to be checked: using deep-first search, the rectangle C is matched to find all t-nodes without outgoing t-links that overlap with it. Those are the tiles still visible. *a2* matches: following the procedure explained in 5.4, C is first cut to be contained within *a2* boundary. The result can be classified as the case M_2: two tiles will be generated and linked to *a2*: *a2.1*, *a2.2*. When done with the t-nodes of A, the z-link between A and its successor is followed. The slice of C which did not overlap with A is then compared with B and there is an overlap of type M_2 again: *b1*, *b2* are generated and linked to B. C is then added through a z-link to B, and dependency links are updated accordingly.

The same process is applied to D: D overlaps with the original rectangle of A, but no visible tile of it (those without any outgoing t-links) overlaps with the new rectangle, so the search can continue without cutting A. *b1* is the first and single visible tile that overlaps with it: it will be cut according to case M_1 to generate 3 new tiles, and the links will be updated like done previously.

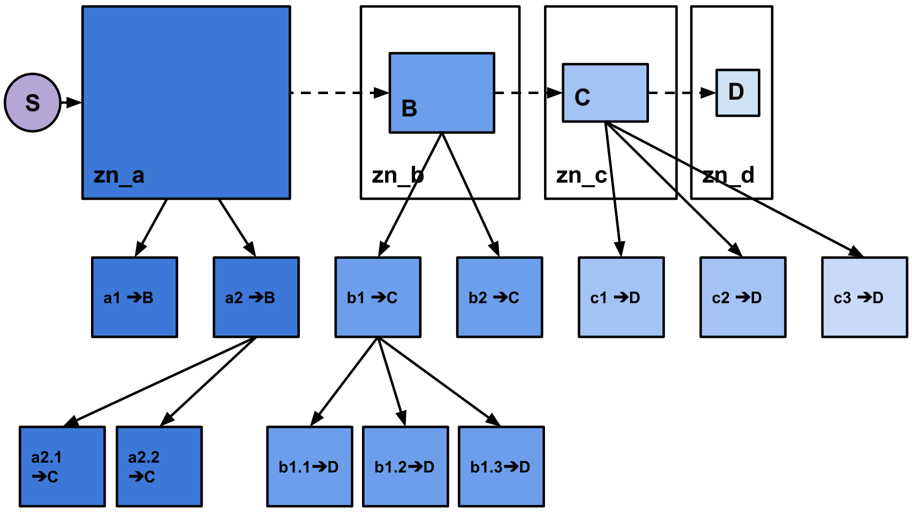


Figure 5.12: Graph built from the example scenario

The graph structure as described presents a few properties that simplify operating on it. When only z-links are considered, the graph behaves like a linked list of windows ordered from the bottom to the top of the screen. In order to find the set of tiles that are visible, the graph can be parsed to collect all the tiles that have no outgoing t-links. In the top right of fig. 5.11 the visible tiles of the example scenario are shown when assembled.

When a new window is inserted not at the top of the scenario, only part of the graph must be regenerated. All the t-links and t-nodes originating from z-nodes above the new window are invalidated, as well as all t-nodes with dep-links pointing to the z-nodes just invalidated. This is done to avoid degradation in the graph: if those tiles were kept, the set of visible rectangles would not be minimal, a few rectangles would be unnecessarily split into two (or more). This tradeoff is acceptable because of the specification: **compose** is performed way more often than **insert**, therefore the data structure is designed to optimize its execution rather than the execution of operations rarely needed. The same is valid for **remove**.

5.6 Compositing strategies

The operation **compose** returns the set of rectangles whose content must be redrawn to display the current frame correctly and efficiently. This can be achieved in multiple ways, which are defined drawing strategies. They can aim for different goals: save CPU time, GPU time, trade offs... Here three compositing strategies will be analyzed: full, tiled and dynamic bitblitting. *Full* bitblitting is the trivial strategy used by non-optimized compositors, see 4.2 and 4.4. The full strategy presented does not differ from those in the related work, it is here described only for comparison. Tiling compositing strategies are available in more sophisticated compositors which aim to minimize overdraw. The tiling strategy here presented is using the tiling algorithm presented in 5.4, which is more advanced compared to that analyzed in 4.1, because the number of tiles generated is minimal (not considering merging of tiles). *Dynamic* is a new approach proposed in this work, that also requires a *calibration* step to generate a prediction model able to estimate the time required by every bitblitting operation.

A further *caching* optimization that can be applied to all the compositing strategies, and especially to the *dynamic* one, is also in this section described.

5.6.1 Full compositing

The simplest **compose** drawing mode is *full compositing.* Starting from S as described in 5.5, every z-node will be checked: if marked, it will be added to the output set, together with all the windows above which overlap it (dependencies). This requires just following the z-links between z-nodes. The advantage of this strategy lies in the number of rectangles bitblitted, which is minimal, and in the speed of execution on the CPU: the complexity is just $O(N)$. The big drawback is that any area of the screen covered by more than one window will be drawn more than once, wasting time to draw pixels that will be immediately overdrawn because of another window on top.

```
compose_full(g, marks) {
  z = g.start
  while(z) {
    if(marks[z.id])
      bitblit_full(z)
    z = z.z_link_next
  }
}

bitblit_full(z) {
  BITBLIT(z.rect)
  mark_dependencies(z.id)
}
```

Listing 5.3: Full compositing

In the pseudocode in listing 5.3, `g` holds the graph structure. `z` is a z-node, which is checked for updates. If it was marked the rectangle contained in it will be bitlbitted with `BITBLT` and the dependencies marked.

5.6.2 Tiled compositing

In order to achieve zero overdraw *tiled compositing* should be used. To obtain the minimal set of visible rectangle can be done by traversing the z-linked list one entry at time, walking its tree composed of t-nodes and adding to the output set every tile which:

- belongs to a marked window
- has no outgoing t-links
- is not completely covered by another window

Considering how rectangles can overlap, inserting a new rectangle can generate at worst 5 new visible tiles. This happens when every corner of the rectangle lies on top of a different tile: every corner generates 1 additional tile (overlap of type M_2) plus the new rectangle. In this case there will be 5 visible tiles more than previously. This guarantees that the number of total tiles in the system will grow at most linearly with the number of windows, although with a theoretical maximum factor of 5.

By fragmenting the rectangles in many smaller tiles and selecting only the visible ones, *tiled compositing* reduces to the maximum extent overdraw.

```
compose_tiled(g, marks) {
  z = g.start
  while(z) {
    if(marks[z.id])
      bitblit_tiled(z)
    z = z.next
  }
}

bitblit_tiled(z) {
  for(t : z.tiles)
    if(t.visible)
      BITBLIT(t)
    else
      bitblit_tiled(t)
}
```

Listing 5.4: Tiled compositing

The tiling pseudocode in listing 5.4 describes the recursive approach used to draw the tiles of a marked rectangle. For every marked z-node, its t-nodes `t` are checked for visibility. Only visible tiles (without children nodes) must be bitblitted. In this case no dependencies must be marked since there is no overdraw.

5.6.3 Dynamic compositing

While tiled compositing provides a solution to achieve zero overdraw, the high number of tiles that could be generated even for simple scenarios motivated the development of a more efficient solution. The idea behind *dynamic compositing* is to choose at runtime what is more convenient between *full* and *tiled* compositing.

The *dynamic* strategy works by comparing the predicted bitblitting time for different combinations of rectangles: a calibration step is therefore necessary before running it, to find a prediction model able to estimate the time that a bitblitting operation requires given its rectangle size. The model is then accessed by the compositing algorithm during its execution.

With the assumption that bitblitting a rectangle requires a fixed time plus an amount of time proportional to the pixels to draw, in some cases bitblitting two big rectangles could be faster than bitblitting more rectangles which together cover a smaller area. In this way the prediction model generated in the calibration step can be used to find the set of rectangles that requires minimal time to be drawn. For every tile marked, the time required to bitblit its descendants (its tiles) is compared to the time required to bitblit the tile itself together with its dependencies. In this way, the most efficient method can be chosen at tile level, not just a window level, saving bitblitting time by choosing the rectangle set that minimizes the bitblitting effort.

```
compose_dynamic(g, marks) {
  z = g.start
  while(z) {
    if(marks[z.id])
      bitblit_dyn(z)
    z = z.next
  }
}

bitblit_dyn(z) {
  mode = best_mode(z)
  if(mode == "full") {
    BITBLT(z.rect)
    mark_dependencies(z.id)
  } else {
    for(t : z.tiles)
      bitblit_dyn(t)
  }
}

best_mode(n) {
  if(has_tiles(n) && time_for_full(n) > time_for_tiled(n))
    return "tiled"
  else
    return "full"
}

best_time(n) {
  return min(time_for_full(n), time_for_tiled(n))
}

time_for_full(n) {
  time_full = time_for_rectangle(n.rect)
  for(d : n.dependencies)
```

```
    time_full += best_time(d)

  return time_full
}


time_for_tiled(n) {
  time_tiled = 0
  if(n.visible)
    time_tiled += time_for_rectangle(n)
  else
    for(t : n.tiles)
      time_tiled += best_time(t)

  return time_tiled
}

time_for_rectangle(r) {
  return REFERENCE_MODEL(r.width, r.height)
}
```

Listing 5.5: Dynamic compositing

The pseudocode in listing 5.5 describes how dynamic compositing is executed. `compose_dyn` checks for the marked z-nodes. If a z-node is marked, the dynamic strategy is applied to it. The time required for the rectangle contained in z-node to be bitlitted in full or in tiled mode will be compared. For every tile in the t-nodes contained in the tree linked to the z-node, the same comparison is done recursively. This process is applied to every marked z-node, as well as to the z-nodes which are marked as dependencies if the full strategy was chosen for any rectangle in the previous steps of the algorithm.

When a tile with descendants is drawn fully (because of an efficiency choice), the window that generated its descendants has also to be redrawn, being it a dependency of the tile. The architecture in section 5.1 describes an asynchronous system, demanding that when the content of a window is redrawn from the off screen buffer the whole onscreen buffer must be updated. This implies that even if a window is only partially covered by another one, it is not possible to redraw the bottom window completely and the window on top partially. This could cause *tearing*, being the swapbuffers commands asynchronous with the compositor thread it would be possible that data originated in two different frames is displayed. For this reason, dependencies are always drawn completely and therefore considered as such in the dynamic time cost prediction.

5.6.4 Caching

The focus of *dynamic compositing* is to save bitblitting time by finding the optimal set of rectangles to draw the visible area of all windows that have been updated since the last frame. The analysis of a typical example case leads to think that a considerable amount of calculation is necessary to find the optimal set of rectangles to bitblit, requiring additional CPU time compared to simpler strategies.

A significant optimization can be devised observing that the compositing strategies behave like mathematical functions: given a set of window ids, the optimal output set of rectangles and tiles is returned to be bitblitted, and it is always the same. In other words the rectangle set to be found depends only on the marked windows as long as no insertion or removal has been made. It is possible then to store the resulting rectangles in a cache, using to the input marks as a key to find its position in the cache. The pseudocode in listing 5.6 describes exactly this process when applied to the dynamic strategy.

```
compose_dynamic_cached(g, marks) {
  key = HASH(marks)
  if(cache[key].match) {
    for(r : cache[key])
      BITBLIT(r)
  } else {
    cache_enable(key)
    graph_walk_dyn(g, marks)
    cache_disable()
  }
}
```

Listing 5.6: Cached dynamic compositing

While this optimization could be applied to any compositing strategy, it suits especially those which require a high number of steps to find the rectangle set to be bitblitted.

5.7 Compositing strategies applied to an example scenario

Fig. 5.13 shows a scenario with 4 windows that when inserted in the graph appear as represented in fig. 5.14. In the example since the rendering of the last frame two *mark* events have been received: on A and on B. The three compositing strategies are applied to the described scenario and compared.

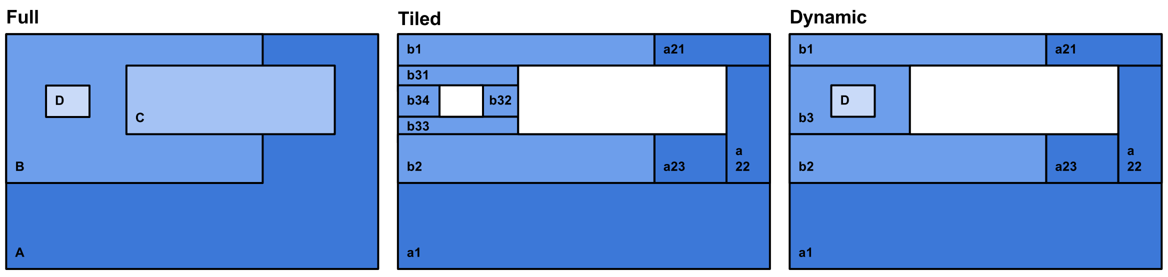


Figure 5.13: Compositing strategies applied to an example scenario

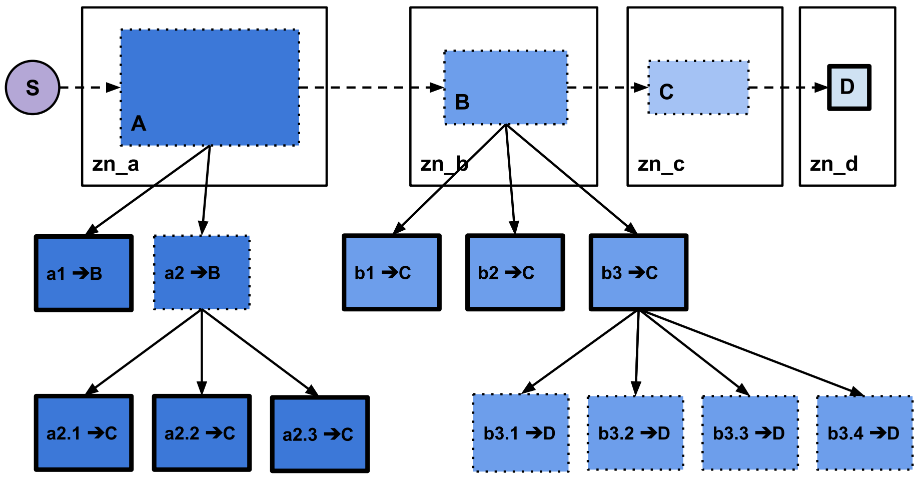


Figure 5.14: Graph generated from the scenario in fig. 5.13

Full compositing

The *full* strategy traverses the z-links bitblitting the marked rectangles and their dependencies, so it will start with A, then it will mark the dependencies of A: B (already marked) and C. Then B will be drawn, marking both C and D since they overlap it. When done with that, C and then finally D are bitblitted. In the the left side of fig.5.13 the rectangles that are bitblitted when the *full* strategy is used are represented.

Tiled compositing

The *tiled* strategy does not care about dependencies because only the visible part of the rectangles will be bitblitted, so since only A and B are marked, the tiles with no children belonging to the marked windows are bitblitted. In the example the drawn rectangles will be *a1*, *a21*, *a22*, *a23*, *b1*, *b2*, *b31*, *b32*, *b33*, *b34*. In the middle of fig.5.13 the rectangles that are selected by the *tiled* strategy to be bitblitted are shown.

Dynamic compositing

The *dynamic* strategy starts with A as well. First of all, its *full* bitblitting time must be calculated, t_A^{full}. Using the prediction model, the time necessary to draw A is composed by a value which is fixed for every rectangle, plus an amount proportional to its area. To this the time required to bitblit its dependencies must be added: B and C. C has no dependencies so it is just the bitblit time of the rectangle: $t_C^{full} = t_C^r$. t_B must not be included because it is already marked for bitblitting, so it won't affect this decision.

$$t_A^{full} = t_A^r + t_C^r$$

Now the *tiled* time is calculated: The tile *a1* has no children but *a2* does, so the algorithm must decide if it is more efficient to bitblit *a2* and the rectangle that splits it (C) along with its dependencies, or go deeper and bitblit the tiles only (*a21*, *a22*, *a23*).

$$\begin{aligned} t_{a1}^r + t_{a2}^r + t_C^{full=r} &\sim t_{a1}^r + t_{a21}^r + t_{a22}^r + t_{a23}^r \\ t_{a2}^r + t_C^r &\sim t_{a21}^r + t_{a22}^r + t_{a23}^r \end{aligned}$$

This decision must be taken using the prediction model: it depends on the size of the rectangles involved. For this example let's say that the prediction model indicated that bitblitting just the tiles would be faster. So:

$$t_A^{tiled} \sim t_{a1}^r + t_{a21}^r + t_{a22}^r + t_{a23}^r$$

Now this can be compared to the *full* bitblitting value from the previous calculation:

$$\begin{aligned} t_A^{tiled} &\sim t_A^{full} \\ t_{a1}^r + t_{a21}^r + t_{a22}^r + t_{a23}^r &\sim t_A^r + t_C^r \end{aligned}$$

This requires again the prediction model, that will finally decide which is the most efficient way to bitblit A. Let's say that it chose *tiled* bitblitting. Now that A is done, the algorithm proceeds in z-order checking the next marked window, which is B. The bitblit time of B is still unknown: its *full* bitblit time consists in $t_B^r + t_D^{r=full} + t_C^{r=full}$. t_c^r can not be left out because it was previously decided to bitblit A in tiled mode, hence not marking C. Now this must be compared with t_B^{tiled}, which at the first depth of the tiles tree means $t_{b1} + t_{b2} + t_{b3}$. Luckily

$t_{b1} = t^r_{b1}$ and $t_{b2} = t^r_{b2}$ but for t_{b3} a deeper research must be made. A comparison between t^{full}_{b3} and t^{tiled}_{b3} is then done and the reference model takes the decision:

$$\begin{aligned} t^{full}_{b3} &\sim t^{tiled}_{b3} \\ t^r_{b3} + t^r_D &\sim t^r_{b31} + t^r_{b32} + t^r_{b33} + t^r_{b34} \end{aligned}$$

Let's say that the model indicated t^{full}_{b3} as the cheapest option. The best combination to bitblit B in tiled mode would be *b1*, *b2*, *b3* and D. Now the algorithm goes back comparing *tiled* and *full* bitblitting for the whole B rectangle:

$$\begin{aligned} t^{full}_B &\sim t^{tiled}_B \\ t^r_B + t^{r=full}_C + t^{r=full}_D &\sim t^r_{b1} + t^r_{b2} + t^r_{b3} + t^r_D \\ t^r_B + t^{r=full}_C &\sim t^r_{b1} + t^r_{b2} + t^r_{b3} \end{aligned}$$

Again the algorithm turns to the prediction model, as the decision depends on the rectangle sizes. In this case, let's say it will opt for *tiled* bitblitting on B. The next rectangle in z-order is C but it is not marked, so it will be skipped. Then D shows up as marked because it was a dependency of *b3*, so it will be bitblitted: $t^{full}_D = t^r_D$.

The final set of rectangles is: *a1*, *a21*, *a22*, *a23*, *b1*, *b2*, *b3*, D which in fig. 5.14 are displayed with a wide black border. In the top right of fig. 5.13 the final bitblitting set is shown.

As seen in this example, some rectangles are checked multiple times and not marked (like C) and this could lead to some non optimal rectangle sets. For this reason when a rectangle is checked negatively and then checked again, the final set of rectangles is compared to the pure *full* bitblitting set and, in case the *full* strategy results more effective, it is adopted for the current frame.

$$\begin{aligned} t^{full} &\sim t^{dyn} \\ t^r_A + t^r_B + t^r_C + t^r_D &\sim t^r_{a1} + t^r_{a21} + t^r_{a22} + t^r_{a23} + t_{b1} + t_{b2} + t_{b3} + t^r_D \end{aligned}$$

5.8 Locking

The flexibility that the graph offers in handling window combinations is not only usable to boost the performance of bitblitting. A feature that can be introduced thanks to it is *rectangle locking*. The system model in chapter 3 described windows that for performance reasons required exclusive access to a dedicated region on screen.

Having the screen split in tiles permits to have granular access on the single regions of screen, even if they overlap with existing active windows. Inserting a special *locked* rectangle into the graph flags all the rectangles overlapping it to be drawn using the *tiled* strategy, therefore ensuring that the locked area is not written at all, no matter which windows are updated. An application that requires direct access to the screen can therefore insert a *locked* rectangle in the graph, so that *full bitblitting* will never used to boost the drawing performance in that region, hence guaranteeing that only the external application actually has access to that area of the screen and the content that it will draw wil not be overwritten by other applications.

6 Implementation

The chapter covers the implementation of a compositor featuring the data structures and algorithms presented in chapter 5.

The complete system architecture is presented, taking in consideration how the compositor interacts with the system libraries and processes. The API functions that the compositor exposes are then listed and their functionality analyzed. A few performance optimization developed to speedup the execution of the compositor's are then introduced.

6.1 Architecture

Fig. 6.1 displays the components of the system that interact with the compositor. The compositor itself is made of a *compositing process* and a *compositing library*. The library exposes an API that other components of the system use to communicate with the compositor.

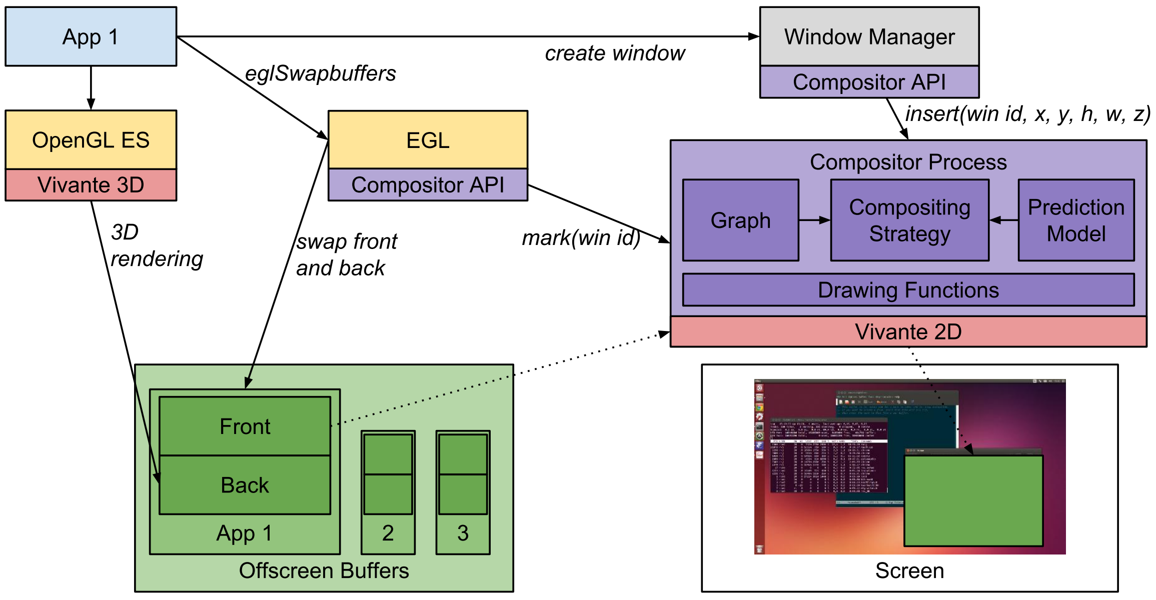


Figure 6.1: System Architecture

The compositor has been developed in C to achieve minimal latency and maximum compatibility with the rest of the system libraries. The hardware chosen to develop and test the compositor is a Freescale i.MX6 Quad Automotive embedded board, running the Linaro Linux distribution.

When an application creates a new visible window through the window manager API, the compositor is immediately informed with the window id, size and position of the new window and uses this information to update the internal graph structure accordingly.

The graphical applications in the system access through EGL and OpenGL the graphics hardware. 3D rendering commands are issued through the OpenGLES 2.0 API that the Vivante GPU supports through their proprietary drivers. Applications initialize a context allocating through EGL a surface in an offscreen buffer. EGL uses the double-buffering method, keeping always a consistent buffer that can be accessed and bitblitted. This is handled by EGL allocating two surfaces instead of one, front and back buffer, and alternating them: when a surface has been rendered completely the *eglSwapbuffers* command is called to swap the pointers to the two surfaces. In this way the application can keep issuing OpenGL commands to render on the backbuffer, while the front one can be accessed by the compositor.

The compositor runs in its own rendering process at a fixed (but configurable) framerate. A framerate of 60fps means, for example, that every 16.6ms the compositor will bitblit on screen the windows that have been updated since the last frame. When an application issues a *eglSwapbuffers* command, the EGL layer notifies the compositor with the id of the buffer that has just been updated using the compositor API. The compositor adds this id to the list of those which have been updated, *marking* the window. It will bitblit the marked windows when the next frame is scheduled, according to the configured framerate, clearing then all the marks to prepare for the next frame. The notifications from EGL are non-blocking: an application can start rendering the next frame while the compositor still updates the screen. The compositor's behavior can therefore be described as asynchronous with respect to the applications.

When the screen must be updated the compositor takes the list of marked windows ids and runs the chosen *compositing strategy* using the marked ids, walking through the *graph* data structure using the *prediction model* generated beforehand to find the set of rectangles that minimizes the bitblitting time. The output set of rectangles will be then bitblitted on screen using the 2D graphics library of the system vendor.

Before running the compositor a *calibration* step is required, necessary to find and store the *prediction model* used at runtime.

6.2 Interface

The graph data structure used to store the windows represents them with rectangles. The definition follows what described in section 5.3, requiring the tuple (x, y, w, h, z) to place a rectangle on the screen but tracking also the window id, necessary to fetch the actual struct of a window to locate its framebuffer. The graph stores in fact only metadata.

```
typedef struct rect {
  uint32_t x, y, w, h, z;
  ...
  w_id id;
} rect_t;
```

The compositor is accessed using API functions. The functions are divided in three groups: functions implementing the *basic operations* described in section 5.2, additional function which improve the API usability and implement the functionality required by a compositor for automotive systems and *compositing* functions implementing the compositing strategies described in section 5.6 and

6.2.1 Graph Operations API

Insert

```
void graph_insert(graph_t * g, rect_t * r)
void graph_insert_alpha(graph_t * g, rect_t * r)
```

graph_insert implements the **insert** operation. Given a graph and a rectangle as input, it inserts the rectangle at the correct location in the graph, generates its tiles and updates all the tiles of the rectangles inserted previously. Inserting a new rectangle invalidates the tiles belonging to the rectangles above the new one, while those below must only be updated slicing the visible tiles that overlap with the newly inserted rectangle. Therefore creating a new window on top of all others (a common case) results in a faster operation compared to inserting anything at the bottom/in the middle, as shown also in the example in section 5.7. The tiles store an additional flag which is set at insertion time that states if the tile, even if without children, is overlapped by other rectangles and therefore is not visible. This case happens when a newly inserted rectangle completely overlaps a previously visible tile. This is done at insertion time to avoid unnecessary overhead when running the compositing functions.

graph_insert_alpha inserts the rectangle into the graph like in a normal insertion, taking also care of the backwards dependencies caused by transparency. It should be noted that the input is exactly the same as in `graph_insert`, since the graph needs only to know that the rectangle must be handled as a transparent one and has no actual knowledge on the content itself.

Inserting transparent rectangles in the graph requires additional effort: transparent rectangles have as dependency not only the rectangles above them, but also those below. When a transparent rectangle is marked to be updated, the rectangles below it must first be drawn, followed by the transparent rectangle and then by the dependencies, if any. Therefore inserting a transparent rectangle could alter the graph above and below it: new conditions are added to the rectangles overlapping the new one (recursively) to trigger the redraw of the rectangle when the transparent one is marked.

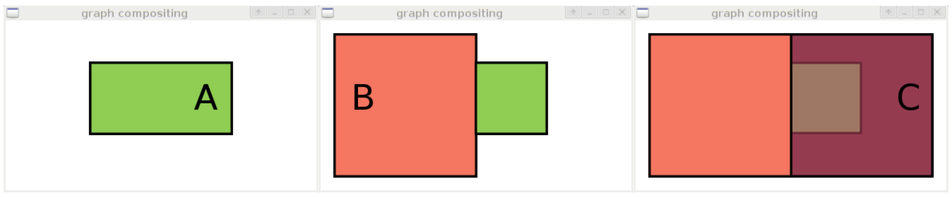


Figure 6.2: Tiled transparency example

If a rectangle is covered by a transparent one, when either of them is updated only the visible area of the rectangle on bottom will be redrawn. An example is shown in fig. 6.2: when rectangle A is covered by the partially transparent rectangle C, only the part of A that is still visible must be bitblitted. This saves significant bitblitting time, since that rectangle must be drawn every time it or the transparent rectangle above it are marked. Any number of transparent windows is supported in this way. A full scenario with multiple transparent windows is shown in fig. 6.3 when rendered with the *tiled* strategy. The tiles are drawn with black borders, and it can be seen that there are tiles generated and displayed also for windows which are not on top.

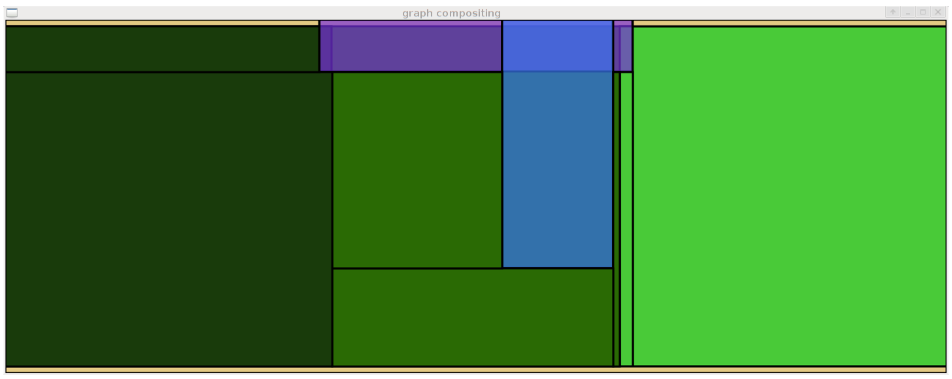


Figure 6.3: Transparent rectangles rendered with tiled strategy

System wise, the support of transparency depends on the framebuffer configuration and the drawing API. Typically this is done in two ways: using an alpha channel or defining a transparent color [24]. The former solution allows partial transparency, making possible to superimpose images on top of one another, while with the latter is mostly useful to draw non-rectangular windows.

Remove

`void graph_remove(graph_t * g, w_id id)`

graph_remove takes a rectangle and all references of it out of the graph, and triggers a rebuild of the tiles belonging to rectangles above it. In this function the garbage collector is involved: given a window id, the garbage collector can free all the tiles of that window without walking through the graph.

Modify

`void graph_modify(graph_t * g, rect_t * r)`

graph_modify inspects the window id contained in the rectangle and applies the modifications requested, updating the graph is the changes require it. Any modification is allowed, also those changing the z-value.

Mark

`void graph_mark(graph_t * g, w_id id) void graph_mark_all(graph_t * g)`

graph_mark is called to notify that the a window performed a *eglSwapbuffers* command and its content must be updated on screen. Marks are stored in an indexed array structure, not directly in the graph for performance reasons.

graph_mark_all is a helper function that marks every window in the graph. It is provided because with a single function call it is possible to trigger a full screen redraw, which is an operation often needed.

6.2.2 Graph Additional API

`graph_t * graph_init()`

graph_init allocates the buffers necessary for the struct `graph_t`. All buffers can dynamically grow or shrink depending on the number of windows in the system. For the graph data structure a custom garbage collector (GC) is used to speedup the *free* operations, so in this function the GC is also initialized and assigned to the `graph_t` that is returned to the caller.

The graph structure requires a considerable amount of memory allocations when tiles are generated, even more when windows are inserted with transparency enabled. While the overall space requirements are not heavy since the content of windows is not duplicated, when a window must be removed from the graph its sub-graph and the sub-graph belonging to the windows above it must be freed completely, and in case of a scenario including many windows this would not be a trivial operation.

For this reason all the *t-nodes* belonging to a *z-node* and the buffers stored within them are allocated through the garbage collector module. The module requires a window id and size in bytes as input parameters to return a pointer to a memory buffer of the requested size.

The garbage collector allocates the memory and stores it in dynamically growing arrays. When a window is removed, its tiles and the tiles of the windows above it are freed just looping through their arrays, indexed by the windows ids. After that, the tiles that have been generated by such windows and are stored in t-nodes belonging to windows below them have to be removed from the graph too: the garbage collector won't be activated yet on those windows ids, a counter will be incremented instead. When the counter reaches the configured value, a *graph rebuild* is triggered: all the buffers stored in the garbage collector are freed to claim the memory currently not in use, then all the tiles are regenerated from scratch.

To preserve consistency in the links between a removal event and the next graph rebuild, all the dependency links between different windows track only the window id and do not point directly to the z-nodes: this permits to use efficiently the garbage collector without having to parse the graph to remove dangling pointers.

In the garbage collector internals, the buffers used to store allocated pointers grow dynamically doubling in size when they are full and receive a new pointer. On the other hand removing a window does not shrink immediately the buffer because there is a high chance that the window will be reinserted (like in case of a graph rebuild), therefore storing a similar amount of pointers.

```
void graph_raise(graph_t * g, w_id id)
```

While *raise* is not a fundamental operation required by the graph, but it is commonly executed by window managers so a helper for it is provided. The operation brings the window with the specified id to the front changing its z-value. *graph_raise* creates a temporary copy of the rectangle with the given id, finds the topmost one in the graph and performs an insertion of the saved rectangle applying $z_{new} = z_{min} - 1$.

```
void graph_animate(graph_t * g, rect_t * r, uint32_t ms)
```

Animations are a feature not directly related to the data structure but it is implemented with a dedicated helper for performance reasons. In this way the window manager does not have to execute *modify* events to manually animate a rectangle, but rather rely on the compositor's animation API. *graph_animate* sets all the variables for the execution of an animation. It does not actually animate the rectangle, since this is done frame by frame in the *compose* step. The graph is searched for a rectangle with the same id of the input rectangle, and this is used as *source rectangle.* The function parameter `r` is used as *destination rectangle.* The other input is a value in milliseconds that defines the animation duration. Animation on the z-axis are not allowed because it would not be possible to have transition effects on them, so it is necessary to fall back to *graph_modify* to change the z-value of a window.

In the function body the boundary rectangle of the animation is calculated, which is the smallest rectangle containing both source and destination. The pseudocode in listing 6.1 shows how the boundary parameters (`b.x`, `b.y`, `b.w`, `b.h`) are calculated using `src` (source) and `dst` (destination) rectangles.

```
b.x = min(src.x, dst.x);
b.y = min(src.y, dst.y);
b.w = max(src.x + src.w, dst.x + dst.w) - b.x;
b.h = max(src.y + src.h, dst.y + dst.h) - b.y;
```

Listing 6.1: Formulas for animation boundaries

All the rectangles overlapping the boundary one are set to be redrawn every frame until the animation is finished, to prevent *motion trail effects*, which happen when a window is moved across the screen but the background is not redrawn, therefore leaving a trail where the window was drawn in the previous frames. A timer is started that will be used in the *compose* functions to animate the window. Multiple animations are also supported, tracking their start-end timestamps individually.

```
void graph_lock(graph_t * g, w_id id)
```

graph_lock locks a rectangle like described in section 5.8, forcing tiling for all the rectangles overlapping it. *Locking* interacts with animations by checking their boundary box first: when the locked rectangle does not overlap the animation, they do not affect each other. If a rectangle happens to be locked when an animation is ongoing and it does overlap with the boundary box, the locked rectangle has priority, since the functionality is meant to guarantee no other application can draw on the rectangles belonging to external applications. The chosen behavior is to terminate the animation by setting the animated rectangle immediately to its final shape and location.

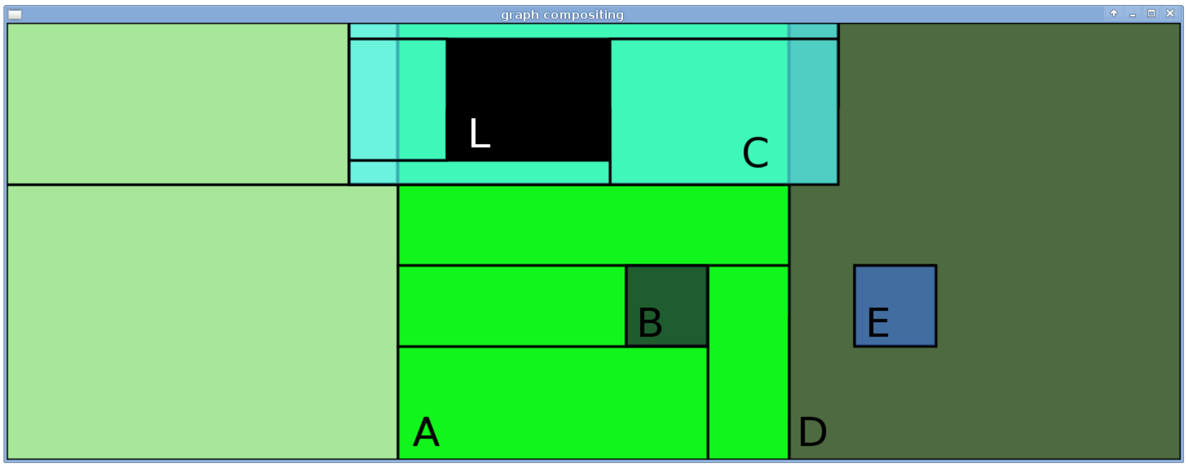


Figure 6.4: A locked rectangle (the small black one) forces tiled bitblitting around it

An example is shown in fig. 6.4: the locked area is the black one in the top of the screen marked with L: nothing is shown there since the area is marked as reserved. This forces all the rectangles overlapping it to be bitblitted in *tiled* mode, like rectangle A for example: A is bitblitted using 4 tiles around B, and 4 additional ones to provide the correct background around the locked rectangle, since there is a transparent rectangle (C) between A and the locked one. On the other hand, D does not overlap with the locked rectangle. For this reason an optimization is applied according to the *dynamic* strategy: both D and E are drawn in *full* mode.

```
void graph_unlock(graph_t * g, w_id id)
```

graph_unlock reverts a lock operation, resuming with the normal *compositing* behavior and displaying its background again if transparency on it was enabled.

```
void graph_reset(graph_t * g)
```

graph_reset clears all the marks applied to the windows in the graph. This is typically called after a *compose* function completed its execution.

```
void graph_free(graph_t * g)
```

graph_free activates the garbage collector to free the graph structure completely and then manually frees the remaining buffers.

6.2.3 Graph Compositing API

The graph API offers 4 *compositing* functions to output the rectangles that have to be bitblitted on screen: *full*, *tiled*, *dynamic*, *dynamic cached.* The behavior of the compositing functions follows what defined in 5.6, adding a 4th option which consists in the caching optimization of the dynamic strategy, described in 5.6.4. They all require a function pointer as input compatible with the following prototype:

```
void (*fun)(rect_t * r, void * v);
```

and a generic void pointer for custom usage. The function passed will be called internally once for every rectangle (or tile) that must be drawn, setting the input parameters to the rectangle to bitblit (including the window id to find the source surface) and the user supplied pointer v. This solution permits to store in the pointer v all the necessary parameters to perform the bitblit. They all return the predicted time required to bitblit the output rectangle set, so it can be compared with the actual time required.

The compositing API functions are also responsible to update the animations. Changes to size or position of the windows can be applied immediately or over time. Changes that have to be finalized immediately are considered *modify* operations. When the modification requires a transition effect it is handled as *animation*. Animation can affect size, position or both at the same time.

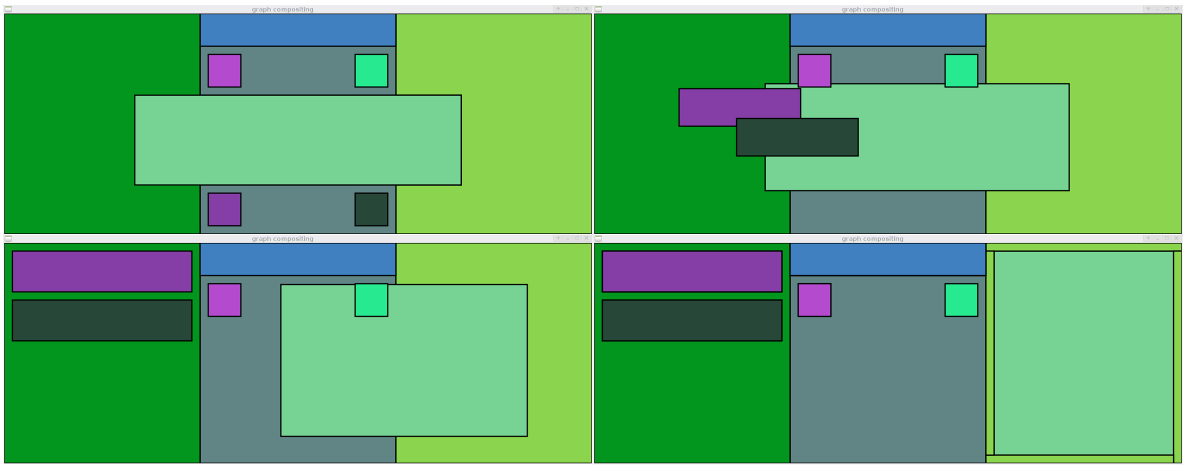


Figure 6.5: Multiple simultaneous animations

The animations are implemented as a transitions meant to be completed within the milliseconds given as a parameter when `graph_animate` is called. The temporary shape and position of the rectangle are set using the formulas in listing 6.2. `r` represents the rectangle shown in the current frame of the animation, `src` and `dest` source and destination rectangles respectively. `elapsed` and `total` represent elapsed time since the start and total animation time.

```
r.x = src.x + (dst.x - src.x) * elapsed/total;
r.y = src.x + (dst.y - src.y) * elapsed/total;
r.w = src.w + (dst.w - src.w) * elapsed/total;
r.h = src.h + (dst.h - src.h) * elapsed/total;
```

Listing 6.2: Formulas for animation transitions

In this way animations are displayed correctly even if the framerate is variable. Fig. 6.5 shows 4 screenshots of a few animations running concurrently: the big window in the middle of the screen is moved to the right to cover the right part of the screen almost completely, while the two small icons on the bottom are moved to the top left, increasing their width and height.

Due to the fixed nature of the graph structure, it is clearly not possible to update it on every frame reflecting the current status of translation and resize effects. The correct visual representation of the graph is then obtained temporarily removing the window that has be animated and dividing the other windows in two groups: those above and those below the animated window. In the next frame all the windows in the group *below* are drawn according to the graph rules. Then, the animated window is drawn calculating position and size for the first frame of the animation (using the elapsed time between the last frame and the current one). After that, all the windows *above* it can be drawn. This approach permits to optimize the bitblitting using the dynamic strategy while supporting animated windows. Transparency is handled in the same way: since the windows are bitblitted in order, the background content of transparent windows is also updated correctly at time of bitblitting the transparent window. As mentioned before, animation take as input a new size or new coordinates to place the window at, and a time in milliseconds to complete the animation within. The correct animation speed is then achieved also at variable framerates, by tracking the elapsed time between frames.

Full compositing

```
float graph_compose_full(graph_t * g, void (*fun)(rect_t * r, void * v), void * v)
```

graph_compose_full implements the full compositing strategy described in section 5.6.1. When a window is updated, the rectangle representing it is bitblitted completely together with the rectangles above it (listed as dependencies). The strategy matches what analyzed in section 4.1 of related work.

Tiled compositing

```
float graph_compose_tiled(graph_t * g, void (*fun)(rect_t * r, void * v), void *
v)
```

graph_compose_tiled implements the tiled compositing strategy described in section 5.6.2. The strategy completely avoids overdraw by bitblitting only the visible tiles of the updated windows. The amount of bitblitting commands executed is on average higher than what required by the *full* strategy, since windows are split into smaller tiles that have to be bitblitted individually. The implemented version of tiled compositing appears more optimized compared to what analyzed in section 4.1 of related work. Taking advantage of the graph structure, the implementation is able to reduce the number of tiles generated for overlapping rectangles. Moreover, rectangles not overlapping with any other one are not split, in contrast to what is done in the algorithm described in section 4.1.

Dynamic Compositing

```
float graph_compose_dyn(graph_t * g, void (*fun)(rect_t * r, void * v), void * v)
```

graph_compose_dyn implements the dynamic compositing strategy described in section 5.6.3. It aims to minimize the bitblitting time, deciding for every tile if it would be better to bitblit it completely along with its dependencies, or just bitblit the tiles that together make for the visible part of it. Using the values obtained in the calibration phase, it is possible to find a rectangle set whose bitblitting time is always at least as good as the best option between *tiled* and *full* bitblitting, and in case of not trivially simple scenarios, better than both of them.

The output of the calibration phase is a function in the form

$$t = f(width, height)$$

that returns the predicted time required to bitblit a rectangle of size $width \cdot height$ pixels. An example function would be:

$$t = a \cdot width + b \cdot height + c \cdot width \cdot height + d$$

with a, b, c and d are values found with the calibration process. The graph API function `graph_compose_dyn` walks through the graph comparing *tiled* with *full* costs for every rectangle/tile marked following the pseudocode shown in listing 5.5.

Cached dynamic compositing

```
float graph_compose_dyn_cached(graph_t * g, void (*fun)(rect_t * r, void * v), void
* v)
```

In section 5.6.4 an optimization is described which involves a cache to reduce the impact on the CPU that the computations of the output rectangle set could have in case of complex window scenarios. The cache must be structured so that every combination of window identifiers is mapped to a single output set (if any). In the current implementation the cache is used to improve the performance of `graph_compose_dyn`.

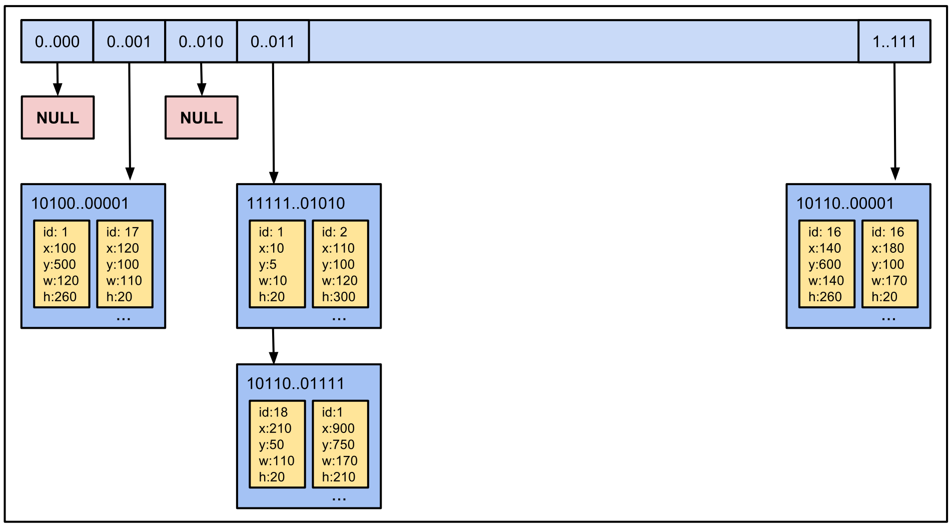

0..000
0..001
0..010
0..011
1..111
NULL
NULL
10100..00001
id: 1 x:100 y:500 w:120 h:260
id: 17 x:120 y:100 w:110 h:20
...
11111..01010
id: 1 x:10 y:5 w:10 h:20
id: 2 x:110 y:100 w:120 h:300
...
10110..00001
id: 16 x:140 y:600 w:140 h:260
id: 16 x:180 y:100 w:170 h:20
...
10110..01111
id:18 x:210 y:50 w:110 h:20
id:1 x:900 y:750 w:170 h:210
...

Figure 6.6: Rectangle set cache

The cache is indexed in a hash-like way to minimize access time. The ids of the windows marked are stored as a bitmap in a 64 bit unsigned integer. This value modulo 16 is then used to index an array of 2^{16} pointers, shown at the top of fig. 6.6. The pointers are null unless in a previous frame an input set matching that index has been bitblitted already. The linked lists are used to avoid collisions: the structs pointed by the array carry the additional

(64-16=48) bits to find the correct set. At configuration time the system can be set to enable cache support for more than 64 windows too, and everything will scale accordingly.

When `graph_compose_dyn_cached` is called, first the cache will be searched for a match of the current marked windows. If no match is found `graph_compose_dyn` will be called like normally but this time storing the rectangles in the cache as well as bitblitting them. In case of positive match the optimal set of rectangles is already stored in the cache, so it is possible to just loop through the array containing the rectangle pointers and bitblit them. In this way a considerable amount of CPU time is saved, since the same input set of marks will never be processed twice.

The current design of the cache requires its invalidation when a window is inserted, removed or the graph structure changes.

6.3 Drawing functions

The compositing API described in section 6.2.3 does not perform the bitblitting directly, but rather only the output set of rectangles is calculated and then passed to a callback. The compositor implements the bitblitting in the callback function using the Vivante 2D API to access the 2D graphics functionality of the GPU. The library operates on framebuffers devices, bitblitting the content of those which are offscreen and contain the graphical rendered applications into the on screen ones. The framebuffers devices appear in Linux as character devices in the `/dev/` directory. A setup with two attached displayshas them typically has them accessible as `/dev/fb0` and `/dev/fb1`. The offscreen framebuffers are initialized by a custom driver that allocates video memory and creates as many virtual framebuffer devices as needed (`/dev/fb2`, `/dev/fb3`..) which are not directly shown on screen. The compositor keeps internally a mapping of window ids to offscreen framebuffers, so when a callback is called with a rectangle to be bitblitted and the corresponding window id, it is possible to find the correct offscreen buffer and to issue a bitblitting command with that framebuffer as source and thescreen framebuffer as destination. The framebuffer name is provided by EGL when `eglCreateWindowSurface` is called with the window id generated by the window manager.

```
void bitblit(rect, data) {

  app_fb = get_framebuffer_from_window_id(rect.wid)
  set_source_fb(app_fb)
  set_source_rectangle(rect)

  screen_fb = get_screen_framebuffer(rect)
  set_dest_fb(screen_fb)
  set_dest_rectangle(rect)

  bitblit_copy_over()
}
```

Listing 6.3: Drawing functions pseudocode

The pseudocode in listing 6.3 describes how the callbacks are implemented. The 2D API requires a source and a destination framebuffer to be set, as well as the rectangles defining the regions of the framebuffers that have to be read and written. A bitblitting command for the operation *copy over* is then executed that takes also into account the alpha channel of the framebuffers to blend sources with partial transparency correctly on the destination.

7 Evaluation

To correctly evaluate the compositor, first the hardware configuration has to be calibrated to provide suitable prediction values to be used in the compositing algorithms. Then the tests to verify the correctness of the *dynamic* strategy are presented, which compare its efficiency with *full* and *tiled* compositing using only the predicted time values. The same set of tests is then run actually measuring the time required for bitblitting, and then the predicted time values are compared with the measured ones. In the next set of tests, the CPU time required to run each strategy is measured. Then the time window where collisions on the GPU bus can occur is measured and the impact of different compositing strategies on it is compared. The last evaluation test concerning the compositing strategies considers the relationship between the strategy choice and the frame rate of the graphical applications bitblitted. The functionality of transparency and animations is then tested, analyzing the time requirements for insertion and then removal of a transparent/animated window respectively. The final test involved an realistic automotive scenario using OpenGL applications, receiving windows updates directly from EGL.

7.1 Hardware/software configuration

The implementation platform is a Freescale i.MX6 Quad Automotive embedded board, which is focused on low power consumption although featuring four 1.2 GHz ARM Cortex A9 cores and 2GB of ram, as well as both 3D and 2D GPUs for graphics acceleration [34]. The board provides hardware graphic acceleration: a *3D Graphics Processing Unit*, a *2D Graphics Processing Unit]* and a *Vector Graphic Processing Unit*.

The performance of the 2D and 3D GPUs is in focus, as their interaction is especially relevant. The 3D GPU is a Vivante GC2000, capable of processing 200M triangles/s and 1000M pixels/s. The API offered by the hardware vendor to access the GPU is OpenGL ES 2.0, which is the OpenGL version optimized for embedded systems [35]. The 2D GPU is a Vivante GC320, which can process 600M pixels/s through a proprietary *2D API* that can be used for bitblitting [9]. The board is connected to a HDMI display to check that the graphics commands are effectively executed and completed. The board supports natively HDMI1.4, LVDS, MIPI and parallel displays. According to the board data sheet [36], the 2D and 3D graphic accelerators are accessed through the same bus. This implies that while the single GPUs handle independently 2D and 3D operations, the commands issued to them will compete for the same bus bandwidth. Therefore using the 2D and 3D GPU at the same time in a test can influence the results due to the bandwidth restriction, and it is accordingly considered in the evaluation.

The SDK provided by the hardware vendor includes two runtime environments, both Linux variants: A Linaro Ubuntu 10.10 system [37] for development/testing and a lightweight distribution called LTIB [38] which is meant as final deployment target. The default kernel, graphics drivers and user space graphics libraries do not differ between the platforms. To ease the development of the tests, Linaro was chosen. The latest Linux kernel available from Freescale is 3.0.35-4.1.0_130816, already patched with the graphics drivers for the GPUs. The user space graphic libraries are distributed only as binaries, no source code is available, so the latest build (released on the 16.09.2013) was chosen.

The tests to evaluate the compositor have been developed in C, therefore a function of the standard library was chosen to measure the execution times. The POSIX function `clock_gettime` offers access on Linux systems to a monotonic clock. Running `clock_getres(CLOCK_MONOTONIC)` on the board returns a resolution of $1ns$. The advantage of this approach is that it can be plugged in any test, inside and/or outside loops to get the raw results or an average value obtained on multiple runs. The accuracy offered by *clock_gettime* was good enough to evaluate the results of the tests with the desired precision.

7.1.1 Tools

The compositor was developed to be compiled into a Linux shared object for easy linking with the rest of the target system designed to run on an embedded board. This allows other programming environments to pick it up for testing and evaluation purposes. As development and debugging platform a gtk client was developed to directly test and display the results of the graph compositing. Figures 6.2, 6.3 and 6.5 have been generated with this tool.

```
Usage: ./compositor [options]
   -s, --scenario=file load a scenario file
   -r, --random    generate a random scenario
   -n, --num=val   number of windows to generate randomly (default 10)
   -e, --events=file load an events file
   -a, --markall   mark all the windows every frame
   -m, --mark=val  random windows to mark every frame (default 2)
   -w, --width=val screen width
   -h, --height=val screen height
```

The gtk compositor client can read fixed scenarios from a file as well as generate random ones. Since it is meant as a debugging tool, windows are just paint with RGBA colors and do not reflect the content of real applications. It can also read timed events from a configuration file to simulate user input like window insertion, removal, modification, animation, locking and unlocking at timed intervals. In this way it is possible to execute specific behaviors in a controlled and reproducible way. The events can be specified in the following way:

```
add <id>            - add a window
ada <id>,a          - add a window with alpha channel (float 0..1)
all                 - add all windows in the scenario
rem <id>            - remove a window
rai <id>            - raise a window
mrk <id>            - mark a window
```

```
mod <id>,x,y,w,h,z  - modify a window. "-" if unchanged
ani <id>,x,y,w,h,ms - animate a window. "-" if unchanged
lck <id>            - lock a window
unl <id>            - unlock a window
```

and an event file looks like:

```
# time(s) action parameters
1        add    3
2        add    5
3        ada    7,0.7
3        ani    7,760,0,170,380,1000
5        ani    3,1040,-,-,-,1500
5        ani    5,400,-,640,-,1500
5        lck    7
5        add    6
6        ani    6,540,50,20,20,1000
```

7.2 Calibration

The following set of tests aims to provide the data necessary to generate a calibration model approximating the expected duration of rectangular bitblits for a wide range of rectangle sizes. The calibration model is used in the *dynamic compositing* strategy to predict which are the most efficient rectangles to bitblit. Different states of GPU load are taken also into account, to validate the generated model in the presence of additional GPU intensive applications.

7.2.1 Bitblitting time without any additional load

The purpose of this test is to measure the time required to execute bitblitting operations for a wide range of rectangle sizes. Using the hardware/software setup described in section 7.4, a test was built to run bitblit commands on all the rectangle sizes between 1x1 and 512x512 pixels. In this first test no other graphical application is concurrently running to avoid contention of the GPU bus. After every command, a *commit* command is also issued to have the GPU command pipeline flushed and executed.

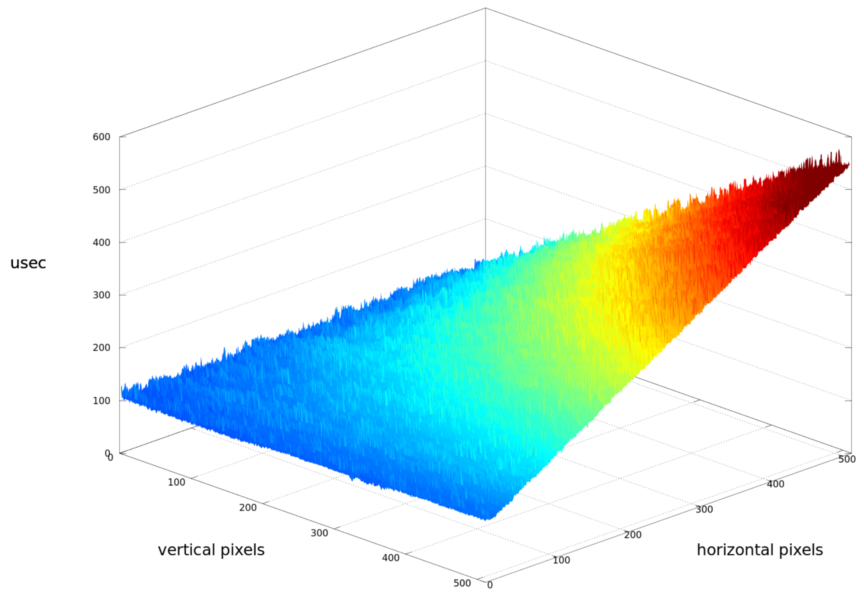


Figure 7.1: Bitblitting time of rectangles from 1x1 to 512x512

To minimize the error introduced by unpredictable context switches by the CPU each bitblitting command is repeated 100 times and then mean and standard deviation of the samples are calculated. Since the time measurements are done in user space, interrupts from the CPU scheduler could affect the results stopping the process while the timer is active. This is merely a limitation of the testing procedure and therefore should not affect the interpretation of the results. For this reason, mean and standard deviation of the samples are used to calculate the cumulative distribution function (cdf) of the normal distribution describing the samples. A threshold of 0.95 is then applied to the cumulative distribution function to filter the values significantly higher than the mean value obtained in this first test. The resulting samples are used to calculate the mean value used as final result.

According to general knowledge about GPUs and their buses, a linear time growth for bigger rectangle sizes is expected. The average measured time in microseconds for rectangle sizes between 1x1 and 512x512 is displayed in the plot in fig. 7.1. A fixed overhead slightly above 100μs can be noted, as well as a growth proportional to the rectangle size.

7.2.2 Prediction model

The data generated in the test of section 7.2.1 represents the average measured time that a bitblit operation of the given rectangle dimensions takes on the hardware/software setup. The data was then processed to make it easily available as a *prediction model* to the compositing algorithms, since shipping the whole dataset with the algorithm is not optimal for size reasons.

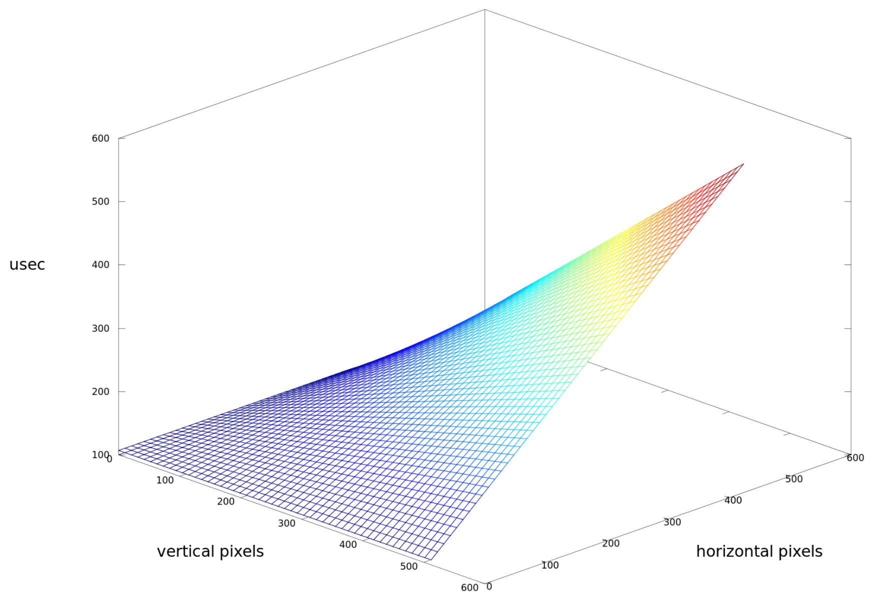


Figure 7.2: Linear interpolation of the measured bitblitting times

Linear

The data has been manipulated with gnu octave to generate an approximating model. The first approach chosen to extract such a model is *bilinear least squares fitting*, and led to equation 7.1. In the equation, t is the time in μs required to bitblit a rectangle of x width and y height, in pixels.

$$t = 1.0676 \times 10^2 - 1.1223 \times 10^{-3}x + 2.1861 \times 10^{-3}y + 1.7267 \times 10^{-3}xy \tag{7.1}$$

In fig. 7.2 the interpolating function is plotted for rectangles sizes from 1x1 to 512x512.

Cubic

The same data processing done to generate the linear model is then executed using *bicubic least squares fitting*, leading to equation 7.2. In the equation, t is the time in μs required to bitblit a rectangle of x width and y height, in pixels.

$$\begin{aligned} t = {} & 1.0813 \times 10^2 - 2.6170 \times 10^{-2}x - 4.5436 \times 10^{-3}y + 1.6784 \times 10^{-3}xy \\ & + 1.4019 \times 10^{-4}xx + 4.7150 \times 10^{-5}yy + 8.8723 \times 10^{-8}xxy \\ & + 5.6186 \times 10^{-9}xyy - 1.9715 \times 10^{-7}xxx + 5.7373 \times 10^{-8}yyy \end{aligned} \tag{7.2}$$

Evaluation

The two approaches were then compared subtracting the value yielded by the model to the actual value of the measurement. Both approached showed an average error below 2%, and were therefore considered both equally good to predict the bitblitting time. Since the linear model requires less CPU instructions to calculate the time required to bitblit a rectangle of given dimensions, it was chosen as reference model for the next tests.

7.2.3 Blitting times with additional GPU load

The test aims to demonstrate that the assumption taken on the bitblitting time is also valid in presence of higher GPU load. The bitblitting test is run with the exact same parameters as the test run in section 7.2.1 but with a different system status: other applications employ the GPU (specifically the 3D GPU) at the same time as the bitblitting test, simulating a real scenario where many 3D intensive applications are issuing commands while the windows on screen are being composed together.

Rectangles of size between 1x1 and 512x512 pixels are bitblitted while 2 instances of es2gears, a 3D benchmark, are executed at a high resolution (1024x1024), replicating a "worst case" GPU load. The expected behavior is that the programs will compete for bandwidth and the average bitblit times will grow.

In order to clearly see how the additional GPU load affects the measurement, part of the obtained results is first shown in fig.7.3. The plot depicts the average times for bitblitting a rectangle of a fixed height of 200px and variable width (1-512px). It is possible to notice that having 3d applications running will slow down the bitblitting unit. In green (top line) the average measurements with GPU load, in blue those without any load (bottom line). The results show an average overhead of approximately 108μs.

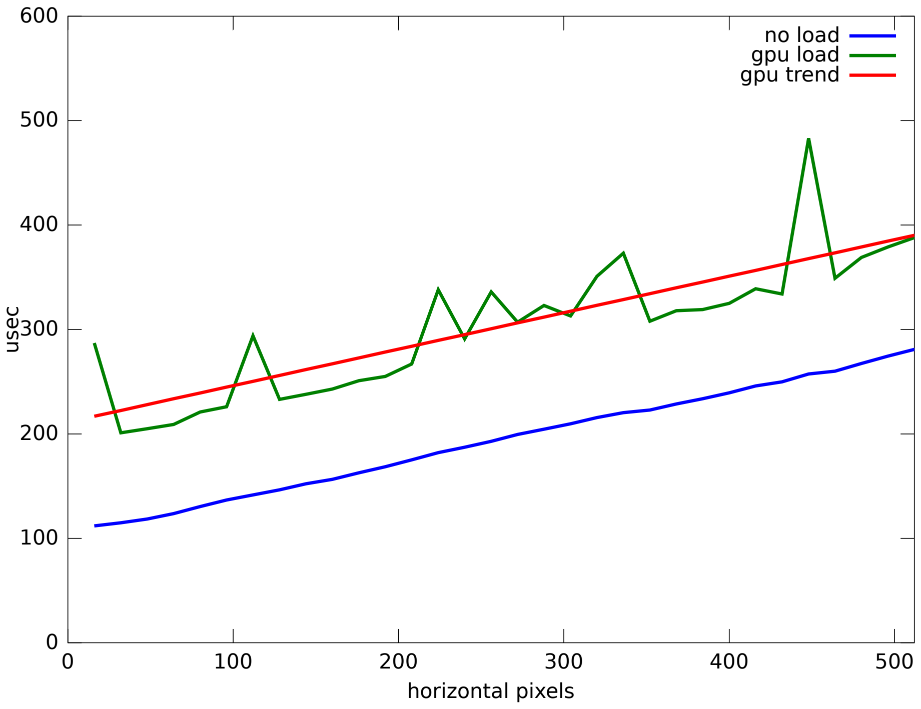


Figure 7.3: Comparison of bitblit times on a fixed height (200px) and variable width (1-512px) rectangle: green with additional GPU load and blue without

Comparing the measured times with additional GPU load and the reference linear model previously generated, it can be seen that the bitblitting performance is indeed affected by the 3D applications running concurrently, although not invalidating the general assumption of a linear growth proportional with the rectangle size. The difference between the measured values under load and the linear model previously generated is calculated on the whole data set (up to 512x512) and plotted in the histogram in fig. 7.4. The histogram confirms the values obtained in the previous test, showing overhead times mostly between 50 and 500μs. The average overhead on the whole dataset was approximately 110μs.

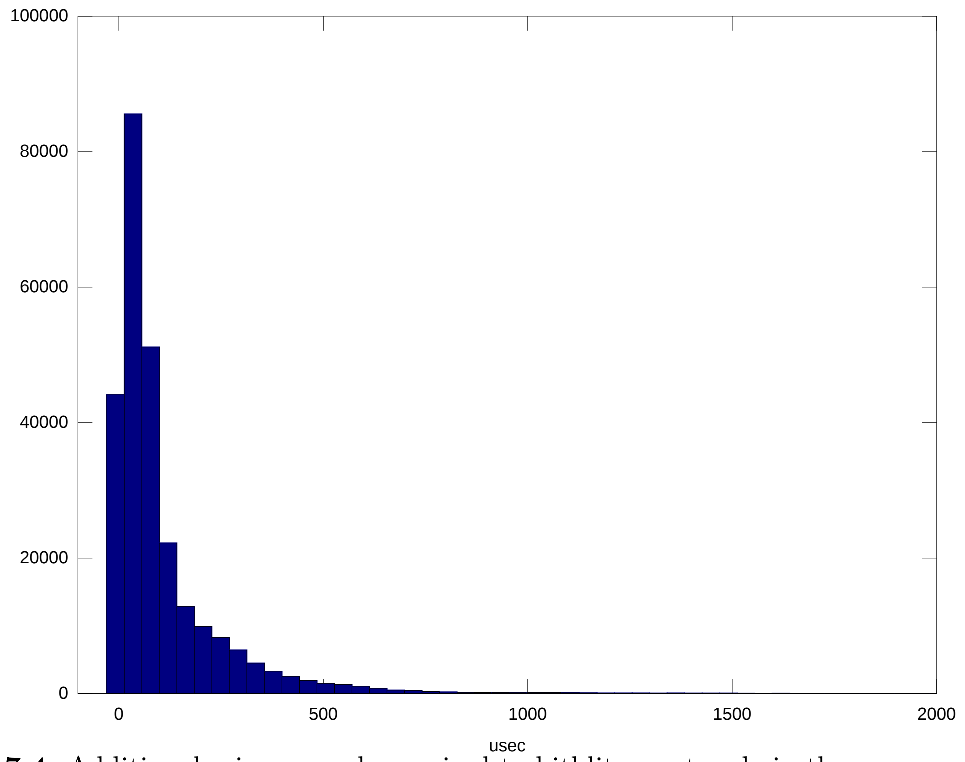


Figure 7.4: Additional microseconds required to bitblit a rectangle in the presence of heavy 3D GPU load

7.2.4 Bitblitting performance hit on applications

The test wants to prove that on the chose hardware platform, bitblitting performance has a direct impact on the frame rate of applications. The test involved executing the benchmark glmark2-es2 and measuring the frames/second obtained with different bitblitting loads running concurrently. The command line used for glmark2-es2 was the following:

```
./glmark2-es2 -s 1024x768 -b texture:duration=30
```

An application was set to bitblit on the framebuffer 60 times a second, simulating a compositor rendering on the screen at 60fps while glmark2-es2 was running.

The test was run first without any extra bitblitting, and then with rectangles bitblitted by the compositing application: 20 rectangles of 100x100 pixels each, then 200x200, 300x300 and 400x400. The results plotted in fig. 7.5 show that heavier bitblitting commands cause a loss of performance in 3D applications.

This test proved that 3D applications performance and bitblitting are closely related on the platform tested due to bandwidth restrictions, confirming the initial assumptions. An improved bitblitting algorithm can therefore relieve the GPU of some load and permit to achieve higher frame rates on the running applications.

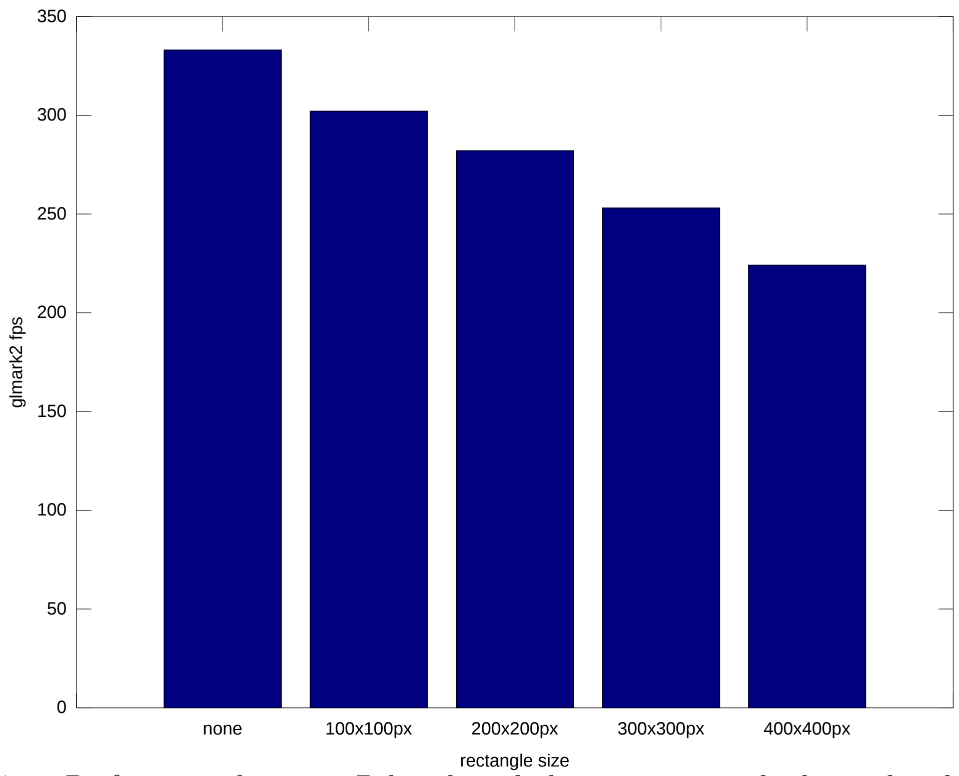


Figure 7.5: Performance hit in a 3D benchmark due to increasingly demanding bitblitting commands run in parallel.

7.3 Compositing strategy evaluation using the prediction model

The verification of the compositing strategies aims to test the validity of the decisions taken in the algorithms, covering the widest possible range of test cases. For this reason, a random *scenario* generator has been developed. A scenario is defined as a set of overlapping windows, defined as rectangles with a position on screen, a size and a z-value. Swapbuffer commands are simulated assigning a random but plausible (fixed or variable) frame rate to the windows and *marking* them in the frames according to that value.

7.3.1 Test description

The test does not measure the actual time spent bitblitting, but rather uses the output rectangles of the compositing strategies to predict their performance. For every frame, every strategy outputs a set of rectangles that has to be bitblitted for the frame, depending on the applications that updated their windows.

The prediction model generated for calibration in section 7.2.2 is used to estimate the bitblitting times of the output sets of rectangles for 3 compositing strategies: *full*, *tiled* and *dynamic*. *Cached* is here not considered since it does not affect the output set of rectangles, being an optimization of the *dynamic* algorithm aimed to save CPU time. The results of *dynamic* are therefore valid for *cached* too. For every rectangle, the estimated bitblitting time is calculated with the equation 7.1. x and y correspond to width and height of the rectangle whose bitblitting time must be predicted.

$$t_{ms} = 1.0676 \times 10^2 - 1.1223 \times 10^{-3} \cdot w + 2.1861 \times 10^{-3} \cdot h + 1.7267 \times 10^{-3} w \cdot h \tag{7.3}$$

The test aims to verify the correctness of the *dynamic* compositing strategy, comparing the rectangle sets generated by it with those generated by the other strategies. The expected result is to have predicted times for the *dynamic* strategy always less or equal than the values obtained using *full* or *tiled* compositing, therefore confirming that the implementation of the *dynamic* strategy is conform to what specified in section 5.6. Before showing the aggregated results, the development and execution of a sample run are described.

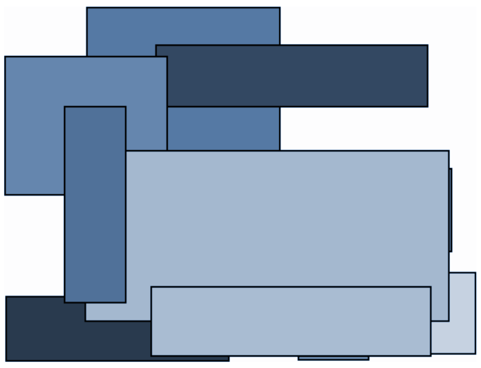


Figure 7.6: Scenario used to obtain the results shown in fig. 7.7

An example of a single run of the test is now presented. The scenario in fig. 7.6 was used. The graph is initialized and the rectangles inserted. This sample run is set to run at 60 frames/second, and for displaying purposes only 8 frames will be considered. On every frame, some of the windows will execute a `swapbuffers` command updating their content, therefore calling `graph_mark` with their window id. After the frame period ends (in this case 16.6ms for 60fps), the ids of the updated windows are used in the different bitblitting strategies (implemented in the `graph_compose_full`, `graph_compose_tiled`, `graph_compose_dynamic` functions) to the predicted bitblitting time value. No actual bitblitting is performed in this test. Every strategy outputs a set of rectangles, and the predicted time required for bitblitting each of those rectangles is then summed.

In fig. 7.7, 8 sets of 3 bars are displayed. Every set represents a frame where some windows issued a swapbuffers command and were therefore marked. In every frame the predicted time for *full* (1st bar, red), *tiled* (2nd, green) and *dynamic* (3rd, blue) are plotted in microseconds. The expected result is that the *dynamic* strategy will always find a set of windows, which minimizes the bitblitting time. Since the frame rates of windows are different, a frame can not be compared with other ones. Instead, the strategies within a single frame should be compared.

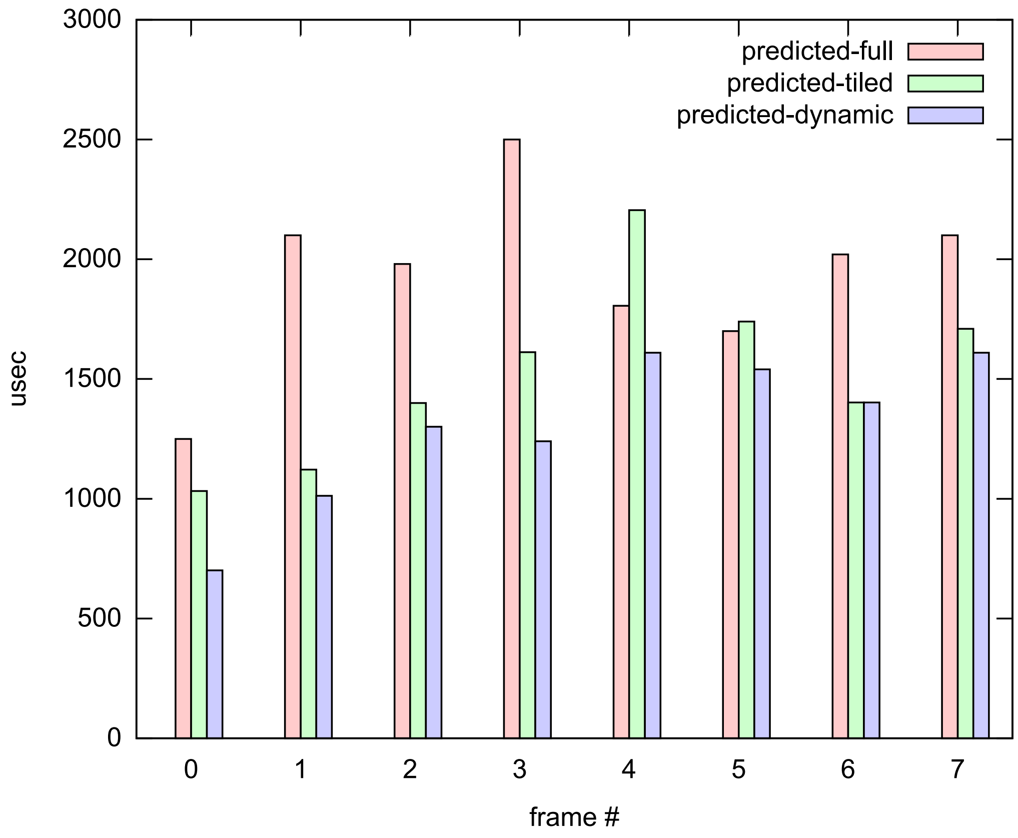


Figure 7.7: Comparison of predicted bitblitting times between different strategies in a sample of 8 frames

From the plot in fig. 7.7 it can be seen that while the choice between *full* and *tiled* strategies may vary, the *dynamic* one is always able to save GPU time. At worst it matches the best one among the other strategies, like in case of frame 6.

7.3.2 Test setup

The overlapping windows used in the test are chosen by a random scenario generator. The variables randomized are number of windows, their position, size, and frame rate. The number of windows is defined randomly between 8 and 12 and their sizes are bound to 100-800 pixels in width and 100-600 pixels in height. They can appear anywhere on screen as long as they are fully contained by the screen boundary. The frame rate is chosen between 20 and 60 frames per seconds.

For a wide coverage of the performance of the *dynamic* strategy the test is run on 100 random scenarios. The screen resolution is set to 1280x800. The estimated time to draw the frame is then calculated for different strategies: *full*, *tiled*, *dynamic*. The test was run on 100 frames on each of the 100 scenarios, for a total of 10000 sample frames.

7.3.3 Test results and evaluation

Comparison of full and tiled strategies

The histogram on the left in fig. 7.8 shows the difference between the time predicted for *full* and *tiled* compositing: $t_{full} - t_{tiled}$. The plot shows that, depending on the scenario layout and the marked windows, the choice between *full* and *tiled* may vary. In 76.88% of the 10000 frames considered, *tiled* was the best strategy. In 17.75%, showed better prediction times better. In 5.37% of the frames the two strategies did output the same rectangle set.

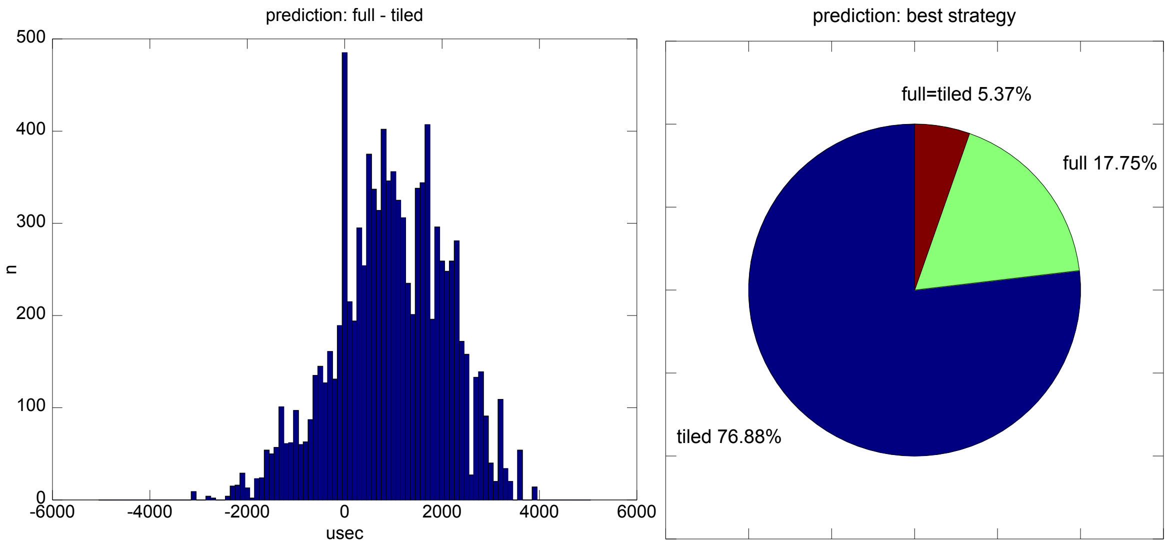
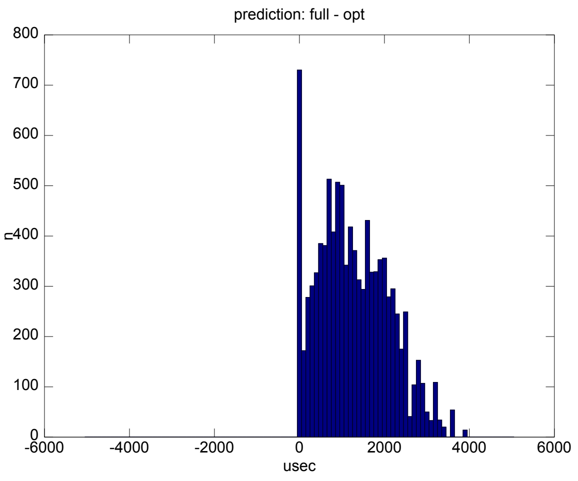


Figure 7.8: Predicted time difference between full and tiled compositing

Comparison of dynamic with full and tiled

The histogram on the left in fig. 7.9 shows the difference between the time predicted for *full* and *dynamic* compositing: $t_{full} - t_{dynamic}$. On the right, $t_{tiled} - t_{dynamic}$ is shown. The plots are centered on zero, and it can be seen that in every single case the *dynamic* strategy has found a set of rectangles is at least as good as the other strategy, since no negative values are included. When no optimization was found, *dynamic* is able to match the best among *tiled* and *full*.



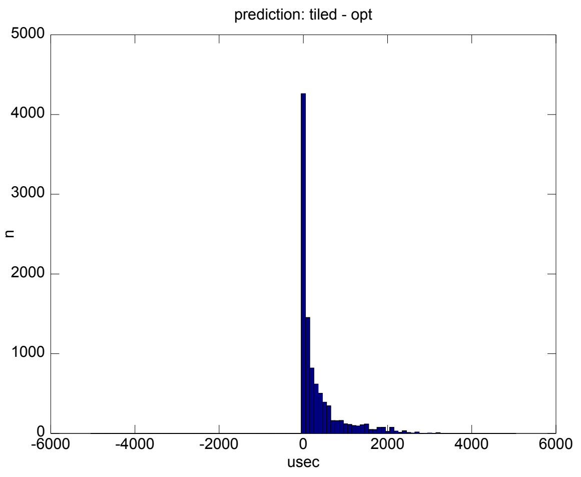


Figure 7.9: Predicted time difference between full and dynamic compositing, tiled and dynamic compositing

Concerning *full*, in 92.70% of the 10000 frames considered a better set of rectangles was found, while in the 7.30% the *full* strategy was already optimal. The average saved bitblitting time on 10000 frames by the *dynamic strategy* was estimated to be 43% of the time required by the *full* one.

In 57.39% of the 10000 frames considered the *dynamic* strategy found a better set of rectangles to bitblit than the *tiled* one, while in 42.61% the *tiled* strategy was already optimal. The average saved bitblitting time on 10000 frames was estimated to be 16%. A summary of the results in shown in table 7.1.

Dynamic vs:	Improved frames (predicted)	Time improvement (pred.)
Full	92.70%	43%
Tiled	57.39%	16%

Table 7.1: Percentage of frames improved and bitblitting time saved using dynamic compositing (predicted values).

Comparison of dynamic with the best result among full and tiled

The final results obtained using the generated prediction data involve the comparison between the best result among *full* and *tiled* and the time value predicted by the *dynamic* strategy. In this way it is possible to find the percentage of frames where the *dynamic* strategy found a set of rectangles which resulted in a more efficient bitblitting than the sets proposed by both *full* and *tiled*.

Plotting $min(t_{full}, t_{tiled}) - t_{dynamic}$ in fig. 7.10 shows that the *dynamic* prediction strategy never performs worse than the other two: in 53.02% of the 10000 frames considered a better set of rectangles was found, while in 46.98% one of the other two strategies was already optimal and *dynamic* was able to match it. The average saved bitblitting time on 10000 frames was estimated to be 10%. This shows that in more than half of the cases the *dynamic* strategy is expected to be improve the bitblitting time of the other strategies. Table 7.2 sums up the results.

Dynamic vs:	Improved frames (predicted)	Time improvement (pred.)
Best of full and tiled	53.02%	10%

Table 7.2: Percentage of frames improved and bitblitting time saved using dynamic compositing (predicted values).

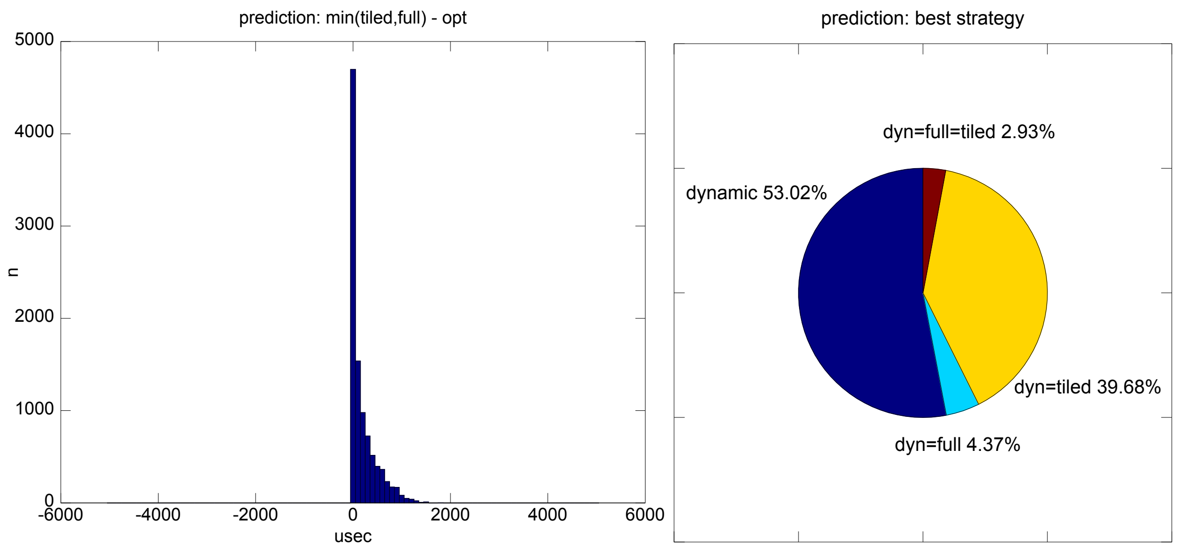


Figure 7.10: Predicted time difference between *minimum(full,tiled)* and dynamic compositing

In the right plot in fig. 7.13 the strategies are displayed according to the frequency of their occurrence as optimal strategy.

7.4 Compositing strategy evaluation by measuring bitblitting time

The focus of this test it to measure the effective time required to bitblit a frame with different compositing strategies. The measurements are done using the hardware/software setup described in , wrapping the single bitblitting API calls with time measurement functions.

7.4.1 Test description

The test is set up in the same way the verification test in section 7.4.4 was executed, so the results can be later correctly compared.

The test measures the actual time that the bitblitting functions require to draw frames, comparing how the compositing strategies perform on the same frame. For every frame, every strategy outputs a set of rectangles that has to be bitblitted for the frame, depending on the applications that updated their windows. The time required to bitblit those rectangles on screen is measured.

The *dynamic* strategy uses the prediction model generated in the calibration step described in section 7.2.2 to take the decisions required by the algorithm.

The *cached* strategy is again not included since it is an CPU-optimized version of the *dynamic* algorithm, and this test measure the time spent to execute the bitblitting commands on the GPU, therefore the results of *dynamic* are valid for *cached* too.

7.4.2 Test setup

The overlapping windows used in the test are chosen by a random scenario generator. The variables randomized are number of windows, their position, size, and frame rate. The number of windows is defined randomly between 8 and 12 and their sizes are bound to 100-800 pixels in width and 100-600 pixels in height. They can appear anywhere on screen as long as they are fully contained by the screen boundary. The frame rate is chosen between 20 and 60 frames per seconds.

For a wide coverage of the performance of the *dynamic* strategy the test is run on 100 random scenarios. The screen resolution is set to 1280x800. The effective time required by the bitblitting functions to draw the frame is then measured for different strategies: *full*, *tiled*, *dynamic*. The test was run on 100 frames on each of the 100 scenarios, for a total of 10000 sample frames.

7.4.3 Test results and evalution

Comparison of full and tiled strategies

The comparison that was done in section 7.4.4, using the prediction data, is now done again using the actual time measurements obtained on the board. *full* and *tiled* compositing are compared on the 10000 frames rendered and shown in the histogram on the left side of fig. 7.11). In 77.16% of the frames, *tiled* performed better. In 18.38% of the frames, *full* was the fastest compositing strategy to render the frame, while in in 4.46% of them no significant difference was observed (within 100μs). The results of this test confirm that there is no strategy between *tiled* and *full* compositing that outperforms the other one in all cases. It depends instead on the scenario and the marked windows.

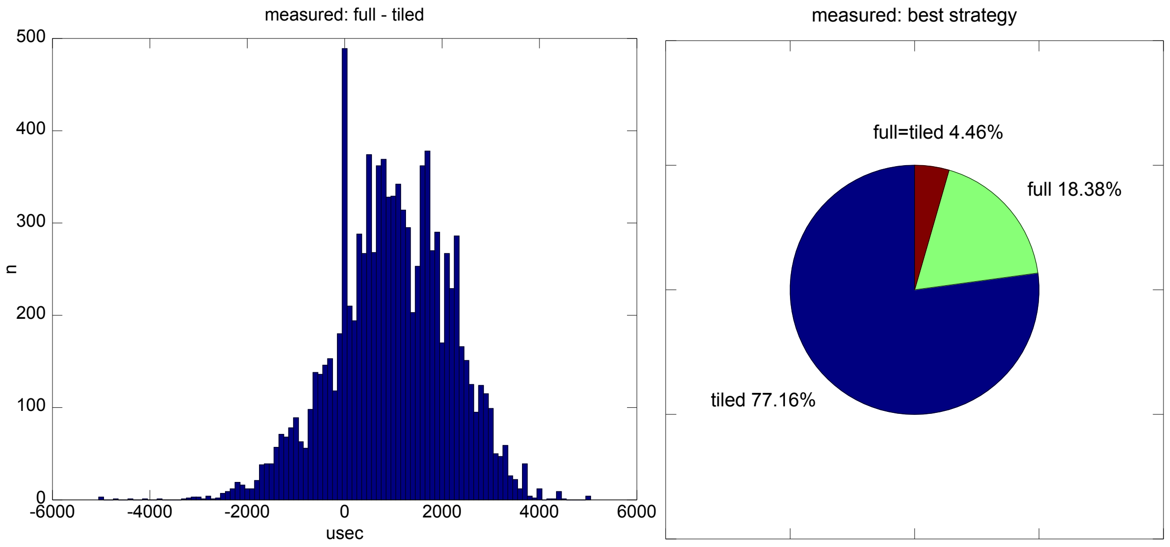


Figure 7.11: Measured time difference between full and tiled compositing

Comparison of full and tiled with dynamic

The histograms in fig. 7.12 show the difference between the time measured executing *full* and *dynamic* compositing (on the left), *tiled* and *dynamic* compositing (on the right).

In the first plot $t_{full} - t_{dynamic}$ is shown. In 92.66% of the 10000 frames rendered a better set of rectangles was found by the *dynamic* algorithm. In 6.47% of the frames the strategies perform in the same way (within 100μs). A small and not disruptive measurement error is also visible in the plot, accounting for 0.87% of the frames, since according to the predicted values no frame should ever be faster when rendered with *full* compositing. This is considered to be unavoidable because of the limitations of the time measurement functions, which have a non-zero overhead and a finite precision. Nevertheless, the results do not differ significantly from what expected after running the prediction tests. The average saved bitblitting time on 10000 frames was 43%/frame.

The histogram on the right of fig. 7.12 shows $t_{tiled} - t_{dynamic}$. The *dynamic* strategy performed better than the tiled one in 56.07% of the frames, while in 40.85% of the frames they perform similarly. The frames with faster rendering times in *tiled* mode amount to 3.08%. The error is higher compared to the *full* case, and this is suspected to be due to the higher number of rectangles that are on average bitblit in *tiled* mode compared to *full*. More rectangles required more time-measurement function calls, therefore increasing (although marginally) the overall error.

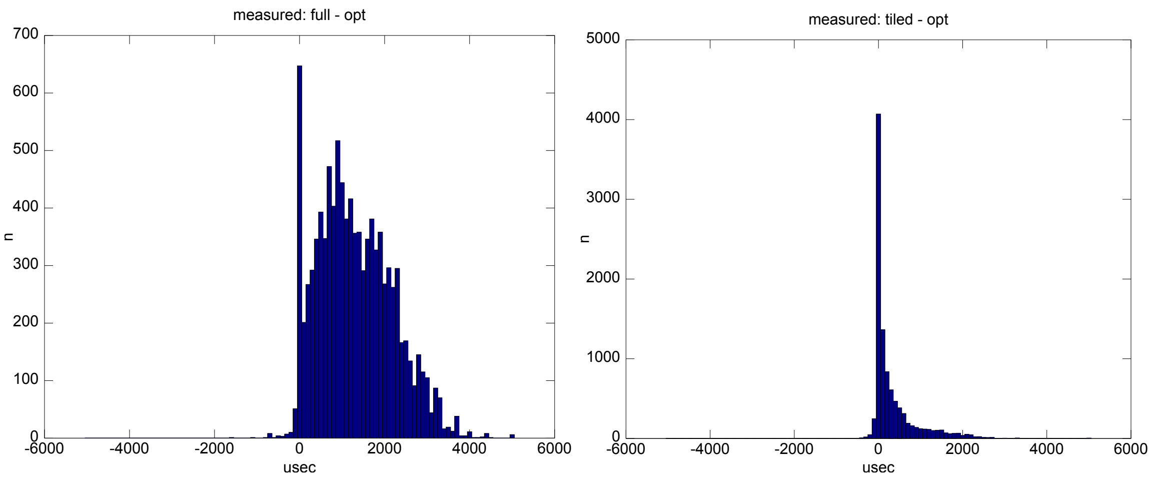


Figure 7.12: Measured time difference between full and dynamic compositing, tiled and dynamic compositing

The results are fully in line with those obtained in the prediction test, showing that the *dynamic* approach is always (within the precision of the time measurement functions used) able to find a set of rectangles to outperform both full and tiled compositing. The results are summed up in table 7.3.

Dynamic vs:	Improved frames	Time improvement
Full	92.66%	43%
Tiled	56.07%	15%

Table 7.3: Percentage of frames improved and bitblitting time saved using dynamic compositing (measured values).

Comparison of dynamic with the best strategy available

Following the same pattern used to analyze the predicted values, $min(t_{full}, t_{tiled}) - t_{dynamic}$ is plot in fig. 7.13. The focus of this test is to identify the percentage of frames which are more efficiently bitblitted using the *dynamic* strategy, therefore implying that the *dynamic* algorithm found a rectangle set to optimize the frame rendering which is better than those proposed by both *full* and *tiled*.

In 51.70% of the 10000 frames rendered, *dynamic* performed better than any other strategy, while in 44.42% one of the other two strategies was already optimal and their result did not differ more than 10μs. In this case the error was similar to what obtained in the *tiled* test, 3.88%, which is plausible since most of the frames displayed a trend of being more efficiently bitblitted with *tiled* than *full*. Table 7.4 sums up the results, displaying also that the average time saved by using *dynamic* compositing was 9%.

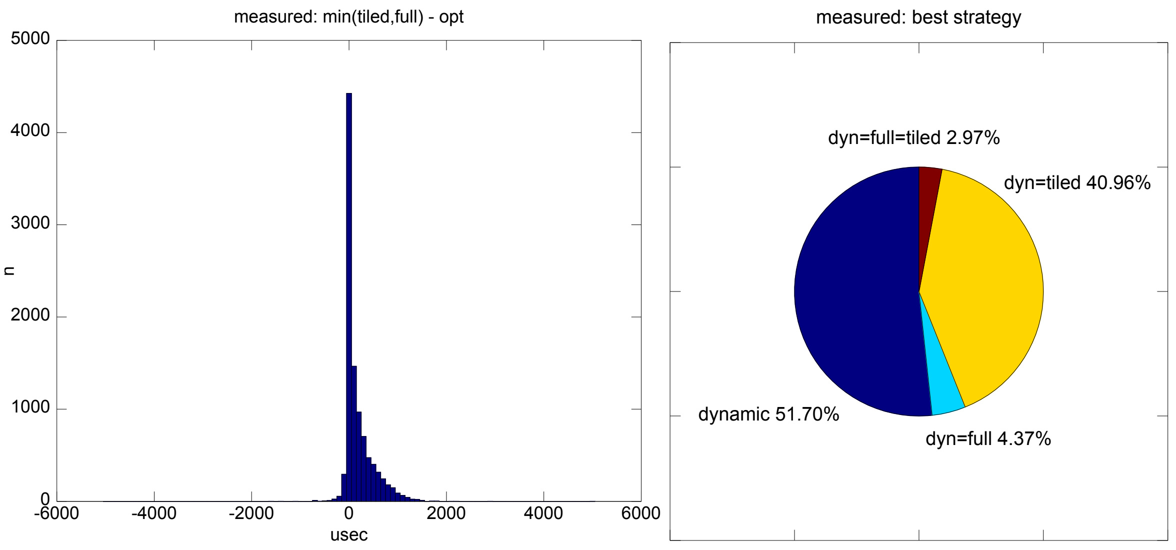


Figure 7.13: Measured time difference between *minimum(full,tiled)* and dynamic compositing

Dynamic vs:	Improved frames	Time improvement
Best of full and tiled	51.70%	9%

Table 7.4: Percentage of frames improved and bitblitting time saved using dynamic compositing (measured values).

In the right plot in fig. 7.13 the strategies are displayed according to the frequency of their occurrence as optimal strategy.

7.4.4 Comparison with predicted values

The results obtained measuring the effective bitblitting time (M) of the *dynamic* strategy can be directly compared with the values predicted (P) in the previous section. Table 7.5 shows the percentage of frames improved while table 7.6 shows the percentage of bitblitting time saved.

In general, the measured results are in line with the predicted ones. The *dynamic* strategy has been proven to be the most efficient in terms of bitblitting time, improving the basic strategies in more than 50% of the cases saving on average 9% of the bitblitting time, when compared

Dynamic vs:	Improved frames % (P)	Improved frames % (M)
Full	92.70	92.66
Tiled	57.39	56.07
Best of both	53.02	51.70

Table 7.5: Percentage of frames improved by using dynamic compositing, predicted (P) and measured (M)

Dynamic vs:	Avg time improvement % (P)	Avg time improvement % (M)
Full	43	43
Tiled	16	15
Best of both	10	9

Table 7.6: Bitblitting time improvement by using dynamic compositing, predicted (P) and measured (M)

with the best suited compositing approach among the traditional ones. When comparing it directly with the traditional compositing approaches it performs even better. The measured results show some jitter which is caused by the time measurements done in user space, and therefore in some cases negative values are present. The error does although not interfere with the results obtained, which show the trend expected after executing the tests in section with the prediction model.

7.5 Algorithm CPU overhead

The algorithms implementing the compositing strategies analyzed vary greatly in complexity and therefore show different execution times for the same input, which consists in the graph of windows and their marks in the current frame.

The plot in fig 7.14 shows how they behave CPU wise, measuring the average execution time necessary for calculating the output set of rectangles in 10 different scenarios randomly generated, following the same rules used to generate the scenarios in section 7.3.1. The values shown are in microseconds, and they were obtained measuring the average time necessary to run the algorithm on 100 frames for every scenario.

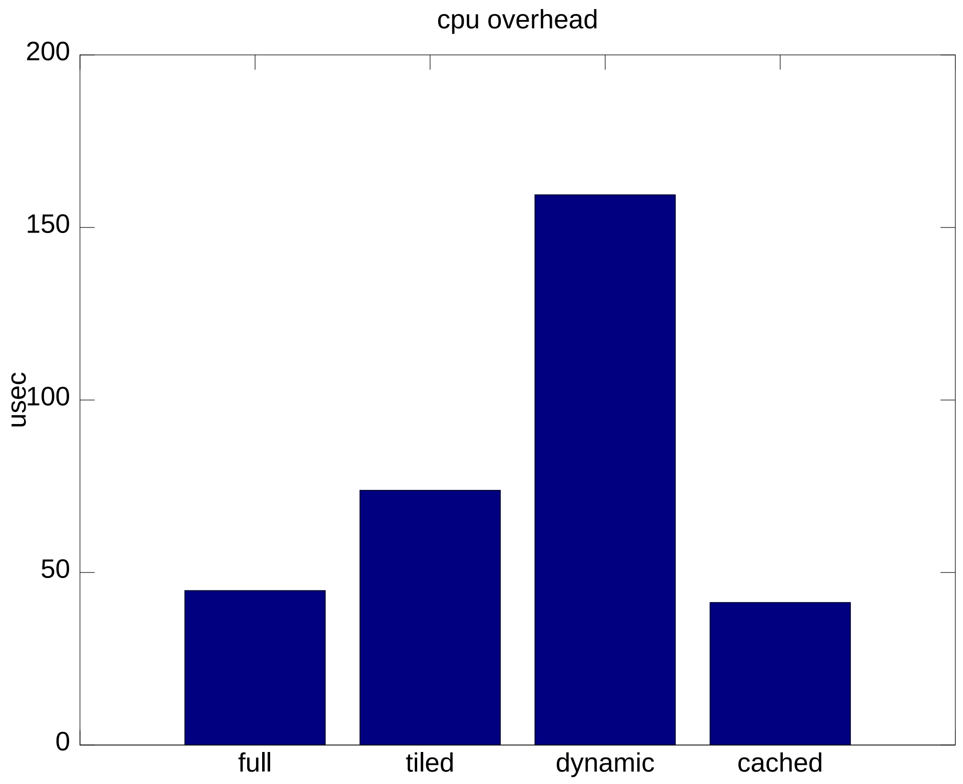


Figure 7.14: Average CPU execution time of the algorithms on 100 frames on 100 scenarios

The results show that the *cached* algorithm is 3.8 times faster than the *dynamic* one, saving in average 73.6% of CPU time while bitblitting the same set of rectangles. The rectangle set yielded is, as previously shown in section 7.4, always equal or better than the sets produced by the other strategies considering the bitblitting time, therefore indicating that the *cached* version of the *dynamic* strategy is the most efficient compositing strategy tested.

7.6 Overall bitblitting time

The tests in the previous sections considered individually the time the GPU was kept busy to execute the bitblitting commands and the CPU time required to run the strategy algorithm. Now the overall time from start to finish of every **graph_compose** function is measured. It is a significant value since this is the time frame where the compositor thread will try to execute bitblitting commands on the GPU, competing for the bus with other graphical applications. For this reason it is important to minimize the window of possible GPU bus contentions.

To help visualizing the results, the plot in fig. 7.15 shows a sample of 8 frames where the overall time of the four algorithms, implementing 3 strategies, is compared for each frame. For the *dynamic* strategy two implementations are here tested, the normal and the *cached* one. The bars represent full (1st, red), tiled (2nd green), dynamic (3rd blue) and cached (4th grey) compositing algorithms. The time, in microseconds, includes GPU and CPU time. It was measured checking the timestamp before entering and after exiting every **graph_compose** function. When the *dynamic* estimated time is very close to one of the other strategies, it can be possible for its effective time to be slightly higher that the other ones, due to the greater CPU requirements of such strategy. In fig. 7.15 this happens on frames 2 and 6, for example. This undesirable result can be mitigated measuring the average time on a higher number of frames, since in the long run the advantages offered by a more efficient compositing exceed the slightly higher CPU overhead necessary to compute the most efficient rectangle set.

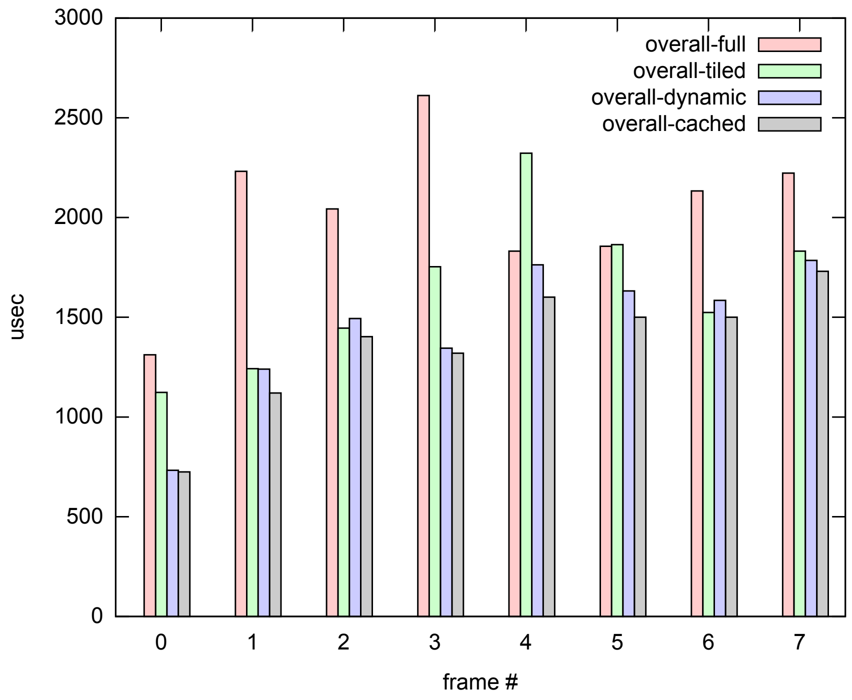


Figure 7.15: Sample of 8 frames showing the overall time spent in the walk functions of 4 algorithms

The actual test was executed running the compositing algorithms on 10000 randomly generated frames on 100 different scenarios (100 frames for every scenario). The scenarios were built using the same parameters described in section 7.3.1, randomizing size and position of windows.

With the data collected the average time spent into the functions was calculated and the plotted in fig. 7.16. The *dynamic* algorithm is performing better than *full* and *tiled*, saving 42% and 14% of the overall bitblitting time respectively. This confirms that even though the CPU requirements to run the more complex algorithm are slightly higher, the time saved while bitblitting guarantees also spending less time in the `compose` function.

On top of that, the *cached* version is performing better than any other one, saving CPU time compared to the *dynamic* one while bitblitting exactly the same rectangle set, saving around 5% of the overall bitblitting time. So using the *cached* algorithm permits to achieve an optimal bitblitting time without increasing the bus contention window due to CPU overhead, further optimizing the *dynamic* solution.

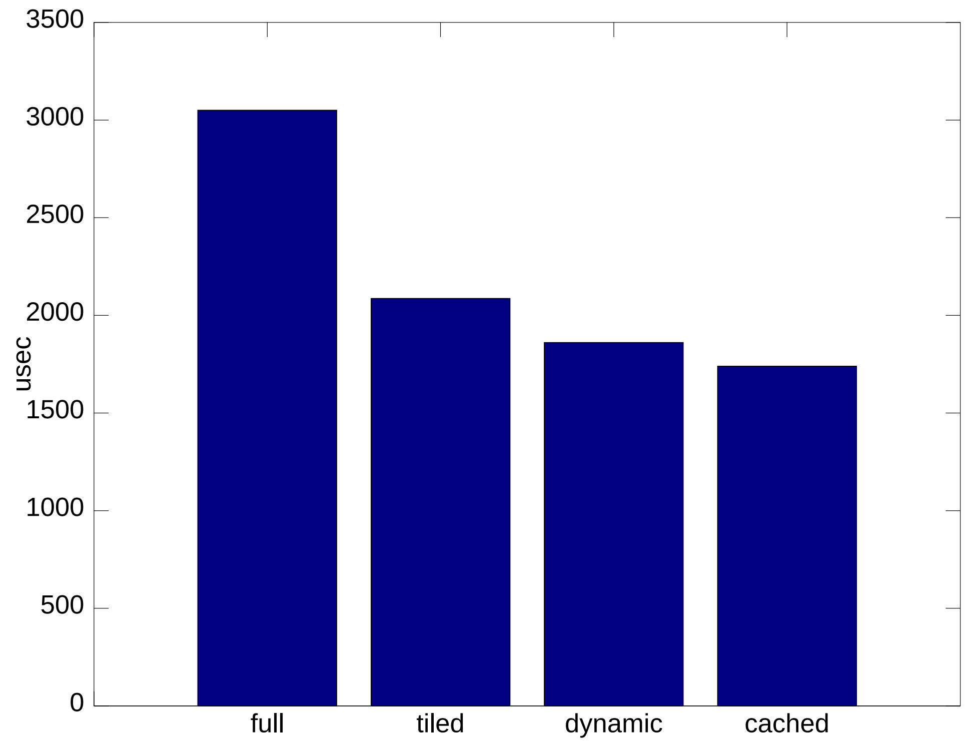


Figure 7.16: Average results on 10000 frames showing the overall time spent in the walk functions of 4 algorithms

7.7 Application frame rate

The test analyzes how different bitblitting strategies applied to a specific scenario affect the frame rate of the graphical applications involved, aiming to prove that a more efficient compositing strategy results in an improvement in the framerate. Multiple instances of es2gears are run logging the average fps while different bitblitting strategies are used to display them on a realistic layout on screen.

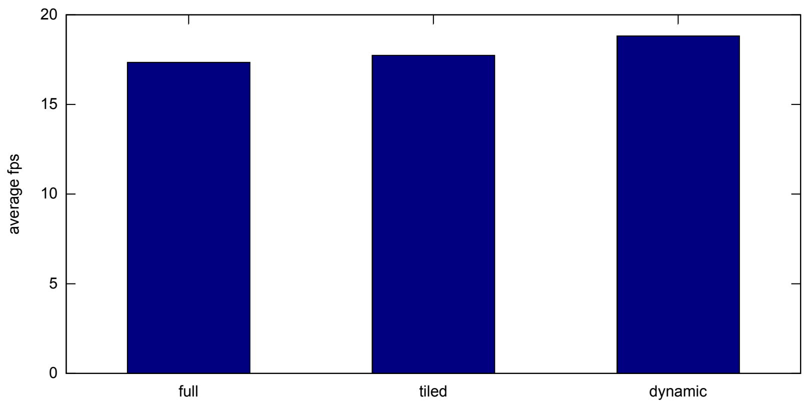


Figure 7.17: Average es2gears framerate with different compositing strategies

Three compositing strategies, *full*, *tiled* and then *dynamic* are used bitblitting on the screen at 60fps, while 7 processes are executing the es2gears benchmark. The results in fig. 7.17 show that bitblitting the screen with the *dynamic* compositing strategy results in a framerate improvement for the applications. In table 7.7 the framerate improvement obtained by using the *dynamic* compositing strategy is shown. Compared to *full* compositing, a 7.9% increment was registered. In respect to *tiled* compositing, *dynamic* results in a framerate improvement of 6%. The results confirm that an improvement in the efficiency of bitblitting commands can directly relate with an increased framerate for 3D applications on the board tested.

Dynamic vs:	FPS improvement %
Full	7.9
Tiled	6.0

Table 7.7: Framerate improvement using the dynamic bitblitting strategy

7.8 Animations

The compositor offers two way to edit a window position and size, **modify** and **animate**. The former applies instantly a modification on the graph, while the latter generates a transition effect over time. In this section the performance impact of the two methods is analyzed.

A scenario composed of 8 windows has been chosen to represent a realistic case with a few windows overlapping. The window to be modified/animated, marked with A in fig. 7.18, lies initially on the bottom right of the screen, and it will be moved or animated to its final destination displayed in fig 7.19.

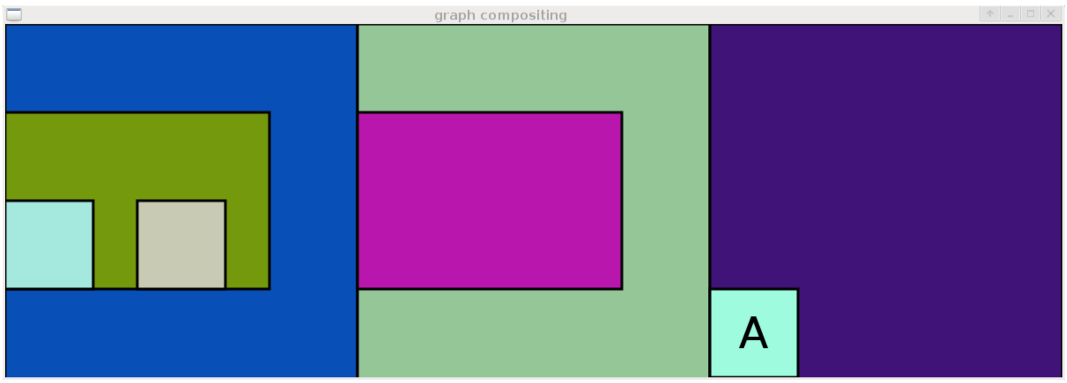


Figure 7.18: Scenario used to test animated windows, initial layout

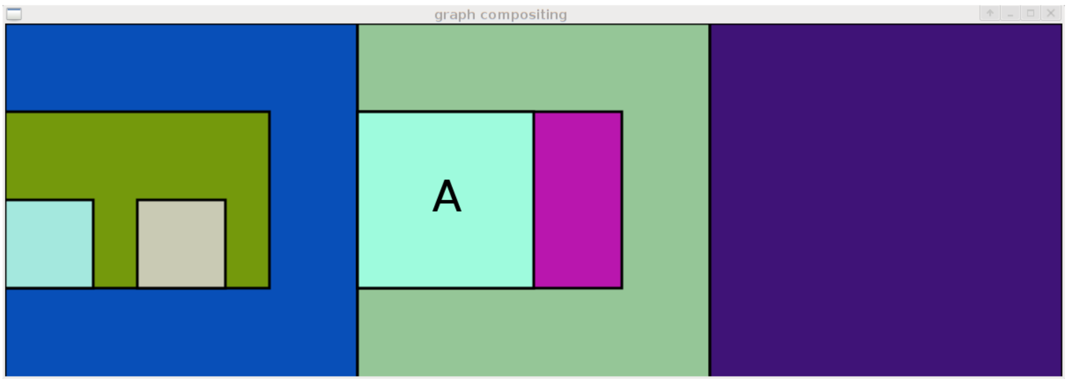


Figure 7.19: Scenario used to test animated windows, final layout

The test runs for 1 second at 60 frames per second. To be able to display the results in a plot which is suitable to be analyzed visually, the 8 windows run at fixed frame rates, 20fps or 60fps. The plots in fig. 7.20 and 7.21 display a timeline of 60 frames (1 second), showing for every frame: how much time the bitblitting time is predicted to be (blue), the measured value of the actual time taken to bitblit (green) and the total time including also the CPU time spent by the algorithm to insert/remove/modify the window (red).

7.8.1 Modify

The plot in fig. 7.20 represents the timeline of a **modify** event triggered at frame 15. It can be seen that the *total* time required for frame 15 was higher compared to the previous ones, since the **graph_modify** API call was executed and the graph rebuilt with the window at the new position. From there on, the new graph layout was then returning new estimated bitblitting times, since moving a window changes the tiling strategy applied.

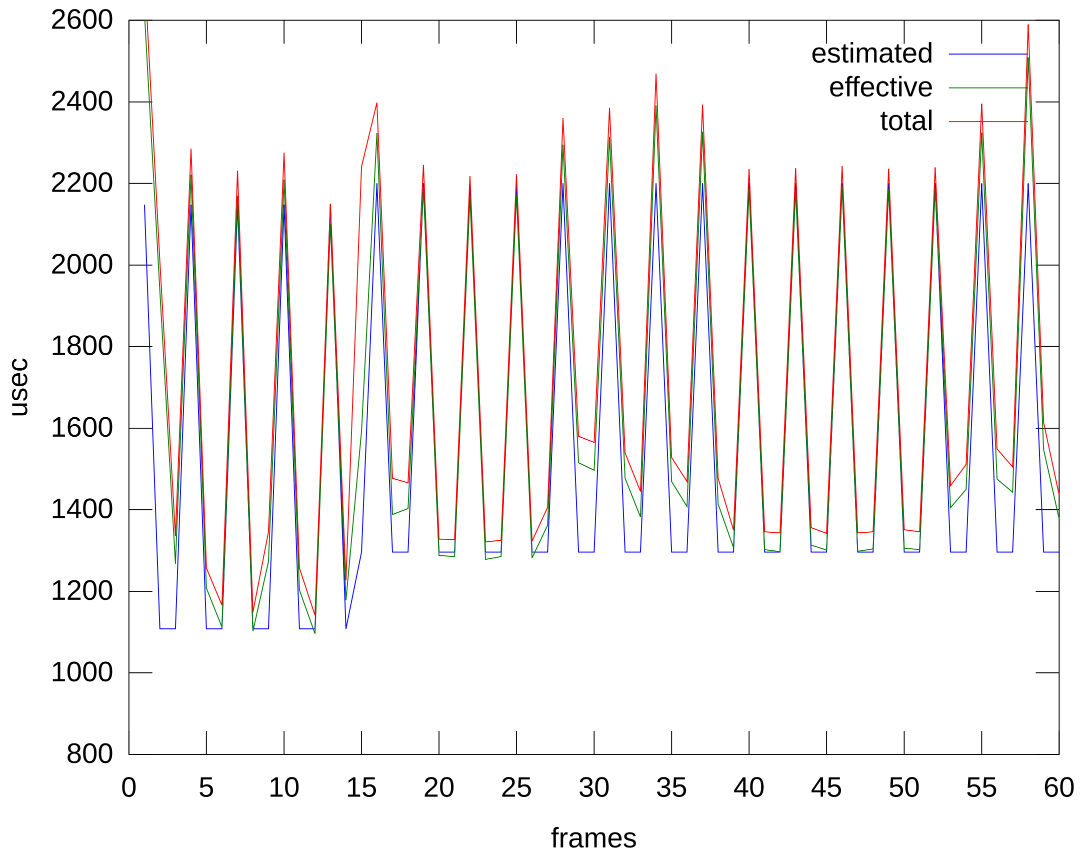


Figure 7.20: Performance analysis of a modify event

7.8.2 Animate

In the plot in fig. 7.21 at frame 15 a **animate** event was instead applied, with a desired duration of 0.5s, which at 60fps means 30 frames. Therefore a transition effect was applied between frames 15 and 45 to move and resize the window to the final destination. This forced the graph to bitblit the windows in full mode to avoid trailing effects. After 30 frames the window had to be inserted again in the graph and the tiles involved regenerated: this can be seen in the peak in the *total* time which appears only on frame 44. From frame 45 on, the execution proceeds exactly like in the *modify* case.

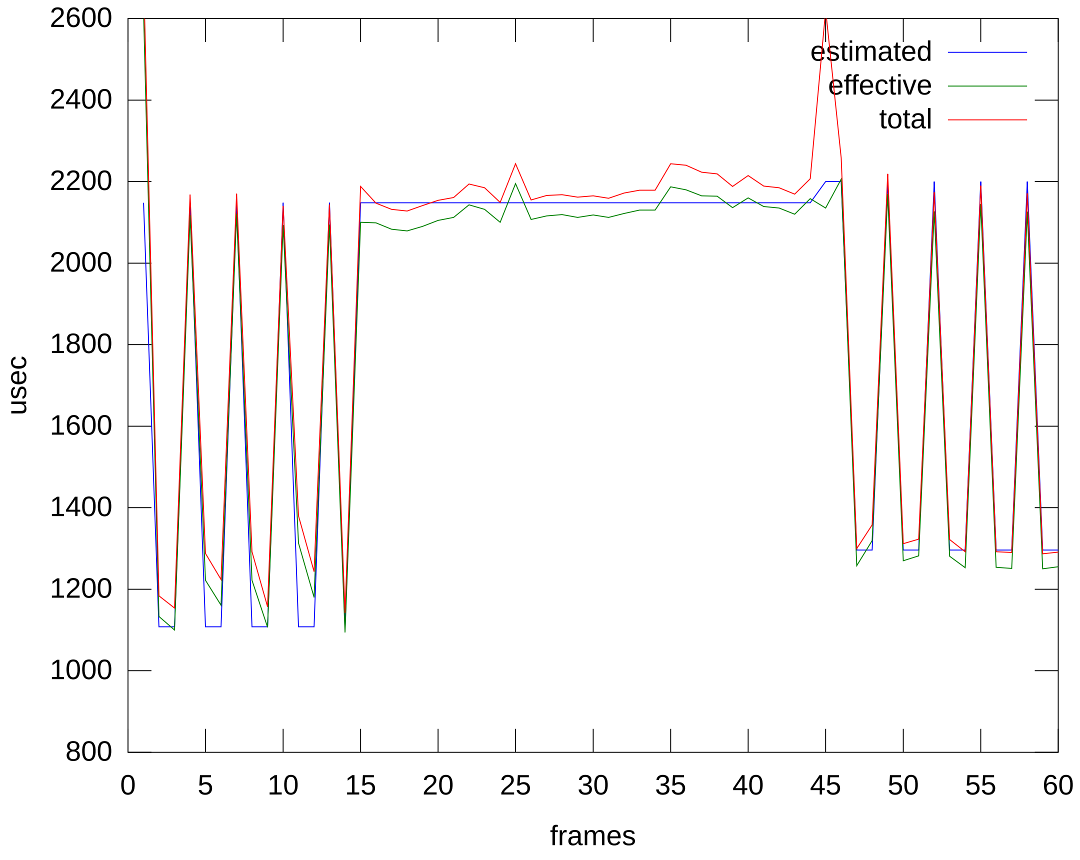


Figure 7.21: Performance analysis of an animation

7.9 Transparency

The compositor supports transparent windows without losing any performance compared to traditional full bitblitting compositors, and occasionally providing an improvement of the bitblitting time compared applying the tiling optimizations to the transparent windows too. An example to show how such scenarios are optimized follows.

The scenario in fig. 7.22 displays 8 windows, labeled from *A* to *H*. Among those, *F* is the single one which has an alpha channel. *F* overlaps *D* as well as emphA. *G* and *H* are on top of it. In presence of transparency, the dependency rules change, since when *D* or *A* are updated *F* must be updated too. A comparison between *full* and *tiled* compositing follows, focusing what could trigger a forced update on *F* and what an update on it could in turn trigger. After that, a test shows how *dynamic* compositing is able to choose the most efficient strategy for every tile also in case of transparent windows.

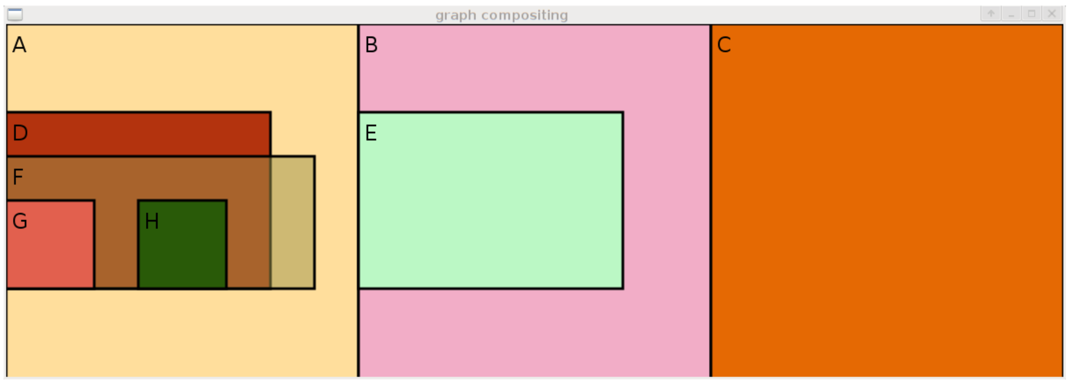


Figure 7.22: Scenario used to test transparency

Table 7.8 shows the direct dependencies of all the windows in the scenario, i.e. the windows that have to be redrawn (with *full* or *tiled* bitblitting) when a window is updated. Notation: *F (3t)* means 3 tiles belonging to window *F*.

Updated window	Dependencies with full	Dependencies with tiled
A	D, F	F (1t)
B	E	-
C	-	-
D	F	F (3t)
E	-	-
F	A, D, G, H	A (1t), D (3t)
G	-	-
H	-	-

Table 7.8: Dependencies table

Applying transitivity to the dependencies (dependencies of a dependent window have also to be redrawn), it is possible to complete the table of updates for the current scenario.

Updated window	Redrawn windows with full	Redrawn windows with tiled
A	A, D, F, G, H	A (3t), F (1t)
B	B, E	B (3t)
C	C	C
D	A, D, F, G, H	D (4t), F (3t)
E	E	E
F	A, D, F, G, H	A (1t), D (3t), F (4t)
G	G	G
H	H	H

Table 7.9: Dependencies table including dependencies of dependencies

On table 7.9 it can be seen that in case of *full* bitblitting, any update to windows *A*, *D* or *F* will force the redraw of the window set *ADFGH*. The *tiled* strategy offers another option, which is updating only the background of the transparent window with the content of the updated windows, and then redrawing the partially transparent content of windows *F* on top of it. This is where the *dynamic* strategy compares the effort necessary for the two approaches, taking rectangle count and size into account to take the best decision.

Using the same testing approach adopted in sections 7.4.4 and 7.4, a wide scale test was executed on 100 scenarios, executing 100 frames on each one of them, to then calculate the percentage of frames that the *dynamic* strategy is able to optimize. The tests have been built using the parameters described in section 7.3.1, with the addition of two transparent windows on every frame.

Dynamic vs:	Improved frames %	Time improvement %
Full	73.13	32
Tiled	53.76	12
Best of both	43.11	8

Table 7.10: Percentage of frames improvement and time saved using dynamic compositing with transparent windows.

The results displayed in table 7.10 show that *dynamic* compositing is also able to optimize the bitblitting time in presence of transparent windows. *Dynamic* compositing performed better than *full* compositing in 73.13% of the frames tested, reducing the bitblitting time by 32% on average. When compared directly with the *tiled* strategy, *dynamic* compositing was able to improve more than 53% of the frames with an average reduction of the bitblitting time of 12%. Even when compared with the best performing of the other two strategies, the *dynamic* strategy optimized 43.11% of the considered frames with an average time improvement of 8%.

7.10 Automotive scenario

The final test involved setting up a realistic automotive scenario using OpenGLES applications which render in offscreen framebuffers allocated through EGL. The compositor receives a notification when a window calls the eglSwapbuffers function, marking the window id. At 60 fps the chosen compositing strategy is executed to bitblit the screen. In fig. 7.23 a screenshot of the scenario is shown, while in fig. 7.24 the windows borders are highlighted to show how the windows overlap.

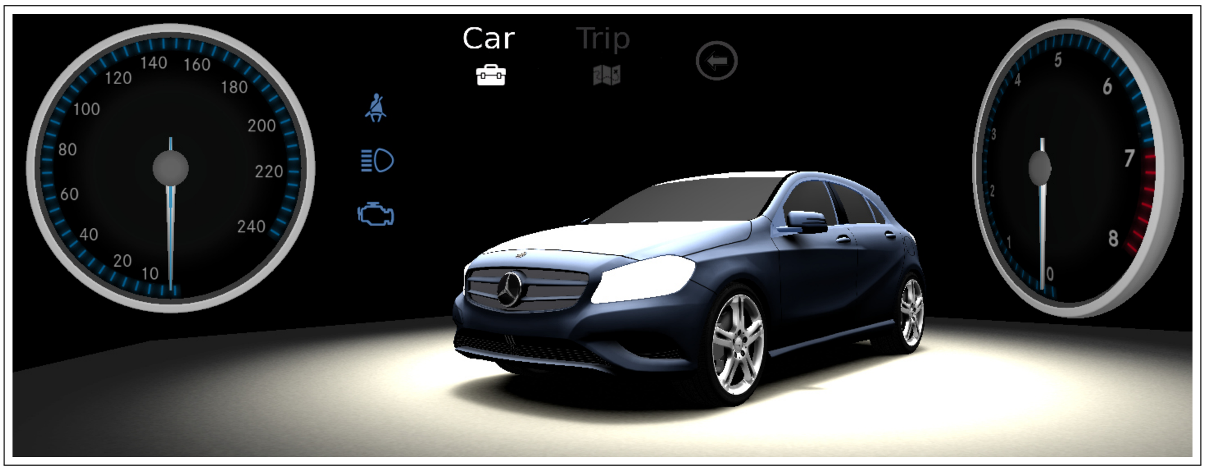


Figure 7.23: Automotive scenario

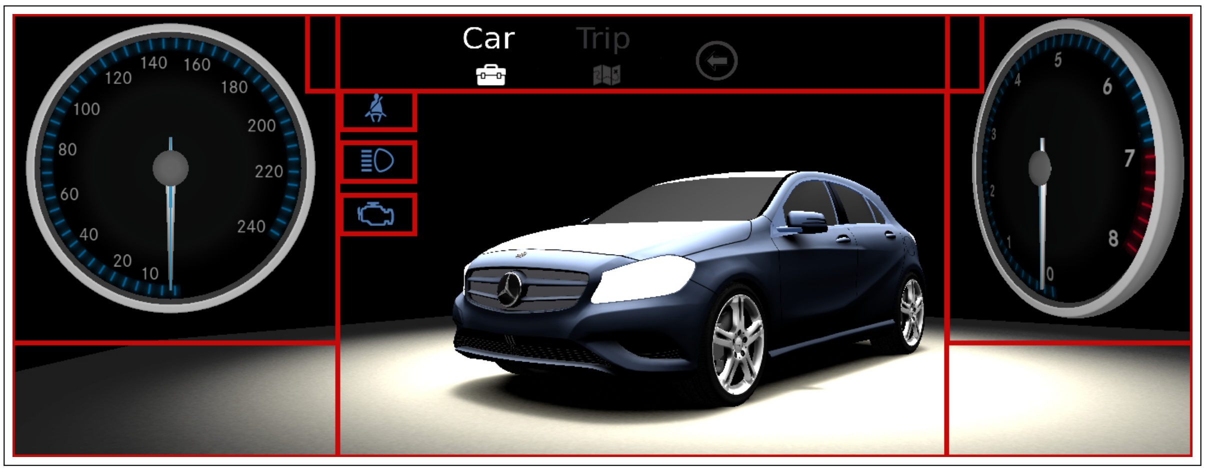


Figure 7.24: Automotive scenario

The 4 compositing strategies *full*, *tiled*, *dynamic* and *cached dynamic* have been run and the overall bitblitting time (cfr. section 7.6) measured. The plot in fig.7.25 shows the average results on the bitblitting of 10000 frames. In the scenario tested, *Dynamic* compositing saves on average 26% of the bitblitting time when compared to *full* and tiled and 7% when compared to *tiled*. The *cached dynamic* strategy improves the bitblitting of the *dynamic* strategy by a further 8%. The results are aligned to what found in the previous tests, confirming that in a realistic scenario running OpenGL applications similar improvements in the bitblitting time can be achieved by using *dynamic* and *cached dynamic* compositing.

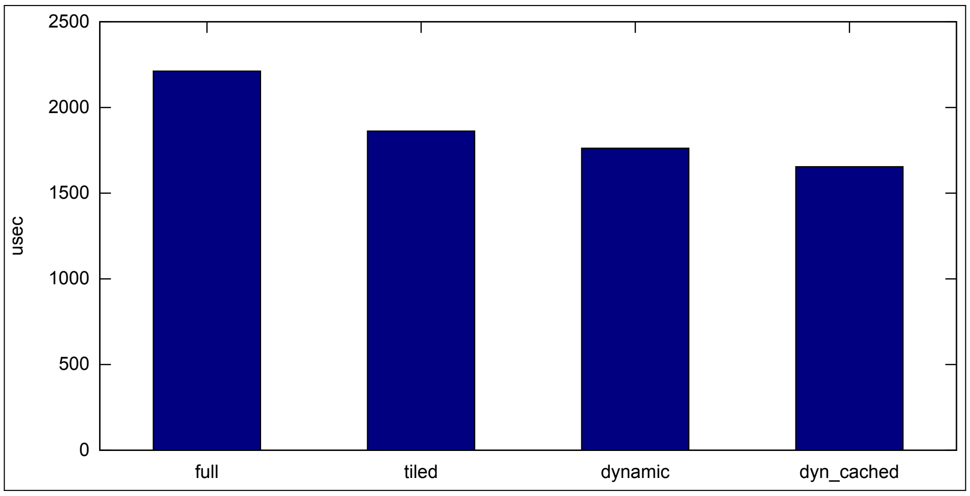


Figure 7.25: Automotive scenario results

8 Summary and conclusions

The usage of 2D and 3D graphical applications in automotive environments is increasing, together with the size and resolution of displays mounted in vehicles. This requires support of windowing systems which are as sophisticated as those running on desktop computers, although being limited by the computing power of the embedded devices.

One aspect of the performance of a windowing system is the behavior of the compositing software. The compositor is responsible to bitblit the content of the windows on screen, handling overlap cases. Compositing strategies like *full compositing* and *tiled compositing* are used by windowing systems, aiming for low algorithm overhead in case of *full compositing*, or for maximal reduction of overdraw in case of *tiled compositing*.

A concept for a compositor has been developed, aiming to minimize the bitblitting time by taking advantage of knowledge on the behavior of bitblitting commands on the system. A graph data structure was developed to store the tiles of the windows according to how they overlap. The graph is designed to be optimal to parse at runtime, when a new compositing strategy called *dynamic compositing* analyzes the graph taking decisions on how the windows should be bitblitted: if only partially because covered by other windows or completely, in order to save bitblitting time. The algorithm requires a system calibration that consists in measuring the time that a bitblitting operation on rectangles of different sizes requires. The calibration results in a prediction model that is used throughout the algorithm.

An evaluation platform was chosen to test the new compositing strategy and compare it with the state of the art. The calibration phase confirmed the initial assumption that bitblitting operations grow linearly together with the amount of pixels to bitblit, requiring also a fixed amount of time in the form of overhead for every bitblitting command executed. The strategy was then tested on a wide range of window combinations. The results confirmed that the *dynamic* algorithm is able to improve the bitblitting time of *full compositing* in 92.66% of the frames analyzed, reducing it average by 43%. When compared with *tiled compositing*, *dynamic* was able to improve 57% of the frames tested reducing the bitblitting time by 16% on average. An further optimization of the new compositing strategy has been also developed, reducing the CPU running time by introducing a cache that is used to store the output of the algorithm, using the windows that updated their content in the last frame as cache index. This approach resulted in 73% of the CPU time saved when used to cache the results of *dynamic* compositing.

A possible further development can be implementing a protocol to communicate to the 3D OpenGL applications which regions of their window are covered by others, and therefore not visible. The applications could use this information to selectively render only the visible content, therefore saving computing resources for the benefit of the rest of the system.

Bibliography

[1] http://www.daimler.com/dccom/0-5-1417330-1-1423615-1-0-0-1417332-0-0-135.html.

[2] http://www.qnx.com/news/pr_5602_1.html.

[3] Image source: http://www5.mercedes-benz.com/en/innovation/welcome_to_2025/.

[4] Brad A. Myers. A taxonomy of window manager user interfaces. *IEEE Comput. Graph. Appl.*, 8(5):65–84, September 1988.

[5] ESOP. *On safe and efficient in-vehicle information and communication systems: update of the European Statement of Principles on human-machine interface.* Commission of the European Communities, 2008.

[6] AAM. *Statement of Principles, Criteria and Verification Procedures on Driver Interactions with Advanced In-Vehicle Information and Communication Systems.* Alliance of Automotive Manufacturers, July 2006.

[7] Michael Abrash. *Michael Abrash's graphics programming black book.* Coriolis, Albany, NY, 1997.

[8] T.J. Olson. Hardware 3d graphics acceleration for mobile devices. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 5344–5347, 2008.

[9] http://www.vivantecorp.com/en/technology/composition.html.

[10] Brad A Myers. A complete and efficient implementation of covered windows. 1986.

[11] http://wayland.freedesktop.org/docs/.

[12] https://android.googlesource.com/platform/frameworks/native/+/android-cts-4.1_r1/services/surfaceflinger/SurfaceFlinger.cpp.

[13] http://www.ubuntu.com/.

[14] D. Rhoden and C. Wilcox. Hardware acceleration for window systems. *SIGGRAPH Comput. Graph.*, 23(3):61–67, July 1989.

[15] Keith Packard and James Gettys. X window system network performance. In *USENIX Annual Technical Conference, FREENIX Track*, pages 207–218, 2003.

[16] James Gettys and Keith Packard. The (re) architecture of the x window system. In *Proceedings of the 2004 Linux Symposium*, volume 1, pages 227–237, 2004.

[17] http://www.opengl.org/.

[18] http://www.opengl.org/registry/doc/glspec44.core.pdf.

[19] http://www.khronos.org/opengles/.

[20] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0363d/CJAJHGHF.html.

[21] Peter Nilsson and David Reveman. Glitz: Hardware accelerated image compositing using opengl. In *USENIX Annual Technical Conference, FREENIX Track*, pages 29–40. USENIX, 2004.

[22] http://directfb.org.

[23] http://bitsavers.informatik.uni-stuttgart.de/pdf/xerox/alto/BitBLT_Nov1975.pdf.

[24] Thomas Porter and Tom Duff. Compositing digital images. *SIGGRAPH Comput. Graph.*, 18(3):253–259, January 1984.

[25] http://www.beyond3d.com/content/articles/43/1.

[26] James Foley. *Computer graphics : principles and practice*. Addison-Wesley, Reading, Mass, 1995.

[27] R. Pike. Graphics in overlapping bitmap layers. *ACM Trans. Graph.*, 2(2):135–160, April 1983.

[28] http://www.pixman.org/.

[29] Robert W. Scheifler and Jim Gettys. The x window system. *ACM Trans. Graph.*, 5(2):79–109, April 1986.

[30] Adrian Nye. *Xlib programming manual for version 11 of the X Window system*. O'Reilly and Associates, Sebastopol, CA, 1992.

[31] http://freedesktop.org/xapps/release/xcompmgr-1.1.tar.gz.

[32] http://elinux.org/images/d/dc/Inside_Android's_User_Interface.pdf.

[33] http://www.android.com/.

[34] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX6Q.

[35] http://www.vivantecorp.com/index.php/en/technology/3d.html.

[36] http://cache.freescale.com/files/32bit/doc/data_sheet/IMX6DQIEC.pdf.

[37] http://www.linaro.org/.

[38] http://ltib.org.

All links were last followed on April 5, 2014.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature