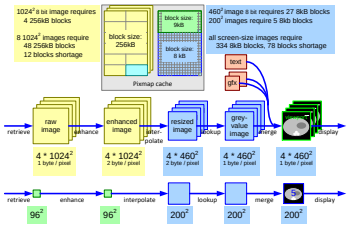
Medical Imaging Workstation: CR Views

-



Gerrit Muller
Buskerud University College
Frogs vei 41 P.O. Box 235, NO-3603 Kongsberg Norway
gaudisite@gmail.com

Abstract

The concepts and realization of the medical imaging workstation are described. The following concepts are described: presentation and processing pipeline, resource management (CPU and memory), including caching and anti-fragmentation strategy, software process decomposition and decomposition rules. The actual realization figures serve as illustration for the justification of some of the concepts.

version: 2.7 status: finished June 23, 2016

1 Introduction

The conceptual and realization views are described together in this chapter. The realization view, with its specific values, brings the concepts more alive.

Section 2 describes the processing pipeline for presentation and rendering, and maps the user interface on these concepts. Section 4 describes the concepts needed for memory management, and zooms in on how the memory management is used to implement the processing pipeline. Section 3 describes the software architecture. Section 5 describes how the limited amount of CPU power is managed.

The case material is based on actual data, from a complex context with large commercial interests. The material is simplified to increase the accessibility, while at the same time small changes have been made to remove commercial sensitivity. Commercial sensitivity is further reduced by using relatively old data (between 8 and 13 years in the past). Care has been taken that the value of the case description is maintained.

2 Image Quality and Presentation Pipeline

The user views the image during the examination at the console of the X-ray system, mostly to verify the image quality and to guide the further examination. Later the same image is viewed again from film to determine the diagnosis and to prepare the report. Sometimes the image is viewed before making a hardcopy to optimize the image settings (contrast, brightness, zoom). The user expects to see the same image at all work-spots, independent of the actual system involved.

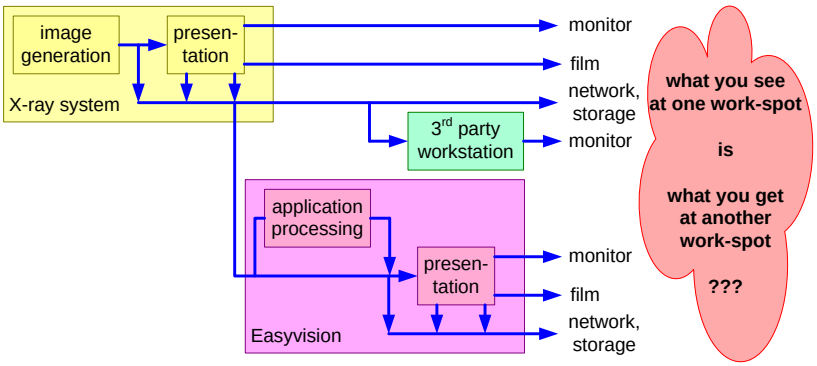


Figure 1: The user expectation is that an image at one work-spot looks the same as at other work-spots. This is far from trivial, due to all data paths and the many parties that can be involved

Figure 1 shows many different possible work-spots, with different media. The user expects *What You See Is What You Get* (WYSIWYG) everywhere. From an

implementation point of view this is far from trivial. To allow optimal handling of images at other locations most systems export images halfway their internal processing pipeline: acquisition specific processing is applied, rendering specific processing is not applied, but the rendering settings are transferred instead. All systems using these intermediate images need to implement the same rendering in order to get the same image perception. The design of these systems is strongly coupled, due to the shared rendering know-how.

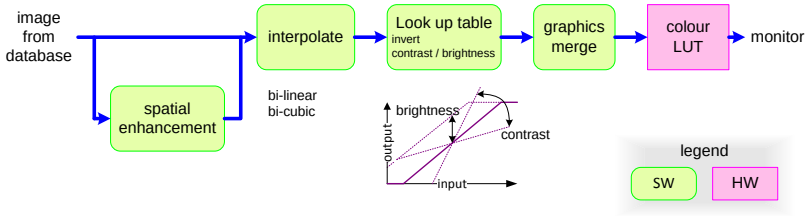


Figure 2: The standard presentation pipeline for X-ray images

Figure 2 shows the rendering pipeline as used in the medical imaging workstation. Enhancement is a filter operation. The coefficients of the enhancement kernel are predefined in the acquisition system. The interpolation is used to resize the image from acquisition resolution to the desired view-port (or film-port) size. The grey-levels for display are determined by means of a lookup table. A lookup table (LUT) is a fast and flexible implementation of a mapping function. Normally the mapping is linear: the slope determines the contrast and the vertical offset the brightness of the image. Finally graphics and text are superimposed on the image, for instance for image identification and for annotations by the user.

The image interpolation algorithm used depends on desired image quality and on available processing time. Bi-linear interpolation is an interpolation with a low-pass filter side effect, by which the image becomes less sharp. An ideal interpolation is based on a convolution with a sinc-function $(sin(x)/x)$. A bi-cubic interpolation is an approximation of the ideal interpolation. The bi-cubic interpolation is parameterized. The parameter settings determine how much the interpolation causes low pass or high pass filtering (blurring or sharpening). These bi-cubic parameter choices are normally not exported to the user interface, the selection of values requires too much expertise. Instead, the system uses empirical values dependent on the interpolation objective.

The monitor screen is a scarce resource of the system, used for user interface control and for the display of images. The screen is divided in smaller rectangular windows. Windows displaying images are called view-ports. Every view-port uses its own instantiation of a viewing pipeline. Figure 3 shows an example of a screen layout, viewing four images simultaneously. At the bottom left a fifth view-port is used for navigational support, for instance in case of zooming this view-port

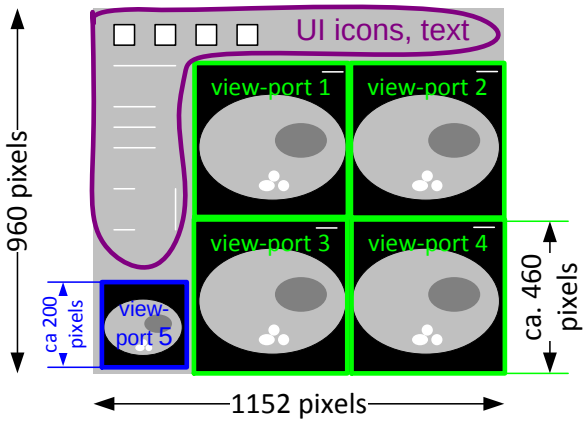


Figure 3: Quadruple view-port screen layout

functions as a roadmap, enabling direct manipulation of the zoom-area. The fifth view-port also has its own viewing pipeline instance.

The concepts visible in this screen layout are view-ports, icons, text, an image area (with the 4 main view-ports), and a user interface area with navigation support. The figure adds a number of realization facts, such as the total screen-size, and the size of the view-ports. The next generation of this system used the same concepts, but the screen size was 1280*1024, resulting in slightly larger view-ports and a slightly larger ratio between image area and user interface area.

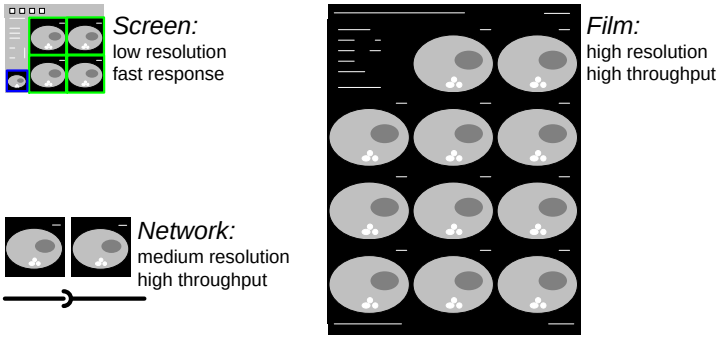


Figure 4: Rendered images at different destinations

At all places where source images have to be rendered into viewable images an instance of the presentation pipeline is required. Note that the characteristics of the usage of the presentation pipeline in these different processes vary widely. Figure 4 shows three different destinations for rendered images, with the different

usage characteristics.

3 Software Specific Views

The execution architecture of Easyvision is based on UNIX-type processes and shared libraries. Figure 5 shows the process structure of Easyvision. Most processes can be associated with a specific hardware resource, as shown in this figure. Core of the Easyvision software architecture is the database. The database provides *fast*, *reliable*, *persistent* storage and it provides *synchronization* by means of active data. The concept of active data is based on the *publish-subscribe pattern* [1] that allows all users of a some information to be notified when changes in the information occur. Synchronization and communication between processes always takes place via this database.

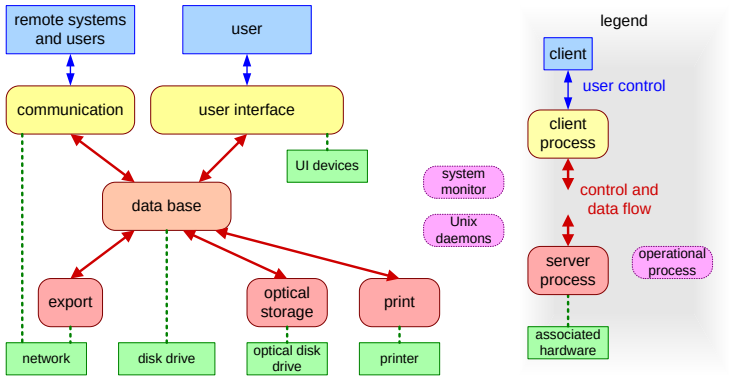


Figure 5: Software processes or tasks running concurrently in Easyvision

Figure 5 shows four types of processes: *client processes*, *server processes* *database process*, and *operational processes*. A client interacts with a user (remote or direct), while the servers perform their work in the background. The database connects these two types of processes. Operational processes belong to the computing infrastructure. Most operational processes are created by the operating system, the so called daemons. The system monitoring processes is added for exception handling purposes. The system monitor detects hanging processes and takes appropriate action to restore system operation.

A process as unit of design is used for multiple reasons. The criteria used to determine the process decomposition are:

management of concurrency Activities that are concurrent run in separate processes.

management of shared devices A shared device is managed by a server process.

unit of memory budget Measurement of memory use at process level is supported by multiple tools.

unit of distribution over multiple processors A process can be allocated to a processor, without the need to change the code within the process.

unit of exception handling Faults are contained within the process boundaries. The system monitor observes at process level, because the operating system provides the means at process level.

Manageability, visibility and understandability benefit from a limited number of processes. One general rule is to minimize the amount of processes, in the order of magnitude of ten processes.

The presentation pipeline, as depicted in Figure 2, is used in the *user interface* process, the *print* server and the *export server*.

Figure 6 shows the software from the dependency point of view. Software in higher layers depends on, has explicit knowledge of, lower layers of the software. Software in the lower layers should not depend on, or have explicit knowledge of software in higher layers.

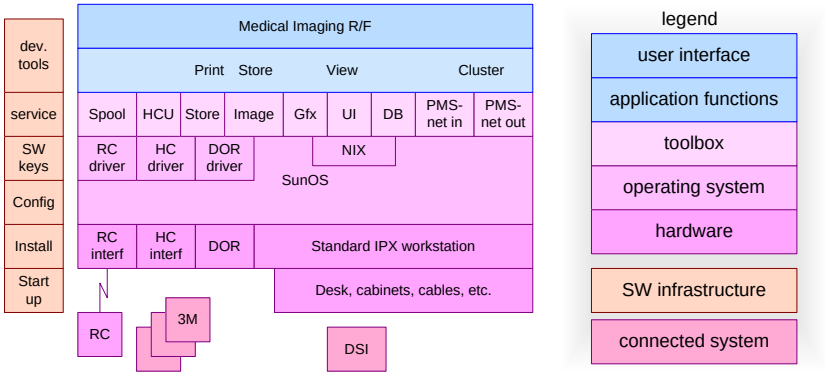


Figure 6: Simplified layering of the software

The caption of Figure 6 explicitly states this diagram to be simplified. The original design of this software did not use the layering concept. The software has been restructured in later years to make the dependency as layering explicit. The actual number of layers based on larger packages did exceed 15. Reality is much more complex than this simplified diagram suggests.

4 Memory Management

The amount of memory in the medical imaging workstation is limited for cost reasons, but also for simple physical reasons: the workstation used at that moment

did not support more than 64 MByte of physical memory. The workstation and operating system did support virtual memory, but for performance reasons this should be used sparingly.

	code		object data		bulk data		total	
memory budget in Mbytes	R1	R2	R1	R2	R1	R2	R1	R2
shared code	6.0	11.0					6.0	11.0
UI process	0.2	0.3	2.0	3.0	12.0	12.0	14.2	15.3
database server	0.2	0.3	4.2	3.2		3.0	4.4	6.5
print server	0.4	0.3	2.2	1.2	7.0	9.0	9.6	10.5
DOR server	0.4	0.3	4.2	2.0	2.0	1.0	6.6	3.3
communication server	1.2	0.3	15.4	2.0	10.0	4.0	26.6	6.3
UNIX commands	0.2	0.3	0.5	0.2			0.7	0.5
compute server		0.3		0.5		6.0		6.8
system monitor		0.3		0.5				0.8
application total	8.6	13.4	28.5	12.6	31.0	35.0	66.1	61.0
UNIX							7.0	10.0
file cache							3.0	3.0
total							76.1	74.0

Figure 7: Memory budget of Easyvision release 1 and release 2

A memory budget is used to manage the amount of memory in use. Figure 7 shows the memory budgets of release 1 and release 2 of Easyvision RF side by side. Three types of memory are distinguished: *program or code*, read-only from operating system point of view, *object data*, dynamically allocated and deallocated in a heap-based fashion, and *bulk data* for large consecutive memory areas, mostly used for images.

Per process, see Section 3, the typical amount of memory per category is specified. The memory usage of the operating system is also specified. The dynamic libraries, that contain the code shared between processes, is explicitly visible in the budget.

The figure shows the realization for two successive releases, for which we can observe that the concepts are stable, but that the realization changes significantly. Release 1 used a rather straightforward communication server, operating on all import streams in parallel, keeping everything in memory. This is very costly with respect to memory. R2 serializes the memory use of different import streams and uses the memory in a more pipelined way. These changes result in a significant reduction of the memory being used. In the same time frame the supplier dictated a new operating system, SunOS was end-of-life and was replaced by Solaris 2. This had a negative impact on the memory consumption; the budget shows an increase of 7 MByte to 10 MByte for the UNIX operating system.

The decomposition in object data and bulk data is needed to prevent memory fragmentation. Fragmentation of memory occurs when the allocation is dynamic with different sizes of allocated memory over time. The fragmentation increases

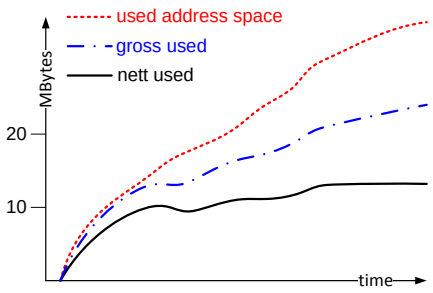


Figure 8: Memory fragmentation increase. The difference between gross used and nett used is the amount of unusable memory due to fragmentation

over time. Due to the paging of the virtual memory system not all fragmentation is disastrous. Figure 8 shows the increase of the amount of memory over time. The net amount of memory stabilizes after some time, but the gross amount of memory increases due to ongoing fragmentation. The amount of virtual memory in use (and the address space) is increasing even more, however a large part of this virtual memory is paged out and is not really a problem.

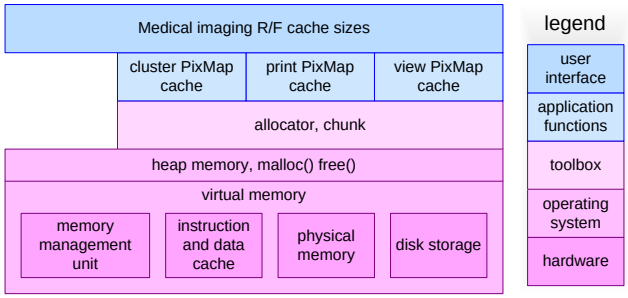


Figure 9: Cache layers at the corresponding levels of Figure 6

The hardware and operating system support fast and efficient memory-based on hardware caching and virtual memory, the lowest layer in Figure 9. The application allocates memory via the heap memory management functions malloc() and free(). From an application point of view a sheer infinite memory is present, however the speed of use depends strongly on the access patterns. Data access with a high locality are served by the data cache, which is the fastest (and smallest) memory layer. The next step in speed and size (slower, but significantly larger) is the physical memory. The virtual memory, mostly residing on disk, is the slowest but largest memory layer.

The application software does not see or control the hardware cache or virtual memory system. The only explicit knowledge in the higher software layers of these

memory layers is in the dimensioning of the memory budgets as described later.

The toolbox layer provides anti-fragmentation memory management. This memory is used in a cache like way by the application functions, based on a *Least Recently Used* algorithm. The size of the caches is parameterized and set in the highest application layer of the software.

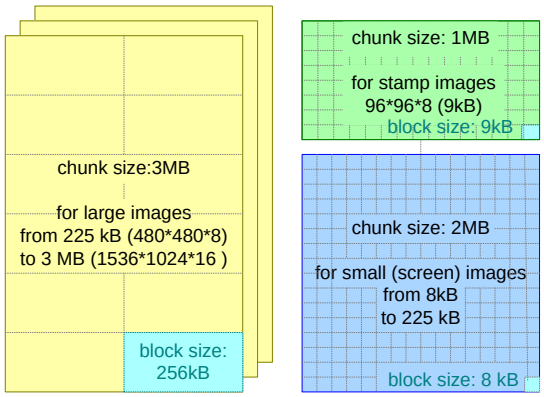


Figure 10: Memory allocators as used for bulk data memory management in Easyvision RF

The medical imaging workstation deploys pools with fixed size blocks to minimize fragmentation. A two level approach is taken: pools are allocated in large *chunks*, every chunk is managed with fixed size *blocks*. For every chunk is defined which bulk data sizes may be stored in it.

Figure 10 shows the three chunk sizes that are used in the memory management concepts chunks, block sizes and bulk data sizes as used in Easyvision RF. One chunk of 1 MByte is dedicated for so-called *stamp* images, 96*96 down scaled images, used primarily for visual navigation (for instance pictorial index). The block size of 9 kbytes is exactly the size of a stamp image. A second chunk of 3 MBytes is used for large images, for instance images with the original acquisition resolution. Small images, such as images at display resolution, will be allocated in the third chunk of 2 MBytes. The dimensioning of the block and chunk sizes is based on a priori know-how of the application of the system, as described in Section **??**. The block sizes in the latter two chunks are 256 kbytes for large images and 8 kbytes for small images. These block sizes result in balanced and predictable memory utilization and fragmentation within a chunk.

The chunks are used with cache like behavior: images are kept until the memory is needed for other images. Figure 11 shows the cached intermediate results. This figure is a direct transformation of the viewing pipeline in Figure 2, with the processing steps replaced by arrows and the data-arrows replaced by stores. In Section 5 the gain in response time is shown, which is obtained by caching the

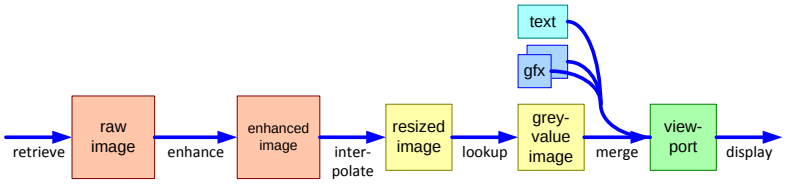


Figure 11: Intermediate processing results are cached in an application level cache

intermediate images.

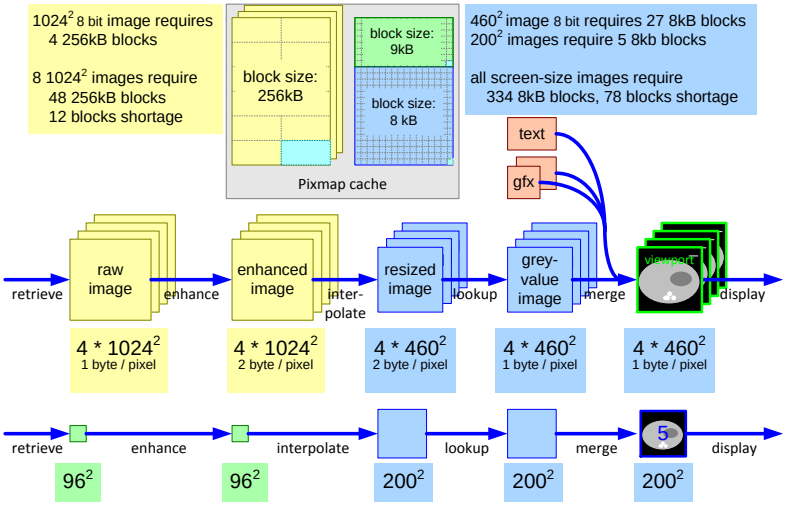


Figure 12: Example of allocator and cache use. In this use case not all intermediate images fit in the cache, due to a small shortage of blocks. The performance of some image manipulations will be decreased, because the intermediate images will be regenerated when needed.

Figure 12 shows how the *chunks* are being used in quadruple viewing (Figure 3). The 1024^2 images with a depth of 1 or 2 bytes will be stored in the 3 MB chunks. The smaller interpolated images of 460^2 will go into the 2 MB chunks, requiring 27 blocks of 8kB for an 1 byte pixel depth or 54 blocks for 2 2 bytes per pixel. Also the screen size images of the navigation view-port fall in the range that maps on the 2 MB chunk, requiring 5 blocks per 200^2 image.

Everything added together requires more blocks than available in the 2 and 3 MB chunks. The cache mechanism will sacrifice the least recently used intermediate results.

For memory and performance reasons the navigation view-port is using the stamp image as source image. This image, which is shown in a small view-port at

the left hand side of the screen, is only used for navigational support of the user interface. Response time is here more important than image quality.

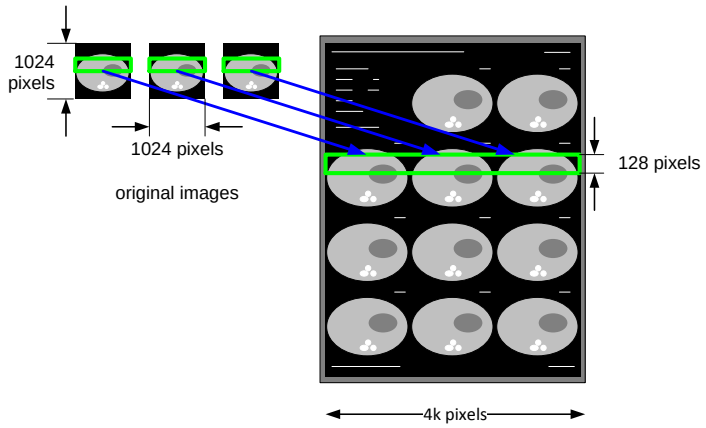


Figure 13: Print server is based on different memory strategy, using bands

The print server uses a different memory strategy than the user interface process, see Figure 13. The print server creates the film-image by rendering the individual images. The film size of 4k*5k images is too large to render the entire film at once in memory: 20 Mpixels, while the memory budget allows 9 Mbyte of bulk data usage. The film image itself is already more than the provided memory budget!

The film image is built up in horizontal bands, which are sent to the laser printer. The size of the stroke is chosen such that input image + intermediate results + 2 bands (for double buffering) fit in the available bulk data budget. At the same time the band should not be very small because the banding increases the overhead and some duplicate processing is sometimes needed because of edge effects.

The print server uses the same memory management concepts as shown in the figure with cache layers, Figure 9. However the application level caching does not provide any significant value for this server usage, because the image data flow is straightforward and predictable.

5 CPU Usage

The CPU is a limited resource for the Easyvision. The performance and throughput of the system depend strongly on the available processing power and the efficiency of using the processing power. CPU time and memory can be exchanged partially, for instance by using caches to store intermediate results.

Figure 14 shows typical update speeds and processing times for a single image user interface layout. Contrast brightness (C/B in the figure) changes must be fast,

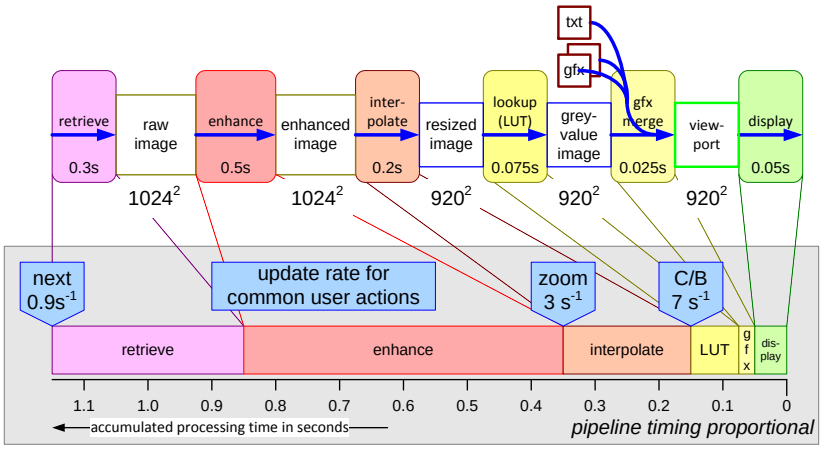


Figure 14: The CPU processing times are shown per step in the processing pipeline. The processing times are mapped on a proportional time line to visualize the viewing responsiveness

to give immediate visual feedback when turning a contrast or brightness wheel. Working on the cached resized image about 7 updates per second are possible, which is barely sufficient. The gain of the cached design relative to the non-cached design is about a factor 8 (7 updates per second versus 0.9 updates per second). Zooming and panning is done with an update rate of 3 updates per second. The performance gain for zooming and panning is from application viewpoint less important, because these functions are used only exceptionally in the daily use.

Retrieving the next image (also a very frequent user operation), requires somewhat more than a second, which was acceptable at that moment in time. This performance is obtained by slightly compromising the image quality: a bilinear interpolation is used for resizing, instead of the better bi-cubic interpolation. For the monitor, with its limited resolution this is acceptable, for film (high resolution, high brightness) bi-cubic interpolation is required.

For background tasks a CPU budget is used, expressed in CPU seconds per Mega-byte or Mega-pixel. This budget is function-based: importing and printing. Most background jobs involve a single server plus interaction with the database server.

Two use cases are relevant: interactive viewing, with background jobs, and pure print serving. For interactive response circa 70% of CPU time should be available, while the load of printing for three examination rooms, which is a full throughput case, must stay below 90% of the available CPU time. Figure 15 shows the load for serving a single examination room and for serving three examination rooms. Serving a single examination room takes 260 seconds of CPU time per examination of 15 minutes, leaving about 70% CPU time for interactive viewing.

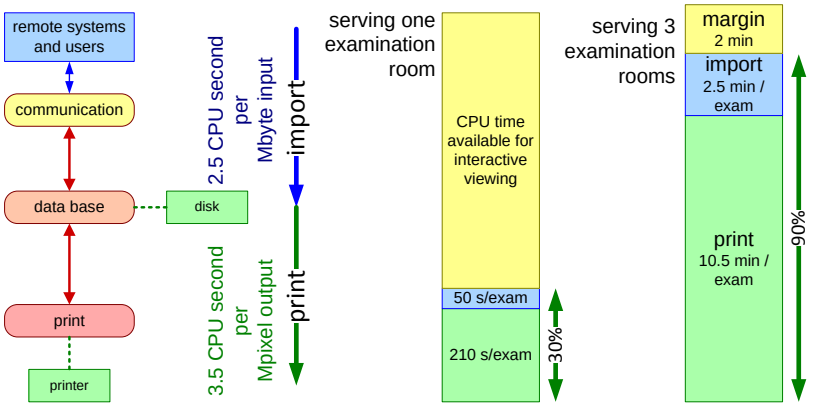


Figure 15: Server CPU load. For a single examination room sufficient CPU time is left for interactive viewing. Serving three examination rooms fits in 90% of the available CPU time.

Serving three examination rooms takes 13 minutes of CPU time per 15 minutes of examinations, this is just below the 90%.

6 Measurement Tools

The resource design as described above is supported in the implementation by means of a few simple, but highly effective measurement tools. The most important tools are: *Object Instantiation Tracing*, *standard Unix utilities* and a *heap viewer*.

The resource usage is measured at well defined moments in time, by means of events. The entire software is event-based. The event for resource measurement purposes can be fired by programming it at the desired point in the code, or by a user interface event, or by means of the Unix command line.

The resource usage is measured twice: before performing the use case under study and afterwards. The measurement results show both the changes in resource usage as well as the absolute numbers. The initialization often takes more time in the beginning, while in a steady running system no more initialization takes place. Normally the real measurement is preceded by a set of actions to bring the system in a kind of steady state.

Note that the budget definitions and the *Unix utilities* fit well together, by design. The types of memory budgeted are the same as the types of memory measured by the Unix utilities. The typically used Unix utilities are:

ps process status and resource usage per process

vmstat virtual memory statistics

kernel resource stats kernel specific resource usage

The *heap-viewer* shows the free and allocated memory blocks in different colors, comparable with the standard Windows disk defragmentation utilities.

class name	current nr of objects	deleted since t_{n-1}	created since t_{n-1}	heap memory usage
AsynchronousIO	0	-3	+3	
AttributeEntry	237	-1	+5	
BitMap	21	-4	+8	
BoundedFloatingPoint	1034	-3	+22	
BoundedInteger	684	-1	+9	
BtreeNode1	200	-3	+3	[819200]
BulkData	25	0	1	[8388608]
ButtonGadget	34	0	2	
ButtonStack	12	0	1	
ByteArray	156	-4	+12	[13252]

Figure 16: Example output of OIT (Object Instantiation Tracing) tool

The *Object Instantiation Tracing* (OIT) keeps track of all object instantiations and disposals. It provides an absolute count of all the objects and the change in the number of objectives relative to the previous measurement. The system is programmed with Objective-C. This language makes use of run-time environment, controlling the creation and deletion of objects and the associated housekeeping. The creation and deletion operations of this run-time environment were rerouted via a small piece of code that maintained the statistics per class of object instantiations and destructions. At the moment of a trigger this administration was saved in readable form. The few lines of code (and the little run time penalty) have paid many many times. The instantiation information gives an incredible insight in the internal working of the system.

The *Object Instantiation Tracing* also provided heap memory usage per class. This information could not be obtained automatically. At every place in the code where malloc and free was called some additional code was required to get this information. This instrumentation has not been completed entirely, instead the 80/20 rule was applied: the most intensive memory consumers were instrumented to cover circa 80% of the heap usage.

Figure 16 shows an example output of the OIT tool. Per class the current number of objects is shown, the number of deleted and created objects since the previous measurement and the amount of heap memory in use. The user of this tool knows the use case that is being measured. In this case, for example, the *next image* function. For this simple function 8 new BitMaps are allocated and 3 AsynchronousIO objects are created. The user of this tool compares this number with his expectation. This comparison provides more insight in design and implementation.

	test / benchmark	what, why	accuracy	when
public	SpecInt (by suppliers)	CPU integer	coarse	new hardware
	Byte benchmark	computer platform performance OS, shell, file I/O	coarse	new hardware new OS release
self made	file I/O	file I/O throughput	medium	new hardware
	image processing	CPU, cache, memory as function of image, pixel size	accurate	new hardware
	Objective-C overhead	method call overhead memory overhead	accurate	initial
	socket, network	throughput CPU overhead	accurate	ad hoc
	data base	transaction overhead query behaviour	accurate	ad hoc
	load test	throughput, CPU, memory	accurate	regression

Figure 17: Overview of benchmarks and other measurement tools

Figure 17 shows an overview of the benchmarking and other measurement tools used during the design. The overview shows per tool what is measured and why, and how accurate the result is. It also shows when the tool is being used.

The Objective-C overhead measurements, to measure the method call overhead and the memory overhead caused by the underlying OO technology, is used only in the beginning. This data does not change significantly and scales reasonably with the hardware improvements.

A set of coarse benchmarking tools was used to characterize new hardware options, such as new workstations. These tools are publicly available and give a coarse indication of the hardware potential.

The application critical characterization is measured by more dedicated tools, such as the image processing benchmark, which runs all the algorithms with different image and pixel sizes. This tool is home made, because it uses the actual image processing library used in the product. The outcome of these measurements were used to make design optimizations, both in the library itself as well as in the use of the library.

Critical system functionality is measured by dedicated measurement tools, which isolate the desired functionality, such as file I/O, socket, networking and database.

The complete system is put under load conditions, by continuously importing and exporting data and storing and retrieving data. This load test was used as regression test, giving a good insight in the system throughput and in the memory and CPU usage.

7 Conclusion

This chapter described several decompositions: a functional decomposition of the image processing pipeline, a construction decomposition in layers and a process decomposition of the software. The image quality, throughput and response time have been discussed and especially the design choices that have been made to achieve the desired performance level. The design considerations show that design choices are related to consequences in multiple qualities and multiple CAFCR views. Reasoning over multiple CAFCR views and multiple qualities is needed to find an acceptable design solution. All information presented here was explicitly available in product creation documentation.

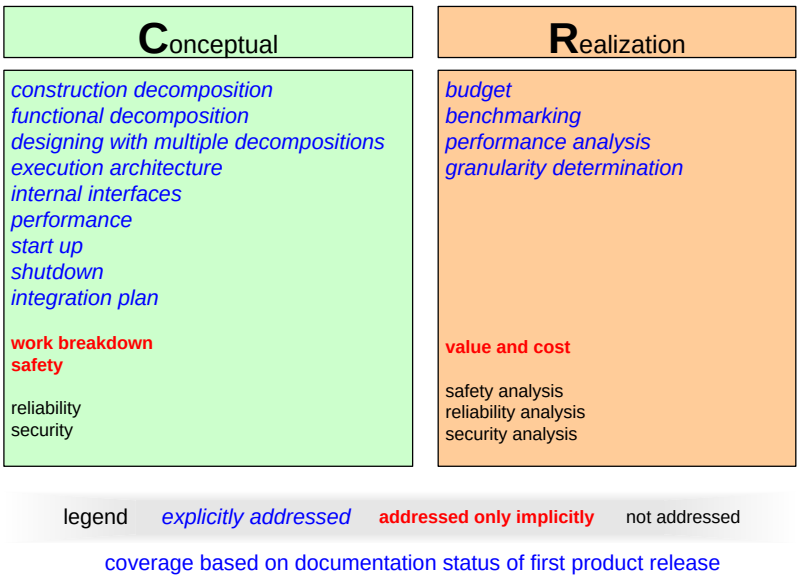


Figure 18: Coverage of submethods of the CR views

A number of submethods has not been described here, such as start up and shutdown, but these aspects are covered by the documentation of 1992. Figure 18 shows the coverage of the submethods described in part II by the documentation of the first release. This coverage is high for most submethods. Safety, reliability and security were not covered by the documentation in 1992, but these aspects were added in later releases of the product.

8 Acknowledgements

Martien Dijks provided input for the benchmarking figure.

References

[1] EventHelix.com. Publish-subscribe design patterns. http://www.eventhelix.com/RealtimeMantra/Patterns/publish_subscribe_patterns.htm, 2000.

[2] Gerrit Muller. The system architecture homepage. http://www.gaudisite.nl/index.html, 1999.

History

Version: 2.7, date: April 6, 2004 changed by: Gerrit Muller

- changed title

Version: 2.6, date: April 6, 2004 changed by: Gerrit Muller
- added description of view-port
- removed figure about memory fragmentation

Version: 2.5, date: March 17, 2004 changed by: Gerrit Muller
- moved sensitivity statement from figure to introduction.
- changed status to finished

Version: 2.4, date: February 27, 2004 changed by: Gerrit Muller
- added section “Conclusion”
- changed status to concept
- changed specialist in consultant

Version: 2.3, date: November 11, 2003 changed by: Gerrit Muller
- many text improvements
- moved criterions for processes from figure into the text.
- updated Figure ”CPU processing time”
- updated Figure ”Server CPU load”
- transformed Figure ”Resource measurement tools” to text
- added text for Figure ”Example OIT output”

Version: 2.2, date: November 5, 2003 changed by: Gerrit Muller
- many text improvements

Version: 2.1, date: September 9, 2003 changed by: Gerrit Muller
- the gain of the cache is made explicit

Version: 2.0, date: March 26, 2003 changed by: Gerrit Muller
- added printserver memory usage
- changed status into draft

Version: 1.4, date: March 13, 2003 changed by: Gerrit Muller
- updated figure benchmarking

Version: 1.3, date: March 10, 2003 changed by: Gerrit Muller
- added benchmarking to the section ”Measurement tools”

Version: 1.2, date: February 10, 2003 changed by: Gerrit Muller
- added section ”Measurement tools”

Version: 1.1, date: February 10, 2003 changed by: Gerrit Muller
- added description of bi-linear and bi-cubic interpolation
- added differentation of concepts and realization facts at several places
- moved section ”software” before ”memory managament”
- added simplified software layering
- added figure with cache layers

Version: 1.0, date: January 28, 2003 changed by: Gerrit Muller
- Added text to all sections
- added server CPU usage
- changed status in preliminary draft

Version: 0, date: January 22, 2003 changed by: Gerrit Muller
- Created, no changelog yet