

Real time hardware for a new 3D display

authors:

B. Kaufmann, M. Akil

Lab. *A²SI* - Groupe ESIEE, Cité Descartes - BP 99, 2 Bd Blaise Pascal, F-93162 Noisy-le-Grand Cedex, France

Abstract

We describe in this article a new multi-view auto-stereoscopic display system with a real time architecture to generate images of n different points of view of a 3D scene. This architecture generates all the different points of view with only one generation process, the different pictures are not generated independently but all at the same time. The architecture generates a frame buffer that contains all the voxels with their three dimensions and regenerates the different pictures on demand from this frame buffer. The need of memory is decreased because there is no redundant information in the buffer.

Keywords: multi-view, auto-stereoscopic display, real-time architecture

1 Introduction

Actually, there are four different ways to display 3D images:

- with a stereoscopic pair of images. Two images are displayed, one for each eye of the observer. To correctly perceive the three dimensions, the observer has to wear a special device which makes each eye see only the correct picture. This device can be goggles which display each picture in front of each eye of the observer (ie. Charles Wheatstone's [12] or David Brewster's [3] stereoscopes, Head-mounted devices, etc.) or special goggles which make each eye see one particular picture out of the two ones projected onto the same screen (ie. Charles d'Almeida's anaglyph, Laurens Hammond and William F. Cassidy's Televue system, Edwin Herbert Land's polarized filters, etc.)

These systems has the advantage to display images which are easy to make and easy to see. But they have the disadvantage to force the observer(s) to wear special device.

- Auto-stereoscopic [6] display systems avoid the wearing of special devices or goggles. They allow the observer to see the 3D image by simply standing at the front of the screen. This can be done with several different techniques, ie. lenticular screens [10], parallax barrier [5], or time-multiplexing image projection [1], etc.

Actually, many auto-stereoscopic display systems exist or are in development phase, but those systems have the two main disadvantages that the number of different points of view are limited for technological reasons and the images are visible only from a certain distance from the screen because of the convergence needed for the observer to see the whole picture.

- Volumetric display systems project multiple images onto multiple [2] or one unique moving plane [11] or helix [7] screen.

These systems have the advantage of being observed from any position in front of or around the display system, but have the disadvantages of only being able to display objects inside the volume delimited by the display system itself and to not be able to display non-transparent objects.

- Holographic display systems which have the advantages of all the other systems, but have the main disadvantage that they need a large number of computing to display a single image. They can't, until now, be used in real time imaging.

Actually, the most effective of these systems is the CAM3D [9] system developed by Cambridge University. This system provides an infinite theoretical number of points of view and allows the observer to move freely in front of the screen.

This article proposes a new 3D display system based upon auto-stereoscopy which also provides an infinite theoretical number of points of view and allows the observer to move freely in front of the screen, but our system

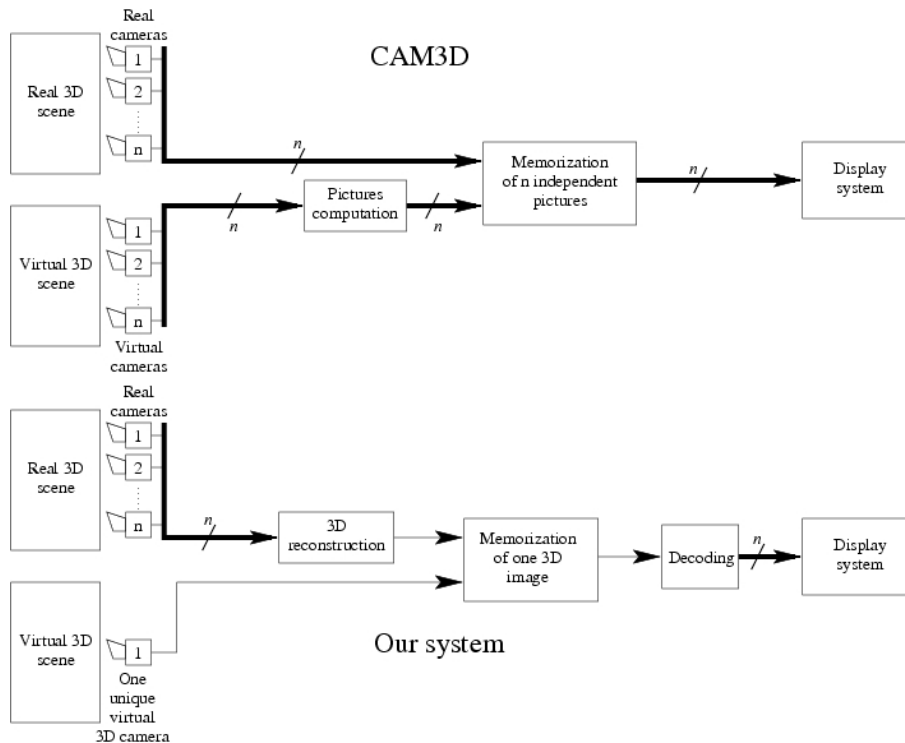


Figure 1: a new 3D display based upon auto-stereoscopy

is based on a dedicated architecture which provides real-time generation of the different points of view, without computing and memorizing independently every picture. The main difference with the CAM3D system is the compression of the 3D data as voxels, without computing explicitly one picture for each possible point of view. Figure 1 shows the parallel between the CAM3D system and ours.

We will present the method used to generate all the different points of view in real time with only one image computation and the algorithms we use, the architecture of the whole processing chain. We will insist on the coding and decoding of the 3D data which permit us to store and reconstruct all the different views. We will finally show how our algorithms can be integrated into the existing hardware accelerated graphics interfaces.

2 AUTO-STEREOSCOPY SYSTEM: 3D VISUALISATION METHOD AND IMAGE PROCESSING CHAIN

2.1 3D Image processing algorithms

This system intends to be a real time 3D computation system which is able to display a large number of different independent points of view. In order to provide a data flow fast enough to ensure the real time condition, we need to use an hardware-accelerated architecture. This architecture is in charge of generating the suitable images to be posted on the 3D screen from the 2D and 3D data sent by the host computer. It provides two main parallel pipelines:

- One of these pipelines is the 3D pipeline. It computes the 3D informations of the objects to display and the 3D environment, ie. coordinates, light, textures, etc. This pipeline is very similar to those used in the actual commercial 3D hardware accelerated graphics cards, at least in its first stages.
- The second pipeline will be perceived as a “classical” 2D interface, but the 2D data will not be memorized and displayed as 2D data. Because our display system is only able to display 3D data, the 2D data need to be transformed into 3D data before being sent to the display system.

The 3D data are sent directly to the processing chain as they are sent to commercial hardware accelerated graphics cards. They are then redirected to the 3D pipeline.

The first stage of this pipeline is the “3D geometry stage”. It will transform the coordinates of the objects and will compute geometrical informations about where and how (lightning, position of the textures, clipping...)

the objects must be displayed on the screen. The algorithms of this stage are the same as those actually used in the 3D graphics cards. The only difference is the parameters of the clipping algorithm which must be adapted to accept a wider viewing area that includes all the possible visions of the different possible points of view. This means that, to be able to reconstruct all the possible points of view, in particular on the sides of the viewing zone, the projection of the objects must be operated on a larger virtual screen. The clipping must be adapted to this by enlarging the virtual screen.

Then the vertices are sent to the second stage of the pipeline which is the rasterization stage. This stage will compute geometrical data to generate the pixels to be displayed. It is mainly composed of the pixel and texture rendering engines, which are the same as those used in 3D graphics cards; and the anti-aliasing engine, which will generate new semi-transparent pixels without writing them directly into the graphics memory: they are sent to the next stage of the pipeline as regular voxels to be coded.

At the end of the 3D pipeline, voxels will be coded with an algorithm that allows us to generate a large number of different points of view with only one reduced-sized frame buffer [8].

The 2D data are sent directly to the 2D pipeline which is only composed of two stages:

- the first one will act as a “classical” 2D graphics card. It will memorize 2D pixels to be displayed and will provide registers to control the display settings. but the data will not “simply” be stored into the frame buffer, they will be synchronously memorized in the 2D memory and sent to the second stage.
- the second stage will recode these pixels with the same algorithm as the one coding the 3D voxels: the 2D voxels are transformed into 3D data to be displayed by the 3D-only display system. The algorithm used is very simple: the 2D data are separated into 2D lines. Each line will be transformed into a 3D line which contains one plane with a predefined depth (the depth of the screen, or else) which contains one patterns which contains all the pixels of the 2D line.

At the end, the 2D and 3D voxels, coded with the same 3D coding algorithm, sent by the two pipelines are merged to generate one only frame buffer that contains all the 2D and 3D pixels.

2.2 Overall hardware architecture of the 3D processing chain

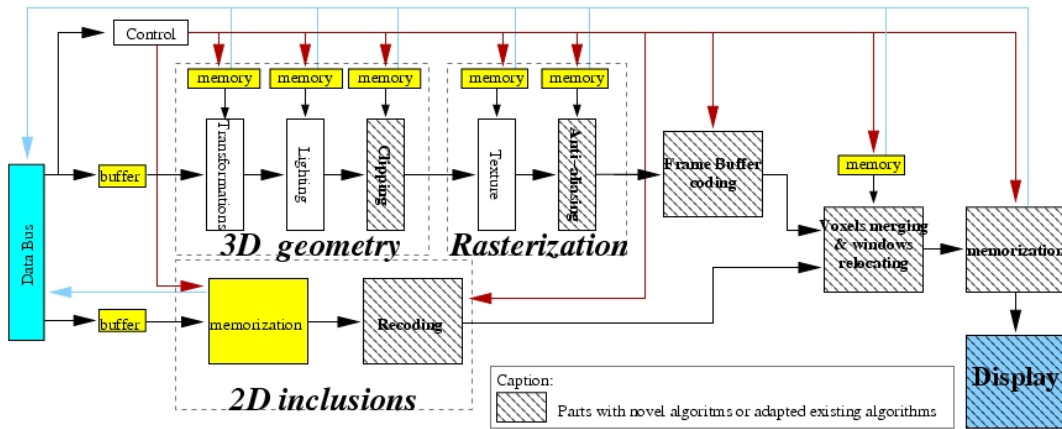


Figure 2: 3D image processing for auto-stereoscopy system

To be considered as a real time 3D computation system, our processing chain must be capable to generate and display at least 30 images/sec with a size of 1024×768 with 24 bits per pixel or each direction corresponding to the possible points of view of the observers. We are actually planning to reach 200 different directions, which leads the system to display $30 \times 200 = 6.000$ different images per second. 200 is the number of different angles needed to obtain the stereoscopic vision for an observer at distance of 2 meters (aprox. 0.5° between his two eyes) and with a viewing area of 100 degrees. With a normally complex image, the system needs to use a dedicated hardware, at least for part of it.

Figure 2 shows an overall view of the processing chain. It shows the two parallel pipelines: the one for the 3D data and the one for the conversion of the 2D data to displayable 3D elements, and the interface with the host computer and the control.

The whole processing chain will be controlled by the host computer through the control stage. This control is similar to a memory access: the host will simply read and write three different types of memory data. These types are:

- hardware informations. This type of memory is only readable by the host, not writable. It informs the host about the type of screen (i.e. plug-and-play support) is connected, the size of the frame buffer, the valid screen resolutions, etc.
- registers of the processing chain. These registers modify the compartment of the different stages or the rendering parameters. Typically, these values determine the resolution of the output screen, the refreshment frequency, the depth of the pixels, etc.
- input data. This type of memory allows the access to some of the working memories: input memory, where data are read before processing, result memory where computed data are stored for displaying and the 2D memory where 2D image is stored, which can be accessed for reading and writing. The input memory contains data like transformation matrix of the virtual camera, locations and color of the light sources in the virtual scene, textures to be applied, etc. The result memory can be accessed for reading the result of the computation in the frame buffer or for writing a previously read frame buffer into the hardware frame buffer to display it.

The data to be displayed are sent by the host computer through the data bus interface, which can be PCI, PCIExpress, AGP or other bus interface. The command request sent by the host is decoded and the data are sent to the corresponding stage.

As described above, the two first stages of the 3D pipeline (3D geometry and rasterization) are hardly the same as those implemented in commercial 3D accelerated graphics cards. There is no transformation needed to adapt the 3 stages that make the 3D geometry stage (Transformation, lightning and clipping stages) usable in this processing chain. The only adaptation needed is the clipping parameters which has to be adapted to the larger width of the viewing area. The second stage of this pipeline (rasterization) is also similar. The pixel generation and the texturing (including computation of the lightening and shadows) algorithms can be used without modification. The anti-aliasing must be adapted: the new pixels must not be merged with the already computed pixels into the frame buffer, but stays as new semi-transparent pixels, which must be coded as “classical” pixels. The hidden pixels erasing stage must be removed because it classically uses the Z-buffer algorithm [4] which is not adapted with the reconstruction of multiple points of view, because it removes pixels that could be seen by the user from several points of view. The erasing of the hidden voxels will be performed by the Frame Buffer coding stage. The last stage of the 3D pipeline will recode the voxels in a way which will allow the system to recreate the different views in a very little time, ie. which will allow the system to display all the images fast enough to be real time: up to 200 points of view \times 30 frames per second = 6,000 images displayed per second.

3 3D VOXELS PROCESSING STAGE

3.1 3D Voxels coding algorithm

To display the different views in real time, we need to be able to display a large number of different images in a very fast time. ie. 6,000 images per second. To do this, we need to memorize the images and code them with an algorithm which doesn't need a long time to decompress those images. We developed such an algorithm [8]: it is based on the idea that if you take several points of view of the same object and from the same side, most of the pixels of each resulting picture will be present in many of the others. The idea is to avoid the repetition of such pixels, coding all them once as voxels. Each voxel will be stored with its coordinates where it must be displayed on the screen for one of the views and its depth coordinate. Because the projection has already been done, it is very easy to reconstruct each view by simply copying the voxels at its correct location. This location can be easily computed with the initial position of the pixel in the reference view, shifted according to its depth and the view to be reconstructed.

This algorithm will store the pixels for each line of the screen, independently. Each line is divided into planes. Each plane represents a set of voxels of a line with the same depth. The depth is the discretized signed distance between the voxel and the screen. This discretization depends on the display system. Each line is composed of planes that contain at least one voxel. Empty planes are not coded to reduce unneeded memory use. Inside the planes, voxels are grouped in patterns. These patterns are stored from left to right and cannot overlap each other. A pattern is a horizontal set of contiguous voxels. This way, regions of the space that do not contain voxels do not consume memory. At last, each voxel corresponds to the 4 components (red, green, blue and transparency) of each pixel to be drawn. The data will be stored in the frame buffer the following way:

For each line of the screen

- Number of used planes in this line

- For each plane of this line
 - Depth of the plane (discretized signed distance between the voxel and the screen)
 - Number of patterns present in this plane
 - For each pattern of this plane
 - * Horizontal position of the far left voxel of the pattern, measured from the left limit of the visible area
 - * Number of voxels present in this pattern
 - * For each voxel of this pattern
 - ARGB component of the voxel
 - * End For
 - End For
- End For

End For

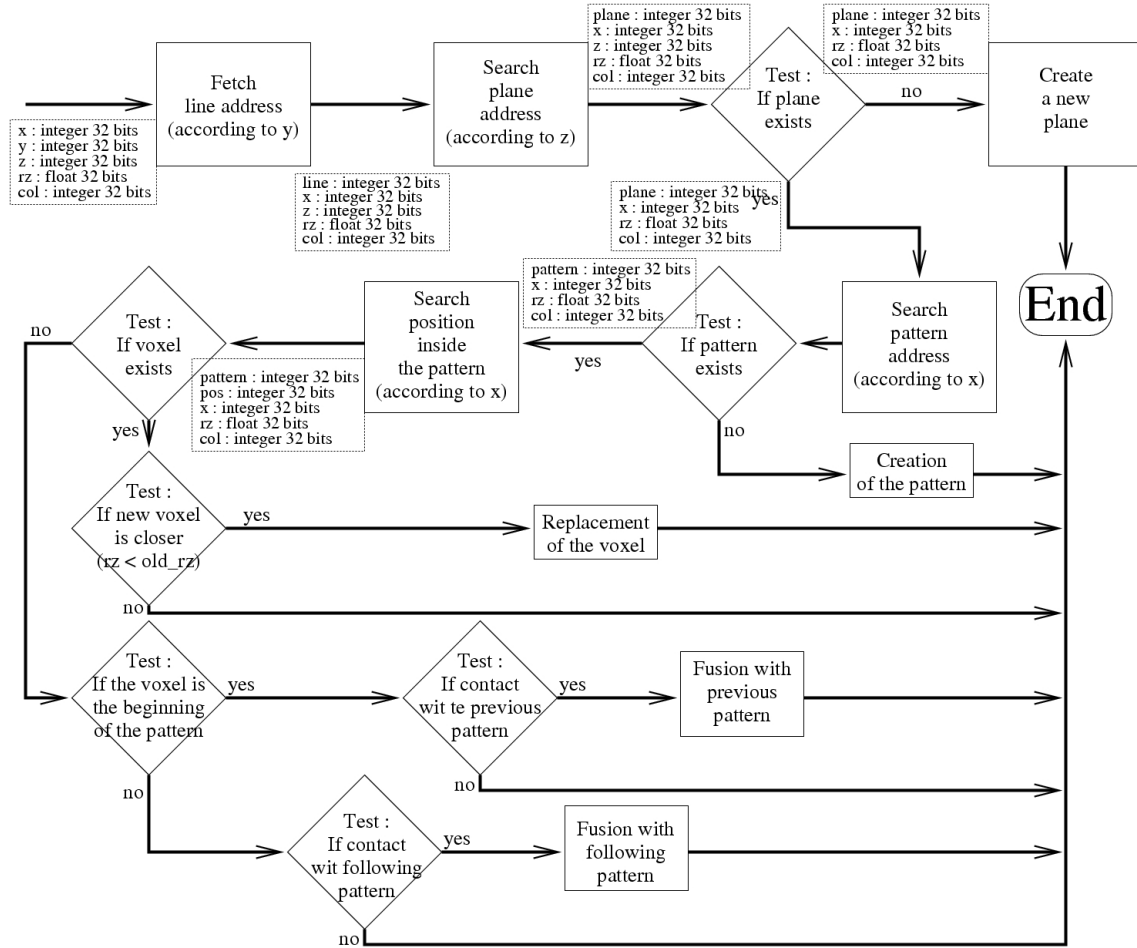


Figure 3: 3D voxels processing stage

The coding algorithm (schematized on figure 3) follows: the Y coordinate of each voxel to be coded determines the line in which it must be inserted. If the line contains no used planes, a new plane is created in this line with the same depth (Z coordinate) as the voxel. If it contains planes, a plane with the matching depth is searched. If one is found, the voxel is inserted into it. If none corresponds, a new one is created.

To insert a voxel into a plane, a corresponding pattern inside this plane must be found. The corresponding pattern is determined with the equation:

$$b_p - 1 \leq x_v \leq e_p + 1$$

where b_p is the coordinate of the first voxel of the pattern,
 x_v is the X coordinate of the voxel, and
 e_p is the coordinate of the last voxel of the pattern (the coordinate of the first voxel of the pattern + the number of voxels in the pattern)
If none can be found, a new one is created. If the voxel can be inserted into an existing pattern, three cases can occur:

- the voxel is inside the pattern, where another voxel is already present. Then the real depths (not the discretized ones which determine the plane in which the voxels must be inserted) of the two voxels are compared and only the closer is kept.
- the voxel is between two patterns and touches them both. (ie. it can be inserted in them both) Then the voxel is inserted into the first one and the two patterns are merged to create one only pattern in the middle of which the voxel is.
- the voxel is beside the pattern and does not touch another one. Then the voxel is simply added to it.

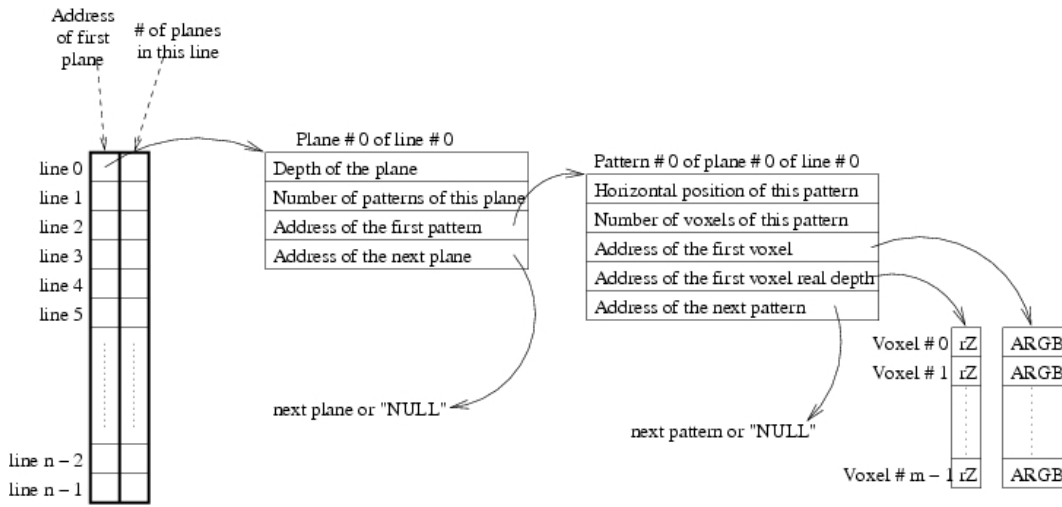


Figure 4: Voxels coding: data structure

There are two main ways to implement the corresponding coding algorithm. The first one uses chained lists to store depths and patterns, as shown on figure 4. It has a time complexity in $o(n \times m)$ (where n is the average number of patterns in each line of the frame buffer and m is the average size of the patterns) and a size need (for the working memory) of $o(n \times m \times 24 \text{ bits} + n \times m \times l \times s)$ (where n is the number of patterns in all the lines and s is the size of a pointer). You can also code the voxels inside a patterns as a chained list to fasten the coding, but it increases the needed size. The second one uses directly the final coding. The size of the needed memory is limited to the size of the final frame buffer but the complexity is $o(S^2)$, where S is the size of the frame buffer. This is a simplified version of the algorithm that is defined here. The full version includes a post-processing which erases voxels hidden by other ones and a mechanism to avoid frame buffer overflows. The needed size of the final frame buffer is not determined and does not only depend on the quality of the displayed pictures, but it also depends on the complexity of the scene to be rendered. Its size must be, at least, the size of the 2D frame buffer for a 2D image of the same resolution. For example, a picture of $1,024 \times 768 \times 24 \text{ bits/pixel}$ will take about 2.3 MB (MB = Mega Bytes, $1 \text{ MB} = 8 \times 2^{20} \text{ bits}$). So this frame buffer will be of a size of about 10 MB, at least. The voxel processing must store data at $10 \text{ MB} \times 30 \text{ images per second} = 300 \text{ MB per second}$ at least to assume real time imaging.

3.2 Dedicated architecture for 3D voxels coding

We chose to implement the coding algorithm using chain lists. The hardware architecture we developed to implement this algorithm consists of a four stages pipeline and an independent recoding stage to convert the chain lists to the final form, with a unique data bus and a unique memory in which the chain lists containing the data are stored.

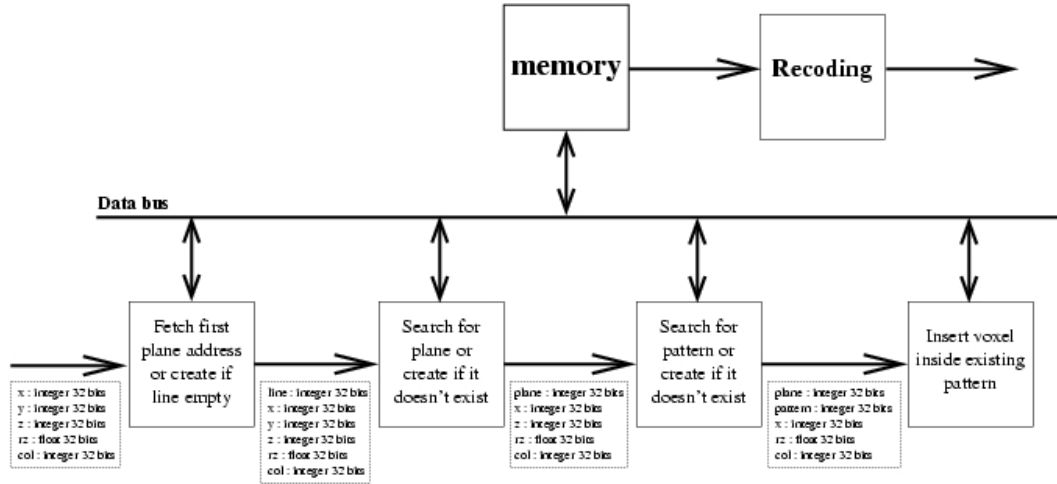


Figure 5: Voxels coding: data and control paths

The voxels to be coded are sent to this pipeline with their 3D coordinates and its ARGB value. The first stage of the pipeline reads the memory to get the address where the first plane of the corresponding line is located and send it to the second stage if it exists. If it doesn't exist, a new plane is created and filled with the voxel. The second stage goes through the chain list containing the planes to find the corresponding one. If it exists, the address of its first pattern is sent to the third stage. If it doesn't exist, it is created and filled with the voxel. The third stage goes through the pattern chain list to find the corresponding pattern. If it doesn't exist, it is created and filled with the voxel. If it exists, its address is sent to the fourth stage. The fourth stage inserts the voxel into an existing pattern and merges the patterns if needed.

Once all the voxels ave been coded into the chain lists, the recoding stage reads all the chain lists and output the raw data in the order defined by the coding algorithm.

The complexity of this algorithm is $o(p \times q)$ for the first coding to the work memory and $o(p \times q \times r)$, where p is the average length of the planes chain lists, q is the average length of the patterns chain lists and r is the average length of the patterns.

The main limitation of this architecture is that all the pipeline stages try to access simultaneously to the same memory. To avoid inter blocking, there are two solutions: adding a cache to each stage of the pipeline for the memory access, or separate the memory into several memories, one for each stage of the pipeline. This second solution seems to be better, but is poses problems for the creation (of planes or patterns) steps.

3.3 Dedicated architecture for 3D voxels decoding

After the frame buffer have been filled with the data, it must be read once for each point of view. Because a very fast decoding process was needed to insure real time decoding with a large number of different points of view, the coding algorithm was designed to provide a limited memory usage, but also a very fast and simple decoding process. This process is so simple it can be limited to 4 loops: one for the lines, one for the planes of each lines, one for the patterns in each plane and one for the voxel inside of each pattern. So it is very easily implementable on specific hardware and can be operated by only a few simple electronic component, as shown on figure 6, which shows a simplified version of the line decoder with a quick anti-aliasing to improve quality of the result pictures. This architecture is composed of three counters that implement three imbricated *for* loops: planes, patterns and pixels. For each pixel, the location in which it must be copied into the display buffer is calculated from the depth of this pixel and the angle into which the current picture will be projected. First, the display sends a signal that initializes the planes counter, setting it to zero. This resets the memory addresses counter to the beginning of the frame buffer and starts the decoding. At each clock signal, a 32 bits value is read inside the frame buffer and the memory addresses counter is incremented (after the value has been read) to be ready to read the next value. The read value is stored inside the planes counter. Those planes are the set of voxels of a line which are at the same depth of the screen, i.e. which have the same Z coordinate. If the first value is not zero, the associated "is zero" comparator unlocks the patterns counter so the next read value will be stored inside this counter. If it is zero (the line is empty), the decoding is blocked on the first value until the line contains voxels to be displayed. After the patterns counter have been set (by a value that cannot be zero), the offset of the first plane is memorized and the display offset is computed and stored in the display offset buffer. This value is the signed X coordinate of the voxel located at position zero in the plane. Then, the pixel

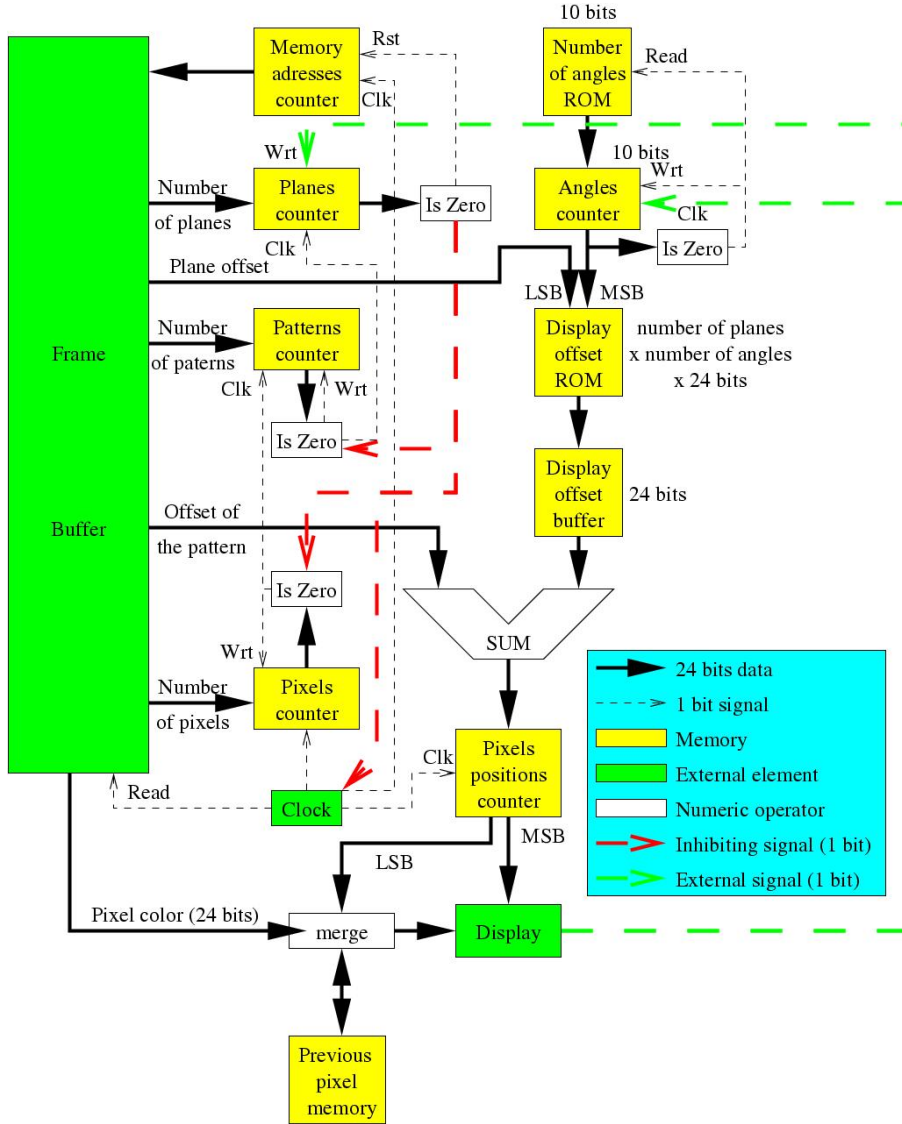


Figure 6: Voxels decoding : frame buffer architecture

counter is unlocked and stores the next value read in the frame buffer. At the following clock beat, the offset of the first pattern is read and the offset of the first pixel of the pattern is computed: the offset of the pattern is added to the display offset stored in the display offset buffer and the result is stored in the pixels positions buffer. For the next clock cycles, the values read in the frame buffer are treated as pixels: they are merged with the previous pixel for anti aliasing, according to the less significant bits of the pixel position which determines the percentage of each pixel (new and previous) that must be displayed, and the new pixel (as it was before the merging) is stored to be merged with the next pixel. The result of the merging is then send to the display memory and will be merged with the previous color (background or pixel in a further plane), according to its alpha channel (transparency) value. Then, the pixels counter and the pixels positions buffer are decremented. If the pixels counter is not zero, we continue for the next voxel. If it is zero, we decrement the patterns counter and, if it is not zero, we store the next pattern offset and size. If the patterns counter is zero, the planes counter is decremented and, if the planes counter is not zero, the decoding continues on the next plane. If the planes counter is zero, the decoding is stopped until the next signal of the display to start the decoding for the next angle of view.

This architecture only decodes one line of the screen. It can be adapted to decode the entire frame buffer and easily parallelized to fasten the decoding in two different ways:

- by parallelizing the decoding of line: each decoder decodes one or a limited number of lines. This can be easily done because the coding of each line is independent. The coding of each line must be in a separate memory or the index of the beginning of each line must be memorized before the beginning of

the decoding.

- by parallelizing the decoding of each view: each decoder decodes one or a limited number of views. Each decoder can read the frame buffer synchronously, using the same data bus because the reading can be completely sequential.

This implementation only requires the memory needed to store one entire line for the resulting line and one pixel for the anti-aliasing. The decoding of one line is directly proportional to the size of the coding needed for each line. Output data are available once the whole line have been decoded and the entire resulting line is available at the same time.

4 3D GRAPHIC AND DISPLAY BOARD BASED UPON AUTO-STEREOSCOPY

4.1 Presentation of the 2D/3D graphic and display board

There is actually a large number of commercial 3D hardware accelerated graphics cards. For obvious reasons, enterprises do not communicate about their products architecture. But we can find a common architecture for these cards.

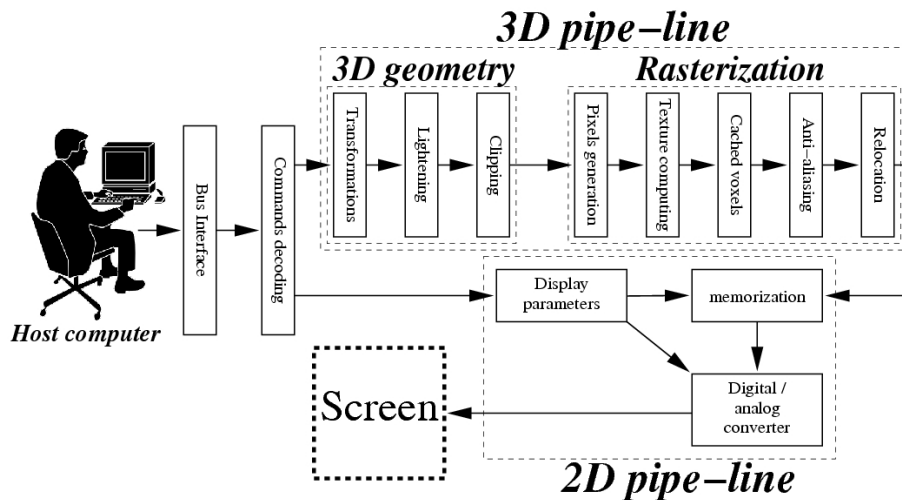


Figure 7: Graphics and display board: overall architecture of the 3D and 2D pipeline

This architecture, represented in figure 7, is defined as following:

- in a first step, data are sent to the card from the host computer by the bus interface, which is most of the time, an AGP, PCI or PCIExpress bus.
- Then, these data are decoded and interpreted by the card to determine the action to perform.
- At last, data are sent to the 2D or 3D data computing pipeline.

The 3D pipeline is divided into two parts: the *3D geometry* part, which will perform the projections of the objects points onto the screen and will compute parameters of lightening, shadowing, etc. of this objects; and the “*rasterization*”, which computes the pixels to display on the screen to represent these objects. The 3D pipeline can be schematized this way:

- The first part computes the 3D transformations of the points of the objects to be displayed and their projection onto the screen. This part computes the coordinates of the points to display on the screen.
- The second part does the first geometrical computings which will allow to later compute the lightening of the different facets of the object. It consists of computing the normal vector of each facet and the scalar product of this normal vector with the normalized vector which indicates the direction of each source of light. This scalar product indicates the quantity of light received by the facet from each light source.

But this computing does not consider the shadows of the objects, which will be computed later, in the *rasterization* stage.

- The third part operates the “*clipping*”. It determines if the points projected onto the screen are inside of the picture which will be displayed. There are three cases:
 1. The displayed object is entirely inside the screen. The object is simply drew without any more computing.
 2. The object is entirely outside the screen, ie. it is behind the observer. In that case, the object is not visible by the observer. So there is no need to draw it and is totally ignored by further stages of the pipeline.
 3. The object is partially visible. In that case, more computing are needed to determine which part of the object is visible and its position inside the screen. Once these informations are known, the corresponding zone can be drawn.

This treatment does not consider the case where the projected object is hidden behind another object already drawn. This case will be treated further in the pipeline.

- The fourth stage of the 3D pipeline generates the pixels, without computing their color. It determines which pixels have to be drawn onto the screen but will not draw them yet. For each of them, this part determines data which will allow to determine the color of each pixel and its depth.
- The fifth stage determines, for each pixel to be effectively displayed, its color and transparency value. These data are obtained with the informations of texture and color of the objects, and with other informations like shadows, light reflects and object reflection.
- The sixth stage determines which pixels, among those generated by the preceding stage of the pipeline, are effectively visible by the observer. This is realized comparing the depth of the pixel generated with the one of the pixel already drawn at the same position of the screen, if it exists. In this case, if the new pixel is further, it will be hidden by the already drawn pixel. It must not be drawn and this point of the screen will have to stay unchanged. If no pixel have been drawn at this position or if the pixel already drawn is further from the observer than the one to be drawn, the new one will be drawn and will erase the old one.
- The seventh stage improves the displayed picture, applying an “*anti-aliasing*” on the objects. This stage adds pixels on the edge of objects or on lines to void “aliasing” effects.
- At last, these color informations of the pixels are directly written into the 2D memory of the graphics card, conforming informations of position of the windows and the transparency informations of the pixels for transparent objects or pixels generated for anti-aliasing. Those actions are operated by the height and last stage of the 3D pipeline which which also computes the address of the video memory in which the value of the pixel has to be written.

The 2D pipeline is much more simple. It is has only two stages:

- The first stage is used to control the display parameters: screen resolution, refresh frequency, etc.
- The second one memorizes the pixels to display. It contains all the 2D, but also the 3D data to be drawn onto the screen. The 3D data are computed and directly written into the 2D display memory by the 3D pipeline.

The memorized data are then read and converted by the digital to analogic converter which generates the video signal sent to the video monitor.

All commercial 3D hardware accelerated graphics cards probably do not correspond exactly to this architecture. It is possible none of them do. But it represents a possible way to hardware implement the algorithms needed to have a real time¹ 3D display system, which is adapted to the modifications we need to implement the processing we want.

All these algorithms used in this architecture are well known and largely implemented and documented in literature.

¹We consider “real time display system” as a system capable to display at least 20 full screen images per second with a resolution of 320×240 pixels with a color depth of at least 16 bits.

4.2 Graphics and display board based upon 3D auto-stereoscopy system

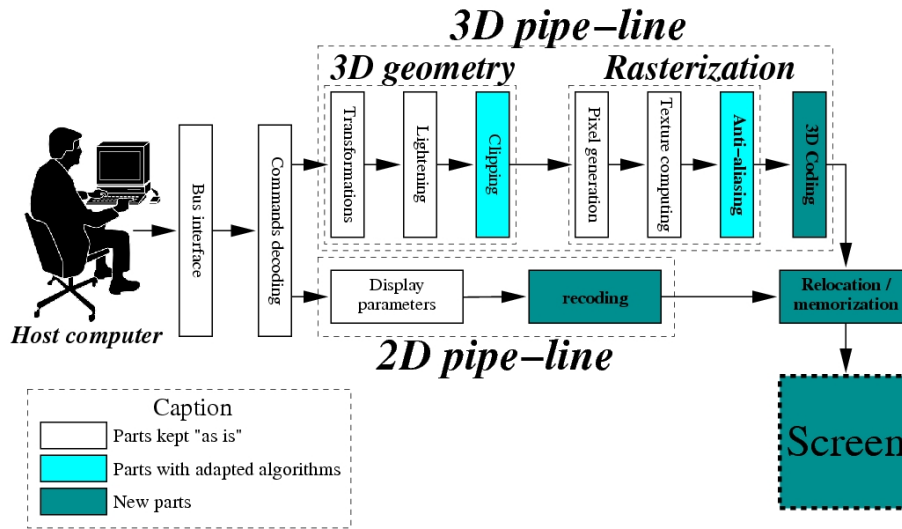


Figure 8: Graphics and display board for 3D auto-stereoscopy system

The figure 8 shows the processing chain modified to implement our processings and indicates which parts have been modified or added. Our adaptation of this chain is limited to the 2D pipeline and to final stages of the 3D pipeline. The bus interface and the command decoding do not need to be modified.

The “3D geometry” stage, which computes the transformations of the coordinates of the points of the objects, the lightening of the facets, the “clipping” and the projection of the pixels onto the screen can also be kept “as is”. Only the parameters of the “clipping” stage will have to be modified to adapt the algorithm to the width of the viewing area and allow multiple points of view, but the algorithm will not be modified. The two first stages of the “rasterization” stage which permit to generate the pixels to be drawn do not need to be adapted. The stage which determines which pixels will be effectively displayed or will be hidden by other pixels already memorized is not useful as it is implemented in commercial hardware accelerated 3D graphics cards any more: memorized data correspond to all the pixels visible from all the different points of view possible with the display system. Some of these pixels will be hidden by others according to the main point of view (the one for which the pixels are projected onto the screen) but must not be erased because they still can be seen by the observer from other points of view and must be displayed into the corresponding pictures. The figure 9 shows an example of

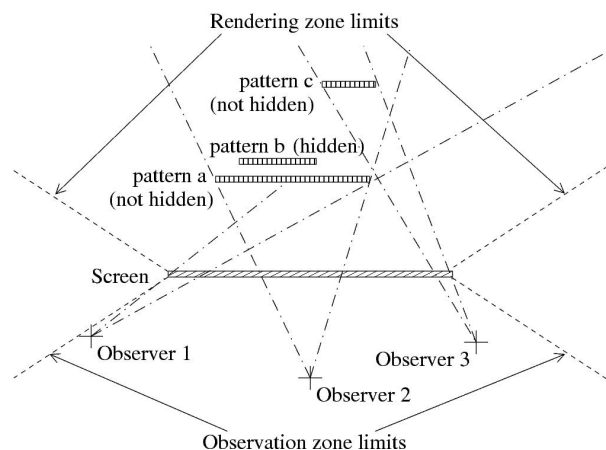


Figure 9: Example of wrongly hidden voxels

this problem: with the “classical” Z-buffer algorithm, with the main point of view “Observer 2”, patterns b and c will be erased because they are hidden by pattern a. But only pattern b is really hidden because it cannot be seen from any of the different points of view and pattern c can be seen from the observer 3’s point of view.

Anti-aliasing stage can be conserved with the same algorithm. The only adaptation is that the new generated

pixels are not merged with existing ones (background or further object pixels) inside the 2D memory, but kept as “normal” semi-transparent pixels which will be coded with the other ones.

Once generated, the pixels has to be coded in a way which will allow the display system to regenerate all the points of view. The pixels are not simply drawn into the 2D data of the graphics card, they are sent to the “3D coding” stage which will code all the pixels with the algorithm we described before, erase the really hidden pixels which are unuseful and check for possible buffer overflows when too many pixels have to be stored. The pixels are then store into the 3D frame buffer before being displayed.

For the 2D data, the same problem occurs: they cannot be stored as 2D data before being displayed. They need to be converted into 3D data to be coded with the same algorithm as 3D data. The first stage of the 2D pipeline doesn’t need to be transformed. Only some registers will probably need to be added to control the new fonctionnalités of the system and others may become obsolete but will probably be kept for compatibility problems. The second stage of this pipeline will recode the 2D pixels as 3D voxels simply adding a depth to them. They are then coded as lines of voxels and sent to the memorization stage to be stored.

The memorization stage memorizes all the voxels from the 2D and the 3D pipelines and relocates them according to the windows positioning informations. So that 2D and 3D data can be correctly displayed without interfering with one another.

5 CONCLUSIONS

We proposed here a novel approach of the generation of data for auto-stereoscopic display systems with specific hardware for real-time imaging.

In the future, we are actually planning to add support for reflection and refraction generation. The coding and decoding algorithms already have this feature, but it as not been yet fully tested, nor the integration into the processing chain, nor the automatic generation of the reflected and refracted images.

We are also planning to implement a part of the computing chain (in particular the coding stage) on GPU (Graphics Processing Unit) for testing.

This research is supported by “Conseil Régional d’Île de France” through SESAME program, convention n° E1755.

References

- [1] J. R. Moore A. R. L. Travis, S. R. Lang and N. A. Dodgson. Time-manipulated three-dimensional video display. *J. Soc. for Info. Disp.* 3(4), 1995. Dec.
- [2] Ph.D. Alan Sullivan. Depthcube solid-state 3d volumetric display. pages 279–284. *Proceedings of Stereoscopic Displays and Applications XV*, SPIE 5291 16th annual symposium, San Jose, California, january 2004.
- [3] Sir David Brewster. *The Stereoscope*. 1856.
- [4] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of CS, U. of Utah, December 1974.
- [5] Daniel J. Sandin et. al. Computer-generated barrier-strip autostereography. In *Proc SPIE, Non-Holographic True 3D Display Technologies, 1083*, pages 65–75, Jan. 1989.
- [6] Michael Halle. Autostereoscopic displays and computer graphics. In *Computer Graphics*, Annual Conference, pages 58–62. ACM Press / ACM SIGGRAPH, 1997.
- [7] E. Rieper K. Oltmann D. Bahr K. Langhans, C. Guill. Solid felix: A static volume 3d-laser display. *Stereoscopic Displays and Applications XIV*, *Proceedings of SPIE*, Volume 5006, Santa Clara, CA, 2003.
- [8] Benoit Kaufmann and Mohamed Akil. Video memory compression for multi-view auto-stereoscopic displays. pages 59–70. EI 5291A-7, *Proceedings of Stereoscopic Displays and Applications XV*, SPIE 5291 16th annual symposium, San Jose, California, january 2004.
- [9] S. R. Lang G. Martin N. A. Dodgson, J. R. Moore and P. Canepa. A 50” time-multiplexed autostereoscopic display. In *SPIE 3957, SPIE Symposium on Stereoscopic Displays and Applications XI*, 23rd-28th Jan 2000, San Jose, California.
- [10] C van Berkel and J H A Schmitz. Multiuser 3d displays. presented at Tile 2000, London 10-11 May 2000.

- [11] Actuality systems web site. <http://www.actuality-systems.com/>.
- [12] Charles Wheatstone. Contributions to the physiology of vision. part 2. on some remarkable, and hitherto unobserved, phenomena of binocular vision (continued). *The London, Edinburgh, and Dublin Philisophical Magazine and Journal of Science, series 4, 3*, pages 504–523, 1838.