

Video memory compression for multi-view auto-stereoscopic displays

authors:

B. Kaufmann, M. Akil

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

presentation date: 19 January 2004 – San Jose, CA.

Abstract

Nowadays, virtual 3D imaging is very commonly used in various domains, i.e. medical imaging or virtual reality. So far these 3D objects are projected to be displayed on 2D visualization systems (i.e. computer monitor or printed paper sheet), by the application itself, a graphic library or a specific hardware.

Now, new displaying systems that allow computers to display 3D objects in real 3D appear, often based on the stereo-vision principle, which ultimate evolution is the multi-view auto-stereoscopic system, that displays different images at the same time, visible from different positions by different observers. When the number of images grows and these different images are directly stored, the needed memory becomes very large.

This article proposes an algorithm for coding multi-view stereograms with very low quality loss and very fast and simple decoding that allows to calculate all the stereoscopic images with a low need of memory.

This algorithm projects the objects on the screen but stores the associated depth of each one. Some of the background voxels are not erased by foreground voxels even if they are projected at the same point of the screen. All those voxels are sorted in a way that fasten the decoding which is reduced only to few memory copies.

Number of words: 205

Keywords: auto-stereoscopic display, image data compression, lossless

1 Introduction

Since Charles Wheatstone, who first described the stereo-vision principle and a way to use it in three-dimensional displays [7], many three-dimensional systems have been conceived. These systems can be divided into two main families :

- the *stereoscopic vision* based systems that display at least two pictures in a way that each eye of the observer perceives a different one (i.e. the system Charles Wheatstone invented)
- and the “*projection*” systems that make the three-dimensional objects appear as if they were floating in the air (i.e. Felix 3D display[2]).

Since the invention of the computer with its growing calculating speed and the apparition of the 3D-rendering dedicated chips, high quality three-dimensional virtual objects display became more and more useful in many domains like medicine [1], or simulation [6].

Most of the existing three-dimensional virtual objects display systems are *stereoscopic vision* based and display a limited number of pre-calculated pictures of the rendered scene. Those pictures are calculated with a simple projection of the scene on as many virtual planes as the number of images required. Then the images are displayed in a way that gives the observer the vision of the three dimension objects: each eye of the observer seeing a different representation of the scene, obtained by the computation of the projection of the scene with two points of view corresponding to the two eyes of the observer. The number of the images calculated can vary from two (one for each eye of the observer) to many (depending on the resolution of the display system).

Until now, most of these systems memorized all the images they needed as 2D images the way 2D video cards did. It was easy to do so if the number of images was quite small (2 to 4), but with systems that can

display much more images at the same time [3], the memorization of a great number of high resolution images requires a very big amount of memory.

This paper proposes a new lossless algorithm for 3D images coding that can carry information to multi-angle 3D image displays [5], using less memory than storing all projected images. We will first describe the principle of the algorithm for coding and decoding. Then we will insist on the details of the implementation. And finally some of the possible extensions.

2 Frame buffer compression using non repetitive data

2.1 Auto-stereoscopic display systems frame buffer inconvenient

If we consider an auto-stereoscopic display system with an image resolution of 1024×768 pixels, 24 bits for each pixel and 20 different observation angles, which directly stores the different images, each image uses $1024 \times 768 \times 3$ bytes = 2359296 bytes = 2304 kilo-bytes. If all the images corresponding with the different observation angles, the memory used is $2304 \times 20 = 46080$ kilo-bytes (45 mega-bytes).

We can imagine that, if the technology allows it, we would increase the number of observation angles. But we would also increase the needed memory size for the images. Even if it is easier now to use large memory and to compute big amount of data, sending this size of memory to the display system fast enough to display 30 images per second seems difficult.

The algorithm that we propose here is based on the idea that when you compute several images of the same object seen from different angles of view, a large part of the pixels of each image is repeated from one picture to another with very little differences. The most evident example of this is when you want to see a virtual scene which represents a wall with a picture on it. When you change the angle of view, you still see the picture and the wall around it: the picture data are the same, you just stretch the picture according to the difference of observation distances.

2.2 Algorithm principle

Before using this algorithm, the 3D scene to be rendered is projected on a virtual screen, as if we wanted to obtain a “classical” 2D picture. So we obtain pixels, defined by their coordinates on the screen (x and y) and the color of the pixel, in ARGB (alpha, red, green and blue channels). The difference is that we keep the depth of each pixel, so each voxel is determined by its coordinates (x, y, z) and its color (ARGB). In fact, it is not really the depth of the voxel as it is calculated in the virtual scene, it is the depth of the voxel multiplied by:

$$\frac{r_p \times d_c}{w_v} \times s$$

where:

- r_p is the physical screen resolution (in pixels)
- d_c is the distance between the virtual observation point and the virtual screen (in virtual scale)
- w_v is the virtual screen width (in virtual scale)
- s is a positive constant value (which can be bigger, smaller or equal to 1) that automatically scales the displayed scene for the observer (without changing any of the x and y coordinates).

It is important to notice here that the voxel coordinates (x, y, z) are not really the coordinates in the scene or in the camera references, or in any virtual reference in the virtual scene, but a sort of “projective” coordinates where the x and the y are the coordinates of the point on the screen after the projection, and the z coordinate is the distance between the screen and the voxel. All those coordinates are indicated in pixels of the physical screen.

Once the coordinates of each voxel are computed, we apply the coding algorithm on the set of computed voxels to obtain the frame buffer.

The principle of this algorithm is to divide the images in patterns which contain voxels that are at the same distance of the screen:

- First, each image is divided into lines, which are independent: each line regroupes all the voxels that would be projected in the same line of the screen if they would be projected on a 2D screen.
- Then, each line is divided into planes that gathers all the voxels of the line that have the same distance to the screen.

- Finally, each plane is divided into patterns that regroup the contiguous voxels of the plane.

The coded 3D data will be:

For each line of the screen:

```

| Number of planes where voxels of this line are present
|   For each plane of this line:
|     | Depth of the plane (signed distance between the screen and the plane)
|     | Number of the patterns present in this plane
|     |   For each pattern of this plane:
|     |     | Horizontal position (from the left limit of the plane) of the pattern
|     |     | Number of the voxels present in this pattern
|     |     |   For each voxel of the pattern:
|     |     |     | ARGB component of the voxel
|     |     |     |   end for
|     |     |   end for
|     |   end for
|   end for
end for

```

2.3 Coding

This algorithm can be used with the voxels obtained with an algorithm like a ray-tracer or with an algorithm of rasterisation after a projection of a facet (like those hardware-implemented in the open-GL compatible 3D graphic accelerator cards).

It can be implemented directly, inserting the new voxels one by one in the pre-computed frame buffer, or first sort all the voxels by their coordinates (y first, then z and finally x) and group them in pattern and add the numbers that indicate the number of planes where voxels of each line are present, depth of each used plane, etc.

The voxels are ignored if they are projected out of the viewing angle range of the display system. This is tested using the coordinates x and y in which the voxel is projected on the screen: The y coordinate must be between 0 and the screen height. The x coordinate must be between two values that correspond to the x coordinate of the points that are at the left and right limits of the area seen by all the viewing angles.

2.4 Decoding

The decoding algorithm is very simple: it is reduced to four imbricate for loops (the first one to count the lines, the second one for the planes of each line, the third one for the patterns and the last one for the voxels in each pattern themselves). For each voxel to be displayed, the corresponding ARGB color data is simply copied from the frame buffer to the right place in the display memory. So the decoding of the frame buffer depends only on the size of the memory stored in the frame buffer.

What we call “*frame buffer*” here is the memory in which the voxels are coded with their pseudo 3D coordinates (x , y and z , as described above) and their complete color (with the alpha channel: ARGB). By opposition with what we call “*display memory*”, which is the memory where the image of the view of each possible angle of view computed for each angle of view from the data in the *frame buffer*.

The correct memory address in the display memory where the data must be copied is determined by:

1. the line where the voxel is located
2. the depth of the plane where the voxel is located, multiplied by a number depending of the viewing angle that is currently rendered
3. the relative position of the voxel in the plane (the relative position of the pattern in the plane added to the relative position of the voxel in the pattern)

The location (x_d , y_d) where the voxel must be drawn in the display memory is determined by the following equation:

$$\begin{cases} x_d = f(d_p, \alpha) + x_v \\ y_d = y_v \end{cases}$$

where:

x_d = x coordinate in the screen (in the display memory)

y_d = y coordinate in the screen (in the display memory)

$f(d_p, \alpha)$ = function to calculate the shift of the plane according to its depth (d_p) and the observation angle that is rendered (α)

x_v = x coordinate of the voxel (= x coordinate of the voxel in the pattern + x coordinate of the pattern in the plane)

y_v = y coordinate of the voxel

2.5 Projection of the voxels

The main idea of this algorithm is that the voxels are pre-projected. So the projection must only be computed once when creating the frame buffer and not for all the possible angles of view. When decoding the frame buffer for one angle, the algorithm is quite like a parallel projection, as shown on figure 1. So the computation is very

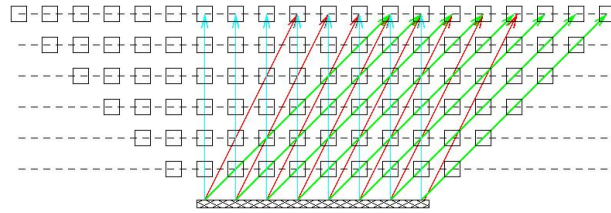


Figure 1: The voxels are projected on the screen using a projection which is parallel to the angle of view. Each square represents a possible position of a voxel and the lines represent the axis to project the voxels onto the screen (the lower line)

simple and fast: it is reduced to a simple memory area copy. Only the location to where the patterns are copied changes according to the angle of view and the depth of the pattern. Like when you look at a paysage through the window of a car or a train, the objects close seem to move faster than the objects far, the shift of each voxel depends only on its depth. Here, because the projection has already be done, the apparent size of the objects are already computed and we only have to compute the shift of the patterns.

With the use of a display system, when the user moves, he changes the angle from which he is looking. This is why the shift of the far planes is bigger than the shift of close ones. It is like if he looks thro a key hole: when he moves, he still can see what is just behind the hole, but his look makes a long distance on the opposite wall. This is why the planes can contain more voxels if they are far (as you can see on figure 1).

That way, we can display objects that can be located on the screen to an infinite distance. The limit of the number of different possible depth depends only on the number of pixels columns of the screen and the number of different possible angles of view. Using the same principle, we can also display objects that are before the screen: we only have to invert the shift of the planes.

The only problem is that if we take the example of a cube with one of its faces parallel to the screen, the width of this face does not move when the observer looks at it from the side, like it would in the reality. But this is not a problem because he sees the screen with a small angle to. So it is the display system itself that shrinks the image because it is looked at from the side.

3 Implementation tips

3.1 Depth discretization

The first problem that appears when you try to implement this algorithm is “What depths of planes must we use to obtain an optimal coding?” By “optimal”, we mean the smallest number of chosen depth planes that allow the screen to display the virtual scene without deformation.

Smaller the number of those planes will be, best will be the compression. The choice of the planes determines the number of possible values the depth of the voxels can take. A too large distance between two successive planes can lead to a loss of quality of the perceived scene. And a too low distance will lead to a frame buffer that contains unused voxels that will never be displayed.

The only plane that never moves is the one which depth is zero (the plane of the screen itself). Actually, the voxels of this plane are seen from the same point of the screen whatever the observation angle. This is what happens when you look at a 2D screen (like a TV set): you see the same picture wherever you are. If all the voxels of the frame buffer are in this plane, we have a 2D image located on the screen.

The shift of the other planes is calculated with the distance to the first plane and the angle of view. We can consider that each plane must be shifted of two pixels more hat the previous one between the two extreme angles (one between the center angle and each of the extreme ones). So we have an obvious depth discretization depending on the number of angles and the the screen resolution.

The problem is that nothing proves that this is the optimal discretization. And this discretization depends only on the wide of the viewing area, but it must also depends on the screen resolution. The only way to determine a “good” discretization is to test several discretizations to evaluate what is the best.

3.2 Erasing hidden voxels

3.2.1 Proposed algorithm

When we compute the virtual objects in order to transform them into voxels, we cannot simply use an algorithm which considers that a voxel hides another voxel which is projected on the same pixel of the screen but is at a bigger distance from it. This sort of method can be found in algorithms like z-buffer or painter (also called z-sorting) algorithm.

This is because this algorithm is made for display systems which allow to see the objects from different points of view. So if a voxel is hidden by another one in one view, it is probably not in another viewing angle.

But, once the voxels have been coded into the frame buffer, using our algorithm, we can see that some voxels are indeed hidden by several others. So those voxels are useless and should be removed from the frame buffer. For example, they can be inside or behind a voluminous object and the observer cannot see them, from any position.

As an example, on the figure 2, the patterns b and c would be hidden if we use the “classical” projection (as seen by Observer 2). But the Observer 3 can see the pattern c. So this pattern must not be deleted. On the other hand, the pattern b is masked by the pattern 1 wherever the observer is. So this pattern cannot be seen, from any of the possible viewing angles. So it can be erased.

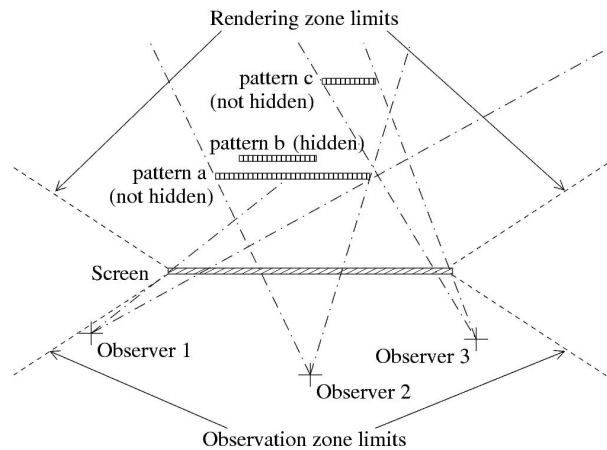


Figure 2: Example of a pattern (a) hiding voxels (pattern b) and a pattern(c) that seems hidden but is not

First, we must detect such voxels. The idea that we proposed is to say that each pattern (even a pattern composed by only one voxel) generates a mask in which the further voxels are not seen. This mask is calculated with the maximal angles from which an observer can look beyond the pattern (see details on figure 3).

The second idea is that a voxel can be hidden by several patterns which would not hide it if they were alone. This case occurs when the masks of two or more patterns touch each other. In that case, we transform the two masks in a unique bigger mask as shown on figure 4, where the resulting mask is the union of the mask of pattern a, the mask of pattern b and the created mask.

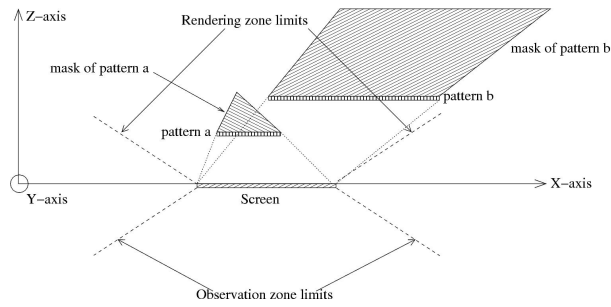


Figure 3: Example of masks used to determine unused voxels. The masks can take several aspects. the area under the x-axis is the area where the observer is. The area above the x-axis is the virtual scene.

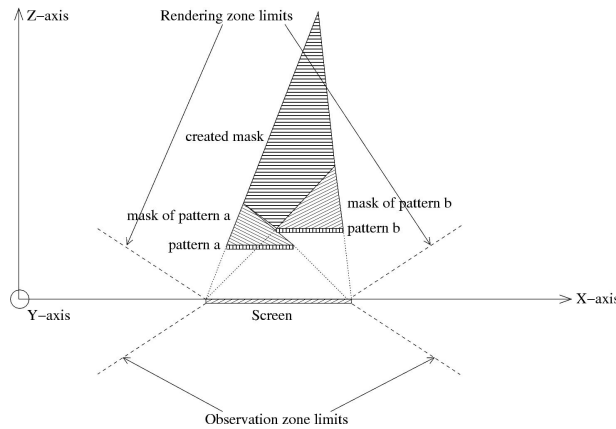


Figure 4: Merge of two masks when two patterns can hide a bigger area together than they do alone

This angles are the angles from which an observer can look at the voxels which are at the two extremities of the pattern. The maximal angle from which the observer can look at a given voxel is determined by the width of the screen, relatively to the distance between the screen and the voxel.

The algorithm we proposed is to analyze the voxels of each plane, taking these planes from the closest to the furthest (closest and furthest from the observer). For each pattern of this plane, we determine each voxels are in the mask of a pattern which has already been computed (which is closer to the observer). We remove them. Figure 5 shows how a pattern can be partially or completely erased if it is in a mask.

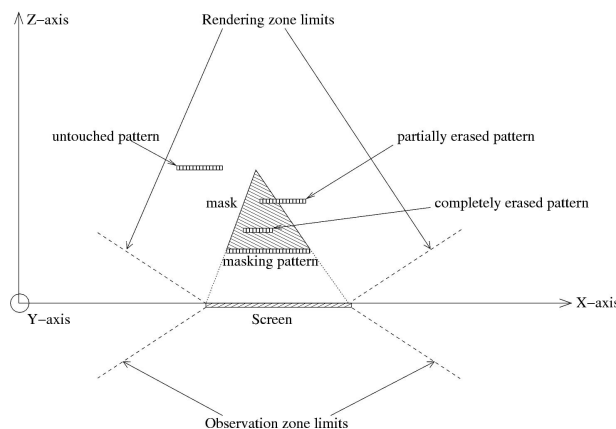


Figure 5: Erasing of voxels when they are occulted by other patterns

If there are still voxels in the pattern (the pattern was not entirely in the mask so it was not entirely hidden), we create a new mask with them. If this mask touches a mask from another pattern, we can amalgamate them.

Then we can continue with the other patterns.

3.2.2 Representation of the masks

In order to implement this algorithm, we must choose the representation of the masks in the computer. The figure 6 shows the different informations usable to define a mask. A mask is determined by three characteristics:

- the depth of the pattern which created it (z in figure 6)
- the half-line that starts from the far left limit of the screen (x_0 in figure 6) and passes by the far left limit of the far left voxel of the pattern (x_1 in figure 6)
- the half-line that starts from the far right limit of the screen (x'_0 in figure 6) and passes by the far right limit of the far right voxel of the pattern (x'_1 in figure 6)

The two half-lines are also defined by the angles they have with the surface of the screen (α and β in figure 6) or with the normal vector of this screen.

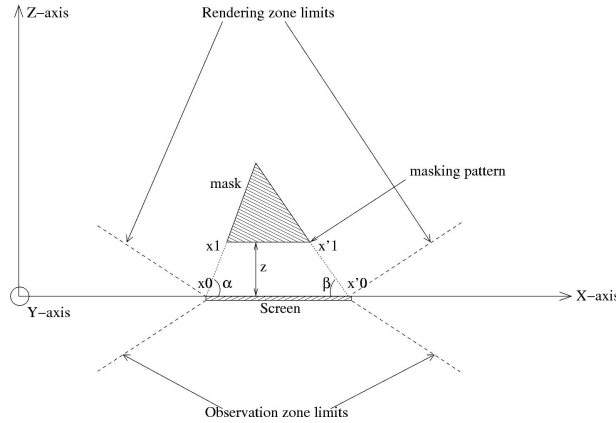


Figure 6: Illustration of the different informations usable for the representation of masks

We saw that the algorithm treats the planes from the closest to the furthest. So the depth of the pattern which created each mask cannot be bigger than the depth of the actual plane, so we can ignore it and only consider the two half-lines.

We also saw that the two half-lines passed by one of the extremity points of the screen. Because each half-line is associated to one of the extremity points, we only have to store the direction of each half-line. These directions can be stored as:

- the angle (or an associated value like sinus, cosine or tangent) of the half-line with the surface of the screen or its normal vector (α and β in figure 6) as a float number or an integer, which has the disadvantage to have a low precision and to accumulate errors when merging the masks.
- one point (other than the already known one) by which the half-line passes, which has the advantage to need only two integers. Moreover, if we decide to always use the intersection point between the half-lines and furthest possible plane, we only have to store the x coordinate of this point and save memory usage and computation time and we minimize the computation errors. Figure 7 illustrates this representation of the masks.

With this representation of the masks, merging two masks become quite simple: we only have to take the two x coordinates associated to the half-lines that give the largest mask. In fact, the computing of the new mask is only two comparisons: we always have to take the further left point of the two left half-lines and the further right point of the two right half-lines.

If the masking pattern is on the first plane (the one that corresponds to the depth of the screen), the x coordinates of the mask can't be computed (this leads to a division by zero). In that case, we have to use coordinates that correspond to the limits of the half-lines rendering zone.

3.3 Calculating the shift of the planes

The second problem that appears when you try to implement this algorithm is the shift of the different planes when rendering the scene in different angles of view. When you look it from the front, there is no problem because the planes are not shifted from their original position. But when the observer changes the angle, the

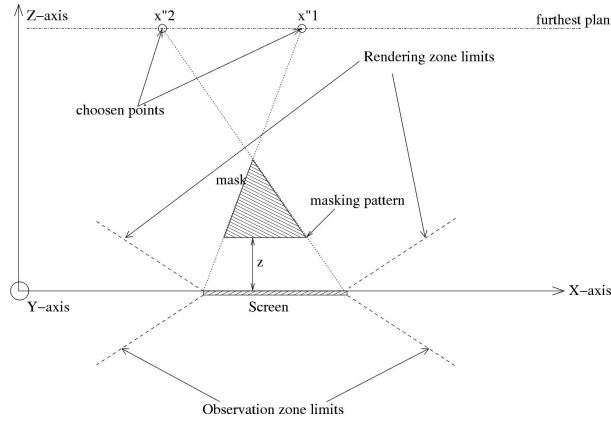


Figure 7: Use of the intersection between the half-lines and the furthest plane for the representation of masks

planes must be shifted (translated along the x -axis) according to this angle, without deforming the objects.

The first idea that we had was to apply the classical shift computation formula $s = d \times \cos(\alpha)$, where:

- s is the signed shift (in pixels) of the plane, along the x -axis
- d is the depth of the plane
- α is the signed observation angle

The problem is that, due to the parallel projection of the the pre-projected image, this formula doesn't fit at all and leads to a big distortion of the rendered objects. The result images shows that the furthest plans are scaled too much more along the x -axis that they should be. It seems that the solution would be to applicate a logarithmic scale on the apparent depth of the planes what would be equivalent to transform the previous formula in something like $s = \log(d) \times \cos(\alpha)$.

3.4 Continuity of the facets

Another problem appears on some objects as holes in them, as shown in figure 8. This problem occurs

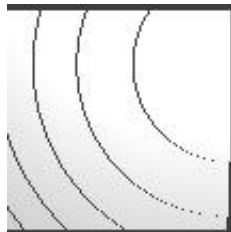


Figure 8: Continuity of the facets problem. This is an example of a cube where holes (the black circles) appear.

when an object has a face which normal vector is close to be parallel with the x -axis and the observer look at it from an angle that is far from the normal vector of the screen. It can be explained by the creation of the voxels: the voxels are calculated by projecting on the virtual screen. So they correspond with the pixels of the screen. When a face has its normal vector close to the x -axis, two voxels which are side-by-side can have very different depths. So the shift of their respective planes can be very different and holes can appear where they wouldn't. In the figure 9, the voxels are calculated using a projection as seen by the observer 1 who sees it without problem. But the observer 2 sees holes where the circles are.

The solution of this problem is to anticipate the holes in the facets by verifying that two side-by-side voxels of the same object and which should touch each other don't have more than one depth plane of difference. If they do, two solutions may be applied:

- If the computation of new voxels from the initial object is possible, we can compute additional voxels at the missing depths. For example, if the two initial voxels are in the coordinates of x and $x + 1$ and have 3

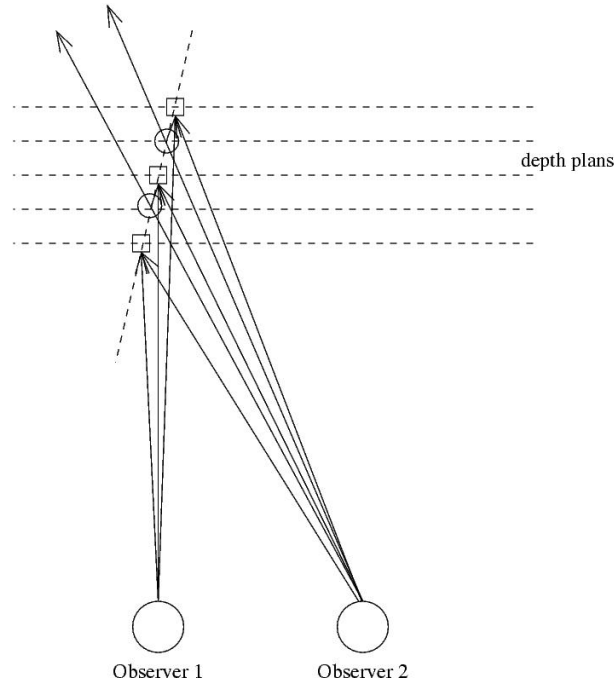


Figure 9: Illustration of the continuity of the facets problem. This figure represents what the observer sees when he moves around some facets. The squares represent the voxels and the circles represent the holes where voxels should be added

empty depth planes between them, the additional voxels to compute would be at the coordinates of $x + \frac{1}{4}$, $x + \frac{1}{2}$ and $x + \frac{3}{4}$. This is true if we assume that the object is composed of plane facets. If the real nature of the object is known, more precise computation can be made. If the nature of the object is not known, this is still a good approximation.

- If the computation of new voxels from the initial object is not possible, we can choose, for example, the color of the further voxel for the additional ones. We can also interpolate the color of the additional voxels with the color of the two initial voxels, which will give a better quality result.

All these additional voxels will then be introduced into the frame buffer behind the closest initial voxel. At this location, they will be seen only when the holes would appear, but not in the other angles of view.

3.5 Complexity of the algorithm

The main characteristic we wanted for this algorithm was its very fast decoding phase. This was for allowing the hardware implementation in the multi-view auto-stereoscopic display systems. For the systems that provide a large amount of angles, so a large number of images to display per second, the decoding has to be fast enough to decode and send images to the display system.

As we said before, the decoding is only a set of memory copies. So the time of the decoding only depends on the size of the frame buffer. But this size hardly depends on the type of the image coded. If the scene is only composed of a flat image placed on the screen, each line will contain only one plane with only one pattern. So each line will use $X + 4$ integers (X is the number of pixels width of the screen). If the scene contains no object, the frame buffer will only contain a zero for each line, so Y integers (Y is the number of pixels height of the screen). But if the scene is composed by a cloud of isolated voxels, the size of the frame buffer is equals to $4 \times N$ (N is the number of isolated voxels) because no motif can surely hide other voxels. In that case, the solution is to limit the maximal depth used: we can merge the far voxels in a plane that will take less memory space. Because the close voxels are close, the resulting plane will not be very visible and the observer would not see the difference. But if the cloud does not cover the whole screen, we mustn't reduce the maximal depth in the whole scene, but only in the area that is not well seen by the observer, typically, the inner and the back of the cloud. If the cloud touches one side of the rendering area, this side too. This is easy to enunciate but harder to program.

For the coding algorithm, the insertion of one voxel in the general case (insertion of voxels in a random order) must start with the search of the correct place. If all the data (lines, planes, patterns and voxels) are stored

as arrays, we can use a dichotomic search, which complexity is $o(\log(N))$, where N is the number of data in which the search is done, the complexity of this step is $C_1 + \log(N_{planes}) + \log(N_{patterns}) + \log(N_{voxels})$. Where C_1 is a constant representing the search of the good line in the known size array of lines, N_{planes} , $N_{patterns}$ and N_{voxels} are respectively the average numbers of planes in each line, patterns in each plane and voxels in each pattern. Then the insertion in the pattern has a complexity of $C_{alloc} + N_{voxels} + C_{test}$, where C_{alloc} is a constant representing the reallocation of the dynamic array containing the voxels of the plane and C_{test} is a constant representing the test of the merge of the pattern where the voxel is inserted and the next one. Finally, for the hidden voxels erasing, the complexity is $N_{lines} \times N_{planes} \times N_{patterns} \times N_{voxels}$.

4 extensions

4.1 Coding reflection, refraction and specular lightening

The proposed algorithm here only allows to display objects that are non-reflective, absolutely diffuse and without refraction effects. But if we want that the perceived images to be realistic, we must, at least, include Phong's specular effects. The problem is that, in a 3D scenes, the hot spots are not fixed on the objects: they move according to the observation angle. So we had to find a way to code this sort of effects in our algorithm. If the reflection is perfect (all light rays are reflected without being modified), we have a plane mirror. In that case, what we see in the mirror is like a second virtual scene, scene through the reflective object. So we can add a special pattern which describes a number of voxels and another virtual scene to compute like if the pattern was another multi-view auto-stereoscopic display system. If the reflection is not perfect, we just have to apply an alpha mask to the other scene and merge it (according to the alpha channel) with the colors of the voxels of the reflective object. The refraction can also be represented this way.

The same way, specular hot spots can be considered as a virtual scene composed by a white (or another color, depends on the color of the light) disk and a black background. For a more natural hot spot, we can add a flag that indicates if we are in "classical" reflection or in "additive" reflection. For the hot spot, the flag should be set and the color of the pixels of the reflected scene would be added to the ones of the reflective objects, not a weighted average, as it is done for the reflection.

4.2 Adapting this algorithm with animated images

This algorithm has been conceived to store still images. But, in the future, we can imagine to adapt actual video coding algorithms to increase the memory used by an animation.

For example, we saw that the image is tiled in numerous different patterns. We can imagine that a moving object is composed by a number of patterns that contain very less voxels of other objects. In that case, we can imagine that the algorithms that use block translation (like MPEG[4]) can take advantages of the pattern tiling used in our algorithm to facilitate and to make more efficient compressing.

4.3 Hardware-implementation

Our decoding algorithm is easily-hardware implementable for two main reasons: First, it is reduced to a fixed number of for loops and memory areas copies. Secondly, because the decoding of each line is independent of the others. So it would be very easy to implement it on a specific hardware.

But for the coding algorithm it is more difficult. This is because the coding algorithm needs sorting and inserting of voxels in existing patterns and things more difficult like merging of patterns, etc. For these reasons, the implementation of the coding algorithm seems difficult on a specific hardware. It would be easier to use programmable hardware like DSP or micro-controllers.

5 Conclusion

This algorithm is intend to be a base for hardware implementation of memory compression for multi-view auto-stereoscopic display systems. Its main characteristic is the very fast decompression algorithm and the facility to implement it on a specific hardware.

The main idea of it is that it codes each voxel once, and not at each possible angle of view. This leads to a large gain of memory space. The obtained frame buffer is also easily compressible to reduce even more the memory needs, but this reduces the speed of decompression.

References

- [1] J. R. Moore T. D. Wilkinson C. W. Connor, N. A. Dodgson and A. R. L. Travis. Design of an autostereoscopic endoscope for abdominal surgery. Proc SPIE 3595 Biomedical Diagnostic, Guidance, and Surgical-Assist Systems, 26th-27th Jan 1999, San Jose, California, pp.130-137, 1999.
- [2] M. Gerken C. Vogt D. Bezecky D. Homann D. Bahr, K. Langhans. Felix: A volumetric 3d laser display. Projection Displays II, Proceedings of SPIE Vol. 2650, pp. 265-273, San Jose, CA, 1996.
- [3] S. R. Lang J. R. Moore, A. R. L. Travis and O. M. Castle. A 16-view time-division-multiplexed autostereoscopic display. SID '93, 1993.
- [4] Didier Le Gall. Mpeg: a video compression standard for multimedia applications. *Commun. ACM*, 34(4):46–58, 1991.
- [5] J. R. Moore N. A. Dodgson and S. R. Lang. Multi-view autostereoscopic 3d display. IBC '99 (International Broadcasting Convention), 1999. 10th-14th Sep 1999, Amsterdam.
- [6] A. R. L. Travis. Three dimensional video and virtual reality. ID UK & Ireland 10th Anniversary Meeting, 1995. Cambridge, UK, July 7, 1995.
- [7] Charles Wheatstone. Contributions to the physiology of vision. part 1. on some remarkable, and hitherto unobserved, phenomena of binocular vision. pages 371–394. Royal Society of London Philosophical Transactions 128, 1838.