

3D images compression for multi-view auto-stereoscopic displays

authors:

B. Kaufmann, M. Akil
Lab. A^2SI - Groupe ESIEE
Cit  Descartes - BP 99, 2 Bd Blaise Pascal
F-93162 Noisy-le-Grand Cedex, France

Abstract

As auto-stereoscopic displays offer more and more different possible views, the problem of the size of the data to display is becoming more and more urging. If many 3D images compression algorithms already exist, using various methods, many are not lossless and lead to zones where data are not well displayed.

This article details our lossless 3D compression algorithm based on 3D reconstruction of the initial 3D scene and presents some results.

We will insist on the description of the coding and decoding process of a number of particular effects, especially the reflection and refraction reconstitution. We will also show an encountered problem called discontinuity of the facets and the way to solve it.

Keywords—**multi-view, auto-stereoscopic display, real-time implementation, lossless compression**

1 Introduction

Auto-stereoscopic displays are more and more developed, using many different technologies. The next step in this research axis will be the multi-view auto-stereoscopic display systems which will provide a wide-angle 3D viewing with many different points of view. Some prototypes already exist in different laboratories, as Cambridge auto-stereo 3D display [5].

The simplest way to display the correct views is to compute them independently and store the resulting pictures. But because more than eight perspectives are necessary to allow the observer to move his head in front of the display system, with the feeling he sees a real object [4], this solution considerably increases the need of memory.

Some 3D compression algorithms have been proposed, but many of them use depth map and warp the reference image [1], which avoid seeing behind some objects, or use detection of correlations between different images, as Torsten Palfner and Erika M ller’s [6] which uses a MPEG4-based compression, or Shah and

Dogson’s [8] or Fujii and Harashima’s [3] which use DCT or geometric compression. These algorithms are more complicated and need more resources for coding and decoding.

The solution we proposed was to recode the 3D data in a way which will allow to reconstruct very easily and fastly the different pictures from a single file, but not the pictures themselves. This file is obtained with only one projection of the virtual objects whatever the number of different points of view are needed. Therefore, this algorithm provides a fastest generation of multiple views of virtual objects and reduces the size of the memory needed to store them. There can be various use of this algorithm: it can be used to generate views for a multi-view auto-stereoscopic display system or to precompute and store objects views for 3D games, or be used for compressing multi-angle 3D images or films.

This article will present the main principles of the 3D images compression algorithm and will insist on some of the special effects which can be coded with this algorithm and how they are computed within it. We will also demonstrate that this algorithm can be implemented on dedicated hardware, such as hardware accelerated graphics hardware, to provide a real-time coding and decoding process and regenerates 2D images with a visual quality similar to actual hardware rendered pictures.

After describing the principle of the 3D compression algorithm, we will explain how to adapt it to include special effects like reflection, refraction and objects which show another virtual scene, as if it was seen through a window. Then we will analyze a problem that will commonly occur when using this algorithm, which causes holes to appear in some objects, and the way we proposed to solve it. At last, we will expose some of the tests we made and we will discuss their results.

2 3D compression algorithm

2.1 Memory size and time evaluation for auto-stereoscopic displays

The main problem when you have to display many different point of view of a 3D object is the time

needed to compute the different views and the memory size needed to store them. Actually the largest number of different views used in an auto-stereoscopic display is 28, used by the Cambridge auto-stereo 3D display [5]. As an example, if you plan to create a display system which provides 28 different points of view to the observers, you will need to compute and store the 28 pictures resulting from the projection of the objects to the 28 directions. If each of these pictures is a $1,024 \times 768$ picture with 24 bits per pixel, the size of each picture is $1,024 \times 768 \times 3 = 2,359,296$ bytes, aka. more than 2 megabytes. For the 28 different pictures, more than 60 megabytes of memory are needed to store the pictures. The time needed to compute all these pictures is also a restraint to real-time applications. Actual hardware accelerated graphics cards, such as NVidia GeForce 6800 Gtx, can perform up to 100 pictures per second, for resolutions up to $1,600 \times 1,200$. For a lower resolution, such as $1,024 \times 768$, speed may be 2 to 3 times bigger. But, with the previous example, the 28 different points of view must all be computed 30 times a second. The total number of different 2D pictures computed must be $28 \times 30 = 840$ pictures per second, which is too important for actual hardware accelerated graphics interfaces. The use of our coding algorithm will allow to solve this problem because the virtual objects have to be projected only once for all the different views.

Now, if you look closely at these pictures, you can notice that many pixels of each one is also present in many of the others, at different positions in the same line. The main idea of this algorithm is to avoid this data repetition to decrease the size of the needed memory by coding directly the voxels with its three coordinates. The reconstruction of the pictures will only consist of moving the voxels along the line in which it is located, according to which picture we are reconstructing.

2.2 Algorithm

The voxels are coded as described below:

- Background color
- *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:*

- ◀>>> Position of the pattern (defined as the signed horizontal position from the center of the image displayed for the center view) of the first (from the left) voxel of the pattern
- ◀>>> Number of voxels present inside the pattern
- ◀>>> *For each voxel of the pattern:*
⇒ ARGB component of the voxel

The decoding of the pictures is defined by algorithm 1. The main characteristic of this algorithm is the simplicity of the decoding: it is reduced to 4 imbricated loops, each of whom consists on up to three memory reads, two counter incrementations and one memory write. This makes it easily implementable on dedicated hardware. The total decoding of one pattern needs one addition, $Nv+2$ memory reads, Nv counter incrementations and Nv comparisons, where Nv is the number of voxels in the pattern. Therefore, the total decoding of one plane needs $2+(3 \times Np)+(Np \times Nva)$ memory reads, one multiplication, $Np \times Nva$ comparisons, Np additions and $Np \times (Nva+1)$ counter incrementations, where Np is the number of patterns in the plane and Nva is the average number of voxels in the patterns of this plane ($Np \times Nva$ is equal to the total number of voxels in the plane) So the total decoding of a line needs $N \times (2+(3 \times Npa)+(Npa \times Nva))+1$ memory reads, N multiplications, $N \times Npa \times Nva$ comparisons, $N \times Npa$ additions and $N \times Npa \times (Nva+1)$ counter incrementations, where N is the number of planes in the line, Npa is the average number of patterns in one plane and Nva is the average number of voxels in the patterns of this plane ($N \times Npa \times Nva$ is equal to the total number of voxels in the line).

The time complexity of the decoding is $o(n)$, where n is the size of the coded data. The memory needed is limited to the one used to store the incoming data, the one used to store the resulting picture, one integer counter to store the position in the input memory to read, the 4 counters for the 4 imbricated loops, one (8-bits wide in our implementation) memory to store the alpha channel of the voxels and one (24-bits wide in our implementation) memory to temporarily store the old pixel of the result picture and the result of the transparency computing (see algorithm 1).

3 Voxels projection, reflexion and refraction coding algorithms

The algorithm described before can only code diffuse and transparent (without light refraction) objects. But a realistic image must contain at least Phong's [7] specular hot spots, which correspond to

Algorithm 1 Regenerating the pictures

Data: fully coded memory (or file) as described in chapter 2.2 with fitting characteristics (such as screen resolution...)

Data: resolution of the auto-stereoscopic display system (width and height, in pixels)

Data: angle of the picture to generate, defined as an integer value determining the shift of the patterns and depending on the characteristics of the display system (such as horizontal resolution...)

Data: output memory large enough to store one 2D picture (each pixel defined by its RGB values) with the resolution of the auto-stereoscopic display system

Result: the 3D object view from the corresponding angle, written in the given output memory

Data: plane_offset, pattern_offset, alpha and old_pixel are local variables (8 bits wide for alpha, 24 bits wide for old_pixel and 32 bits wide for plane_offset and pattern_offset, in our implementation)

```
1: fill in the output memory with the background color
2:   for each line of the display system do
3:     for each plane of this line do
4:       plane_offset ← angle × depth of plane
5:       for each pattern of this plane do
6:         pattern_offset ← plane_offset + Horizontal position of the pattern
7:         for each voxel of this pattern do
8:           alpha ← alpha channel of the voxel, scaled to [0, 1]
9:           old_pixel ← pixel of coordinates (pattern_offset + voxel number, y) in the output memory
10:          for each component R, G and B do
11:            component of the pixel of coordinates (pattern_offset + voxel number, y) in the output memory ← ( alpha × component of voxel ) + ( 1 - alpha ) × component of old_pixel )
12:          end for
13:        end for
14:      end for
15:    end for
16:  end for
```

the reflection of light on the surface of the objects. With traditional hardware acceleration, these spots [2] are drawn directly onto the objects as textures. The problem of this solution is it doesn't make the hot spot

to move along the objects when the observer changes his point of view, as it should. We proposed an improvement to our algorithm to include this feature.

3.1 Voxels projection and coding

The voxels are projected as if they would be displayed as a 2D picture. The coordinates of the point of the screen where the voxel is projected become x and y . The depth of the voxel becomes rz . Then rz is discretized to limit the number of planes and becomes z . The three coordinates (x , y and z) are used to determine in which pattern the voxel needs to be inserted. If the corresponding pattern or plane does not exist, it is created and filled with the appropriate data. If another voxel already exists in the same pattern, then the rz coordinate is used to determine which one is more suitable to be kept. Algorithm 2 represents the insertion algorithm.

The complexity of this algorithm depends on the implementation: if the voxels are directly coded into the final form, the insertion of a new voxel constrains to move all the following data, which takes a long time, and to go throw all the previous data or use large arrays to store the index of the beginning of every element (line, plane and pattern). The best compromise between memory use and execution speed is to use chained lists. This way, the seeking of the correct element of the list or the position where a new one must be inserted takes L memory reads and $\frac{L}{2}$ comparisons, where L is the average length of the chain lists. The insertion of a new element will take only one memory read and 2 memory writes, whatever the length of the list. The complexity of this algorithm is $o(L)$.

3.2 Reflexion and refraction coding

3.2.1 principle

The coding algorithm defined above allows to display diffuse objects. But more complex effects like reflection and refraction can also be coded. To do that, we simply added the possibility to code an entire scene into a single pattern. This scene, we call "secondary scene", can be different and without any connection to the "main" scene, or can represent the virtual scene as seen by reflection or refraction from the points represented by the pattern. The figure 1 shows these three different uses of this functionality.

This coding can show simultaneously several different effects and can also be used to code specular lighting effects.

3.2.2 coding reflection and refraction effects

We implemented this algorithm using the most significant bit of the number of voxels in a pattern to

Algorithm 2 insertion algorithm

Data: a memory in which write the modified data

Data: 1 voxel to insert, defined by:

- **x, y** and **z**: its 3 coordinates corresponding to the x and y coordinates of the projection of the voxel onto the screen and its discretized depth
- **rz**: its “real” depth, used to choose which voxel to keep when two of them must be inserted at the same coordinates x, y, z.
- **col**: the ARGB component of the voxel

Result: The result memory contains the new voxel inserted at the fitting place

```
1:  fetch the address of the data of the line corresponding to y
2:  if the line does not contain any plane then
3:    create the plane of depth z in the line y
4:    create a pattern at coordinate x and of size 1 in the plane
5:    copy rz and col into the voxel of the pattern
6:  else
7:    search in the line the plane which fits the depth of the voxel or the position where it should be
8:    if the fitting plane does not exist then
9:      create one
10:     create a pattern at coordinate x and of size 1 in the plane
11:     copy rz and col into the voxel of the pattern
12:     insert the new plane at the correct place
13:   else
14:     search in the plane the pattern in which the voxel must be included, or which the voxel touches, or
the position where such a pattern should be
15:     if such a pattern does not exist then
16:       create a pattern at coordinate x and of size 1 in the plane
17:       copy rz and col into the voxel of the pattern
18:       insert the new pattern at the correct place
19:     else
20:       compute position where the voxel should be inside the pattern
21:       if a voxel already exists in the pattern at the same position then
22:         if the new voxel is closer than the old one (comparing the rz) then
23:           copy rz and col into the voxel of the pattern, erasing the old ones
24:         end if
25:       else
26:         insert the voxel into the pattern {comment: The voxel is at one extremity of the pattern}
27:       if the voxel is the last (resp. first) one of the pattern and touches the first (resp. last) voxel of
the next (resp. previous) pattern and the two patterns are compatible (aka. they both do not have
a secondary scene or they have the same secondary scene with same options) then
28:         merge the two patterns
29:       end if
30:     end if
31:   end if
32: end if
33: end if
```

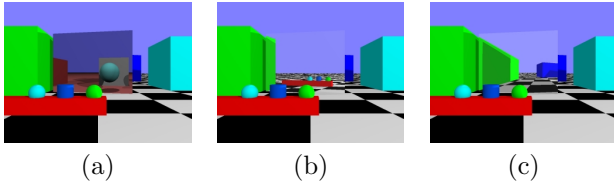


Figure 1: Three different uses of the coding of a “secondary” scene in an object: (a) with an independent scene, (b) as reflexion coding and (c) as refraction coding.

determine if this pattern has a secondary scene coded in it. If this bit has the “true” value then it is immediately followed by the address (the offset inside the data) of the beginning of the data relative to this secondary scene, which is only one line height. The secondary scenes are coded at the end of the memory (or file) after the end of the coding of the last line and can be stored in any order. These choices have been made for the following reasons:

- the secondary scene is associated to one pattern
- one secondary scene can be used by several patterns
- all the data we use are 32 bits wide, but if another implementation uses smaller values or a “huge” resolution screen is used, the limit of the size or the horizontal location of a pattern can be reached. In this case, it is more suitable to use a size twice smaller than the horizontal position than the opposite. The reasons are obvious. This is why we use the MSB of the size of the pattern and not its horizontal position

Then, this address is immediately followed by a second byte of memory (we used 32 bits in our implementation) which will be used in the future for some special features like using the same secondary scene for several patterns with a slight difference. For example, this last byte can be used to make several parts of a reflective sphere or vertical cylinder use the same secondary scene with a little change of angle of observation.

With this functionality, the voxels coding, as described in chapter 2.2, is modified as described below:

- Background color
- *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:*
 - >>> Position of the pattern (defined as the signed horizontal position from the center of the image displayed for the center view) of the first (from the left) voxel of the pattern
 - >>> Number of voxels present inside the pattern, coded in 31 bits plus one bit which indicates if the pattern has a secondary scene
 - >>> If the pattern has a secondary scene, address (offset) in the input memory where the coding of the secondary scene starts
 - >>> If the pattern has a secondary scene, special features data (see text)
 - >>> *For each voxel of the pattern:*
 - ⇒ ARGB component of the voxel

- secondary scenes

3.2.3 decoding reflection and refraction effects

The decoding algorithm is very simple: when reading the size of each pattern, we check if the MSB is equal to true. If so, we read the correct number (without the MSB) of voxels of the pattern, we read the ARGB value of the voxels and write their color in a secondary buffer which size is the screen width and we store the decoding of the secondary scene before being moved into the real buffer. Then the secondary scene is recursively decoded as the main one and the resulting line of the view is written into the secondary buffer. The decoding algorithm becomes as shown on algorithms 3 and 4.

The execution time of this algorithm is the same as for the one described in section 2.2 plus one bit test and, for patterns that have a secondary scene, a bit masking to remove the “pattern has a secondary scene” bit from the number of voxels. The decoding of the secondary scene is the same algorithm as for decoding the main scene. The complexity of this algorithm is still $o(n)$, where n is the size of the coded data, including the secondary scenes.

3.3 Discontinuity of the facets

3.3.1 description of the problem

When implementing this algorithm, we found a recurrent problem when generating the 2D images. We called it the “discontinuity of the facets”. It consists in

Algorithm 3 Regenerating the pictures with secondary scenes

Data: fully coded memory (or file) as described in section 3.2.2 with fitting characteristics, such as screen resolution

Data: resolution of the auto-stereoscopic display system (width and height, in pixels)

Data: angle of the picture to generate

Data: output memory large enough to store one 2D picture (each pixel defined by its RGB values) with the resolution of the auto-stereoscopic display system

Result: the 3D object view from the corresponding angle, written in the given output memory

- 1: fill in the output memory with the background color
 - 2: **for each** line of the display system **do**
 - 3: call the one line decoding with secondary scenes algorithm (algorithm 4) with parameters:
 - fully coded memory (or file) as described in algorithm 3.2.2 with fitting characteristics
 - address (offset) in the input memory where the coding of the line starts
 - resolution of the auto-stereoscopic display system (width and height, in pixels)
 - angle of the picture to generate
 - output memory
 - 4: **end for**
-

holes that appear on the surface of objects when two voxels, which x coordinates difference is equal to 1 are in planes which offset difference can be more than 1. In that case, when the two patterns are drawn, a gap can appear between them. Figure 2.a shows a detail of a result image where such gaps can be seen.

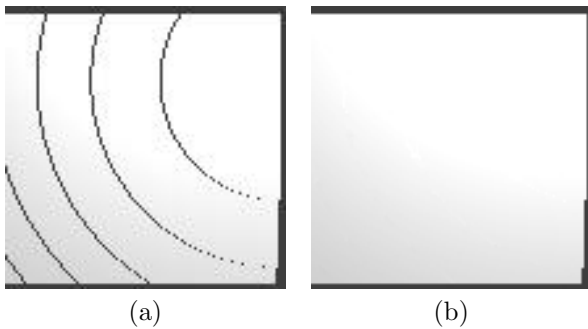


Figure 2: “discontinuity of the facets” problem: detail of a result image (a) before the correction algorithm and (b) after the correction algorithm

This can be imagined as a set of voxels positioned at different depth. When two successive voxels (which x coordinates difference is equal to 1) are separated by more than one plane, when an observer looks at them from one side, he will see through the gap between them. Figure 3 illustrates this representation.

The solution we implemented is to check if there is a gap between each voxel and the following one. If a gap exists and the voxels are supposed to be stuck to each other, then we simply add enough voxels between

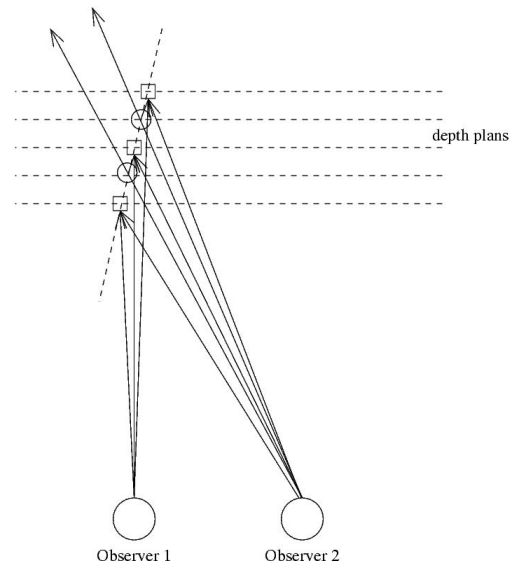


Figure 3: Representation of the reason of the “discontinuity of the facets” problem: the voxels (represented as squares) are first created using the projection of the objects as seen by the observer 1, but the picture computed for the angle corresponding to observer 2’s point of view shows holes where other voxels (represented as circles) should be present.

them and behind the closest one to fill the hole. In fact, we create one new voxel for each plane (with the discretized depth) so there will be no gap between it and the previous and the following. Figure 2.b shows the result of this algorithm. The RGBA values of these

Algorithm 4 one line decoding with secondary scenes

Data: fully coded memory (or file) as described in section 3.2.2 with fitting characteristics

Data: address (offset) in the input memory where the coding of the line starts

Data: resolution of the auto-stereoscopic display system (width and height, in pixels)

Data: angle of the picture to generate

Data: output memory large enough to store one line of a 2D picture (each pixel defined by its RGB values) with the resolution of the auto-stereoscopic display system

Result: the specified line of the 3D object view from the corresponding angle is written in the given output memory

Data: plane_offset, pattern_offset, alpha and old_pixel are local variables (8 bits wide for alpha, 24 bits wide for old_pixel and 32 bits wide for plane_offset and pattern_offset, in our implementation)

```
1:   for each plane of this line do
2:     plane_offset  $\leftarrow$  angle  $\times$  depth of plane
3:     for each pattern of this plane do
4:       pattern_offset  $\leftarrow$  plane_offset + Horizontal position of the pattern
5:       if the pattern does not have a secondary scene in it then
6:         for each voxel of this pattern do
7:           alpha  $\leftarrow$  alpha channel of the voxel, scaled to [0, 1]
8:           old_pixel  $\leftarrow$  pixel of coordinates (decal + voxel number, y) in the output memory
9:           for each component R, G and B do
10:            component of the pixel of coordinates (pattern_offset + voxel number, y) in the output memory
             $\leftarrow$  ( alpha  $\times$  component of voxel ) + ( ( 1 - alpha )  $\times$  component of old_pixel )
11:          end for
12:        end for
13:      else
14:        dynamic allocation of a secondary buffer for secondary scenes large enough to store one line of the
        output image
15:        for each (real) voxel of this pattern do
16:          pixel of coordinates (decal + voxel number, y) in the secondary buffer  $\leftarrow$  voxel RGB channels
17:        end for
18:        recursively call the algorithm with parameters:
        - fully coded memory (or file) as described in algorithm 3.2.2 with fitting characteristics
        - address (offset) in the input memory where the coding of the secondary scene starts
        - resolution of the auto-stereoscopic display system (width and height, in pixels)
        - angle of the picture to generate
        - secondary buffer
19:      end if
20:    end for
21:  end for
```

voxels are computed using an interpolation of the color of the corresponding object.

3.3.2 coding algorithm with avoiding discontinuity of the facets

As we said below, the offset of one plane is equal to $\text{angle} \times \text{depth}$ of the plane, where angle is a relative integer value corresponding to the view angle to be rendered. If we consider two voxels called A and B of coordinates x_a, y_a, z_a, rz_a for A and x_b, y_b, z_b, rz_b for B, such as $x_a = x_b - 1$ and A and B are from the same facet, there shouldn't be a hole between them. But if $z_a \neq z_b$, A and B are in two different patterns. Let max_{angle} and min_{angle} be respectively the maximum and the minimum of the angle value, such as A and B are both visible in the result picture. The offset of A will be $max_{angle} \times z_a$ to the right of the result picture and $min_{angle} \times z_a$ to the left. For B, these values will be respectively $max_{angle} \times z_b$ and $min_{angle} \times z_b$. If $(z_a < z_b)$ and $(\text{offset of A} - \text{offset of B}) > 1$ or $(z_a > z_b)$ and $(\text{offset of A} - \text{offset of B}) < -1$ then a gap will appear between A and B.

To fill this hole, new voxels must be inserted behind the closer voxel: same x and y coordinates, but z and rz bigger. These voxels must verify the following condition:

if ($z_c < z_d$) *then*

$$max_{angle} \times z_c - max_{angle} \times z_d = 1$$

else

$$max_{angle} \times z_c - max_{angle} \times z_d = -1$$

where z_c and z_d are the z coordinates of two successive voxels C and D, respectively, between A and B (C can be A and D can be B), such as $x_c = x_d - 1$. The color associated to each of these voxels is obtained by interpolation of the texture of the corresponding object.

3.3.3 decoding algorithm with avoiding discontinuity of the facets

The decoding algorithm is left unchanged because the added voxels are inserted directly into the result memory. They will be decoded and displayed as normal voxels. If they must not be displayed, they will be hidden by the voxels before them. The complexity of the decoding algorithm is still $o(n)$, where n is the size of the coded memory, including the added voxels.

4 Tests and results

4.1 Visual results

We implemented this coding algorithm writing our own ray-tracing program. The left-hand pictures of figure 4, noted $x.a$, where x can be 1 or 2 or 3, etc. show ray-traced images of different 3D scenes. The

other pictures ($x.b$) show the same 3D scenes, computed with the same ray-tracing program, modified to recode 3D data with our algorithm: The coordinates of the intersections between rays and objects plus the color of the object at these points make a voxel. The set of all the computed voxels is then coded with our 3D coding algorithm and written in a file. This file is then decoded by a simulator we wrote and which implements our decoding algorithm and displays the views from different angles. The results of this simulator are shown on right-hand pictures of figure 4.

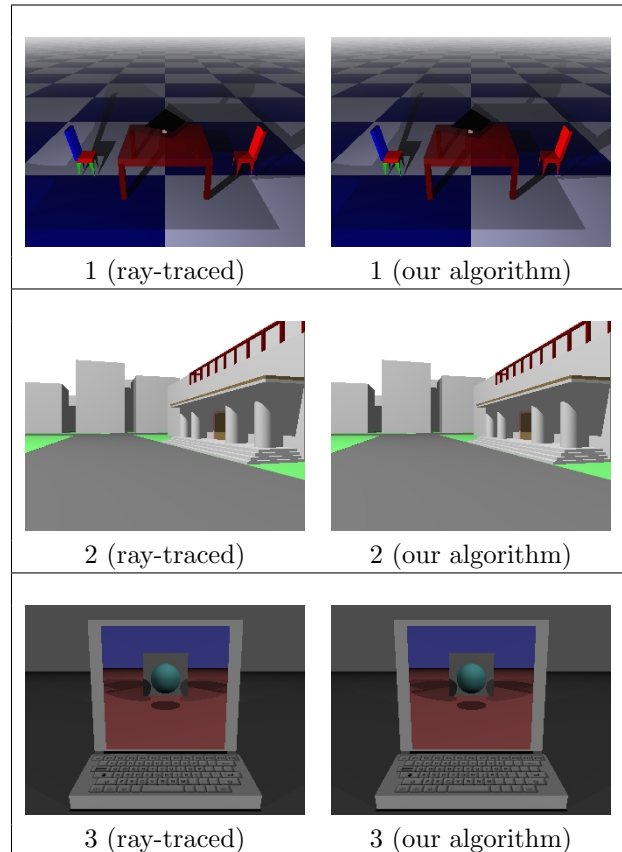


Figure 4: Visual comparison between our algorithm (right) and ray-traced pictures (left)

As you can see, if the scene does not contain effects that cannot be coded by the algorithm, the coded images and the ray-traced images are similar and show no noticeable difference. If the angle chosen to reconstruct the view corresponds exactly (offset = 0) with the one used to create the scene, the reconstructed picture is exactly the same as the corresponding ray-traced one.

4.2 Size of the compressed files and speed of the algorithm

The files coding the scenes shown on figure 4 have the characteristics shown on figure 5.

Figure	resolution	# of voxels	size of file	coding time	decoding time	picture size
1	a	389771	2083233	4660	6.326	230400
1	b	97324	523653	480	0.896	57600
2	a	433996	1909293	4250	7.41	230400
2	b	103246	454733	550	1.62	57600
3	a	726909	4131497	15960	18.67	230400
3	b	123529	666105	720	2.93	57600

Figure 5: Results of the algorithm

All those files have been computed for a virtual auto-stereoscopic display system which is capable to display 200 different views. The resolution corresponds to the resolution of the simulated display system: a means 320×240 and b means 160×120 . The file sizes are given in bytes, using 32 bits data and 8 bits for each of the R, G, B and A component of every voxel. The times are given in milliseconds and have been measured using the Linux system call *clock*. Decoding times do not include file reading nor displaying. The coding program was written in C++ and the simulator in C. These tests were made on a 1.8 GHz mobile AMD Athlon with 256 MB of RAM, running linux Mandriva 2006.

As you can see, the size of the files corresponds to approximately 10 times of the corresponding 2D picture. This means that if you need more than 10 different views, this algorithm uses less memory than store the pictures. The file coding is fast, compared to the time needed to compute the projections of the objects, even if it is not linear. If the coding times shown here do not fit with the real time constraints, the time gained by hardware implementation will reduce the coding time to acceptable values for real-time. The decoding speed is high enough for real time in our sim-

ulator, for the resolution we took. Here again, hardware implementation will increase the decoding speed and allow the real-time decoding of larger images.

5 Conclusion

We demonstrated here the capabilities of our algorithm and its performances. Despite other compression algorithms offer better compression ratio, we consider that this one is the better compromise between compression and decoding speed. This is why this algorithm is particularly useful for real-time imaging, when very fast picture generation is needed with a very little data loss for a large number of different views. It provides the advantage to regenerating all the data, even those which are hidden from the front view.

This work is supported by the Conseil Régional d'Île de France (PRIMPROC project)

References

- [1] ISO/IEC JTC1/SC29/WG11 N4415: PDAM of ISO/IEC 14496-1/AMD4, dec. 2001.
- [2] Gary Bishop and David M. Weimer. Fast phong shading. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 103–106. ACM Press, August 1986.
- [3] T. Fujii and H. Harashima. Data compression of an autostereoscopic 3-d image. In *Proceedings of SPIE: Stereoscopic Displays and Virtual Reality Systems*, pages 108–118, April 1994.
- [4] B. Javidi and F. Okano (eds). *Three-Dimensional Television, Video and Display Technologies*. Springer-Verlag, 2002.
- [5] 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.
- [6] Torsten Palfner and Erika Müller. Coding of multi-view images. pages 47–58. EI 5291A-7, Proceedings of Steroscopic Displays and Applications XV, SPIE 5291 16th annual symposium, San Jose, California, january 2004.
- [7] Bui-T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.
- [8] D. Shah and N. A. Dogson. Issues in multi-view autostereoscopic image compression. In *Proceedings of SPIE: Stereoscopic Displays and Applications'01*, pages 307–316, January 2001.