

Accurate Image Generation and Interactive Image Editing with the A-buffer

Wing Hung Lau and Neil Wiseman

Computer Laboratory,
University of Cambridge,
Cambridge CB2 3QG, England.

Abstract

The A-buffer suggested by Carpenter [2] is a hidden surface removal method which deals with arbitrary overlapping edges in a pixel. In this paper, we suggest a way of increasing the subpixel resolution by storing the bitmask index rather than the bitmask. This allows much more accurate images to be generated while at the same time, minimising memory usage. It also allows zooming to reveal more information. We also suggest an enhancement to the A-buffer by allowing the creation of dynamic objects. These dynamic objects can then be edited (deleted, moved, etc.) interactively as image modification and assembly is going on. This new buffer forms part of the image composition system we have been working on.

Keywords: buffering method, image generation, zooming, interactive image editing, object selection, animation, windowing support.

1. Introduction

To produce an image using a z-buffer type algorithm [3], we need to know the colour value and the depth value of the closest object at each pixel. However, keeping these two values alone would produce images with aliasing in them. This aliasing is the result of the point sampling nature of the z-buffer. To solve this problem means that we need to solve the visibility problem in subpixel coordinates. This would require the calculation of the visible area of each polygon at each pixel.

Existing hidden surface removal methods that calculate subpixel visibility can roughly be classified into *fixed-sized buffering methods*, which use a fixed-sized buffer for generating images, and *variable-sized buffering methods*, which use a variable-sized buffer [10]. Fixed-sized buffering methods normally require a lot of memory [6] or a lot of processing [1] [8]. Variable-sized buffering methods can be further divided into *the span buffering method* [4] [11] [14], which stores polygons as spans in the buffer and *the pixel buffering method* [2], which stores polygons as pixel fragments in the buffer. Because of the very nature of the variable-sized buffering methods, they all require run-time memory allocation.

The aim of our work is to produce a buffering method which helps to:

- Generate accurate images while using minimal amount of memory, and
- Allow interactive editing of the image stored in the buffer.

In order to allow image editing, we need to accumulate the scene information in the buffer and we have chosen the A-buffer method as our basic framework.

2. The A-buffer Method

The A-buffer uses two data structures called *pixelstruct* and *fragment*. Pixelstruct forms the basic element of a two-dimensional pixel array created at start and fragment is allocated at run-time for storing information about an object edge at a pixel. The definitions of them are shown in the following C code:

```

typedef struct {
    float z;          /* normally positive */
    union {
        Fragment *flist; /* fragment ptr */
        struct {
            unsigned char r, g, b;
            unsigned char a; /* coverage */
        } data;
    } fragORdata;
} Pixelstruct;

typedef struct Frag {
    struct Frag *next;
    short r, g, b;
    short opacity; /* transparency */
    short area; /* pixel coverage */
    short object_tag; /* primitive obj id */
    pixelmask m; /* subpixel bitmask */
    float zmin, zmax; /* depth values */
} Fragment;

```

In pixelstruct, **z** normally stores a positive depth value. When the pixel contains visible edges, **z** is set negative and **fragORdata** becomes a fragment pointer which points to a list of fragments.

In fragment, **zmin** and **zmax** are used for detecting surface intersection and for estimating the area of the pixel covered by each of the intersecting surfaces. **Opacity** is used to store the transparency of the surface. To simplify the discussion, we will not consider them here and we will use **z** to represent **zmin** and **zmax** in the following discussion.

At start, the pixelstruct array is initialised to have no fragments in it. When scan-converting polygons, if a pixel is fully covered by a polygon, the contents of **rgb** and **z** in pixelstruct are updated provided that the depth of the polygon at the pixel position is smaller than **z**. If, instead, the pixel is partially covered by the polygon, two fragments are created and are linked to the pixelstruct. The first fragment stores the polygon data and the second stores the information originally in pixelstruct. If polygon data is received while there is already a list of fragments in the pixel, a fragment is created to store the polygon data and is then inserted into the list in such a way as to maintain the list in depth order.

When inserting a fragment into a fragment list, merging is possible. If a fragment in the list is found to have overlapped in depth with the fragment being inserted, their **object-tags** are compared. These **object-tags** are integers assigned to non-self-intersecting primitive objects. If they are the same, the two fragments can be merged into one.

After all the polygons have been scan-converted, a packing process is called to obtain the colour value of each pixel in the buffer. The fragment list in each pixel is traversed in depth order and two operations are performed on each of the fragments in the list.

$$M_{in} = M_{search} \cap M_f \quad (1)$$

$$M_{search} = M_{search} \cap \sim M_f \quad (2)$$

where M_{search} , M_f and M_{in} are the search mask, the bitmask of the current fragment, and the bitmask representing the visible subpixels in M_f respectively. The number of bits set in M_{in} determines the colour contribution of the current fragment. M_{search} is initially set to all 1's. After the two operations, the subpixels covered by the current or any of the previous fragments in the list will be cleared. M_{search} is then passed on to the next fragment in the list and the two operations will be performed again until all the bits in it are cleared. The pixel colour will be the sum of all the weighted colours calculated from the processed fragments.

In fragment, **area** stores the alpha value of the fragment calculated analytically without considering all the other fragments in the list. When there is only one object edge crossing a pixel, **area** is used as the alpha value to calculate the final colour. When there is more than one edge crossing

a pixel, **area** can only be used by the first fragment to calculate its colour contribution to the pixel. The alpha value for each of the other fragments in the list will be obtained by counting the number of bits set in M_{in} of Equation 1. Since most of the edge pixels in a typical scene contain only one object edge, using **area** would give a more accurate result in that situation.

3. The Bitmask Index

In [7], Fiume and Fournier described a method for converting a polygon fragment into an 8×8 bitmask. Initially, a table is pre-computed to store all the possible bitmasks that a single polygon edge can produce. When a polygon fragment is found overlapping a pixel, the positions where each of the polygon edges crosses the boundaries of the pixel are quantised to one of the eight subpixel positions. The pixel boundaries and the subpixel positions where each edge enters and leaves the pixel form a four field index. This index is used to locate a pre-computed bitmask from the table. When the polygon has more than one edge crossing a pixel, separate indices are formed for each of the edges. All the indexed bitmasks will then be ANDed to form a single bitmask representing the polygon fragment.

Here, we apply this technique in the A-buffer. But instead of converting the bitmask index into a bitmask before storing it into the buffer, we store the bitmask index into the buffer. This bitmask index is composed of two fields, the *enterfield* and the *exitfield* as in Figure 1. In each field, 2 bits are used to indicate which of the four pixel boundaries the object edge enters and leaves the pixel; these are called *enterside* and *exitside*. Another 5 bits are used in each field to store the quantised subpixel positions where the edge intersects the pixel boundaries; these are called *enterpos* and *exitpos*. The reason for using as many as 5 bits to represent the subpixel position is to allow the buffer to be used even in applications which require high accuracy as 5 bits would give 32 quantised subpixel positions on each of the four pixel boundaries. We will describe the function of the sign bit later.

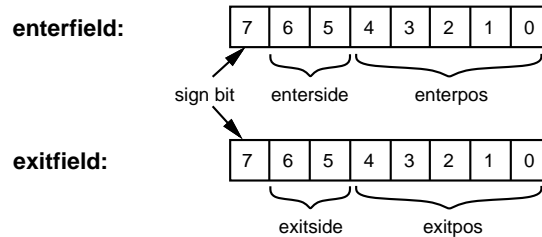


Figure 1: Definitions of the enterfield and exitfield.

The enterfield and exitfield are stored in each fragment structure. To form a bitmask index, which can be used to locate the bitmask from the bitmask table, we combine the enterfield and the exitfield but drop their sign bits. The enterpos and the exitpos will be truncated to the subpixel resolution of the bitmasks in use. Figure 2 shows an example of a bitmask index with 8×8 subpixel resolution.

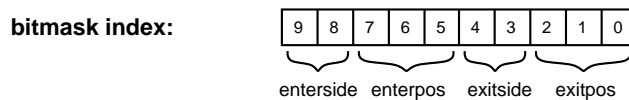


Figure 2: A bitmask index with 8×8 subpixel resolution.

Since the total number of bitmask indices we need to store in a fragment depends on the number of edges crossing the pixel by the same polygon, it is difficult to allocate a fixed amount of memory for them and storing each of these bitmask indices in a separate fragment structure can be very expensive. The method we use is that as in most cases there is only one bitmask index in a fragment, the fragment structure is designed to store just one bitmask index. When there is more than one bitmask index, we create a one-dimensional array to store the indices. The new definition of fragment becomes:

```
typedef struct Frag {
    struct Frag *next;
    float z;
    short opacity;
    DynamicID dynamicID; /* will be described later */
    char enterfield;
    union {
        struct {
            char exitfield;
            unsigned char r, g, b;
        } data;
        unsigned char *indexl;
    } dataORindexl;
} Fragment;
```

Since we have difficulties in implementing the suggested method for detecting surface intersection, the new definition does not include the related fields.

When there is only one bitmask index, the sign bit of **enterfield** is positive and **dataORindexl** is a data structure containing the **rgb** fields and the **exitfield**. If **enterfield** is negative, **dataORindexl** becomes a pointer to a one-dimensional array called the *indexlist* and that bit 0 to bit 6 of **enterfield** stores the total number of bitmask indices. The size of the indexlist depends on the number of edges and the first three locations of which store the rgb values of the fragment. Figure 3 shows an example of a fragment with 2 bitmask indices.

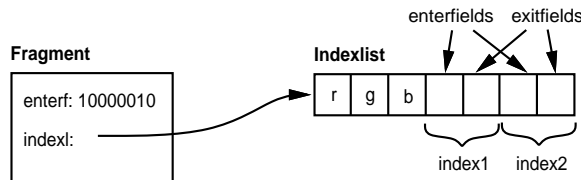


Figure 3: A fragment with 2 bitmask indices.

During the scan-conversion process, each polygon edge is clipped to the pixel boundaries, the **exitfield** of the edge is calculated while the **enterfield** is the **exitfield** of the preceding edge pixel. At polygon vertices, edges may not be touching any pixel boundaries, in such case we need to extend the edges until they intersect with the boundaries.

Merging is also possible with the new buffer (only for fragments with one bitmask index though). When the **enterfield** and **exitfield** of a fragment in the list are the same as the **exitfield** and **enterfield** of the fragment being inserted, if they also have the same depth value, we assume that the two fragments represent two polygons sharing the same edge and together they cover the whole pixel. So, we can merge them into one.

The packing process is similar to the one in the A-buffer except that M_f in Equations 1 and 2 is not available at once and requires a fetching of one or more bitmasks from the bitmask table.

Advantages and Disadvantages

Storing the bitmask index instead of the bitmask has three advantages:

1. *Memory usage*

- Since the size of **enterfield** and of **exitfield** is fixed to one byte each, memory usage is no longer dependent on the size of the bitmask.
- It generally saves memory when the resolution of the bitmask is higher than 4×4 .
- There is no need to allocate memory for **area**. When an accurate alpha value is needed, the **enterfield** and **exitfield** of the fragment can be used to calculate the alpha value analytically. Otherwise the bit count in each entry of the bitmask table is used.

This memory saving will become more important when we have dynamic objects in the buffer as will be described in Section .

2. *Accuracy of image generated*

- Instead of converting the bitmask index to a bitmask, we can modify the buffer to calculate the visible coverage value of each fragment analytically using a method such as [13] by clipping all the fragments in a pixel against each other. In fact, we can implement both methods in the buffer and let the complexity of a pixel determine which method to use.
- The buffer can be made to support more than one bitmask resolution and again the complexity of the pixel can be used to determine which table to use to give the best compromise between efficiency and accuracy. When a very high resolution bitmask is needed, we can generate the bitmask on the fly to save storing all the information.

3. *Zooming*

Since the bitmask index represents a subpixel resolution of 32×32 , zooming into the image can reveal more information. We will describe this next.

One drawback of storing the bitmask index is that the introduction of the indexlist further complicates the system. Since in our application, most of the objects in the scene are reasonably big, the effect of the drawback is minimal. However, when a scene contains a lot of small objects, a large number of indexlists may need to be created which would slow the system down.

Image Zooming

Zooming is a useful function in applications such as CAD/CAM systems. Traditional methods for doing this is by pixel duplication or sometimes by interpolation. These methods do not produce new information when zooming in. Nakamae et al. in [11] suggested a method for scaling by dividing each scanline into multiple sub-scanlines and storing polygons as spans in the sub-scanline buffer. This method is good for horizontal scaling. However, if a non-distorted zooming is wanted, the zooming factor will be limited by the number of sub-scanlines in a pixel.

Because of the high subpixel resolution of the bitmask index, we can perform zooming in our new buffer, which reveals more information. The maximum zooming factor here will depend on the subpixel resolution of the zoom image wanted. For example, if a 2×2 resolution is wanted, the maximum zooming factor will be 16.

To zoom in an image, an $N \times N$ array of integers and an $N \times N$ bitmask are used (where N is the product of the zooming factor and the subpixel resolution of the zoom image). Each pixel of the zooming area in the original image is processed in turn. The integer array is cleared and each fragment in the pixel is given an identifier according to its order in the fragment list. The enterpos and the exitpos of each **enterfield** and **exitfield** respectively are scaled by $\frac{N}{32}$. The bitmask is then

set to represent the coverage of the fragment. Any location which is cleared in the integer array but not in the bitmask will have the same location in the integer array set to the fragment identifier. After all the fragments in the pixel have been processed, the integer array represents the areas of the pixel covered by different fragments. The subpixels in the array corresponding to each pixel in the zoom image are then checked. The identifiers stored in them determine the colour contribution of each of the fragments.

For example, if the zooming factor is chosen to be 3 and the subpixel resolution of the zoom image is 4×4 , N will become 12. Figure 4 shows the mapping of a pixel in the original image to pixels in the zoom image (where '1' represents subpixel covered by the first fragment, '2' the second, and '3' the third). When a non-integer zooming factor is wanted, a pixel in the original image may not map to an integer number of pixels in the zoom image. In that case, we need to propagate colour values between neighbouring pixels.

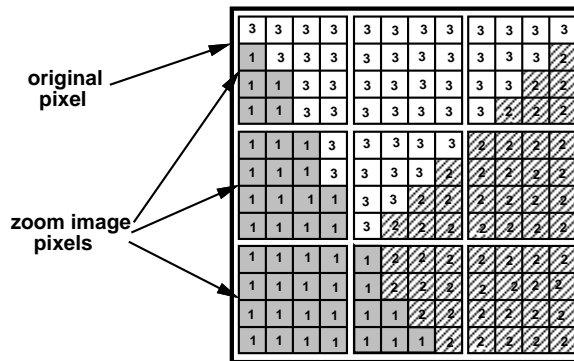


Figure 4: Mapping of pixels.

Plate 1 shows an image generated with the new buffer and Plate 2 shows a zoom image of the centre part of Plate 1 produced using pixel duplication. Plate 3 shows a zoom image of the same area using the method described here. The subpixel resolution of the zoom image is 4×4 and the zooming factor is 8. In it, we may notice blocks of pixels having similar colours. This is due to the fact that we have only one colour value for each fragment. Once a pixel is magnified, multiple pixels in the zoom image may result in the same colour. Another problem is surface intersection. Since z stored in each fragment does not contain any orientation information, performing surface intersection detection with it in the zoom image pixels is not possible. A method to solve these two problems is by interpolating both the colour values and the depth values with neighbouring pixels. However, this would increase the processing time for generating the zoom image and it also requires modification to the fragment structure for identifying surfaces.

4. Image Editing

Since generating a realistic image can take a long time, some image generators or animation systems would provide a scene editor and a fast renderer such as a wireframe renderer in addition to the main renderer which is usually more accurate but much slower [9] [12]. The scene editor allows the designer to edit the scene file and to see the effect of the changes immediately through the wireframe output. Once the editing process is finished, the main renderer is called to generate the final image. However, there are situations where we might not be satisfied with the output image even though we have "previewed" the wireframe output. The problem might be the surface colour of an object due

to some lighting effects or the overall appearance of the image. If this is the case, we will need to run the process again; i.e. re-edit the scene file and re-render the image.

In [14], Whitted described the software test-bed, which breaks polygons into spans and stores them in a span buffer. An anti-aliased image can subsequently be generated from the information stored in the buffer. Since the span buffer accumulates the spans generated, it is possible to use it to assist image editing. However, the problem of using a span buffering method for image editing is that whenever we modify the information stored in the span buffer, a complete shading operation is needed to generate a new image from the updated span buffer. Hence, we have chosen the A-buffer, which stores pixel colours of objects.

The idea here is to extend the A-buffer to a full scale buffer to keep all information (visible or invisible) in it. By doing this, it is possible to delete an object from the buffer and re-render it with an updated description without regenerating the whole image. However, simply keeping each pixel of every object as fragments is very expensive in terms of memory. The method we use is to identify objects as static and dynamic objects; a dynamic object can be edited where a static one cannot. We add a field, **dynamicID**, into the fragment structure. This **dynamicID** stores a unique identifier allocated to each dynamic object. All static objects will have their dynamicID's set to zero. During the scan-conversion process, only fragments with the same dynamicID can obscure each other; again obscured information would be removed from the buffer. With such a mechanism, when a dynamic object is removed, the information behind it will be readily available and re-running of the packing process around the region will reveal the objects behind the removed one.

There is one thing to note here. Since the buffer may contain a lot of fully covered fragments representing different dynamic objects, it is worth allocating a special bitmask index to speed up the detection of this case. The special index chosen is zero. The two reasons for this choice are that we can simply use a boolean operation to detect the case and that an edge is unlikely to have the same enterfield and exitfield. Hence, if we detect a situation that both enterfield and exitfield are zero, we know that the fragment fully covers the pixel.

To be able to locate an object description with a dynamicID, a dynamic object table is needed. This table is indexed with the dynamicID and each entry in the table stores a pointer to the corresponding object structure as shown in Figure 5. We can also associate more than one object to have the same dynamicID so that any editing function applied would always be on the whole group of objects.

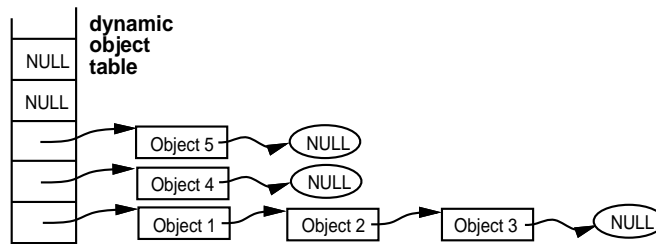


Figure 5: The dynamic object table.

With such a modification to the A-buffer, it is possible to perform many interesting functions. The following outlines some of them.

Object Selection

Object selection is a function used very often in applications such as CAD and interactive graphics systems. By specifying a point on the screen, we want to know which object has been selected. A common way of doing this is to go through the scene database and find out all the objects that contain

the given coordinate and then return the one with the smallest depth. Such an operation is very time consuming. The Geometry System [5] performs the same operation in hardware with a much greater speed.

By adding dynamic objects to the A-buffer, we can provide interactive object selection even without hardware support. Given an x-y coordinate, we can immediately identify the selected dynamic object by checking the dynamicID stored in the first fragment of the pixel. By going through the selected object description, it is also easy to identify the selected polygon and surface within the object.

Editing of Dynamic Objects

Before generating an image, any objects whose definitions may need to be changed are specified as dynamic objects. After the image has been generated, we can move any dynamic object around, change the colour of one of its surfaces or change the size of it by first deleting the object from the buffer and then re-rendering it with the updated description.

Animation

When generating an animation sequence, there are often only a few objects which move from frame to frame in front of a static background. We can specify all objects that move in a particular sequence as dynamic objects. After an image frame has been generated and recorded, the object descriptions of all the dynamic objects that move in the next frame are updated. These objects are then removed from the buffer and are regenerated again from the new descriptions. Generating a 2D cartoon animation sequence in the buffer can also be simple. Moving a cartoon object is simply done by translating all fragments in the region with a given dynamicID by a displacement value.

Windowing Support

As windowing systems are becoming more and more common, hardware support is provided in some advanced workstations. In our new buffer, windowing support is readily available. It is to treat each window as a dynamic object and store it in the buffer. Moving and deleting a window become a very simple task and there is no need to do window clipping. It also allows windows of arbitrary shapes to be used with anti-aliasing effects around the window edges. The major problem of this method is its memory consumption – keeping each pixel of a window as a fragment is very expensive in terms of memory. However, as the price of memory is getting cheaper such an application may be possible in the near future.

Advantages and Disadvantages

The introduction of dynamic objects allows selected objects to be edited after the image has been generated whilst minimising memory usage. Since only the objects involved are regenerated, the method can provide a form of interactive image editing which is difficult to do with conventional methods. (In our implementation, the satellite shown on the right of Plate 1 takes about 0.4s to delete and 2.6s to move it to a new position when the system is running on a DECstation 2100, which has a MIPS R2000 cpu running at 12.5MHz, with all the code written in C.) The new buffer can also be used as a tool for animation by writing scripts to move, for example, cartoon objects around the buffer to generate a sequence of images.

However, the disadvantage of adding dynamic objects into the buffer is that a lot of fragments will be needed which implies a lot of memory. To give the reader some idea, if we use 4 bytes for the depth and 1 byte for the dynamicID, each pixelstruct will take 8 bytes of memory and each fragment will take 16 bytes. In our statistics, a typically complex image of size 512×512 without dynamic objects creates not more than 27 thousand fragments. Such an image will need all together about 2.4MBytes

of memory. If we have 50 dynamic objects each of 100×100 pixels in size, approximately 7.6MBytes of extra memory will be needed for storing the dynamic objects.

5. Fragment Allocation

In order to reduce the frequency of calling the dynamic memory allocator provided by the compiler, which is usually not very efficient and tends to use up extra memory, a simple method is used here for allocating and freeing of fragment memory. A one-dimensional array of fragments are created at the start. Each fragment in the array is made to point to the next to create a list of free fragments. Whenever a new fragment is needed, the first one in the list is returned. If the list is empty, another fragment array can be created. When a fragment is no longer needed, it is put back to the head of the list.

6. Acknowledgement

We would like to acknowledge David Brown for initiating the idea of extending the A-buffer into a full scale buffer.

References

- [1] G. Abram and L. Westover. *Efficient Alias-free Rendering Using Bit-masks and Look-up Tables*. Computer Graphics, Vol. 19, No. 3, July 1985, pp53-59.
- [2] L. Carpenter. *The A-buffer, an Antialiased Hidden Surface Method*. Computer Graphics, Vol. 18, No. 3, July 1984, pp103-108.
- [3] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.D. thesis, Computer Science Department, University of Utah, 1974.
- [4] E. Catmull. *A Hidden-Surface Algorithm with Anti-Aliasing*. Computer Graphics, Vol. 12, No. 3, Aug. 1978, pp6-11.
- [5] J. H. Clark. *The Geometry Engine: A VLSI Geometry System for Graphics*. Computer Graphics, Vol. 16, No. 3, July 1982, pp127-133.
- [6] F. C. Crow. *The Aliasing Problem in Computer-Generated Shaded Images*. Communication of the ACM, Vol. 20, No. 11, Nov. 1977, pp799-805.
- [7] E. Fiume and A. Fournier. *A Parallel Scan Conversion Algorithm with Anti-aliasing for a General-Purpose Ultracomputer*. Computer Graphics, Vol. 17, No. 3, July 1983, pp141-150.
- [8] P. Haeberli and K. Akeley. *The Accumulation Buffer: Hardware Support for High-Quality Rendering*. Computer Graphics, Vol. 24, No. 4, Aug. 1990, pp309-318
- [9] P. Jackson. *MacroMind 3D*. Personal Computer World, Dec. 1990, pp194-197.
- [10] W. H. Lau. *Realistic 3D Image Composition*. Ph.D. thesis, Computer Laboratory, University of Cambridge, 1991.
- [11] E. Nakamae, T. Ishizaki, T. Nishita and S. Takita. *Compositing 3D Images with Anti-aliasing and Various Shading Effects*. IEEE Computer Graphics & Applications, Mar. 1989, pp21-29.

- [12] A. Raja. *3D Studio*. Personal Computer World, Apr. 1991, pp208-212.
- [13] K. Weiler and P. Atherton. *Hidden Surface Removal Using Polygon Area Sorting*. Computer Graphics, Vol. 11, No. 2, July 1977, pp214-222.
- [14] T. Whitted and D. M. Weimer. *A Software Test-Bed for the Development of 3-D Raster Graphics Systems*. Computer Graphics, Vol. 15, No. 3, Aug. 1981, pp271-277.

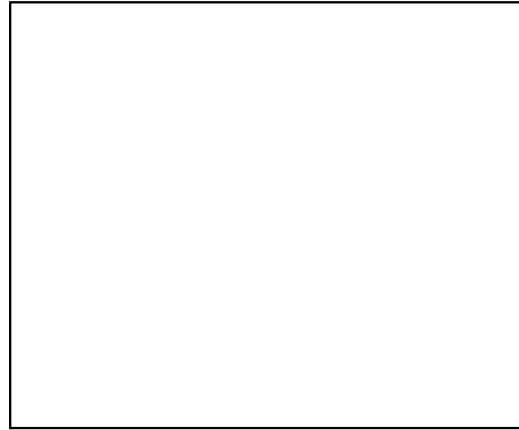
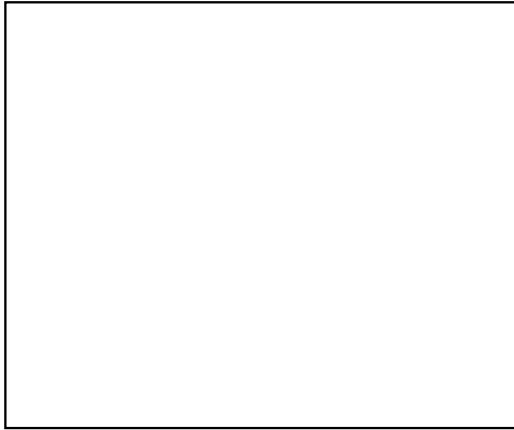


Plate 1: Image generated with the new buffer. Plate 2: Zooming using pixel duplication.

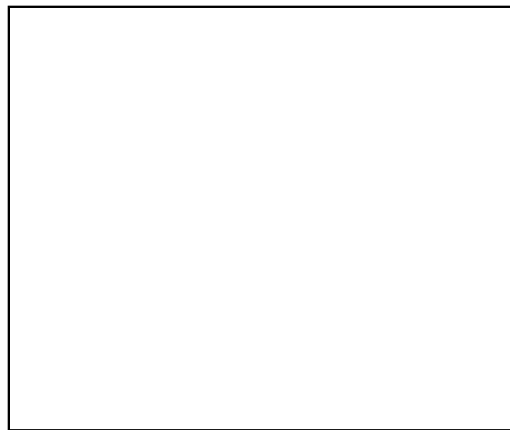


Plate 3: Zooming using the new method.