EUROGRAPHICS '99 / P. Brunet and R. Scopigno
(Guest Editors)

Volume 18 (*1999*), *Number 3*

An Efficient $2\frac{1}{2}$D Rendering and Compositing System

Max Froumentin and Philip Willis

University of Bath, United Kingdom

Abstract

We describe a method for doing image compositing using either 2D geometric shapes or raster images as input primitives. The resolution of the final image is virtually unlimited but, as no frame buffer is used, performance is much less dependant on resolution than with standard painting programs, allowing rendering very large images in reasonable time. Many standard features found in compositing programs have been implemented, like hierarchical data structures for input primitives, lighting control for each layer and filter operations (for antialiasing or defocus).

1. Introduction

With the increasing use of digital techniques in cinema, television and photography, there is a growing need for software capable of efficiently creating and compositing high-quality digital images. Compositing merges picture data usually originating from live film, or possibly digitised artwork and computer synthesised images. Special effects for cinema and television traditionally require compositing live films (e.g. matte techniques), although compositing live film and synthetic images is more and more common. Computer animation usually requires compositing purely synthetic images.

Software for compositing images into high-quality output must be able to produce high resolution pictures, up to 1k for TV, up to 4k for photography and up to 8k for film. Resolution can be higher still if high-quality supersampling is used. In terms of computer memory, a single frame can then take up several hundred megabytes. Moreover, when multiple images are composited into a single frame, multiples of that amount can be required. The total amount of memory needed for convenient compositing is only available today on high-end computers. The process is also very demanding on CPU speed: as compositing computations need to be carried out on every pixel, they must be executed several million times for each image. Again, only high-end computers allow doing this quickly enough.

We have designed an image compositing software library that is able to produce very high resolution images without using a frame buffer. An image is computed scan-line by scan-line and is compressed as it is created by producing and manipulating colour-coherent spans. This allows the compositing process to delay single pixel manipulation as much as possible, in order to save on both memory and processing time. Input images or 2D primitives can be transformed using standard 3D homogenous transformations as well as boolean operations. Compositing is performed using a cel model (traditionally used in cartoon animation) which provides standard layering features as well as complex lighting of each layer. Effects such as defocusing, transparency and anti-aliasing are also available. We have produced images up to 1M by 1M (1000 gigapixels) showing the effectiveness of the approach.

The paper is organised as follows: section 2 presents related research, section 3 introduces our software library from the user's point of view, section 4 gives details on the library's implementation, section 5 discusses performance issues and section 6 presents some of our results.

2. Related work

The main limitations of current compositing software are image size (very few systems go beyond a resolution of 4k) as well as a lack of sources and compositing operations.

The usual method to handle large pictures is to design a memory allocation model which stores in the frame buffer only the part of the picture that is modified by the compositing operation. However the reduced amount of memory that such a model needs is balanced by an increased overhead

time. Shantzis[8], for example, has introduced a model for processing and compositing pictures that optimizes the size of the frame buffer as well as the efficiency of compositing algorithms.

Recent results have also shown significant advances in compositing algorithms themselves, in related areas such as computer-aided animation (CAA) as well as real-time compositing:

Pixel systems

The CAA software system introduced by Wallace for Hanna-Barbera Studios[11] allowed manipulation of raster images of digitised hand-drawn artwork. Although using image buffers was not avoided, the system featured complex lighting and image merging operations.

Berman, Bartell and Salesin's *multiresolution images*[1] are also based on raster images only. However, using wavelet representations, raster images having "different resolutions in different places" can be created and composited.

Lengyel and Snyder[5] have recently proposed an image compositing system based on Microsoft's Talisman hardware architecture[9]. As their main concern is speed rather than image quality, image resolution is strongly limited. The system is thus only suitable for raster display applications such as video games.

Vector Systems

UltraPaint[12, 15] is a 2D system that was designed for producing very high definition pictures. But as it only manipulates vector primitives, no digitised data can be used as input.

Tic-Tac-Toon[3] is a more recent CAA system which aims at being used in every stage of the traditional animation process. Using vector primitives allows saving on frame buffer memory, however it also suffers from the impossibility of importing digitised data. It is also not optimised for high resolutions as a single frame can take up to a few minutes to compute.

3. The programming interface

Our Image Rendering and Compositing Software (IRCS) is a software library written in C providing a data structure for layered 2D graphics objects as well as scan-converting procedures.

3.1. The model

IRCS allows the creation of data structures that reproduce the traditional cartoon animation process (as described by Patterson and Willis[7]): artwork is drawn and colours are painted on transparent celuloid sheets (called *cels*). For each

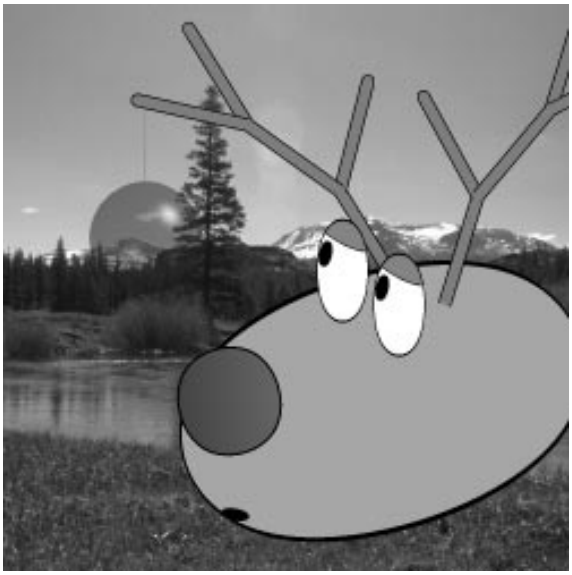


Figure 1: *A sample image made from five cels.*

frame several cels are overlayed for convenience of manipulation. For example, layers may be used in order to separate the moving parts of a sequence (e.g. a character's lips) from the non-moving parts (e.g. backgrounds). The latter can then be reused in other frames. The cels used for a particular frame are stacked and are placed on a *rostrum* (Figure 2) which includes a camera, a set of trays to hold the cels and lights.

The IRCS library allows programmers to create virtual stacks of cels, which include 2D graphics primitives that are combined using planar transforms and boolean operators (Figure 3).

As with a real rostrum, lighting can be controlled. Actually, more control is given to the library's users as front and back lights can be different for each cel, instead of having a single pair of lights for the stack, as in Figure 2. This allows interesting colour effects, particularly in CAA, as was shown by Willis and Oddy[13, 6].

At each stage of the primitive/object/cel hierarchy, a 3×3 homogeneous transformation matrix is specified, allowing standard operations like translations, scalings, rotations or general 2D perspectives (allowing simple 3D effects like vanishing points). More specific operations can be specified at the cel level (e.g. defocus) or at the stack level (e.g. brightness).

In order to simulate the placement of each cel relative to the camera, a *viewport* is specified, indicating the area of that cel to be rendered. This, in conjunction with the multiple layering, permits simulation of panning, zooming and parallax.

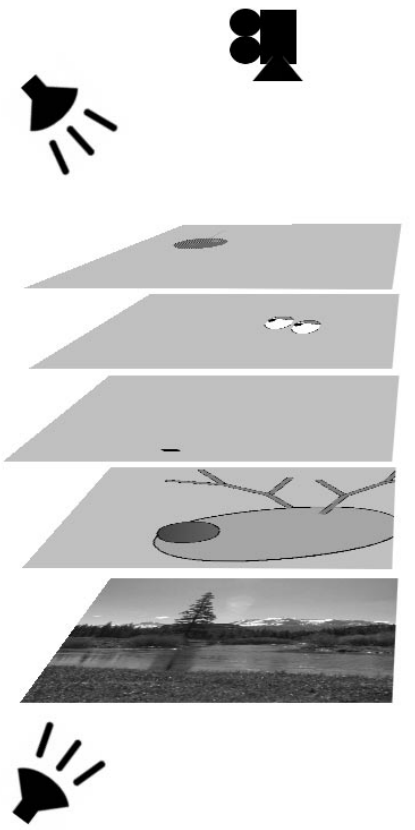


Figure 2: *A rostrum*

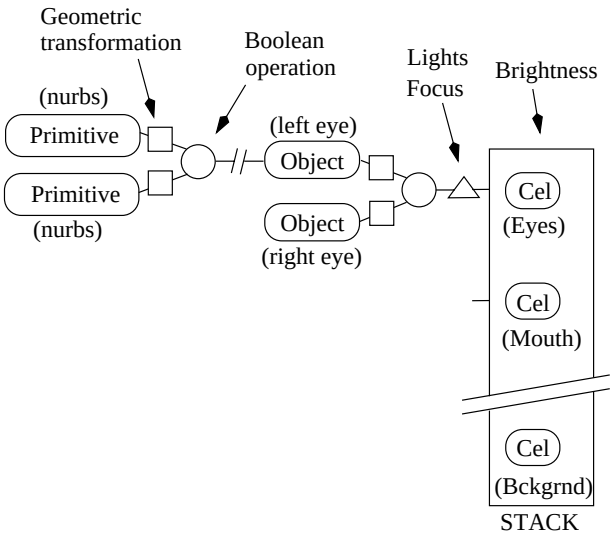


Figure 3: *Stack structure*

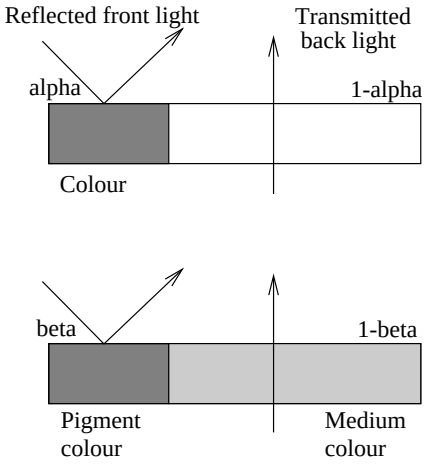


Figure 4: *RGBA (top) and BCM (bottom) colour models*

3.2. Primitives

A primitive is generally defined as a closed region in 2D space. At this time, rectangles, triangles, circles and closed NURBS curves have been implemented. Any other 2D primitive can be added to the system by providing a procedure for rendering the intersection of that primitive and a scan-line as a pixel array.

3.3. Textures

To each primitive is assigned a texture which stores information on the primitive's colour, independently of lighting and other objects. Currently a texture can be of one of three types:

- `flat_colour`: a single colour is specified for all the points inside the primitive.
- `gradient`: a triangle (containing the primitive) is specified, as well as the colour of each of its vertices. The colour of a point inside the primitive is then determined by linear interpolation of these colours.
- `picture`: a raster image is mapped to the primitive's bounding rectangle. This texture type is used to introduce digitised images into the compositing process. The image mapping transformation is modified by the object transformations (as specified in the cel hierarchy), allowing image rotation, translation or perspective effects (see Figure 15 for an example).

Colours are specified using the *Beta colour model*[6] (BCM), an extension of the well-known RGBA model. BCM models filtered colour (back lit) as well as scattered colour (front lit) to correctly simulate the transparency and lighting effects that can be obtained with real cels (Figure 4).

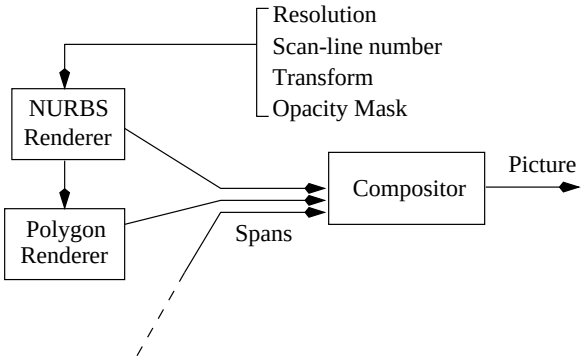


Figure 5: *The rendering/compositing data path*

3.4. Rendering

Once a stack structure has been created and additional parameters (such as lights and viewport) have been specified, rendering can be performed. Rendering a stack structure is equivalent to shooting a real stack on film: a raster image is computed and output to one of the available raster devices (screen, film recorder, image file, etc.)

As no frame buffer is used rendering and compositing are performed scan-line by scan-line, each one being compressed as soon as it has been computed. Because rendering and compositing are combined there is no need to store rendered components for later compositing, saving on both time and storage. The fact that IRCS does not rely on frame memory also allows images to be rendered with almost no limit on resolution or aspect ratio. Antialiasing is also available by supersampling and filtering. The user may supply any digital filter (Turkowski[10] gives some useful examples), though box and Gaussian functions are built-in.

4. Implementation

This section describes the methods used internally for rendering a stack. The computation of a scan-line's pixels is handled by two software components: the *renderer* that computes the intersection of a single primitive with the scan-line into a list of colour-coherent spans, and the *compositor* which merges the results from the renderers (Figure 5). The resulting list of spans is finally converted into a row of pixels.

4.1. Overview of rendering

Given the output resolution and a scan-line number, the rendering procedure queries the stack for the colour-span list that represents its intersection with the scan-line. The stack intersection procedure in turn queries the cels it contains for their intersection with the scan-line, and so on down to the primitive where each renderer returns the pixel data for the intersection between that primitive and the scan-line. The compositor then goes back up the object hierarchy, merging the intermediate results into the final colour span list.

Unlike other systems that often apply operators on input data regardless of whether they will be visible in the end or not, IRCS only renders the part of a cel that is visible: because of the scan-line loop, only the part of the frame inside the defined viewport is rendered (*on-demand* computation). Inside the loop, rendering is performed from the front cel backwards, so that covered parts of the back cels are detected and not rendered. This allows the renderers to avoid unnecessary computation of hidden objects and therefore improves overall rendering speed. The algorithm is described in detail in section 4.4.2.

4.2. The span structure

Image data generated by the renderers and merged by the compositor are stored as lists of colour-coherent spans. A span encodes an array of pixels by storing the indices of the starting and ending pixels in the scan-line as well as colour information for the pixels represented. This information can be of three types, depending on the types of textures that are used (Figure 6):

- `single_colour`: a colour value that represents the common colour of all the span's pixels.
- `colour_gradient`: two colour values that represent the colours of the span's leftmost and rightmost pixels. The colours of the pixels in-between can be computed by linear interpolation of these two colours.
- `colour_array`: the colour of each pixel is explicitely stored in an array.

As a primitive might intersect the scan-line at more than a single span all the span composition functions work on linked lists of span, that can possibly cover the whole horizontal range of the image.

The span structure not only offers a means of compressing pixel data but also allows the rendering to be independent of the width of the output image. Of course this is only possible under the condition that no operations on single pixels are performed during the rendering/compositing process. This is possible if:

- No `colour_array` spans is used.
- The rendering/compositing algorithms make use of the span structure by working directly on spans and not on individual pixels.

IRCS is designed to apply these rules whenever possible, by having all compositing algorithms linear relative to the number of spans, and also by producing either `single_colour` or `colour_gradient` spans. If the renderers do not produce `colour_array` spans, then the compositor will not produce any `colour_array` span either (see section 4.4).

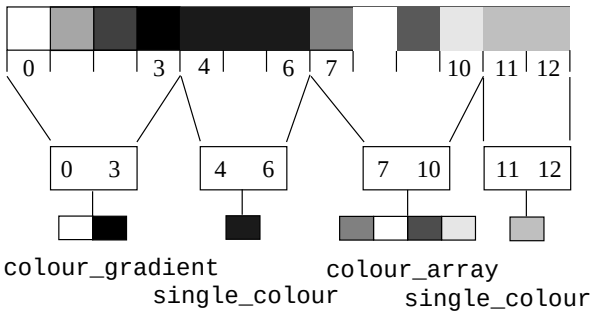


Figure 6: *A pixel array (top) represented as a list of spans (bottom)*

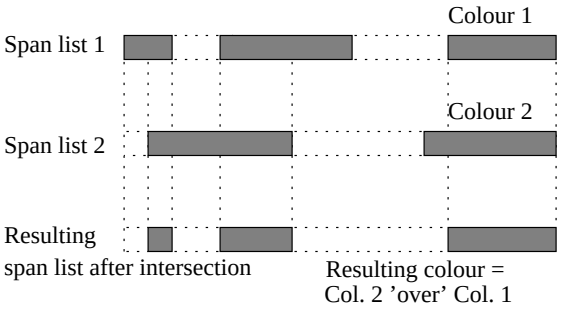


Figure 7: *Example of Span List compositing (intersection)*

4.3. The primitive renderers

As we described above, a rendering function must be written for each primitive type. A description of the part of the primitive to be rendered as well as information on the output resolution will be passed to that function, which will then return a row of colours stored as a span list. More precisely, the compositing modules send the following parameters to the renderers:

- primitive data (geometry and texture),
- primitive transform (a 3×3 matrix),
- image output resolution (in pixels),
- scanline index,
- opacity mask (computed from front cels).

From these parameters, the primitive renderer is able to compute the corresponding span list. The geometry of the span list is obtained by computing the intersection (in object coordinates) and by quantizing the floating-point values to pixel indices. The colours of the resulting span list are then determined according to the object's texture, by simple assignment (in the case of a `flat_colour` texture), by computing each span's left and right colours (for `gradient` textures) or by 2D inverse texture mapping (for `picture` textures). In this last case, fast inverse texture mapping techniques[4] have been implemented to reduce as much as possible the time used to process each individual pixel.

4.3.1. Precision

Although computing the intersection between a line and a planar shape is usually a simple task special care must be taken in our case. Because of the possibly very high resolution of the output images, control of floating-point number precision is essential. For example, the NURBS curve intersection algorithm we use[14] does not perform well if the end points of a NURBS segment lie exactly on the intersecting line: some intersections will be "missed" or incorrect ones will be calculated, resulting in noticeable visual errors. We have therefore implemented an extension of the Willis and Oddy intersection algorithm which slightly moves a curve's control points up or down if it is located such that a precision problem could occur.

4.4. Compositing

The span lists computed from the intersection of the primitives and the scan-line are composited according to the object hierarchy. The problem can be split into two: finding the geometry of the resulting spans, and determining their colour:

- Geometry: the bounds of the resulting span list are computed according to the input spans and the boolean operation specified (Figure 7 shows an example of intersection).
- Colours: the colours of the input span lists are merged in order to simulate the superimposition of paints. The Beta Colour Model defines the *over* operator, which computes the colour resulting from overlaying two colours, as well as the *lighting* operator, which gives the resulting RGB colour resulting from lighting a BCM colour with a front and a back light[6].

The colour type of the spans generated by compositing depends on the type of the spans that are composited. For example, compositing two `single_colour` spans results in a `single_colour` span. Table 1 gives the resulting type in each case. Note that, regarding efficiency considerations pointed out in section 4.2, `pixel_array` spans are only generated by other `pixel_array` spans. This shows that an image that does not use `picture` textures will never use pixel arrays and will therefore be computed independently of horizontal resolution.

	`single`	`gradient`	`array`
`single`	`single`	`gradient`	`array`
`gradient`		`gradient`	`array`
`array`			`array`

Table 1: *Span type compositing*

4.4.1. Compositing objects (inside a cel)

The operation of compositing objects inside a cel is slightly simpler than compositing cels inside a stack as in the latter case cel lights must be taken into account.

Inside a cel the boolean operations specified in the stack data structure (union, intersection, difference or exclusive-or) are applied to the span lists of the corresponding objects or primitives. To compute resulting colours, the *over* operator, which computes the colour of two superimposed colours described in the BCM representation, is used.

4.4.2. Compositing cels

When compositing span lists that represent a cel's colours, handling geometry is simpler than inside a cel: only the *union* boolean operator is used, in order to reproduce the physical superposition of cels. Nevertheless colour operations are somewhat more complex: what is computed is not the colour of an object (as it is done inside a cel) but the *light* that reaches the camera (the colour of which is obtained by mixing the RGBA colours of the light sources and the BCM colours of the objects). The result of rendering a scan-line is then an array of RGBA values, which can be directly written to the output device.

However the colour of the light that comes from a cel depends on that cel's back and front lights as well as the light coming from the cels behind it. This means that computing that colour must be performed in *back-to-front* (whereas Wallace's system[11] computes the composition of two adjacent cels in any order, the BCM colour model and the fact that we use different lights at each cel do not allow us to do so.) This goes against the masking process which as we saw above must be performed in front to back order.

The algorithm we designed allows both features to be implemented by doing two passes: front-to-back, then back-to-front:

1. the cels are processed from front to back (independently of lights and other cels) and the resulting span lists stored in an array of span lists.
2. The span list array is processed from back to front, adding lighting information at each cel and merging colour values to produce the final span-list for the scan-line. Figure 8 shows how this pass is performed: operator $\otimes$ merges the cel's colour, the colour of the cel's front light and the colour of the light coming from behind into a resulting colour propagated forwards, using the BCM *lighting* operator:

$$R = Cel_{\beta} Cel_{pigment} L_F + (1 - Cel_{\beta}) Cel_{medium} L_B$$

where $(Cel_{pigment}, Cel_{medium}, Cel_{\beta})$ is the BCM colour of the cel, L_F is the colour of the front light and L_B is the colour of the light coming from the back. Operator $\oplus$ merges the light coming from the back cels with the back light of the front cel, simply by adding the red, green and blue colour components.

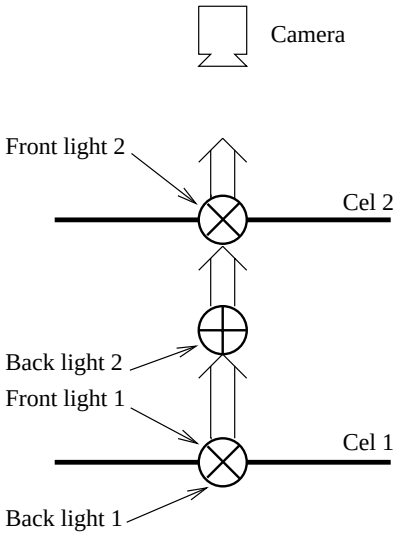


Figure 8: *Back-to-front computation of light reaching the camera through cels*

A more realistic simulation of lighting should in principle take multiple light reflections between cels into account, possibly leading to a complete 1D ray-tracing algorithm. However our method already allows interesting lighting effects without dramatically increasing computation time, and the results are easier to visualize when planning the lighting.

4.4.3. Opacity Masking

The opacity mask is a particular span list (not containing any colour information) that indicates the areas of the current cel that are visible on the current scan-line or that are hidden by front cels. This list is updated each time a cel's span list is computed, by including into it the opaque parts of the span list. The mask can help avoid computing intersections and colours (in particular texture mapping) that will not be visible in the final image. Using an opacity mask is not necessary to compute the result but can greatly accelerate rendering.

Figure 9 shows a span list computed as the intersection of a NURBS primitive with a scan-line. The opacity mask (1), which has been computed from the intersections of the cels that are in front of the current one, is used to avoid unnecessarily computing intersections and to reduce the size of the result, yielding (2). After it has been computed, the result is used to update to opacity mask (3) which can then be used when rendering underlying primitives.

4.5. Filtering

As antialiasing is applied to the final picture, only RGBA colours have to be filtered (section 4.4.2). For the standard box filter (that averages subpixels), averaging each channel

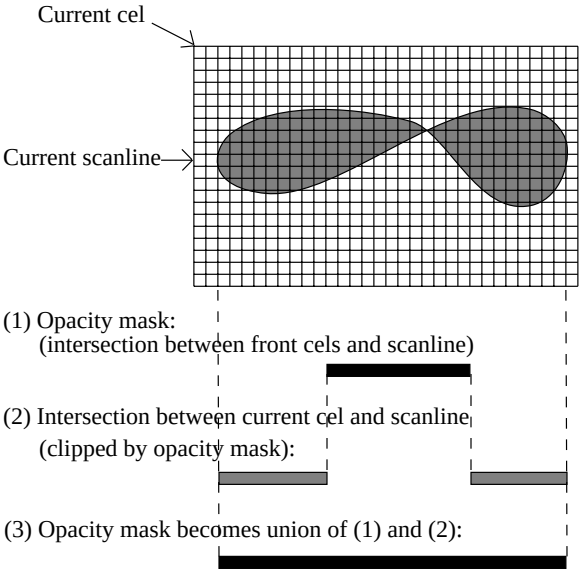


Figure 9: *a span list resulting from rendering a single NURBS primitive*

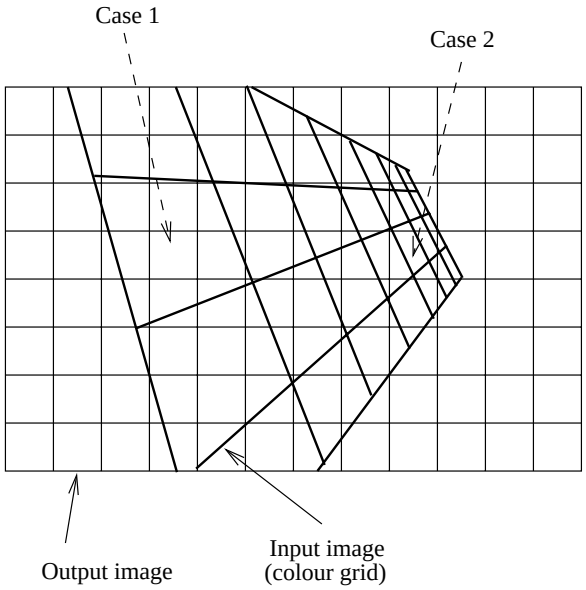


Figure 10: *Texture-mapped primitive under a primitive transform*

of the subpixels gives wrong results[2]. Pre-multiplication of each colour channel by alpha is necessary, leading to the formulas used to filter N RGBA pixels $(Colour_i, \alpha_i)$:

$$\alpha = \sum \alpha_i / N$$
$$\alpha Colour = \sum \alpha_i Colour_i / N$$

When other filters are used (e.g. Gaussian), filtering formulas become:

$$\alpha = \sum \alpha_i Filter_i / N$$
$$\alpha Colour = \sum \alpha_i Colour_i Filter_i$$

where $Filter_i$ are the values of the filter function of width N.

BCM colours are filtered when defocus is applied to a cel. Similarly to RGBA filtering, pre-multiplication of the pigment by the β channel and of the medium by $1-\beta$ is necessary, which results in:

$$\beta = \sum \beta_i Filter_i$$
$$\beta Pigment = \sum \beta_i Pigment_i Filter_i$$
$$(1-\beta) Medium = \sum (1-\beta_i) Medium_i Filter_i$$

5. Performance

As we have shown, using colour-coherent spans instead of pixel arrays allows the design of an image rendering system that does not unnecessarily rely on large amounts of memory. In this section we investigate the advantages of our method in terms of computation time, which we also pointed out is an important problem.

The general principle is pixel-independence: ideally image generation algorithms should be as independent as possible of the resolution of the image. The complexity of standard frame-buffer based rendering systems is $O(2^n r^2)$, where n is the number of objects and r is the resolution. This reflects the fact that all r^2 pixels are computed independently, each time compositing all n objects in a binary way.

Our system reduces this result to: $O((2^n + r)r)$ For each scan-line, a list of spans is computed independently of r in 2^n steps, then the spans are converted into a pixel row in r steps. These two operations are repeated r times to produce all the scanlines of the final image. This shows that for a sufficiently large number of objects, our method is of an order of r times faster.

5.1. `pixel_array` spans

The above result is valid only if no individual pixel is manipulated during the production of the span lists, i.e. no `pixel_array` span is used. `pixel_array` spans occur when a picture texture is applied to an object.

This could be avoided by considering the object as a colour grid (Figure 10) and by rendering it using either a span for each grid cel (when a single texture colour is mapped to several output pixels – case 1) or a single-pixel span covering several cels (several texture colours are mapped to a single output pixel – case 2).

The consequence would be that, as no pixel array is used,

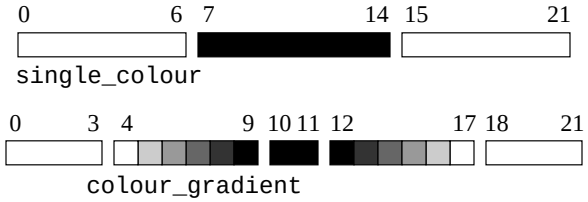


Figure 11: *Span list filtering*

the span list is really independant on output resolution, and then the maximum complexity of the rendering process is indeed $O(2^n r + r^2)$. However, this corresponds to increasing n to include each single input pixel as a primitive. There would be a potentially huge amount of memory used as many single-pixel spans can be created. Moreover, case 1 is in practice unlikely to occur as the final image would distinctly show individual texture pixels, which is usually not desirable.

5.2. Antialiasing and defocus

Antialiasing and defocus are two operations that act on the whole picture and on a single cel respectively. Antialiasing is performed by supersampling the image and then filtering the subpixels, and defocus is performed by filtering pixels at output resolution. Again, to be as pixel-independent as possible, filters are not applied on individual pixels but on span lists. For example, Figure 11 shows how a 3-span list (22 pixels) is filtered into a 5-span list using a box filter.

5.3. Output image format

Most compressed image file formats (like JPEG, Targa or TIFF) use linear encoding methods: the pixels of a scan-line can be computed in linear time with respect to horizontal resolution. As the operation of converting a span list to an array of pixel is comparable to that of decoding an image file, we can save the span list information directly into the image file (if the output of IRCS is a file). The image viewer programme will then decode the pixels at the time of viewing, as is usually done.

This allows not doing pixel decoding during rendering, which therefore makes rendering complexity $O(2^n r)$. Unlike most image generating programmes pixel compression is unnecessary as spans are already a form of pixel compression. Ideally, image rendering might perhaps be made completely linear with respect to resolution (complexity $O(2^n)$ for rendering, and $O(r^2)$ for converting to pixels). However this would require a 2D span encoding scheme that has yet to be developed.

The obvious choice for an image compression scheme adapted to our system is *run-length encoding* (RLE), where consecutive pixels of the same colour are encoded by storing a run-length *packet* containing the pixels common colour, along with a count value. RLE is used in formats such as Targa or BMP. We chose to use an extension of Targa, in order to account for gradient spans:

- `single_colour` spans are stored as run-length packets.
- `colour_array` spans are stored as unencoded pixel packets.
- For `colour_gradient` spans, we introduced a new packet type that contains the colours of the span's ends as well as its length.

We also store all pixel counts on 15 bits instead of 7 on the original Targa format (allowing up to 32768 pixels per packet), to compress high resolution images more efficiently.

Each of the three types of extended-Targa packets can be decoded to pixels linearly, only requiring a linear interpolation of colours in the case of a gradient span.

6. Results

6.1. Test Images

Figure 15 shows four images generated with our system and used to compile performance statistics.

Image 15(a) consists of 1000 randomly generated closed NURBS shapes. Each shape has its colour and transparency randomly allocated. The shape of each is determined by randomly placing four control points within the square, and by computing random knot values. All the shapes are overlayed within the same cel. The lighting is one back light and one front light of fully bright white. The purpose of this picture is to test the fundamental rendering speed of the software (section 6.2).

Image 15(b) combines three pixel images, two for the people and one for text and logo in the background. These are placed as textures on rectangles, which are then positioned to build the composite scene. In addition, the background has a gradient texture giving a shaded effect. This image demonstrates compositing of pixel images (including a perspective distortion of the foreground people), as well as defocus effects (text and logo).

Image 15(c) combines a wider range of objects. The foreground face and the flowers are taken from a single frame of a digitally-encoded movie sequence of full professional quality. Each frame is recorded at 2048 by 1536 pixels, with logarithmic encoding of colour at 30 bits per colour. The movie was recorded under studio conditions with low contrast lighting. The background image is a scene showing a view of Bath University at 768 by 512 pixels resolution. The large red "clover leaf" shape is a NURBS shape and has progressive translucency (top to bottom). The whole picture was built up by two texture mappings onto separate cels and the lighting was adjusted in each cel to correct brightness differences between source images. The purpose of this image

Figure 12: *Sample NURBS image*

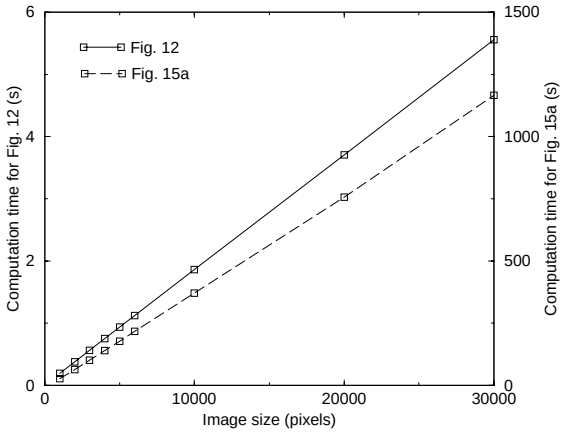


Figure 13: *Rendering time for Figures 12 and 15(a)*

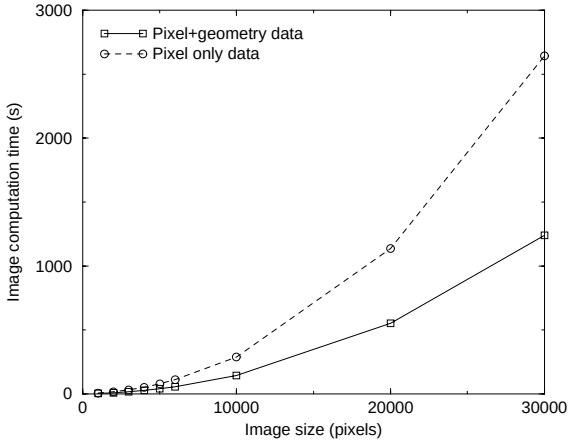


Figure 14: *Rendering time for a complex image*

was to demonstrate combining image data from disparate sources.

Image 15(d) is a high magnification re-rendering of the region of image 15(c) outlined by the black square. It shows quite clearly that the geometrically-defined "clover leaf" shape retains a sharp outline, while the image data reveals the usual pixelisation effects.

6.2. Performance

Having demonstrated the various components of the renderer, we then took the geometrically-defined picture Figure 15(a) and re-rendered at increasing resolutions. The tests were performed on one processor of an SGI Origin 2000 computer.

In what follows, each rendering created a square image. The results are plotted against a horizontal axis which shows the number of pixels along a side of this image. That is, the full image would contain that number of pixels squared. As our tests ran up to 30k along a side, the notional image size ran up to 1 gigapixels. These are non-trivial demands of any renderer.

Figure 13 plots the computation time against image size for two images composed of respectively 10 and 100 NURBS spahes (Figures 12 and 15(a)). It clearly shows that the rendering time is linear with the width (or height) of the picture. In other words, it has a much better performance than simpler renderers. In particular the absolute performance, 30k in 8.5 seconds, means that it can realistically be used at ultra-high definition.

Our second series of tests uses Figure 1 as the basis. This has the same pixel background as Figure 15(c) but a purely synthetic foreground defined by closed NURBS. The foreground element will be rendered at the full output resolution while the background, being pixels, will remain at fixed resolution. It is thus a useful case to determine how the performance degrades when using pixel maps.

We first rendered all the components of the scene separately at a fixed resolution of 1000 by 1000 pixels, as standard commercial software would do. These images were then used as source data for a simple compositing process, using our renderer. The compositing was performed over a range of output resolutions and the dotted line of Figure 14 shows the outcome. As expected, the complete reliance on pixel source data gives a square-law curve for time against resolution. Compute times are now very much longer, up to 2600 seconds at 30k.

However, if we allow the renderer to work with the orig-

inal data for this scene, with the foreground geometrically-defined and pixels only for the background, the solid line curve results. Not only does this halve the time taken but it also produces much higher image quality due to the sharper geometrically-defined edges. Even at the extreme resolution compute times are plausible, especially for feature film work.

7. Future work

As Figure 15(c) shows, we need to investigate ways of better re-balancing lighting at the composition stage. Indeed a common problem is removing the lighting from an element, in order to re-light it in various ways. There is related work at this site on converting pixel images into pure geometry. The attraction of this is partly that re-rendering at higher resolutions becomes possible: there will be no pixelisation. With our software, another attraction is that the rendering times become linear. Work is also in hand to capture image components and to track them across frames of a movie sequence

8. Conclusion

In this paper, we have described a system for producing pictures from layered 2D data. The fact that our system does not rely on frame-buffer memory and is as pixel-independent as possible allows us to render very-high resolution images in reasonable time. Also, as effects like complex lighting and transparency are possible, this tool has applications in cartoon animation as well as in film special effects or digital photography.

Acknowledgements

This work was supported by the European Union *Training and Mobility of Researchers* project "Platform for Animation and Virtual Reality". The authors would also like to thank Createc for providing us with digitally-encoded live action film, as well as Frédéric Labrosse for the photographs that we used to generate the example images. We are also grateful to the anonymous reviewers for their useful comments.

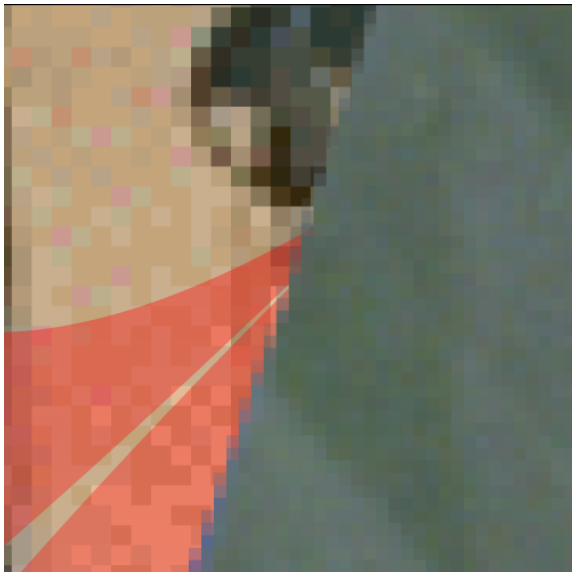
References

1. Deborah F. Berman, Jason T. Bartel, and David H. Salesin. Multiresolution painting and compositing. In *ACM Siggraph '94 Conference Proceedings*, pages 85–90. ACM Press, 1994.
2. Jim Blinn. Compositing, part 1: Theory. *IEEE Computer Graphics & Applications*, pages 83–87, September 1994.
3. Jean-Daniel Fekete, Érick Bizouarn, Éric Cournarie, Thierry Galas, and Frédéric Taillefer. TicTacToon: A paperless system for professional 2-D animation. In *SIGGRAPH 95 Conference Proceedings*, pages 79–90. ACM Press, August 1995.
4. Paul S. Heckbert and Henry P. Moreton. Interpolation for polygon texture mapping and shading. In David F. Rogers and Rae A. Earnshaw, editors, *State of The Art in Computer Graphics: Visualization and Modeling*, pages 101–111, New York, 1991. Springer-Verlag.
5. Jed Lengyel and John Snyder. Rendering with coherent layers. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 233–242. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.
6. Robert J. Oddy and Philip J. Willis. A physically based colour model. *Computer Graphics Forum*, 10(2):121–127, June 1991. 9th annual Eurographics conference.
7. J. W. Patterson and P. J. Willis. Computer assisted animation: 2D or not 2D? *The Computer Journal*, 37(10):829–839, 1994.
8. Michael A. Shantzis. A model for efficient and flexible image computing. In *SIGGRAPH 94 Conference Proceedings*, pages 147–154. ACM Press, 1994.
9. Jay Torborg and James T. Kajiya. Talisman: Commodity realtime 3D graphics for the PC. In *SIGGRAPH 96 Conference Proceedings*, pages 353–363. ACM Press, 1996.
10. Ken Turkowski. *Filters for Common Resampling Tasks*, volume 1 of *Graphics Gems*, chapter 3, pages 147–165. Academic Press, 1990.
11. B. A. Wallace. Merging and transformation of raster images for cartoon animation. *SIGGRAPH 81 Conference Proceedings*, 15(3):253–262, August 1981.
12. Geoff Waters and Philip Willis. UltraPaint: a new approach to a painting system. In *Proceedings of the 1987 Eurographics Conference*, pages 125–132. North-Holland, 1987.
13. P. Willis and T. Nettleship. The animachine renderer. In *Computer Animation '95 Conference Proceedings*. IEEE Computer Society Press, April 1995.
14. P J Willis and R J Oddy. Rendering nurbs regions for 2d animation. *Computer Graphics Forum*, 11(2):35–44 and 465, 1992.
15. Philip Willis and Geoff Watters. Scan converting extruded lines at ultra high definition. In *Proceedings of the 1987 Eurographics Conference*, pages 133–140. North-Holland, 1987.

(a) 1000 random NURBS shapes

(b) Transformation and compositing of images

(d) Compositing NURBS and images



(e) Close-up

Figure 15: *Four examples*