Motion Extrapolation at the Pixel Level

John P. Costella
School of Physics, The University of Melbourne, Parkville Vic. 3052, Australia

(14 January 1993)

Abstract

In this paper, considerations for the implementation of motion extrapolation on a pixel-by-pixel basis are discussed.

1. Introduction

The emerging field of Virtual Reality (VR) has focussed attention on the increasingly onerous performance requirements that are being placed on graphical display architectures by modern applications. In many implementations of VR, the sensing of the head position and orientation of the participant allows the system to change the view presented on head-mounted displays, in real time, to more or less convince the participant that they are exploring a virtual world with similar space-time properties to the real world. But this act places the participant into the closed-loop system of the application in a vastly higher bandwidth way than most conventional computer-human interfaces. The ubiquitous mouse of today, for example, relies on the power of eye-hand coordination for its great utility, which in turn requires that the mouse's effect on the display follows the movements of the physical mouse smoothly, accurately and without delay. This is achievable in today's applications because the mouse is usually considered as a *pointer*; thus, only small pixmaps representing its position and operation need be animated around the display. In contrast, VR requires that the *entire display* be updated with this same high degree of fidelity, to reflect changes in the three translational and three rotational degrees of freedom of the viewer's head, in addition to any inherent changes taking place in the scene being depicted. It is this promise of high-quality, high-bandwidth interaction that is both motivating advancements in this field, and causing persistent headaches for computer graphics engineers.

For most practical purposes, temporal frequencies greater than about 100 Hz may be considered to be insensible to the human visual system. Everyday acts of visual perception, however, frequently stretch towards this 100 Hz mark. Thus, a naïve approach would be to simply design a graphical display system capable of rendering a complete scene every ten milliseconds; this would give a temporal visual equivalent of "hi-fi". But this naïve approach is unnecessarily wasteful of computational resources: the information rate of the human visual system is nowhere near 100 times the information discernable in a single static image, by some orders of magnitude. The conventional response to this fact is to "de-tune" a system design from 100 Hz to some lower update rate, such as 10 or 5 Hz. At such rates, visual information is still being presented at a rate sufficient to convey some sense of "presence" in the virtual world; for example, navigational capabilities are not usually thrown into instability at such rates. But the convinceability of a virtual world is, in general, markedly degraded by such design decisions: the world appears to "clunk along" to the beat of a hidden drum; motion is no longer smooth.

It is clear that, at an abstract level, all that needs to be done to get the best of both of these worlds is to somehow "impedance-match" graphical display architectures to the capabilities of the human visual system. Carrying out this task in practice, however, is not exactly trivial. It is obvious that the higher-frequency sensitivities of the visual system are linked to *motion recognition*, rather than any recognition of fine detail. Thus, some sort of motion compression is clearly in order. (This must be motion *extrapolation*, of course, since the goal is to reduce the latency of the closed-loop system to as close to zero as is practicable.) In this task, VR and related applications have a vast advantage over conventional applications using techniques such as MPEG to effect such compression. Firstly, the locations, velocities, accelerations, *etc.*, of objects in the virtual world are, by definition, already *known* to the system generating that world; thus, motion *estimation* via graphical algorithmical means is not needed at all. Secondly, there are essentially *no low-bandwidth communications bottlenecks* between each VR engine and its connected participant: the purpose of motion compression is not to "squeeze" information over a low-bandwidth link, as it is with MPEG-like compression applications, but rather to relieve the computational and graphical load on the generating system itself. In fact, one may achieve great benefit from highly *parallelising* the motion-encoded data, rather than serialising it.

The ultimate in graphical parallelism is to be able to implement it right down to the pixel level. Thus, one might choose to avoid the small-talk completely, and immediately ask whether it is feasible to perform motion extrapolation on a *pixel-by-pixel* basis with current hardware. On the surface of it, such an idea seems to imply an unwieldy and completely impractical demand for storage and processing capabilities—indeed, almost rivalling the complexity of the original graphical system that it is intending to supplement. However, we shall now discuss the precise practical requirements of such a system explicitly (but nevertheless briefly), and shall show that the required overhead is vastly less than what one might otherwise expect.

2. Discussion

It is worthwhile to have a name to describe a pixel that in some sense has inertial properties—that, in between updates, does not simply sit still on the display (as with conventional sample-and-hold rasterised displays), but rather continues to move of its own accord. A worthy name, honouring the discoverer of inertia, is a *Galilean pixel*, or *galpixel* for short. A pixmap made up of independently-propagating galpixels is therefore most simply referred to as a *galpixmap*.

The first choice that one must make in implementing motion extrapolation is just *how many derivatives* of the galpixel's apparent motion should be stored. (By "apparent" we mean the motion as projected onto the plane of the display, with an appropriate depth measure.) Often only the *velocity* is considered as a candidate for such tasks. But in general we can extrapolate any object's apparent motion $\boldsymbol{r}(t) \equiv (x(t), y(t), z(t))$ about some given time $t = t_0$ to arbitrary accuracy (given enough initial information) via its Taylor series:

$$\boldsymbol{r}(t) = \boldsymbol{r}(t_0) + \boldsymbol{v}(t_0)\,(t - t_0) + \frac{1}{2}\boldsymbol{a}(t_0)\,(t - t_0)^2 + \frac{1}{6}\boldsymbol{j}(t_0)\,(t - t_0)^3 + \ldots, \tag{1}$$

where $\boldsymbol{r}(t_0)$ is its position at time t_0, $\boldsymbol{v}(t_0)$ its velocity, $\boldsymbol{a}(t_0)$ its acceleration, $\boldsymbol{j}(t_0)$ its jerk, and so on. Of course, it becomes more and more difficult to get a meaningful estimation of these derivatives using physical transducers as their order becomes higher—and even the jerk could, in practical situations, be riddled with delta function dependencies that would make its evaluation very error-prone. However, the apparent *acceleration* $\boldsymbol{a}(t_0)$ is a quantity that may in practice be computed or measured quite reasonably; moreover, *without* the acceleration it is impossible to depict with any degree of fidelity an object that is *rotating* (since a straight line is a very poor approximation to the peak of a sinusoid, whereas a parabola is a much better fit). In addition to this, one may note that the human visual system (in distinction to some other animals') is rather adept at transforming away the effects of velocity (you can watch a slowly moving car as easily as if it were still) *and* acceleration (you can catch a ball), but is rather poor when it comes to derivatives greater than the second. Thus, in practice, it is worthwhile to aim to include velocity and acceleration information with each galpixel, but not (in the usual case) any higher derivatives.

Let us now examine (1) in a little more detail. Retaining terms up to second order, we then have

$$\boldsymbol{r}(t) = \boldsymbol{r}(t_0) + \boldsymbol{v}(t_0)\,(t - t_0) + \frac{1}{2}\boldsymbol{a}(t_0)\,(t - t_0)^2\,, \tag{2}$$

where we shall pretend, for simplicity, that the higher derivatives of the motion do in fact vanish. Now, (2) seems to imply that we will need to perform a number of floating point multiplications for *every* pixel—not impossible, but also not exactly desirable. However, this is, of course, an illusion. The only times that are of relevance to us are the times that each frame is placed onto the display device. But the frame rate, f_{frame}, is in general a constant; therefore the time of frame n is (with a suitable choice of origin) given by $t_n = n\tau$, where $\tau \equiv 1/f_{\text{frame}}$ is the frame period. Thus, equation (2) becomes

$$\boldsymbol{r}_{n+m} = \boldsymbol{r}_n + \boldsymbol{v}_n m\tau + \frac{1}{2}\boldsymbol{a}_n m^2\tau^2.$$

But since we wish to propagate the galpixel forward anew for *each* frame, we need never worry about any m except $m = 1$, which propagates us from frame n to frame $n + 1$:

$$\boldsymbol{r}_{n+1} = \boldsymbol{r}_n + \boldsymbol{v}_n\tau + \frac{1}{2}\boldsymbol{a}_n\tau^2. \tag{3}$$

Together with (3), we need to propagate forward the velocity:

$$\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + \boldsymbol{a}_n\tau, \tag{4}$$

and the acceleration,

$$\boldsymbol{a}_{n+1} = \boldsymbol{a}_n. \tag{5}$$

(Note carefully that these are *convective* quantities; in other words, the velocity and acceleration are *carried along* with a particular galpixel, not "left behind" in the pixel location that the galpixel is leaving.) Equation (5) is of course trivial. However, equations (3) and (4) still imply that multiplications are required. Consider, then, choosing our (*a priori* arbitrary) units of time such that $\tau \equiv 1$, *i.e.*, so that time is measured as integral multiples of the frame period; equations (3) and (4) then become

$$\boldsymbol{r}_{n+1} = \boldsymbol{r}_n + \boldsymbol{v}_n + \frac{1}{2}\boldsymbol{a}_n \tag{6}$$

and

$$\boldsymbol{v}_{n+1} = \boldsymbol{v}_n + \boldsymbol{a}_n \tag{7}$$

respectively. But these equations involve no more complicated operations than addition and a division by two (shift right), and are thus trivially implemented in either hardware or machine code.

We must, however, still consider how the components of the velocity and acceleration values are to be *stored* for each galpixel. The naïve approach would be to store floating point values. But this is of course excessively wasteful. Consider a practical graphical system. First of all, for a given frame rate and display dimensions, there will be a maximum apparent velocity in the x or y direction beyond which the human visual system will be unable to detect the fleeting object at all. If we call the minimum recognisability time τ_{recog}, measured in units of frame periods, and if the typical dimension of the display is of order D pixels, then the fastest apparent velocity we need store is of the order of D/τ_{recog} in magnitude (in units of pixels per frame period). However, since we are also allowing *accelerations* to be carried by our galpixels, we must cater for the scenario in which a galpixel starts on one extremity of the display, moving rapidly towards the opposite extremity, but is "pulled back" by the acceleration just sufficiently that it stops at the other side, and accelerates back to the first extremity. If we take the minimum recognisability time to be the total time that such a galpixel is on the display, then it is trivial to verify that the maximum velocity magnitude we must consider is in fact of order $4D/\tau_{\text{recog}}$, and the maximum acceleration magnitude $8D/\tau_{\text{recog}}^2$. Thus, the *number of integral bits* we need to store each component of the velocity and acceleration of each galpixel is

$$N_{\text{int}}^{v_i} = \log_2 D - \log_2 \tau_{\text{recog}} + 3 \tag{8}$$

and

$$N_{\text{int}}^{a_i} = \log_2 D - 2\log_2 \tau_{\text{recog}} + 4 \tag{9}$$

respectively (where $i = x$ or y), where we have added one bit for the sign to each expression. The requirements of the velocities and accelerations of the z-buffer information, on the other hand, are not so clear-cut, but nor are they quite so important visually. If the z-buffer bins are of roughly the same spatial size as the pixel bins in the x and y directions, then a good first approximation is to use the same values (8) and (9) for the integral bits of the z component values also, *i.e.*, we may reasonably consider $i = x$, y or z.

Turning, now, to the *fractional* part of the velocity and acceleration components, it will be noted that the x and y components of a galpixel's position are *implicitly* recorded in its pixel position in the matrix of the galpixmap. But such a position in a matrix carries no "fractional" part at all; thus, it might appear that all velocities and accelerations (which ultimately manifest themselves by shifting galpixel positions) must be integral. However, this would lead to intolerable quantisation errors: the minimum velocity of any compont of a galpixel's motion (of which all velocities would be a multiple) would then be 1 pixel per frame, or $f_{\rm frame}$ pixels per second: an untenably large quantum in almost all practical situations. Thus, we must also store a *fractional position* for each of the x and y components of a galpixel's position, which, when added to the components of its matrix position, gives its "true" position. (In propagation, of course, each galpixel "snaps" into the best-fitting pixel location that it finds.) If we allocate $N^{r_i}_{\rm frac}$ bits for each component of such a fractional position, where $i = x$ or y, then the minimum velocity magnitude will then be

$$v_i^{\rm min} = 2^{-N^{r_i}_{\rm frac}}.$$

We can reinterpret this level of quantisation accuracy by stating that the motion of the galpixel will be accurate to the *single-pixel* level for propagation times up to

$$\tau_{\rm prop} = \frac{1}{v_i^{\rm min}} \equiv 2^{N^{r_i}_{\rm frac}} \tag{10}$$

(again in units of frame periods). We can now turn this argument around, and *specify* the propagation time that we wish to "rate" our system for: for example, $\tau_{\rm prop} = 16$ or 32 frames, which would allow (say) a 100 Hz frame rate system to require galpixmap updates at only 3 or 6 Hz, and still depict smooth motion at the 100 Hz level of fidelity. With such a specification of $\tau_{\rm prop}$, we then invert (10) to yield

$$N^{r_i}_{\rm frac} = \log_2 \tau_{\rm prop}, \tag{11}$$

where it is again stressed that we only need $i = x$ or y here. (Fractional z positions are in principle unnecessary; the required accuracy can be obtained by simply increasing the number of bits allocated to the z-buffer, since the z position is not matricised.)

We can now proceed to compute how many fractional bits are required for the velocity and acceleration data of each galpixel. It is important to start with the acceleration and work back down the differential hierarchy to the position, to avoid round-off errors in the conceptual argument. To maintain accuracy of the propagated galpixel position to within one pixel over $\tau_{\rm prop}$ frame periods, we must note (from (2)) that the effect of the initial acceleration on the position is *quadratic* with time, but contains a factor of one-half. Thus, the number of fractional bits required for the acceleration is just

$$N^{a_i}_{\rm frac} = \log_2\left(\frac{1}{2}\tau^2_{\rm prop}\right),$$

or, in other words,

$$N^{a_i}_{\rm frac} = 2\log_2 \tau_{\rm prop} - 1. \tag{12}$$

We now must worry about how this level of accuracy is to be propagated through to the position component using the frame-by-frame propagation equations (6) and (7). Clearly, from (7), the velocity $\boldsymbol{v}$ *must* contain at least $N^{a_i}_{\text{frac}}$ fractional bits per component also; otherwise, we would simply be throwing away bits of $\boldsymbol{a}$ every time we invoked (7). But our earlier requirement of a minimum velocity yielded a minimum number of fractional bits given by (11). However, this is always less than or equal to (12) since τ_{prop} must be 2 or more for the technique to be worthwhile at all. Thus, each of the x and y velocity components must also have the number of fractional bits specified by (12); in other words,

$$N^{v_i}_{\text{frac}} \equiv N^{a_i}_{\text{frac}} = 2\log_2\tau_{\text{prop}} - 1. \tag{13}$$

We are now in a place to consider how expensive it would be in terms of storage to convert a standard pixmap into a galpixmap. (Implementing the propagation hardware is a different question, of course, to which we shall return.) Assume we start with a pixmap in which some sort of colour information (*e.g.*, 24-bit RGB) as well as sufficient z-buffer information is already implemented. To append motional information to each pixel, we must add the various quantities considered above. Fractional position information is in general only required for the x and y directions, but if we are *adding* to an existing implementation then it is desirable to add the same number of "fractional" bits onto the z-buffer data also. Likewise, for simplicity, let us consider D to be of the same order of magnitude for all three directions; the velocity and acceleration components then require the same number of bits for each. We can then simply add the bits up for *one* component,

$$N^{i}_{\text{total}} = N^{v_i}_{\text{int}} + N^{a_i}_{\text{int}} + N^{r_i}_{\text{frac}} + N^{v_i}_{\text{frac}} + N^{a_i}_{\text{frac}},$$

using the explicit formulæ (8), (9), (11), (12) and (13):

$$\begin{aligned} N^{i}_{\text{total}} = {} & (\log_2 D - \log_2\tau_{\text{recog}} + 3) + (\log_2 D - 2\log_2\tau_{\text{recog}} + 4) \\ & + (\log_2\tau_{\text{prop}}) + (2\log_2\tau_{\text{prop}} - 1) + (2\log_2\tau_{\text{prop}} - 1)\,; \end{aligned}$$

or, on simplifying,

$$N^{i}_{\text{total}} = 2\log_2 D - 3\log_2\tau_{\text{recog}} + 5\log_2\tau_{\text{prop}} + 5. \tag{14}$$

Let us now plug some real-world numbers into (14) to evaluate its feasibility. Take, for example, $D = 1024$, $\tau_{\text{recog}} = 8$ and $\tau_{\text{prop}} = 16$. Equation (14) then tells us that we need to add 36 bits of storage per Cartesian component for each pixel to convert it into a fully-fledged galpixel, or in other words 13.5 additional bytes all up. (We of course need at least two galpixmaps in order to compute the propagation algorithm.) This is not at all excessive by 1993 standards; it will appear less excessive as technology advances. Even in cases where it does seem beyond reach, one can always "shave" bits off this estimate by settling for a somewhat smaller degree of position accuracy, a larger minimum recognition time, or a smaller maximum velocity or acceleration.

So far, we have only considered the *motional* components of a galpixel's required additional storage. Clearly, for additional fidelity, the derivatives of the *colour* data should be stored also, so that "shading inertia" may be automatically implemented by the hardware. The largest number of bits for this purpose should be dedicated to changes in the *luminosity* of each galpixel; a simple estimate of luminosity (not involving floating-point arithmetic) such as simply the sum or red, green and blue values, or perhaps using the fractions 1/2 and 1/4, would be in order, so that a simple hardware demultiplexer into RGB values would suffice. However, we shall not discuss this topic in detail in this paper, but merely continue to consider the issues involved in implementing pixel-level motion extrapolation (or *Galilean spatio-temporal antialiasing*) in a minimal system.

The major concern for the implemention of the propagation algorithm is that there is no *a priori* reason why a galpixmap full of arbitrarily-moving galpixels at frame n should happen to rearrange itself via a one-to-one mapping into a complete galpixmap at frame $n+1$. Clearly, in the general case, some galpixels will propagate out of the display area, some will want to occupy the same x–y pixmap position as other galpixels, and (as a result of these two phenomena) some pixmap positions will be left unoccupied. The propagation algorithm must cater for these events, but it must do so in a way that can be implemented in the simplest possible hardware (or machine code), with a running time that is independent of the image being propagated.

Firstly, consider galpixels that propagate right out of the display area. There could be two reasons for this: the image as a whole is being panned (because the participant is moving her head), or because the object that that galpixel belongs to happens to be departing the scene. We shall treat the latter case in more detail shortly. In either case, one may simply clip the galpixmap so that galpixels moving out of its bounds are discarded. But in the former case (where the image is being panned), the disappearance of galpixels off one boundary will be accompanied by a need for new galpixels on the opposite boundary. A simple way to alleviate this problem somewhat is to render a galpixmap area that is *larger* than the display area—with perhaps the precise areas of greatest "margins" decided on-the-fly on the basis of participant-motion data—so that galpixels expected to enter the display area before the next update are "in the wings" at update time, waiting for their grand entrance.

Next, consider two galpixels in the galpixmap at frame n which the propagation algorithm decides should occupy the *same* position in the galpixmap at frame $n+1$. For a minimal implementation, we might simplify matters by assuming that partially-transparent objects, shadows, *etc.*, are to be ignored for this purpose. (Extensions of the technique to include such qualities as these are simple, if somewhat more demanding in terms of storage requirements.) With such an assumption, the propagation algorithm need only use the stored z-buffer information to decide which galpixel should be displayed; the other is discarded.

The most troublesome problem is what to do when there are empty positions in the new galpixmap. Clearly, they must be filled with something. A minimal approach is to simply follow the conventional sample-and-hold frame buffer philosophy, and leave "debris" behind that is just an RGB copy of what occupied that pixel location in the previous frame. This trick (and indeed any conventional rasterised display device simply employing a frame buffer) relies on the visual system's integration properties for its effectiveness: if the debris is not lying around for too long, the "jerkiness" of the resulting motion is, to a greater or lesser extent, integrated out. In the current case, we are using this property in only those areas of the display for which we have no information, rather than for the entire display plane. This debris would of course need to be flagged in some way, so that if in a later frame a fully-fledged galpixel were to be validly propagated to that point, it would "win out" over any debris.

However, this approach is rather poor, and can be improved for little cost. One reason it is poor is that objects that are *approaching* the participant will (by the perspective transformation) appear to *expand*; with only a debris approach, such an object would (in the general case) be riddled with holes, through which the background could be seen, since the fixed number of galpixels constituting the original rendering cannot possible fill the now-larger area covered by the object's image. To repair this "bullet-hole" problem, however, we need some way to determine, *on the fly*, whether an empty galpixmap location is in the *interior* of a surface, or whether it is in an area that has been "uncovered" by the bulk motion of an object that previously occupied that position. To determine, at the galpixmap level, whether a hole is indeed interior to a surface, we can simply consider the z-buffer information (stored in the galpixmap) as a *function* of x and y, namely, $z = z(x, y)$. Now compute the two-dimensional Laplacian of this function:

$$\nabla^2 z(x,y) \equiv \left\{ \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right\} z(x,y). \tag{15}$$

For any function $z(x, y)$ of x and y that is continuous and has a continuous first derivative at some point (x_0, y_0) (such as that describing a smooth surface with a continuous first derivative), the Laplacian vanishes. (This is most intuitively verified by considering that, as one shrinks one's attention to a smaller and smaller surface area around (x_0, y_0), the surface approximates a *plane* better and better; and a plane's x–y Laplacian vanishes trivially.) On the other hand, if, on the x–y image plane, the images of two objects at *different distances* abut each other, then the step-function in the z-distance data (which is not smooth) will lead to a *source term* in (15), *i.e.*, we will find

$$\nabla^2 z(x,y)\Big|_{(x_0,y_0)} = \rho(x_0,y_0) \neq 0. \tag{16}$$

Thus, we can find the *edges* of all surfaces in the original galpixmap by computing equation (16)—Poisson's equation—at each pixel location, and then testing whether the source term is zero or non-zero. The Laplacian (15) is of course trivially estimated for a discrete rectangular grid, being simply given by

$$\begin{aligned} \nabla^2_{\text{disc.}} z(x,y)\Big|_{(x_0,y_0)} \equiv {} & z(x_0+1,y_0) + z(x_0-1,y_0) \\ & + z(x_0,y_0+1) + z(x_0,y_0-1) - 4z(x_0,y_0), \end{aligned} \tag{17}$$

which, involving only addition, subtraction and multiplication by 4 (shift left two bits) is trivially implemented in hardware or machine code. With the assumption that the surfaces are approximately flat over distances of the order of a single pixel (without which the "interior" would not be discernable anyway), we can then use the discrete Laplacian (17) to estimate the value of Poisson's equation (16), which can be compared to zero, within some suitably-chosen bound to account for the approximate nature of the discrete derivative.

With such an analysis of the original galpixmap (with the results stored in a 1-bit-deep map of flags denoting either "in surface" or "on edge of surface"), we can proceed to fill the gaps in the new galpixmap after we have propagated the old galpixmap into it. Scanning through the new galpixmap, we look for any empty locations; when we find one, we propagate *backwards* the positions of any of its neighbours that happen to be filled, back to their original galpixmap locations. If any of them are inside a surface, according to the Poisson map, then the chances are that the given empty galpixel location is inside the same surface. Thus, the galpixel properties (colour, velocities, *etc.*) of these surrounding filled galpixels may be averaged (or copied, if there is only one) to provide a reasonable interpolation to "fill in the bullet-holes".

The above procedure, however, seems to treat the interiors of just-unobscured areas (due to the bulk motion of an object to reveal what was behind it) in the same way as the interiors of the surfaces of objects themselves. This does not, on the surface of it, seem to be correct: we need to come up with a different technique to deal with areas for which we have no information. However, the answer to this comes from the *boundary conditions* of just-unobscured areas. Consider the top-left corner of a just-unobscured area (which for simplicity we take to be roughly square, although the argument holds true for arbitrary shapes). The galpixel to the left of this top-left corner is (by necessity) at the edge of a surface (the one to the left of the unobscured area), as will be discovered when it is propagated back to the original Poisson map. So too is the galpixel above. The galpixels to the right and below, on the other hand, are *themselves* empty. Thus, the algorithm at this point decides that the empty galpixel location *is* in fact just-unobscured, and invokes its procedure to deal with this case (to be discussed shortly). Such a galpixel is then *flagged* as having been just-unobscured. As the algorithm proceeds to the pixel to the right (and, later, the pixel below), it continues to find that there are no "interior" (and not just-unobscured) galpixels surrounding empty galpixel locations, and so continues to invoke the just-unobscured algorithm. In this way, the interiors of just-unobscured areas are built up by virtue of their boundaries' lack of "interiorness".

Finally, we are left with the problem of filling regions that are just-unobscured in some intelligent and non-visually-offensive way. It is towards this end that there is most scope for an individual system designer to implement as simple or as sophisticated an approach as their hardware resources will allow them. The simplest approach is a more sophisticated version of the above simple "debris" aproach. Rather than just leaving behind static debris, which would simply remain there until the next update, or until another galpixel arrives to fill the void, one can copy across the galpixel occupying that pixel location in the previous galpixmap *but including motional information*. It is then flagged as debris: if it (in a future propagation) clashes with a non-debris galpixel, then the latter will always "win". This form of "galdebris" is implementable by copying the original galpixmap across to the new galpixmap *without* propagation, flagging all galpixels as "galdebris", and then propagating the galpixmap proper, overwriting galdebris where it occurs. The "copied" galpixels that survive are now delayed behind their correct positions by one frame period; if a piece of galdebris is subsequently required to *again* be delayed, then it will be two frames behind, and so on. The net effect is a "smudging" of the edge of the object that is moving out of the way, if there is nothing else that moves in to fill the breach—not unlike the back edge of the USS Enterprise when it hits warp speed. This approach has the advantage over the static-debris approach that its *relative* effects are the same regardless of the object's velocity or acceleration, a relativity property that the static debris approach does not share.

Clearly, an even better solution to the unobscuration problem would be to somehow "summon" an image of the object that was previously obscured, rather than simply filling the area with the least-visually-offensive junk that can be created from the remaining galpixmap.

However, the whole aim of the motion extrapolation technique is to allow the graphical system to *not* have to generate updates within a single frame period; thus, the display hardware cannot demand such an update to be created in time to be worthwhile. One approach is to *cache* galpixmaps, so that an object that is obscured and then obscured could be summoned back from the cache. However, to gather the greatest benefit from this approach, one must remove the "global-update" philosophy of most current VR rendering systems (namely, that the whole scene must be churned out at once), and implement display buffers that allow *individual* objects' images to be *grafted onto* the existing (propagating) galpixmap. This may necessitate the use of object tags in the galpixmap structure, so that the object's previous rendering (which might be becoming reasonably inaccurate by that stage) can be deleted by the propagation hardware as it grafts on the new rendering. Such an approach also implies a much higher level of object-orientation in the graphical architecture: each object in the virtual world must be responsible for maintaining its *own* graphical representation on the display, and "knowing" when its Galileanly-propagating image has become out-of-date, rather than just having a single rendering process traverse the entire virtual-world database at regular intervals, spitting out global updates. This implies a much more sophisticated approach to real-time graphics, integrated with the operating system and display hardware in a highly intimate way, but it promises to yield the highest amount of leverage, for a given level of inherent graphical power, from the technique of motion extrapolation.

3. Acknowledgments

Helpful discussions with readers of the Usenet newsgroup Sci.virtual-worlds, following the posting of a long paper on this topic to that group, are gratefully acknowledged.

This work was supported in part by an Australian Postgraduate Research Allowance.