Galilean Antialiasing for Virtual Reality Displays

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

(25 October 1992)

Abstract

In this paper, a method is described that improves the perceived "smoothness" of motion depicted on rasterised Virtual Reality displays, by utilising the powerful information already contained in the virtual-world engine. Practical implementation of this method requires a slight realignment of one's view of the nature of a rasterised display, together with modest modifications to current image generation and rasterisation hardware. However, the resulting improvement in the quality of the perceived real-time image is obtained for only modest computational and hardware cost—offering the possibility of increasing the *apparent* graphical capabilities of existing technology by up to an order of magnitude.

1. Introduction

Numerous definitions of the term *Virtual Reality* abound these days. However, they all share one common thread: a successful Virtual Reality system *convinces* you that you are somewhere other than where you really are. Despite engineers' wishes to the contrary, just *how* convincing this experience is seems to depend only weakly on raw technical statistics like megabits per second; rather, more important is how well this information is "matched" to the expectations of our physical senses. While our uses for Virtual Reality might encompass virtual worlds bearing little resemblance to the real world, our physical senses nevertheless still expect to be stimulated in the same "natural" ways that they have for millions of years.

Two particular areas of concern for designers of current Virtual Reality systems are the *latency* and *update rate* of the visual display hardware employed. Experience has shown that poor performance in either of these two areas can quickly destroy the "realness" of a Virtual Reality session, even if all of the remaining technical specifications of the hardware are impeccable. This is perhaps not completely surprising, given that the images of objects that humans interact with in the natural world are never delayed by more than fractions of milliseconds, nor are they ever "sliced" into discrete time frames (excluding the intervention of the technology of the past hundred years). On the other hand, the relative psychological importance of latency and update rate can, conversely, be used to great advantage: with suitably good performance on these two fronts, the "convinceability factor" of a Virtual Reality system can withstand relatively harsh degradation of its other capabilities (such as image resolution). "If it *moves* like a man-eating beast, it probably *is* a man-eating beast—even if I didn't see it clearly" is no doubt a hard-wired feature of our internal image processing subsystem that is responsible for us still being on the planet today.

A major problem facing the Virtual Reality system designer, however, is that presenting a sufficiently "smooth" display to fully convince the viewer of "natural motion" often seems to require an unjustifiably high computational cost. As an order-of-magnitude estimate, a rasterised display update rate of around 100 updates per second is generally sufficiently fast enough that the human visual system cannot distinguish it from continuous vision. But the actual amount of information gleaned from these images by the viewer in one second is nowhere near the amount of information containable in 100 static images—as can be simply verified by watching a "video montage" of still photographs presented rapidly in succession. The true rate of "absorption" of detailed visual information is probably closer to 10 updates per second—or worse, depending on how much actual detail is taken as a benchmark. Thus, providing a completely "smooth" display takes roughly an order of magnitude more effort than is ultimately appreciated—not unlike preparing a magnificent dinner for twelve and then having no one else turn up.

For this reason, many Virtual Reality designers make an educated compromise between motion smoothness and image sophistication, by choosing an update rate that is somewhere between the ~ 10 updates per second rate that we absorb information at, and the ~ 100 updates per second rate needed for smooth apparent motion. Choosing a rate closer to 10 updates per second requires that the participant mentally "interpolate" between the images presented—not difficult, but nevertheless requiring some conscious processing, which seems to leave fewer "brain cycles" for appreciating the virtual-world experience. On the other hand, choosing a rate closer to 100 updates per second results in natural-looking motion, but the reduced time available for each update reduces the sophistication of the graphics—fewer "polygons per update". The best compromise between these two extremes depends on the application in question, the expectations of the participants, and, probably most importantly, the opinions of the designer.

In the remaining sections of this paper, we outline enhancements to current rasterised display technology that permit the motion of objects to be consistently displayed at a high update rate, while allowing the image generation subsystem to run at a lower update rate, and hence provide more sophisticated images. Section 2 presents an overview of the issues that are to be addressed, and outlines the general reasoning behind the approach that is taken. Following this, in section 3, detailed (but platform-independent) information is provided that would allow a Virtual Reality designer to "retrofit" the techniques outlined in section 2 to an existing system. For these purposes, as much of the current image generation design philosophy as possible is retained, and only those minimal changes required to implement the techniques immediately are described. However, it will be shown that the full benefit of the methods described in this paper, in terms of the specific needs of Virtual Reality, will most fruitfully be obtained by subtly changing the way in which the image generation process is currently structured. These changes, and more advanced topics not addressed in section 3, are considered in section 4.

2. The Basic Philosophy

We begin, in section 2.1, by reviewing the general methods by which current rasterised displays are implemented, to appreciate more fully why the problems outlined in section 1 are present in the first place, and to standardise the terminology that will be used in later sections. Following this, in section 2.2, we review some of the fundamental physical principles underlying our understanding of motion, to yield further insight into the problems of depicting it accurately on display devices. These deliberations are used in section 2.3 to pinpoint the shortcomings of current rasterised display technology, and to formulate a general plan of attack to rectify these problems. A brief introduction to the terminology used for the general structures required to carry out these techniques is given in section 2.4—followed, in section 2.5, by a careful consideration of the level of sophistication required for practical yet reliable systems. Specific details about the hardware and software modifications required to implement the methods outlined are deferred to sections 3 and 4.

2.1. Overview of Current Rasterised Displays

Rasterised display devices for computer applications, while ubiquitous in recent years, are relatively new devices. Replacing the former *vector display* technology, they took advantage of the increasingly powerful memory devices that became available in the early 1970s, to represent the display as a digital matrix of pixels, a *raster* or *frame buffer*, which was scanned out to the CRT line-by-line in the same fashion as the by then well-proven technology of *television*.

The electronic circuitry responsible for scanning the image out from the frame buffer to the CRT (or, these days, to whatever display device is being used) is referred to as the *video controller*—which may be as simple as a few interconnected electronic devices, or as complex as a sophisticated microprocessor. The *refresh rate* may be defined as the reciprocal of the time required for the video controller to refresh the display from the frame buffer, and is typically in the range 25–120 Hz, to both avoid visible flicker, and to allow smooth motion to be depicted. (Interlacing is required for CRTs at the lower end of this range to avoid visible flicker; for simplicity, we assume that the display is *non-interlaced* with a suitably high refresh rate.) Each complete image, as copied by the video controller from the frame buffer to the physical display device, is referred to as a *frame*; this term can also be used to refer to the time interval between frame refreshes, *i.e.*, the reciprocal of the refresh rate.

Most Virtual Reality systems employ two display devices to present a stereoscopic view to the participant, one display for each eye. Physically, this may be implemented as two completely separate display subsystems, with their own physical output devices (such as a twin-LCD head-mounted display). Alternatively, hardware costs may be reduced by interleaving the two video signals into a single physical output device, and relying on physically simpler (and sometimes less "face-sucking") demultiplexer techniques to separate the signals, such as time-domain switching (*e.g.*, electronic shutter glasses), colour-encoding (*e.g.*, the "3-D" coloured glasses of the 1950s), optical polarisation (for which we are fortunate that the photon is a vector boson, and that we have only two eyes), or indeed any other physical attribute capable of distinguishing two multiplexed optical signals. For the purposes of this paper, however, we define a *logical display device* to be *one* of the video channels in a twin-display device (say, the left one), or else the corresponding *effective* monoscopic display device for a multiplexed system. For example, a system employing a single (physical) 120-Hz refresh-rate CRT, time-domain-multiplexing the left and right video channels into alternate frames, is considered to possess two *logical* display devices, each running at a 60 Hz refresh rate. In general, we ignore completely the engineering problems (in particular, *cross-talk*) that multiplexed systems must contend with. Indeed, for most of this paper, we shall ignore the stereoscopic nature of Virtual Reality displays altogether, and treat each video channel separately; therefore, all references to the term "display device" in the following sections refer to *one* of the two logical display devices, with the understanding that the other video channel simply requires duplicating the hardware and software of the first.

We have described above how the video controller scans frames out from the frame buffer to the physical display device. The frame buffer, in turn, receives its information from the *display processor* (or, in simple systems, the CPU itself—which we shall also refer to as "the display processor" when acting in this rôle). Data for each pixel in the frame buffer is retained unchanged from one frame to the next, unless it is overwritten by the display processor in the intervening time. There are many applications for computer graphics for which this "sample-and-hold" nature of the frame buffer is very useful: a background scene can be painted into the frame buffer once, and only those objects that change their position or shape from frame to frame need be redrawn (together with repairs to the background area thus uncovered). This technique is often well-suited to traditional computer hardware environments—namely, those in which the display device is physically fixed in position on a desk or display stand—because a constant background view accords well with the view that we are using the display as a (static) "window" on a virtual world. However, this technique is, in general, ill-suited to Virtual Reality environments, in which the display is either affixed to, or at least in some way "tracks", the viewer, and thus must change even the "background" information constantly as the viewer moves her head.

There are several problems raised by this requirement of constant updating of the entire frame buffer. Firstly, if the display processor proceeds to write into the frame buffer at the same time as the video controller is refreshing the display, the result will often be an excessive amount of visible flicker, as partially-drawn (and, indeed, partially-erased) objects are "caught with their pants down" during the refresh. Secondly, once the proportion of the image requiring regular updating rises to any significant fraction, it becomes more computationally cost-effective to simply erase and redraw the entire display than to erase the individual objects that need redrawing. Unless the new scene can be redrawn in significantly less time than a single frame (untrue in any but the most trivial situations), the viewer will see not a succession of complete views of a scene, but rather a succession of scene-building drawing operations. This is often acceptable for "non-immersive" applications such as CAD (in which this "building process" can indeed often be most informative); it is not acceptable, however, for any convincing "immersive" application such as Virtual Reality.

The standard solution to this problem is *double buffering*. The video controller reads its display information from one frame buffer, whilst at the same time a second frame buffer is written to by the display processor. When the display processor has finished rendering a complete image, the video controller is instructed to switch to the second frame buffer (containing the new image), simultaneously switching the display processor's focus to the first frame buffer (containing the now obsolete image that the video controller was formerly displaying). With this technique, the viewer sees one constant image for a number of frames, until the new image has been completed. At that time, the view is instantaneously switched to the new image, which remains in view until yet another complete image is available. Each new, completed image is referred to as an *update*, and the rate at which these updates are forthcoming from the display processor is the *update rate*.

It is important to note the difference between *refresh* rate and *update* rate, and the often-subtle physical interplay between the two. The refresh rate is the rate at which the video controller reads images from the frame buffer to the display device, and is typically a constant for a given hardware configuration (*e.g.*, 70 Hz). The update rate, on the other hand, is the rate at which complete new images of the scene in question are rendered; it is generally lower than the refresh rate, and usually depends to a greater or lesser extent on the complexity of the image being generated by the display processor.

It is often preferable to change the video processor's focus only *between* frame refreshes to the physical display device—and not mid-frame—especially if the display processor is comparable in speed to one update per frame. This is because switching the video controller's focus mid-frame "chops" those objects in the particular scan line that is being scanned out by the video controller at the time, which leads to visible discontinuities in the perceived image. On the other hand, this restriction means that the display processor must wait until the end of a frame to begin drawing a new image (unless a third frame buffer is employed), effectively forcing the refresh-to-update rate ratio up to the next highest integral value. This is most detrimental when the update rate is already high; for example, if an update takes only 1.2 refresh periods to be drawn, "synchronisation" with the frame buffer means that the remaining 0.8 of a refresh period is unusable for drawing operations.

Regardless of whether the frame-switching circuitry is synchronised to the frame rate or not, if the update rate of the display processor is in fact slower than the refresh rate (the usual case), then the same static image persists on the display device for a number of frames, until a new image is ready for display. For *static* objects on the display, this "sample-and-hold" technique is ideal: the image's motion (*i.e.*, no motion at all!) is correctly depicted at the (high) refresh rate, even though the image itself is only being generated at the (lower) update rate. This phenomenon, while appearing quite trivial in today's rasterised-display world, is in fact a major advance over the earlier vector-display technology: the video processor, utilising the frame buffer, effectively *fills in the information gaps* between the images supplied by the display processor. Recognition of the remarkable power afforded by this feat of "interpolation"—and, more importantly, a critical assessement of how this "interpolation" is currently carried out—is critical to appreciating the modifications that will be suggested shortly.

As mentioned in section 1, the *latency* (or "time lag") of a Virtual Reality system in general, and the display system in particular, is crucial for the experience to be convincing (and, indeed, non-nauseous). There are many potential and actual sources of latency in such systems; in this paper, we are concerned only with those introduced by the image generation and display procedures themselves. Already, the above description of a double-buffered display system contains a number of potential sources of lag. Firstly, if the display processor computes the apparent positions of the objects in the image based on positional information valid at the *start* of its computations, these apparent positions will already be slightly out of date by the time the computations are complete. Secondly, the rendering and scan-conversion of the objects takes more time, and is based on the (already slightly outdated) positional information. Finally—and perhaps most subtly—the very "sample-and-hold" nature of the video processor's frame buffer leads to a significant average time lag itself, equal to *half the update period*. While a general mathematical proof of this figure is not difficult, a "hand-waving" argument is easily constructed. For simplicity, assume that all other lags in the graphical pipeline are magically removed, so that, upon the first refresh of a new update, it describes the virtual environment at that point in time accurately. By the time of the second refresh of the same image, it is now one frame out-of-date; by the third refresh, it is two frames out-of-date; and likewise for all remaining refreshes of the same image until a new update is provided. By the "hand-waving" argument of simply averaging the out-of-datedness of each refresh across the entire update period, one obtains

$$\langle \tau_{\text{lag}} \rangle \sim \frac{1}{\tau_{\text{update}}} \int_0^{\tau_{\text{update}}} t \, dt = \frac{1}{2} \tau_{\text{update}},$$

where τ_{update} is the update period. Thus, a long update period not only affects the *smoothness* of the perceived display, but also its *latency*—thus rendering it a particularly insidious enemy of real-time Virtual Reality systems, and a doubly worthy target of our attention.

It is this undesirable feature of conventional display methodology that we will aim to remove in this paper. However, to provide suitable background for the approach we shall take, and to put our later specifications into context, we first review some quite general considerations on the nature of physical motion.

2.2. The Physics of Motion

As noted in section 1, while our applications for Virtual Reality technology may encompass virtual worlds far removed from the laws of physics, our physical senses nevertheless expect to be stimulated more or less in the same way that they are in the real world. It is therefore useful to review briefly the evolution of man's knowledge about the fundamental nature of motion, and note how well these views have or have not been incorporated into real-time computer graphics.

Some of the earliest questions about the nature of motion that have survived to this day are due to Zeno of Elea. His most *famous* paradox—that of Achilles and Tortoise—is amusing to this day, but is nevertheless more a question of mathematics than physics. More interesting is his paradox of the Moving Arrow: At any instant in time, an arrow occupies a certain position. At the next instant of time, the arrow has moved forward somewhat. His question, somewhat paraphrased, was: How does the arrow know how to get to this new position by the very next instant? It cannot be "moving" at the first instant in time, because an instant has no duration—and motion cannot be measured except over some duration.

Let us leave aside, for the moment, the flaws that can be so quickly pointed out in this argument by anyone versed in modern physics. Consider, instead, what Zeno would say to us if we travelled back in time in our Acme Time Travel Machine, and showed him a television receiver displaying a broadcast of an archery tournament. (Ignore the fact that, had television programmes been in existence two and a half thousand years ago, Science as we know it would probably not exist.) Zeno would no doubt be fascinated to find that the arrows that moved so realistically across the screen were, in fact, a *series of static images* provided in rapid succession—in full agreement (or so he would think) with his ideas on the nature of motion. The question that would then spring immediately to his lips: *How does the television know how to move the objects on the screen?*

Our response would, no doubt, be that the television *doesn't* know how to move the objects; it simply waits for the next frame (from the broadcasting station) which shows the objects in their new positions. Zeno's follow-up: How does the *broadcasting station* know how to move them? Answer: It doesn't either; it just sends whatever images the video camera measures. And eventually we return to Zeno's original question: How does the real arrow itself "know" how to move? Ah, well, that's a question that even television cannot answer.

Ignoring for the moment the somewhat ridiculous nature of this hypothetical exchange, consider Virtual Zeno's first question from first principles. Why *can't* the television move the objects by itself? Surely, if the real arrow somehow knows how to move, then it is not unreasonable that the television might obtain this knowledge too. The only task then, is to determine this information, and tell it to the television! Of course, this is a little simplistic, but let us fast-forward our time machine a little and see what answers we obtain.

Our next visit would most likely be to Aristotle. Asking him about Zeno's arrow paradox would yield his well-known answer—that would, in fact, be regarded as the "right answer" for the next 2300 years: Zeno wrongly assumes that indivisible "instants of time" exist at all. Granting Aristotle this explanation of Zeno's mistake, what would his opinions be regarding "teaching" the television how to move the objects on its own? His response, no doubt, would be to explain that every object has its *natural place*, and that its *natural motion* is such that it moves towards its natural place, thereafter remaining at rest (unless subsequently subjected to *violent motions*). Heartened by this news, we ask him for a mathematical formula for this natural motion, so that we can teach it to our television. "Ah, well, I don't think too much of mathematical formulæ," he professes, engrossed in a re-run of *I Love Lucy*, "although I can tell you that heavier bodies fall faster than light ones." So much for an Aristotelian solution to our problem.

Undaunted, we tweak our time machine forward somewhat—2000 years, in fact. Here, we find the ageing Galileo Galilei ready and willing to answer our questions. On asking about Zeno's arrow paradox, we find a general agreement with Aristotle's explanation of Zeno's error. On the other hand, on enquiring how a television might be taught how to move objects on its own, we obtain these simple answers: If the body is in *uniform motion*, it moves according to $x = x_0 + vt$; if it is *uniformly accelerated*, it moves according to $x = x_0 + v_0t + at^2/2$. Furthermore, *gravity* acts as a uniform acceleration—changing the velocity of an object smoothly (and not discontinuously, as earlier propounded); and, what is more, this rate of acceleration is a constant for every object. To this, one must add accelerations other than gravity (such as the force of someone in throwing a ball) into the equation. If we teach these principles to our television, he explains, it *will* then know how to move objects by itself. And thus, from the first modern physicist, we get the information we desire—meanwhile leaving him fascinated by images of small white projectiles following parabolic paths, subtitled "British Open Highlights".

The tale woven in this section is, admittedly, a little fanciful, but nevertheless illustrates most clearly the thinking behind the methods to be expounded. Very intriguing, but omitted from this account, is the fact that Aristotle's solution to Zeno's arrow paradox, which remained essentially unchanged throughout the era of Galilean relativity and Newtonian mechanics, suffered a mortal blow three-quarters of a century ago. We now know that, ultimately, the "smooth" nature of space-time recognised by Galileo and Newton, and which underwent a relative benign "warping" in Einstein's classical General Relativity, must somehow be fundamentally composed of quantum mechanical "gravitons"; unfortunately, no one knows exactly how. Zeno's very question, "How does anything move at all?", is again *the* unsolved problem of physics. But that is a story for another place. Let us therefore return to the task at hand, and utilise the method we have gleaned from seventeenth century Florence.

2.3. Galilean Antialiasing

Consider the rasterised display methology reviewed in section 2.1. How does its design philosophy fit in with the above historical figures' views on motion? It is apparent that the "slicing up" in time of the images presented on the display device, considered simplistically, only fits in well with Zeno's ideas on motion. However, we have neglected the human side of the equation: clearly, if frames are presented at a rate exceeding the viewer's visual system's temporal resolution, then the effective integration performed by the viewer's brain combines with the "time-sampled" images to reproduce continuous motion—that is, at least for motion that is slow enough for us to follow visually.

Consider now the "interpolation" procedure used by the video processor and frame buffer. Is this an optimal way to proceed? Aristotle would probably have said "no"—the objects, in the intervening time between updates, should seek their "natural places". Galileo, on the other hand, would have quantified this criticism: the objects depicted should move with either constant velocity if free, or constant acceleration if they are falling; if subject to "violent motion", this would also have to programmed. Instead, the sample-and-hold philosophy of section 2.1 keeps each object at one certain place on the display for a given amount of time, and then makes it *spontaneously jump* by a certain distance; and so on. In a sense, the pixmap *has no inertial properties*. As noted, this *is* the ideal behaviour for an object that is not moving at all; its manifest incorrectness for a moving object is even more simply revealed by simple Galilean mechanics: Consider how an object travelling at constant apparent velocity $\boldsymbol{v}$, with respect to the display, is depicted with this system. Mathematically, the trajectory displayed by the video processor is

$$\boldsymbol{x}(t) = \boldsymbol{x}(0) + \boldsymbol{v}\,\mathrm{floor}(t), \tag{1}$$

where $\boldsymbol{x}(0)$ is the two-dimensional pixel-position vector at $t = 0$, t is measured in units of frame-periods, $\boldsymbol{v}$ is the (constant) velocity of the real object being simulated (in units of pixels per frame period), and floor(y) returns the greatest integer that is smaller than or equal to y. Now consider applying a Galilean transformation of velocity $\boldsymbol{v}$ to the *viewer* of the system, in the same direction that the object is moving (*e.g.*, by having the viewer standing on a "moving footpath" purloined from LA International Airport, which travels past the display device). The new trajectory seen by this moving viewer, $\boldsymbol{x}'(t)$, is obtained from the stationary-viewer trajectory $\boldsymbol{x}(t)$ by the Galilean transformation

$$\boldsymbol{x}'(t) = \boldsymbol{x}(t) - \boldsymbol{v}t. \tag{2}$$

The *correct* trajectory of the object, of course, should simply be

$$\boldsymbol{x}'(t) = \boldsymbol{x}'(0) \equiv \boldsymbol{x}(0),$$

i.e., a stationary object. On the other hand, application of the transformation (2) to the video-controller trajectory (1) yields

$$\boldsymbol{x}'(t) = \boldsymbol{x}'(0) + \boldsymbol{v}\,\{\mathrm{floor}(t) - t\}\,. \tag{3}$$

The function $f(t) \equiv \text{floor}(t) - t$ appearing here can be recognised as simply a "saw-tooth" function, ramping linearly from $f(0) = 0$ to $f(1^-) = 1^-$, at which instant it jumps back to $f(1^+) = 0^+$; it then ramps linearly back up to $f(2^-) = 1^-$, and jumps back down to $f(2^+) = 0^+$; and so on. *It is this spurious motion, and this motion alone, that causes the sample-and-hold display philosophy to perform poorly for uniformly moving objects.* The basic idea of a frame buffer is not the problem: rather, the fault lies with the naïve way in which it is used. It is also seen why a longer update period worsens the effect: the object "wanders" further—and for a longer time—before "jumping" back to its correct position. It is not surprising that such an effect is nauseous; the amount of inebriation required to simulate this effect in Real Reality is more than enough to separate the participant from his or her last meal.

This spurious motion can also be viewed in another light. If one draws a *space-time diagram* of the trajectory of the object as depicted by the sample-and-hold video display, one obtains a staircase-shaped path. The *correct* path in space-time is, of course, a straight line. The saw-tooth error function derived above is the difference between these two trajectories; the "jumping" is the exact spatio-temporal analogue of *the jaggies*—the (spatial) "staircase" effect observable when straight lines are rendered in the simplest way on bitmapped (or rectangular-grid-sampled) displays. The mathematical description of this general problem with sampled signals is *aliasing*; in rough terms, high-frequency components of the original image "masquerade as", or *alias*, low-frequency components when "sampled" by the bitmapping procedure, rendering the displayed image a subtly distorted and misleading version of the original.

As is well-known, however, aliasing *can* be avoided in a sampled signal, by effectively filtering out the high-frequency components of the original signal before they get aliased by the sampling procedure. This technique, applied to any general sampled signal, is termed *antialiasing*; in the field of computer graphics, reference is often made to *spatial antialiasing* techniques used to remove "the jaggies" from scan-converted images. (This is often shortened, in that field, to the unqualified term "antialiasing"; we shall reject this trend and reinstate the adjective "spatial".) For the same reasons, the "jerky motion" of sample-and-hold video controllers is thus most accurately referred to as *spatio-temporal aliasing*; any method seeking to remove or reduce it is *spatio-temporal antialiasing*.

One form of spatio-temporal antialising is performed every time we view standard television images. Generally, television cameras have an appreciable *shutter time*: any motion of an object in view during the time the (electronic) shutter is "open" results in *motion blur*. That such blur is in fact a *good* thing—and not a shortcoming—may be surprising to those unfamiliar with sampling theory. However, the fact that the human eye easily detects the weird effects of spatio-temporal aliasing if motion blur is *not* present, even at the relatively high field rate of 50 Hz (or 60 Hz in the US), can be appreciated by viewing any footage from a modern sporting event, such as the Barcelona Olympics. To improve the quality of the now-ubiquitous slow-motion replay (for which motion blur is stretched to an unnatural-looking extent), such events are usually shot with cameras equipped with *high-speed* electronic shutters, *i.e.*, electronic shutters that are only "open" for a small fraction of the time between frames. The resulting images, played at their natural rate of 50 fields per second, have a surreal, "jerky" look (often called the "fast-forward effect" because the fast picture-search methods of conventional video recorders lead to the same unnatural repression of motion blur). This effect is, of course, simply spatio-temporal aliasing; that it is noticeable to the human eye at 50 fields per second (albeit only 25 *frames* per second) illustrates our visual sensitivity. (For computer-generated displays, for which simulating motion blur may be relatively computationally expensive, increasing the refresh and update rates to above 100 Hz and relying on integration by the CRT phosphor or LCD pixel, and our visual system, may be the simplest solution.)

This *frame*-rate spatio-temporal aliasing, which is relatively easy to deal with, is not usually a severe problem. Our immediate concern, on the other hand, is a much more pronounced phenomenon: the *update*-rate spatio-temporal aliasing produced by the sample-and-hold nature of conventional video controllers (the spurious motion described by (3)). Correcting the video controller's procedures to remove this spurious motion is thus our major task. Again, we recall Zeno's question: how does the arrow know where to move, if it only knows where it is, not where it's going? The answer, supplied first by Galileo (albeit in a somewhat long-winded form, in pre-calculus days), is that we need to know the *instantaneous time derivative of the position* (*i.e.*, instantaneous velocity) of the object at that particular time, in addition to its position. We shall refer to the use of such information (or, in general, any arbitrary number of temporal derivatives of an object's motion) to perform update-rate spatio-temporal antialiasing as *Galilean Antialiasing*. Suggested methods for carrying out this procedure with existing technology are described in the remainder of this paper.

To carry out this task, we need to reexamine the video controller philosophy described in section 2.1. The most obvious observation that strikes one is that, using that design methodology, *velocity information is not provided to the video controller at all!* The reason for this omission is easily understood in historical perspective. *Television* applications for CRTs preceded computer graphics applications by decades. At least initially, all television images were generated by simply transmitting the signal from a video camera, or one of a number of available cameras. However, normal video cameras have no facilities for determining the *velocity* of the objects they view. (Although this is not, in principle, impossible, it would be technically challenging, and quite possibly of no practical use.) Rather, the high frame and field rate of a television picture alone, together with suitable motion blur, were sufficient to convince the viewer of the television image that they were seeing continuous, smooth motion.

When CRTs were first used for computer applications, in vector displays, the voltages applied to the deflection magnets were directly controlled by the video harware; such displays' only relation to television displays was that they both used CRT technology. However, when simple *rasterised* computer displays became feasible in the early 1970s, it was only natural that their development was built on the vast experience gathered from television technology—which, as noted, has no notion of storing velocity information. In fact, it is only in very recent years that memory technology has been sufficiently advanced that the *physics of the display devices*—rather than the amount of amount of video memory feasible—is now the limiting factor in developing ever more sophisticated displays at a reasonable price. To even contemplate storing the velocity information of a frame—even if it *were* possible to determine such information—is something that would have been unthinkable ten years ago. It is, of course, no coincidence that the field of Virtual Reality has also just recently become cost-effective: the immature state of processor and memory technology was the critical factor that limited Sutherland's pioneering efforts twenty-five years ago. It is thus no surprise that the fledgling commercial field of Virtual Reality requires new approaches to traditional problems.

Of course, the very nature of Virtual Reality, while putting us in the position of requiring rapid updates to the entire display, conversely provides us with the *very* information about displayed objects we need: namely, velocities, accelerations, and so on, rather than just the simple *positional* information that a television camera provides. Now, it is of course a trivial observation that all virtual-world engines already "know" about the laws of Galilean mechanics, or Einsteinian mechanics, or nuclear physics—or indeed any system of mechanics that we wish to program into them, either based on the real universe, or of a completely fictional nature. In that context, our rehash of the notions behind Galilean mechanics may seem trivial and unworthy of the effort spent. What existing virtual-world engines do *not* do, however, is *share some of this information with the video controller*. On this front, apparent triviality is magnified to enormous importance; our neglect of these same physical laws is, in fact, creating an artifical and unnecessary degradation of performance in many existing Virtual Reality hardware methodologies.

There is no reason for this omission to continue; the physics has been around for over three hundred and fifty years; and, fortunately, the technology is now ripe. The following sections will provide, it is hoped, at least a very crude and simplistic outline of the paths that must be travelled to produce a fully-functional Virtual Reality system employing Galilean Antialiasing.

2.4. Galpixels and $G^{(n)}$ pixmaps

Historically, rasterised computer graphics came into existence as soon as solid-state memories of suitable capacity were able to be fabricated. It is therefore not difficult to guess the number of bits that were initially allocated to each pixel: one. Such technology was, for this reason, also referred to as *bitmapped graphics*: the bits in the memory device provided a rectangular "map" of the graphics to be displayed—which, however, could therefore only accomodate bi-level displays.

As memory—and the processor power necessary to use it—became even more plentiful, rasterised display options "fanned out" in a number of ways. At one extreme, the additional memory could be used to simply improve the spatial resolution of the display, while maintaining its bitmapped nature. At the other extreme, the additional memory could be used exclusively to generate a multi-level response for each pixel position—for grey-scale, say, or a choice of colours—without increasing the resolution of the display at all; the resulting memory map, now no longer accurately described as a "bit" map, is preferentially referred to as a *pixmap*. In between these two extremes are a range of flexible alternatives; to this day, hardware devices often still provide a number of different "video modes" in which they can run.

Increasing the memory availability yet further led, in the 1980s, to the widespread use of *z-buffers*, both in software and, increasingly, hardware implementations (whereby the "depth" of each object displayed on the display is stored along with its intensity or colour). We can see here already an extension to the concept of a pixel: not only do we store on–off information (as in bitmaps), nor simply intensity or colour shading information (as in early pixmaps), we also include additional, *non-displayed* information that assists in the image generation process. (Current display architectures also routinely store several more bits of *control* information for each pixel.)

We now extend this concept of a "generalised pixel" still further, with our goal of Galilean Antialiasing firmly in our sights. As well as storing the pixel shading, z-buffer and control information, we shall also store the *apparent velocity* of each pixel in the pixmap. We use the term *apparent motion* to describe the motion of objects in terms of display Cartesian coordinates: x horizontal, increasing to the right; y vertical, increasing as we move upwards; and z normal to the display, increasing as we move out from the display towards our face. This motion will typically be related to the *physical motion* of the object (*i.e.*, its motion through the 3-space that the system is simulating) by perspective and rotational transformations; however, in section 4, more sophisticated transformations are suggested between the apparent and physical spaces.

Thus, for z-buffered displays (assumed true for the remainder of this paper), three apparent velocity components must be stored for each pixel—one component for each of the x, y and z directions. The motional information stored with a pixel, however, need not be limited to simply its apparent velocity. In general, we are free to store as many instantaneous derivatives of the object's motion as we desire. The rate of change of velocity, the *acceleration* vector $\boldsymbol{a}$, is an obvious candidate. The *rate of change of acceleration*, which the physicist Richard P. Feynman christened the *jerk*, may likewise be stored; as can the rate of change of jerk (for which the author knows no proposed name); the rate of change of the rate of change of jerk; and so on.

We shall defer to the next section the process of deciding just how many such motional derivatives we should store with each pixel. For the moment, we shall simply refer to any pixmap containing motional information about its individual pixels as a *Galilean pixmap*, or *galpixmap*. The individual pixels within a galpixmap will be referred to as *Galilean pixels*, or *galpixels*. Of course, in situations where distinctions need *not* be made between these objects and their traditional counterparts, the additional prefix *gal-* may simply be omitted.

Finally, it will be useful to have some shorthand way of denoting the highest order derivative of (apparent) motion that is stored within a particular galpixmap. To this end, we (tentatively) use the notation *$G^{(n)}$ pixmap*, or, more verbosely, *Galilean pixmap of order n*, where n is the order of the highest time-derivative of the apparent position of each galpixel that is stored in the galpixmap. Thus, a conventional pixmap, which only records the position of each pixel (encoded by its position in the pixmap, together with its z-buffer information), and *no* higher time derivatives, may be described as a $G^{(0)}$ pixmap. Galpixmaps that store velocity information as well are $G^{(1)}$ pixmaps; those that additionally store acceleration information are $G^{(2)}$ pixmaps; and so on.

As will be seen shortly, additional pieces of information, over and above mere motional derivatives, will also be required in order to effectively carry out Galilean Antialiasing in practical situations. Although the amount of information thus encoded may vary from implementation to implementation, we shall not at this stage propose any notation to describe it; if indeed necessary, such notation will evolve naturally in the most appropriate way.

2.5. Selecting a Suitable Galpixmap Structure

We now turn to the question of determining *how much* additional information should be stored with a galpixmap, in order to maximise the overall improvement in visual capabilities of the system that are perceived by the viewer. Such questions are only satisfactorily answered by considering *psychological* and *technological* factors in equal proportions. That a purely technological approach fails dismally is simply shown: consider the sample-and-hold video controller philosophy described in section 2.1, as (successfully) applied to static objects on the display. We noted there that the video controller effectively boosted the perceived information rate of the display from the *update* rate up to the *refresh* rate, simply by repeatedly showing the same image. Shannon's information theory, however, tells us that this procedure *does not*, in fact, increase the information rate one bit: the repeated frames contain no new information—as, indeed, can be recognised by noting that the viewer could, if she wanted to, reconstruct these replicated frames "by hand" even if they were not shown. Thus, even though we *know* that frame-buffered rasterised displays "look better" than display systems without such buffers (*e.g.*, vector displays), information theory tells us that, in a raw mathematical sense, the frame buffer itself doesn't do anything at all—a fact that must be somewhat ironically amusing to at least one of Shannon's former PhD students.

Raw mathematics, therefore, does not seem to be answering the questions we are asking. A better way to view this *apparent* increase in information rate is to examine the viewer's subconscious prejudices about what her eyes see. She may not, in fact, even realise that the display processor *is* only generating one update every so often: to her, each frame looks just as fair dinkum as any other. All of this visual information—a static image—is simply preprocessed by her visual system, and compared against both "hard-wired" and "learnt" consistency checks. Is a static image a reasonable thing to see? Did I really see that? Was I perhaps blinking at the time? Am I moving or am I stationary? What do I *expect* to see? It is the lightning-fast evaluation of these types of question that ultimately determines the "information" that is abstracted from the scene and passed along for further cogitation. In the case described, assuming (say) a stationary viewer sitting in front of a fixed monitor, all of the consistency checks balance: there appears to be a fair-dinkum object sitting in front of her. In other words, the display is providing sufficient information for her brain to conclude that the images seen are consistent with what would be seen if a real object were sitting there and reflecting photons through a transparent medium in the normal way; that is all that ultimately registers.

We now turn again to our litmus test: an object with a *uniform apparent velocity* being depicted on the display. Using a $G^{(0)}$ display, such as described in section 2.1, results in the motion depicted in equation (1). What does the viewer's visual system say now? Is that an object that keeps disappearing and popping up somewhere else? Is it really something moving so jerkily? Why doesn't it move like any animal I've ever seen before? The answers to these questions depend on just how slow the update rate is, the context that the images are presented in, and, most likely, the past experiences of the viewer. Let us assume, however, that her brain *does* decide that the scene is, in fact, depicting a single object in motion, rather than the spontaneous and repetitive destruction and creation of similar-looking objects. Immediately after this decision is reached her visual processing system performs a hard-wired Galilean transformation to that part of the scene in which the object moves, such as described in section 2.3. Why? Because as animals we learnt the hard way that it isn't enough to simply know whether something is moving as a whole—one also needs to know what the motion of the *parts* of the object are. Is that human walking towards us with arms swinging by its sides, or with arms outstretched ready to throttle us? The Galilean transformation (2) removes the (already-decided-on) uniform motion of the object, to let the viewer then determine the relative motion of its constituent parts. The result, as we have already shown in section 2.3, is a weird-looking saw-tooth motion, involving spontaneous teleportations every time the frame buffer is updated. Does this look like any animal we have ever seen? No, not really. OK, then, maybe we didn't see it too well? Yes—that must be it—I probably didn't see it properly. How well this rationalisation can be tolerated depends on how low the update rate is: seeing *is* believing—but only if you see it for at least 100 milliseconds.

Let us now assume that the display system is not a $G^{(0)}$ device at all, but is rather one of the freshly-unpacked $G^{(1)}$ models. At some initial time, the object appears on the display; the frame buffer has been updated to show that it is there. (Ignore, for the moment, this instantaneous birth.) One frame later, the display processor is still busy redrawing things; the video controller must decide itself what to do with the image. Firstly, it clears a *third* frame buffer (in addition to the one that it has just finished scanning from, and the one that the display processor is talking to), into which it is going to generate a new image. Secondly, it goes through the entire frame buffer that it has just displayed, galpixel by galpixel. At each pixel, it retrieves the velocity information for that galpixel. It then adds this velocity (measured in pixels per frame) to the current position, to find out where that galpixel would be one frame later. It then writes this information into the new frame buffer at the appropriate position; and repeats the process for all the galpixels in the original frame buffer. Thirdly, it worries a bit about those galpixels in the new frame buffer that didn't get written to; let us ignore these worries for the moment, and just leave the "background" colour in those pixels. Finally, it scans the new frame buffer onto the display device.

Ignore, for the moment, that the procedure described seems to double the amount of time for the video controller to do its work. (A moment's reflection reveals that, in any case, there is no fundamental reason why the new-frame-drawing procedure cannot occur at the same time that the *previous* frame is being scanned to the display device.) What will the viewer think that she is seeing? Well, the object will clearly jump a small distance each frame—with each jump exactly the same size as the last (at least, to the nearest pixel), until the new update is available. If the object really *is* travelling with constant apparent velocity (our assumption so far), then upon receipt of the new image update, the object will jump *the same* small distance from the last (video-controller-generated) frame as it has been jumping in the mean time (assuming focus-switching is appropriately synchronised, of course). Now, the *refresh* rate of the system is assumed to be significantly faster than the visual system's temporal resolution; therefore, the motion will look like convincingly like uniform motion. Uniform motion has been Galilean antialiased!

Let us examine, now, what "residual" motion we are left with when this uniform motion is "subtracted off", via a Galilean transformation, from the perceived motion. We now—thankfully—do not end up with the horrific expression (3), but rather with an expression that is *almost* zero. In the setup described the error is not *precisely* zero—if the apparent velocity of the object does not happen to be some integral number of pixels per frame, then the best we can do is move the pixel to the "closest" computed position—leading to a small pseudo-random saw-tooth-like error function in space-time, *i.e.*, we are hitting the fundamental physical limits of our display system. However, the fact that the *amplitude* of the error is at most one pixel in the spatial direction, and one frame in the temporal direction, means that it is a vastly less obtrusive form of antialiasing than the gross behaviour described by (3). (If so desired, however, even this small amount of spatio-temporal aliasing can be removed with suitable trickery in the video controller; but we shall not worry about such enhancements in this paper.)

Having successfully convinced the viewer of near-perfect constant motion, let us now worry about what happens if the object in question is, in fact, being *accelerated* (in terms of display coordinates), rather than moving with constant velocity. For simplicity, let us assume that the object is undergoing *uniform acceleration*. Fortunately, such a situation is familiar to us all: excluding air resistance, all objects near the surface of the earth "fall" by accelerating ("by the force of gravity", in 19th century terminology) at the same constant rate. How do our various display systems cope with this situation?

Let us assume that the object in question is initially stationary, positioned near the "top" of the display. Let us further assume that the acceleration has the value 2 pixels per frame per frame. Firstly, let us consider the optimal situation: the display processor is sufficiently fast to update the object each frame. Clearly, if we shift our axes in such a way that $y = 0$ corresponds to the initial position of the object, its vertical position in successive frames will be given by

$$-y = 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, \ldots, \tag{4}$$

as is verified from the formula $y = y_0 + v_0 t + at^2/2$, where in this case $y_0 = 0$, $v_0 = 0$ and $a = -2$, and t is measured in frame periods. Since this motion is depicted at the *refresh* rate—which, by our assumptions, is sufficiently fast that our visual system does not perceive any inherent temporal aliasing—the object should look a real object falling, without air resistance, to the ground. (We defer a more complete discussion of *motion blur* to another place.)

Let us now examine how this object is depicted on a $G^{(0)}$ display device, such as described in section 2.1. For simplicity, assume that the display processor takes precisely *two* frame periods to render each completed image. Clearly, the sample-and-hold nature of the video controller will simply yield the positions

$$-y = 0, 0, 4, 4, 16, 16, 36, 36, 64, 64, \ldots. \tag{5}$$

The error in each frame can be obtained by simply subtracting from the values in (5) their counterparts in (4), yielding

$$\mathit{\Delta} y_{\text{error}} = 0, 1, 0, 5, 0, 9, 0, 13, 0, 17, \ldots.$$

It is apparent that the error gets worse as the object accelerates: it does not simply "ramp" between two bounds as was the case for uniform velocity. Whether or not this can be recognised by our viewer as smooth motion or not depends on the resolution of the device. However, it should be noted that the *psychological* mismatches that are accumulating here are of a worse nature than for simply uniform motion. The reason for this is that, firstly, the viewer's visual preprocessing system must decide, at each point in time, if the object on the display really is moving at all—*i.e.*, whether it has a *velocity*, such as described above. Secondly, her visual processing system must then subconsciously determine whether the object is in fact *accelerating*. How do we know that she cares about acceleration? *Toss a tennis ball her way.* The fact that humans can catch projectiles moving under the acceleration of gravity shows that we are capable (by some mechanism) of mentally computing the effects of acceleration. This is not surprising, given that everything on the surface of the earth not held up by something else accelerates downwards at a constant rate. Whether *space*-born and -bred humans would develop their visual systems in the same way, or whether they would "evolve" to reduce the psychological importance of acceleration in their thinking, is a question that is beyond the author's reckoning; but it is nevertheless a hypothetical (or, perhaps in time, a not-so-hypothetical) question that emphasises the all-important fact that *the way we perceive the world depends greatly on how we are used to seeing it behave*.

Let us now return to the case of the falling object, and determine how its motion will be depicted on the $G^{(1)}$ display device that served our purposes so admirably in our uniform-velocity thought experiment above. Again, assume that the display processor updates the image only once every two frames. On each update, the *velocity* of each galpixel must also computed; for the object in question, the formula $v = v_0 + at$, with $v_0 = 0$, yields the velocity for each successive frame:

$$-v_{\text{stored}} = 0, 0, 4, 4, 8, 8, 12, 12, 16, 16, \ldots . \tag{6}$$

We have here copied the velocity from every even-numbered frame (the ones being updated) to the odd-numbered frames: the video controller, having no better information, simply continues to assume that the velocity of the galpixel is constant until the next update arrives, and thus copies this information from frame to frame as it copies the galpixel. In the current example, this "propagated" velocity (*i.e.*, the velocity that is assumed constant from frame to frame) is not actually used to compute anything (as the video controller only fills in *one* frame itself after each update), but we shall shortly examine a case in which it *is* used (namely, when the display processing takes longer than two frames to generate each update).

It is straightforward to compute how the $G^{(1)}$ display system will depict our uniformly-accelerated object's motion: using the velocities in (6) to extrapolate the object's *position* for every odd frame, we obtain

$$-y = 0, 0, 4, 8, 16, 24, 36, 48, 64, 80, \ldots .$$

Subtracting from this the sequence of *correct* positions, (4), we find that

$$\mathit{\Delta} y_{\text{error}} = 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, \ldots .$$

Obviously, we have here a vast improvement over the $G^{(0)}$ display: the errors, while not zero, are at least *bounded*.

Let us now consider making these last two examples just a tad more realistic. Keeping the other parameters constant, let us assume that the display processor is now only fast enough to generate one update every *three* frames. Our dust-gathering $G^{(0)}$ display system will successively show the object to be at the positions

$$-y = 0, 0, 0, 9, 9, 9, 36, 36, 36, 81, \ldots .$$

The errors represented by these positions are, respectively,

$$\mathit{\Delta} y_{\text{error}} = 0, 1, 4, 0, 7, 16, 0, 13, 28, 0, \ldots .$$

Obviously, the errors are worse than even the horrible performance with two frames per update. Let us, therefore, turn our attention immediately away from these horrible figures (in the best spirit of politicians), and consider instead our now-worn-in $G^{(1)}$ video controller. The velocity values stored with the galpixmap, as copied across by the video controller where necessary, will now be given by

$$-v_{\text{stored}} = 0, 0, 0, 6, 6, 6, 12, 12, 12, 18, \ldots.$$

The position values used to display the object, as determined by the video controller, can then be computed to be

$$-y = 0, 0, 0, 9, 15, 21, 36, 48, 60, 81, \ldots, \tag{7}$$

which represent errors of

$$\mathit{\Delta} y_{\text{error}} = 0, 1, 4, 0, 1, 4, 0, 1, 4, 0 \ldots.$$

We now see the precise capabilities of a $G^{(1)}$ system when dealing with a uniformly-accelerated object. In between updates, the positional error increases *quadratically*—*i.e.*, in a parabolic shape. Once the display processor provides an update, the positional error returns (as it always does) to zero. It then proceeds to increases quadratically to the *same* maximum error as before (not to ever-worse values, as is the case for a $G^{(0)}$ system); and repeats the cycle. This bounded-error feature of a $G^{(1)}$ display system with uniformly *accelerated* motion resembles, to some extent, the (bounded) positional errors associated with a $G^{(0)}$ system for an object uniform *velocity* (apart from the change of shape from ramp to parabola, of course). This is not very surprising when one considers that, in both of these cases, the video controller "knows" about all of the non-zero temporal derivatives of the object's motion save for the highest order one. On the other hand, this raises a worrying problem: in general, the apparent motion of an object on the display will be an arbitrary analytical function of time (excluding, for the moment, object "teleporting", and any non-physical exotic mathematical functions introduced by a devious programmer). The problem is that an arbitrary analytical function has an *infinite number* of non-zero positive-exponent Taylor series coefficients, *i.e.*, one needs to know *all* of its derivatives to extrapolate its motion indefinitely. How can we possibly justify going on to $G^{(2)}$ systems, then $G^{(3)}$ systems, then $G^{(4)}$ systems, *et cetera ad infinitum*? Are we doomed?

Clearly, this paper would not be in public circulation were this problem a real one, rather than a case of mathematics-gone-wild. Already, we have seen that using a $G^{(1)}$ display system vastly improves upon the performance of a $G^{(0)}$ system, even if it cannot *fully* extrapolate accelerated motion. If all we are interested in is *some* low-cost performance improvement, rather than full mathematical rigour in extrapolation, then we already know how to obtain it. However, we shall shortly see that we can do better than this: there is clearly a point of diminishing returns beyond which it is not worthwhile going to the next $G^{(n+1)}$ system. To justify this statement, however, we must first remove our attention from the abtract world of mathematics, and focus again on the most important component in our physical system: *the participant*.

We first return to the above thought experiment of using a $G^{(1)}$ display system for uniformly accelerated motion. We earlier obtained results for the *positional* error of the display (the repeated-parabola): what about the *velocity* error? To assess this, we must first invent some crude model for the way in which the viewer's brain computes velocities in the first place. The simplest suggestion for such a model, based (perhaps very erroneously) on traditional mathematical methods, might be that the visual system simply takes *differences in positions* between "brain ticks"—or, here, between frames—to compute an average velocity for each "tick". Admitting, for the moment, that this model is not too outrageous, we can proceed to compute the values that the viewer's brain would compute, using such an algorithm, when viewing the $G^{(1)}$ display results in (7). Taking these differences between positions at adjacent times, we obtain

$$-v = 0, 0, 9, 6, 6, 15, 12, 12, 21, \ldots. \tag{8}$$

The problem is, however, the following: how do we ascribe a particular *time* to each of these values, being as they are differences between *two* adjacent times? For example, if we take the difference $y(5) - y(4)$, does this velocity "belong" to $t = 4$ or $t = 5$? The least complicated choice is to simply decide that this velocity evaluation "belongs" to $t = 4.5$—"split the difference", as it were. With this ansatz, the sequence of values (8) corresponds to the perceived velocity computed at the times $t = 0.5, 1.5, 2.5, 3.5, \ldots$. The *correct* velocity values, on the other hand, evaluated at these particular times, are simply computed via $v = v_0 + at$ as

$$-v = 1, 3, 5, 7, 9, 11, 13, 15, 17, \ldots. \tag{9}$$

Taking the difference between (9) and (8) thus yields the errors in the computed velocities:

$$\Delta v_{\text{error}} = -1, -3, +4, -1, -3, +4, -1, -3, +4, \ldots.$$

Now, this is the first error sequence that we have encountered that has had *both positive and negative values*. In fact, it is clear that the *time-average* of this sequence is zero: $(-1) + (-3) + (+4) = 0$. In other words (as would have already been blindingly obvious to anyone who has ever designed a feedback-control system), using $G^{(1)}$ antialiasing results in the correct *average* velocity being depicted for an object (where we average over a complete update period), *even if the motion itself contains higher derivatives*. It is this observation that will tell us when to stop increasing the n in $G^{(n)}$.

Let us now entertain the notion that we could, at this stage, be forever more happy with our $G^{(1)}$ display technology. Excluding the fact, for the moment, that modern computer equipment already has a half-life shorter than most pairs of socks, would the general *principles* of $G^{(1)}$ antialiasing be such that we would never desire to venture to any $G^{(n)}$ for $n > 1$? That this is not an unreasonable notion is supported by the simple fact that $G^{(0)}$ technology has been widespread for about twenty years, and no one seems to be complaining about *it*. (Yet.) On the other hand, since this section does not end at the end of this sentence, the reader is no doubt anticipating that the story of our hypothetical viewer is not yet complete.

The main problem with our simplistic examples above, which will show why they are not completely representative of the real world, is that they are all *one-dimensional*. Of course, we can tranform the "falling" example to a "projectile" situation by simply superimposing an initial horizontal velocity on the vertical motion; this would simply require applying the uniform motion and uniform acceleration cases cojointly. However, we have also simplified the world by only talking about some unspecified, featureless, Newtonian-billiard-ball-like "object", without worrying about the extended three-dimensional structure of the object. A 2-D world, of course, is infinitely more interesting than a 1-D world, not least because it becomes possible to step *around* other people, instead of simply bouncing into them. In a "$2\frac{1}{2}$-D" system—such as in many video games, or Microsoft Windows—*three*-dimensional structure is represented by painting the object with depth-emulating visual clues; but the objects still only move, effectively, in two dimensions. On the other hand, in a manifestly three-dimensional environment such as Virtual Reality, the two-dimensional display is a subtle *perspective projection* of the virtual three-space. Objects in such virtual worlds, unlike their 2-D or $2\frac{1}{2}$-D counterparts, are free to rotate about arbitrary axes, as well as to move closer to or further from the participant. The *apparent* motion of the objects on the display device is a complicated interplay between the physical motion of the objects, the physical motion of the participant, and the mathematical transformations yielding the perspective view. It is important that we consider more general forms of motion, such as this, before drawing any general conclusions about whether $G^{(1)}$ antialiasing is good enough for practical purposes.

In fact, a fairly simple example will suffice to illustrate the problems that are encountered when we move up from 1-D to 2-D or 3-D motion. Consider a two-dimensional square, in the plane of the display, which is also *uniformly rotating* in that same plane about the centre of the square. Choose any point on the boundary of this square. The path traced out by this point in time will be a *circle* (as is simply verified by considering that point alone, without the complication of the rest of the square). Now, a $G^{(0)}$ display will represent this rotating point as a series of discrete points on the circle: the overall display will show successive renderings of a square in various stages of rotation, from which the viewer will (hopefully) be convinced that it is a single square undergoing a somewhat jerky rotation.

Now consider how a $G^{(1)}$ display will render this rotating square. The *velocity* of our chosen point on the square will depend on how far the point is from the centre of the square; its direction will be perpedicular to the line joining it to the centre. Consider what happens when we propagate forward in time from this position, using only the velocity information: the point moves in a *straight line*. But we really want it to travel in a circle! What will happen to the square? Well, for very small times, the square will indeed be rotating beautifully—a vast improvement on the $G^{(0)}$ situation. If we keep propagating the square further, however, we find a disconcerting feature: *the square is slowly growing larger as it rotates!* Propagate yet further and this growth dominates, and the rotation slows to a trickle. Of course, when the next update comes along, the square spontaneously shrinks again—not a good solution.

It might be argued that this example merely shows that you cannot stretch the Galilean Antialiasing procedure indefinitely; ultimately, you must update the display at a *reasonable* rate, even if that is considerably lower than the refresh rate. To a certain extent, this is indeed true. However, rejecting $G^{(1)}$ antialiasing as an optimal solution, due to its poor performance for rotation, can be justified on the basis of a good "rule of thumb" in Science: Approximating an arbitrary curve by a straight line is usually pretty bad; approximating it with a parabola is infinitely better; approximating it with a cubic *may* be worthwhile; using anything of higher order is probably a waste of resources, unstable, or both. For sure, this is not a definitive law of Nature, but in any case it suggests that we may be much better off going to $G^{(2)}$, or perhaps $G^{(3)}$, rather than sticking simply with $G^{(1)}$ technology.

This purely mathematical line of reasoning, while most helpful in our deliberations, nevertheless again overlooks the fact that, ultimately, all that matters is what the viewer *thinks* she is seeing on the display, not how fancy we are with our mathematical prowess. To investigate this question more fully, it is necessary to perform some investigations of a psychological nature that are not completely quantitive. These deliberations, however, shall require an additional piece of equipment, readily available to the author, but (unfortunately) not to all Virtual Reality workers: a *Melbourne tram*. (Visitors to the Virtual Reality mecca of Seattle can, however, make use of the authentic Melbourne trams that the city of Seattle bought from the Victorian Government ten years ago, which now trundle happily along the waterfront with a view of Puget Sound rather than Port Phillip Bay.) Melbourne trams (the 1920s version, not the new "torpedo" variety) have the unique property that, no matter how slowly and carefully they are moving, they always seem to be able to hurl standing passengers spontaneously into the lap of the nearest seated passenger (which may or may not be an enjoyable experience, depending on the population of the tram). This intriguing (if slightly frivolous) property of such vehicles can actually be used as a reasonably quantitative experimental investigation of the kinematical capabilities of humans.

Consider a Melbourne tram located in the Bourke Street Mall, sitting at the traffic lights controlling its intersection with the new Swanston Street Mall. A passenger standing inside the tram looks out the window. Apart from wondering why on earth Melbourne needs so many malls—or indeed why one needs traffic lights at all at the intersection of two malls—the passenger is unperturbed; she can take a good look at the surrounding area. She drops a cassette from her Walkman into her handbag: it falls straight in.

Now consider the same tram thirty seconds later, as it is moving at a constant velocity along Bourke Street. The standing passenger is again simply standing around; cassettes drop fall straight down; if it were not for the passing scenery, she wouldn't even know she was moving. And, of course, physics assures us that this is always the case: inertial motion cannot be distinguished from "no" motion, except with reference to another moving object. The laws of physics are the same.

We now take a further look at our experimental tram: it is now accelerating across the intersection at Russell Street. Old Melbourne trams, it turns out, have quite a constant rate of acceleration when their speed controls are left on the same "notch", at least over reasonable time periods. Let us assume that this acceleration is indeed constant. With our knowledge of physics, we might predict that our standing passenger might have to take some action to avoid falling over: the laws of Newtonian physics *change* when we move to an accelerated frame. Inertial objects do not move in straight lines. Cassettes do not fall straight down into handbags. Surely this is a difficult environment in which to be merely standing around?

Somewhat flabbergasted, we find our passenger standing in the accelerated tram, not holding onto anything, unperturbedly reading a novel. How is this possible? Upon closer examination, we notice that our passenger is *not* standing exactly as she was before: *she is now leaning forward*. How does this help? Well, to remain in the same position in the tram, our passenger must be accelerated at the same rate as the tram. To provide this acceleration, she merely leans forward a little. The force on her body due to gravity would, in a stationary tram, provide a torque that would topple her forwards. Why does this not also happen in the accelerating-tram case then? The answer is that the additional forward frictional force of the tram's flooring on her *shoes* both provides a counter-torque to avoid her toppling, as well as the forward force necessary to accelerate her at the same rate as the tram. Looked at another way, were she to *not* lean forward, the frictional force forward of the accelerating tram would produce an unbalanced torque on her that would topple her *backwards*.

Leaning forward at the appropriate angle is, indeed, a fine trick on the part of our passenger. Upon questioning, however, we find to our dismay that she knows nothing about Newtonian mechanics at all. We must therefore conclude that "learning" this trick must be something that humans do spontaneously—everyone seems to get the hang of it pretty quickly.

It should be noted that *this trick would not work in space*. Without borrowing the gravitational force, there is no way to produce a counter-torque to that provided by the friction on one's shoes. Of course, this friction *itself* should not be relied on too much: without any gravitational force pushing one's feet firmly into the floor, there may not be any friction at all! This means that there will, in general, be no unbalanced torque to topple one backwards anyway: the passengers in an accelerating space vehicle, (literally) hanging around in mid-air, will simply continue to move at constant velocity. But the accelerating vehicle will then catch up to them: they will slam against the *back* wall of the craft! Of course, this is precisely Einstein's argument for the Equivalence Principle between gravity and acceleration—if you are a standing passenger, you had better find what will become the "floor" in your accelerated spacecraft quick smart—but it shows that life on earth has prepared us with different in-built navigation systems than what our descendents might require.

What lessons do we learn from this? Firstly, it is unwise to put a Melbourne tram into earth orbit. More importantly, it shows that humans are quite adept at both extrapolating the effects of constant acceleration (as shown by our ability to catch projectiles), as well as being able to function quite easily in an accelerated environment (as shown by our tram passenger).

Let us now examine the tram more closely *before* it accelerates away. It is now sitting at the traffic lights at Exhibition Street. The lights turn green. The driver releases the tram's air-brakes with a loud hiss. Our passenger spontaneously takes hold of one of the overhanging stirrups! More amazingly, another passenger spontaneously starts to fall forwards! The tram jerks, and accelerates away across Exhibition Street. Our passenger, grasping the stirrup, absorbs the initial jerk and, as the tram continues on with a constant acceleration, lets go in order to turn the page of her novel. The second passenger, the spontaneously-falling character, magically did not fall down at all: the tram accelerated at just the right moment to hold him up—and there he is, still leaning forward like our first passeneger! However, all is *not* peaceful: a Japanese tourist, who boarded the tram at Exhibition Street, has tumbled into the lap of a (now frowning) matronly figure, and is struggling to regain his tram-legs.

What do we learn, then, from this experience? Clearly, the Japanese tourist represents a "normal" person. Accustomed to acceleration, but *not* to the discontinuous way that it is applied in Melbourne trams, he became yet another veteran of lap-flopping. Our first passenger, on the other hand, who grabbed the stirrup upon hearing the release of the air-brakes, had clearly suffered the same fate in the distant past, and had learnt to recognise the audible clue that the world was about to shake: such is the Darwinian evolution of a Melburnite. The second passenger, who spontaneously fell forward, appears to be an even more experienced tram-dweller: by simply falling forward he had no need to grasp for a stirrup or the nearest solid structure. This automatic response, which relies for its utility on the fact that Melbourne tram-driving follows a fairly standard set of procedures, is perhaps of interest to behavioural scientists, but does *not* indicate that the passenger had any infallible "trick" for avoiding the effects of jerks (to employ Feynman's term)—the "falling" method does not work at all if the jerk is significantly delayed for some unknown reason. (A somewhat mischievous tram driver once confessed that his favourite pastime was releasing the air-brake and then not doing anything—and then watching all of the passengers fall over.) Of course, the Melbourne trams on the Seattle waterfront have an extra reason for unexpected deceleration: in a city covered by decrepid, unused *train* tracks, motorists turning across the similar-looking *tram* tracks get the fright of their lives when they find a green, five-eyed monster bearing down on them!

Returning, now, to the task at hand, our admittedly simplified examples above show that people are, in general, relatively adept at handling *acceleration* (not surprising, considering our need to deal with gravity), but not too good when it comes to *rate of change* of acceleration, or *jerk*. Numerous other examples of this general phenomenon can be constructed: throw a ball and a person can usually catch it; but half-fill it with a liquid, so that it "swooshes around" in the air, and it can be very difficult to grab hold of. Sitting in a car while it is accelerating at a high rate "feels" relatively smooth; but if the driver suddenly lets off the accelerator just a little, your head and shoulders go flying forward—despite the fact that you are still being accelerated in the *forward* direction! In each of these examples, it is the (thus appropriately named) *jerk* that throws our inbuilt kinematical systems out-of-kilter; not too surprisingly, it is difficult to formulate uncontrived examples in the *natural* world in which jerks are prevalent (apart from falling out of a tree, of course—but repeatedly hitting the ground is not a technique well suited to evolutionary survival).

We now have two pieces of information on which to base a decision about what n should be in a practical $G^{(n)}$ display system. Firstly, we have a purely mathematical "rule of thumb": n should be either 2 or 3 to represent most efficiently (and stably) arbitrary motion around $t = 0$. Secondly, we have a psychological criterion: we have recognised that, in rough terms, the human kinematical-computation system is comfortable with dealing with motional derivatives up to the second, but is quite uncomfortable dealing with the third derivative; this suggests that n should probably be 2, or, at the most, 3. There is, however, a *third* consideration to be taken into account in this question, that has nothing to do with mathematics nor psychology, but rather basic physics and information theory: How many derivatives of the motion can we *accurately specify* in a realistic Virtual Reality situation, in which many of the relevant physical quantities are not merely generated by the computer, but are, in fact, measured by physical transducers? To answer this question, we need to look a little more closely at the physical transducers that are used in real-life Virtual Reality systems, as well as the way that the data they produce is analysed by the system itself.

Clearly, positional–rotational data is the common denominator among existing Virtual Reality transducer technology: find some physical effect that lets you determine how far away the participant is from a number of fixed sensors, as well as her orientation with respect to these sensors, and you "know where she is". Transducers for *velocity* information—which tell you "where she's going" (but not "where she is")—are also commonplace in commerical industry, albeit less common in Virtual Reality. However, even if such transducers are *not* used, quite a reasonable estimate of the true velocity of an object may be obtained by taking differences in positional data (as was used to calculate the results in (8)). On the other hand, this "numerical differentiation" carries two inherent dangers: firstly, computing *any* differentiation on physical data enhances any high-frequency noise present; and, secondly, performing a *discrete-time* numerical differentiation introduces lags into the data (*i.e.*, in rough terms, you need to wait until $t = 5$ to compute $f(5) - f(4) \approx f'(4.5)$). The first problem can be somewhat ameliorated by low-pass filtering; the second by "extrapolating" the data forwards half a time interval; unfortunately, these two solutions are largely mutually exclusive. However, in practice, quite reasonable velocity information *can* in fact be obtained—as long as it is treated with care.

In a similar way, acceleration can either be measured directly, or computed from the velocity data by discrete numerical differentiation. In many respects, the laws of Nature make accelerometers *easier* to make than speedometers. This is, of course, due to the fact that *uniform motion is indistinguishable from no motion at all*, as far as Newtonian mechanics is concerned: it is vitally necessary to measure velocity "with respect to something" (such as the road, for an automobile; or the surrounding air, for an aeroplane—"ground speed" being much more difficult to measure because there is [hopefully] no actual contact with the ground!). On the other hand, *accelerations* cause the laws of physics to change in the accelerated frame, and can be measured without needing to "refer" to any outside object. (Of course, this is not completely true: if (classical) laws of physics are written in a *generally relativistic* way, they will also hold true in accelerated frames; but that is only of academic interest here.) Nevertheless, even though these properties make acceleration inherently easier to measure than velocity, the Virtual Reality designer must ultimately worry about both minimising the cost of the system, and minimising the number of gadgets physically attached to the participant—and it is unlikely that *both* velocity and acceleration transducers would be deemed necessary; one or the other (or, indeed, both) would be omitted. Of course, acceleration may be deduced from velocity data numerically, and carries the same dangers as velocity data obtained numerically from positional data. Most dangerous of all is if acceleration data must be numerically obtained from velocity data that was *itself* obtained numerically from positional data; the errors compound.

Of course, it is also possible actually *omit* measuring positional information altogether, and instead obtain it by integrating measured velocities. This integration actually *reduces* high frequency noise—but what one gains on the swings one loses on the roundabouts: the *low* frequency noise is boosted—manifested, of course, in "drift" in the measured origin of the coordinate system, which must be regularly calibrated by some other means. Alternatively, the relative simplicity of the physics may lead a designer to simply use *accelerometers* as transducers, integrating this information once to obtain velocity data, and a second time to obtain positional data. Of course, this double-integration suppresses high-frequency noise even further, but requires regular calibration of not only the origin of the *positional* coordinate system, but also of the origin of the *velocity* information (*i.e.*, knowing when the transducer is "stationary" with respect to the laboratory)—which is again a manifestation of the general Wallpaper Bubble Conservation Law (*i.e.*, whenever you get rid of one problem it'll usually pop up somewhere else).

Keeping in mind the above technical and design concerns involved in determining even positional, velocity and acceleration information from physical transducers, what chance is there for us to extend this methodology to measuring *jerk* data? On the physics side, there are few (if any) genuinely simple physical effects that could inspire the design of a *jerkometer* (to coin a somewhat obscene-sounding term). It is not even clear that a jerkometer would be all that reliable an instrument anyway: since (by Newton's Second Law, $\boldsymbol{F} = m\boldsymbol{a}$) *accelerations* are caused by *forces*, we see that *jerks* are caused by *rates of change of forces*. Think, now, of what happens when (for example) you lift your leg: at some instant in time you decide, "Hmm, I'd like to lift my leg"; your leg muscles then start to apply a force which overcomes gravity and begins to accelerate your leg upwards. The point is that *the force itself is applied rather abruptly*—in other words, the *jerk* is almost like an *impulse function* (or an "infinite prick" as it is sometimes derogatorily termed). The main problem with impulse functions is that, due to fact that they reach extremely high peak values (for short values of time, such that the area underneath their curve is constant), many physical devices encountering them either *clamp* them to their peak-allowable value (rendering the area under the impulse inaccurate), or get driven into *non-linear behaviour* (which can not only scramble the area-under-curve information, but may indeed drive the whole system into instability). Thus, one would need to be very careful in implementing directly a jerk transducer. Of course, even if one *were* able to manufacture such a jerkometer, incorporating it into Virtual Reality equipment would again come up against the abovementioned barriers of transducer overpopulation and excessive cost.

On the numerical-differentiation side, on the other hand, whether it would be wise or not to perform an extra differentiation of the acceleration data to obtain the jerk data depends largely on where the acceleration information originates. If it is obtained from an actual physical accelerometer, such a numerical differentiation would probably be reasonable. However, if the physical transducer is in fact a velocity- or position-measuring device, then one would not wish to place too much trust on a second- or third-order numerical derivative for a quantity that is already subject to concerns in terms of basic physics: most likely, all one would get would be a swath of potentially damaging instabilities in the closed-loop system. Thus, a numerical approach depends intimately on what order of positional information is actually yielded by the physical transducers.

We now make the following suggestion: The visual display architecture of Virtual Reality technology should make only *minimal* assumptions about the nature of the physical tranducers used elsewhere in the system. This suggestion is based, of course, on the concept of *modularity*: if groups of functional components in any system cohere, by their very intrinsic nature, into readily identifiable "modules", then any one "module" should not be made unnecessarily and arbitrarily dependent on the internal nature of another "module" *unless* the benefits gained from whole outweigh the loss of encapsularity of the one. If this suggestion is accepted (which, in some proprietary situations, may require consideration of the future health of the industry rather than short-term commerical leverage), then it is clear that it would be inappropriate for a display system to assume that a given Virtual Reality environment obtains anything more than raw positional–rotational information from physical transducers. With such a minimalist assumption, our above considerations show that it would *not* be wise, in general, to insist that jerk information about the physical motion of the Virtual Reality participant be provided to the display system. Of course, this does not prevent jerk information being obtained about the other *computer-generated* objects in the virtual world—their trajectories in space are (in principle) knowable to arbitrary accuracy; arbitrary orders of temporal derivative may be computed with relative confidence. However, if a display system *did* use jerk information for virtual objects, but not for the participant herself, it would all be for naught anyway. To appreciate this fact, it is only necessary to note that *all* of the visual information generated in a virtual world scenario *depends solely on the relative positions of the observer and the observed*. Differentiating the previous sentence an arbitrary number of times, it is clear that the *only* relevant velocities, accelerations, jerks, *etc.*, in a Galilean antialiased display environment are the *relative* velocities, accelerations, jerks, *etc.*, of the observer and the observed. Of course, this property of *relativity* is a fundamentally deep and general principle of physics, whether it be Galilean Relativity, or Einstein's Special Relativity or General Relativity; it will probably not, however, be an in-built and intuitively obvious part of humanity's subconscious until we more regularly part company with *terra firma*, and travel around more representative areas of our universe. (Ever played *Wing Commander*? Each Terran spacecraft has a maximum speed. But *with respect to what?!* Galileo would turn in his grave. . . .) Thus, using jerk information for one half of the system (the virtual objects) but not the other half (the participant) brings us no benefits at all—and, indeed, the inconsistencies in the virtual world that would result may well be a significant degradation.

Returning, again, to the task at hand, the above deliberations indicate that, all in all, the most appropriate order of Galilean antialiasing that should be used, at least for Virtual Reality applications, is probably $G^{(2)}$. Of course, much of the above relies on only back-of-the-envelope, first-principles arguments, which may well be rejected upon a more careful investigation; and, of course, anyone is free to develop technology to whatever specifications they like, if they believe that their end product will be marketable. However, for the purposes of the remaining sections of this paper, we shall assume that $G^{(2)}$ antialiasing is used exclusively; the results obtained, and conclusions drawn, would need to be extended by the reader if a different order of antialiasing were desired.

3. A Minimal Implementation

The previous sections of this paper have been concerned with the development of the underlying philosophy of, and abstract planning for, Galilean Antialiasing in general. In this section, we turn directly to the practical question of how one might retrofit these methods to existing Virtual Reality technology. Section 3.1 outlines the minimal modifications to existing display control hardware that must be implemented; section 3.2 describes, in general terms, the corresponding software enhancements necessary to drive the system. More advanced enhancements to the general visual-feedback methodology of Virtual Reality—which would, by their nature, be more amenable to implementation on new, ground-up developments—are deferred to section 4.

3.1. Hardware Modifications and Additions

Our first task in modifying an exisiting Virtual Reality system is to determine precisely what changes must be made to its hardware: if such changes are technically, financially or politically unattainable, then a retrofit will not be possible at all, and further speculation would be pointless. Clearly, the area of existing $G^{(0)}$ technology that has been subjected to most scrutiny in this paper is the *video controller subsystem*. As would by now be obvious, the sample-and-hold frame buffer methodology of conventional display systems, as described in section 2.1, must be completely gutted; in its stead must be installed a tightly-integrated, relatively intelligent video controller subsystem capable of correctly propagating from frame to frame the moving, accelerated (but pixelated) objects passed to it by the display processor. We shall assume, in this paper, that *refresh-rate Galilean Antialiasing* is to be retrofitted; in other words, the video controller computes a new, correctly propagated frame for *each* physical refresh of the display device. This is, of course, the most desirable course of action: the motion depicted on the display device should then (if its physical refresh rate is suitably high) be practically indistinguishable from smooth motion. However, in a retrofit environment, memory- and circuitry-speed constraints may quite possibly render this goal unachieveable. In such a situation, *sub-refresh-rate Galilean antialiasing* may be opted for: the actual "frame period" used for the propagation circuitry would, in that case, be chosen to be some integral multiple of the physical refresh period, *i.e.*, the propagator update rate would then be a sub-harmonic of the physical refresh rate. For example, for a display system with a 60 Hz physical refresh rate, a "Galilean refresh rate" of 30 Hz, or 20 Hz, or 15 Hz, *etc.*, may be chosen instead of the full 60 Hz. Such an approach, however, carries the potential for many headaches: the existing pixmap frame buffers used by the scan-out circuitry may (or may not) need to be retained, with *additional* pixmap buffers for the propagation process; the results may then need to be copied at high speed from the propagation frame buffer to one of the scan-out frame buffers; and so on. In any case, sub-refresh-rate Galilean Antialiasing will not be specifically treated in the following; practitioners wishing to use this technique will need to take references to "refresh rate" and "frame buffers" to mean the corresponding objects in the *propagation* circuitry; requirements for the *scan-out* versions of these objects must then be inferred by the practitioner.

As noted in section 2.4, the galpixmaps that will be stored in the (now necessarily multiple) frame buffers extend significantly on the simple intensity or colour information stored in a regular pixmap. However, there is clearly no need for a galpixmap to be *physically* configured as a rectangular array of galpixel structures (in terms of physical memory); rather, a much more sensible configuration—especially in a retrofit situation—is to maintain the existing hardware for the frame buffer pixmap (and duplicate it, where necessary), and construct new memory device structures for storing the additional information, such as velocity and acceleration, that the galpixmap requires. The advantage of this approach is that, properly implemented, the detailed circuitry responsible for actually scanning out the frame buffer to the physical display device may be able to be left unchanged (apart, perhaps, for including a frame-buffer multiplexer if hardware double-buffering is not already employed by the system). This is a particularly important simplification for retrofit situations, since, in general, the particular methodology employed in the scan-out circuitry depends largely on the precise nature of the display technology used.

We now turn to the question of what information *does* need to be stored in the extended memory structures that we are adding to each frame buffer. Clearly, the ("display", or "apparent") *position vector* of the i-th galpixel, $\boldsymbol{x}_i$ (where i runs from 1 up to the number of pixels on the display), is already well-spoken for: its x and y components are, by definition, already encoded by the galpixel's physical location in the pixmap matrix; and its z component is stored in the hardware z-buffer memory structure. (If a hardware z-buffer is already implemented in the system, it can be used unchanged; if a hardware z-buffer is *not* present, it *must* be added to the system at this point in time; its presence is vital for Galilean antialiasing, as will soon be apparent.) Is this all the positional information that we require? What might be surprising at first sight is that, in fact, *it is not*; we must, however, first turn to the other memory structures required, and the propagation algorithm itself, to see why this is so.

The *velocity vector* of the i-th galpixel, $\boldsymbol{v}_i$, is a new three-component piece of data that the display processor must now provide for each galpixel; likewise, the *acceleration vector* of the i-th galpixel, $\boldsymbol{a}_i$, must also be supplied. With a $G^{(2)}$ system, of course, the acceleration of a particular galpixel does not change at all while the video controller propagates the scene from frame to frame; however, it would be wrong to think that the *matrix* of galpixel acceleration values similarly stays constant under propagation. The reason, of course, is that *each galpixel carries its velocity and acceleration data along with it*; in physics terms, these quantities are *convective* temporal derivatives ("carried along" with the flow), *not* partial temporal derivatives (which stay put in space). It should be noted that this concept—that a "*gal*pixel" is a little pixelated logical object moving around the display, whereas a plain "pixel" is simply a static position in the frame buffer matrix—will be used extensively in the following description.

The next questions that must be considered are: What numerical format should we store the velocity and acceleration information in? How many bits will be needed? These questions are of extreme importance for the implementation of any Galilean Antialiased technology. That this is so can be recognised by calculating just how many such quantities need to be stored in the video hardware subsystem: We need both a velocity and an acceleration value for every galpixel in a frame buffer. Velocity and acceleration each have three components. We need at least three such frame buffers for each display device (two for the video controller to propagate between, and one for the display processor to play with at the same time); and, for stereoscopic displays, will need two (logical) display devices. Even for a bare-minimum display resolution of (say) 320×200, we are looking down the barrel at 2.3 million quantities that need to be stored somewhere; increasing the resolution to 640×480 blows this out to 11 million quantities. Of course, with RAM currently on the street for A\$45 per megabyte, even a relatively wasteful implementation of memory would not blow out the National Deficit on RAM chips alone; however, configuring such a memory structure in terms of electronic devices, in such a way that it can be processed at video-rate speeds, is a challenging enough task without having to deal with an explosion of complexity.

Let us, therefore, make some crude estimates as to how we would like our physical system to perform. Assume, for argument's sake, that we are implementing a 50 Hz refresh rate display system; each frame period is then 20 milliseconds. Propagating quantities forward with a finite numerical accuracy leads to accumulated errors that increase with time. In particular, the position of an object will be "extrapolated" poorly if we retain too few significant figures in the velocity and acceleration—even ignoring the fact that the acceleration of the object may have, in fact, changed in the mean time. How poor a positional error, arising from numerical accuracy alone, can we tolerate? Let us say that this error should be no worse than a single pixel or so. But the error in position will, in a worst case scenario, increase linearly with time, *i.e.*, number of extrapolated frames. How many frames will we want to extrapolate forwards while still maintaining one-pixel accuracy? Well, since we will be using binary arithmetic eventually, let's choose a power of 2—say, 16 frames. This corresponds to an inter-update time (*i.e.*, the time for which the video controller itself happily propagates the motion of the galpixels between display processor updates) of 320 milliseconds—which should be *more* than enough, considering that the participant's acceleration (not to mention that of the objects in the virtual world) will have no doubt changed by a reasonable amount by then—and the view, if it has not been updated by the display processor, will thus be reasonably inaccurate anyway. Of course, the whole display system won't suddenly fall over if, in some situation, we don't actually get a display processor update for more than 16 frames—it is just that the inherent numerical inaccuracy of the propagation equations will simply grow larger than 1 pixel. We shall say that the display system is *rated for a* $N_{\text{prop}} = 16$ *propagation time*—a number that would appear in its "List of Specifications" at the back of the Instruction Manual.

OK, then, how do we use our design choice of N_{prop} (16, in our case) to determine the accuracy of the velocity and acceleration that we need to store? The answer to that question is, in fact, inextricably intertwined with the particular equation that our hardware will use to propagate galpixels across the display—and, in particular, how it is implemented with finite-accuracy arithmetic—which must therefore be brought to the focus of our attention. Now, we have learnt from our Virtual Galileo, in section 2.2, that the formula for the trajectory of a uniformly accelerated object is given by

$$\boldsymbol{x}(t) = \boldsymbol{x}(0) + \boldsymbol{v}(0)t + \frac{1}{2}\boldsymbol{a}t^2, \tag{10}$$

and, from this, the equation of motion for its velocity is given by

$$\boldsymbol{v}(t) = \boldsymbol{v}(0) + \boldsymbol{a}t. \tag{11}$$

Since our video controller only knows about our galpixel's initial position $\boldsymbol{x}_i(0)$, initial velocity $\boldsymbol{v}_i(0)$ and initial acceleration $\boldsymbol{a}_i(0)$ anyway, assuming a *constant* acceleration $\boldsymbol{a} \equiv \boldsymbol{a}_i(0)$ for the galpixel (until the next display processor update) is about the most reasonable thing to do. If we measure t in units of the frame period—which we shall always do in the following—we can compute *exactly* where the uniformly-accelerated object would be, and what its velocity would be, at the time of the next frame, by simply inserting $t = 1$ into (10) and (11), giving

$$\boldsymbol{x}_i(1) = \boldsymbol{x}_i(0) + \boldsymbol{v}_i(0) + \frac{1}{2}\boldsymbol{a}_i \tag{12}$$

and

$$\boldsymbol{v}_i(1) = \boldsymbol{v}_i(0) + \boldsymbol{a}_i \tag{13}$$

respectively. We now note that, to our delight, the arithmetical operations needed to compute (12) and (13) are not only simple—*they can actually be hard-wired!* The most "complicated" operation we need to perform is (signed) addition, for which a hardware implementation is trivial. And since we will be employing binary numbers, taking half of $\boldsymbol{a}_i$, as required in (12), amounts to simply shifting it right one bit—or, in the hard-wired-adder version of

(12), simply connecting the signals appropriately shifted. Thus, we can already see why it *is* technologically feasible to propagate motion using $G^{(2)}$ antialiasing, even at video rates—the required computations are, by coincidence, trivial for the digital devices that we have at our disposal.

Now, using (12) and (13), how many fractional bits do we need to store for $\boldsymbol{x}_i$, $\boldsymbol{v}_i$ and $\boldsymbol{a}_i$ to ensure 1-pixel accuracy at $N_{\text{prop}} = 16$? At this point, we come to a horrible realisation: *we cannot have any fractional bits at all!* Why do we reach this conclusion? Because, as noted earlier, the x and y components of the $\boldsymbol{x}_i$ information for the galpixel is already encoded in its position in the pixmap matrix. . . and there's no fractional part to a position in a matrix! And without any fractional bits in $\boldsymbol{x}_i$, performing the addition in (13) with fractional bits in $\boldsymbol{v}_i$ or $\boldsymbol{a}_i$ would be a complete waste of time: the fractional bits would be thrown away each frame as we write the galpixel into its new position! *A catastrophe!*

Recovering our composure, we ask: What happens if we *are* restricted to having integral $\boldsymbol{v}_i$ and $\boldsymbol{a}_i$? Is that good enough? Well, let us consider just the velocity for the moment, and assume the acceleration is zero. Clearly, if $\boldsymbol{v}_i = 0$ also, the galpixel won't move anywhere at all until the next display processor update. This is appropriate if the galpixel wasn't supposed to move more than half a pixel in any direction anyway. OK, then, what is the next smallest speed possible? Let us consider just the x direction, for simplicity. The smallest value for v_i^x, in integer arithmetic, is, obviously, $v_i^x = \pm 1$. What will the video controller do with such a galpixel? It will move it one pixel per frame until the next update. But this is terrible—that's a whole 16 pixels (if we wait 16 frames for a display processor update); the thing is moving at 50 pixels per second! And that's the *smallest* non-zero speed we can define! What happens if the galpixel should have been moving at 24 pixels per second? Too bad—it stays put. And if it should have been going at 26 pixels per second? Sorry, 50 is all I can give you. You'll just have to overshoot. *A catastrophe!*

Regaining our composure again, let us reconsider the reason why the proverbial hit the fan in the first place. Our basic problem is that a pixmap matrix has no such thing as a "fractional row" or "fractional column". Maybe we could increase the resolution of our display. . . and call the extra pixels "fractions"? Hardly a viable proposition in the real world—and in any case we'd be simply palming off the problem to the *new* pixels. Maybe we could leave the display at the same resolution, but replace each entry in the galpixmap with a little galpixmap matrix of its own? Then we could have "fractional rows and columns" no problems! Well, how big would the little matrix need to be? Seeing as a velocity of $v_i^x = 1$ pixel per frame moves us by 16 pixels in 16 frames—and we only want to move one pixel, max., in this time period—we should therefore reduce the minimum computable velocity to 1/16 of a pixel per second. This, then, requires a 16×16 sub-galpixmap for each display pixel. We'd need 256 times as much memory as we've already computed before—2.3 millions quantities blows out to almost 600 million! *A catastrophe!*

Regaining our composure for the third (and final) time, let us consider this last proposal a little more rationally. We uncontrollably assumed that what was needed at each pixel location was a little galpixmap, with all the memory that that requires. Do we really need all this information? What would it mean, for example, to have a whole lot of little "baby galpixels" moving around on this hugely-expanded grid? What if the babies of two different (original-sized) galpixels end up on the same (original-sized) galpixel submatrix—does such a congregation of babies make any conceptual sense? Well, our display device only has *one* pixel per submatrix: so who gets it? Do we add, or average, the colour or intensity values for each of the baby galpixels? No—the object *closer* to the viewer should obscure the other. Should it be "most babies wins"? No, for the same reason. Then is there any reason for having baby galpixels at all? It seems not.

Let us, therefore, look a little more closely at these last considerations. Since the display device only has one physical pixel per stored galpixel (of the original size, that is—the baby galpixels having now been adopted out), then, obviously, a galpixel can only move one pixel at a time anyway. But we only want to move the galpixel by one pixel every 16 frames—or every 3 or 7 or 13 frames, or whatever time period will, on the average, give us the right average apparent velocity of the galpixel in question. So how does one specify that the video controller is to "sit around" for some number of frames before moving the galpixel? Simple—put in a little counter, and tell it how many frames to wait. Of course, we need a little counter for each galpixel, but it need only count up to 16, so it only needs 4 bits anyway (in each of the x and y directions)—not a large price. Thus, we *can* get sub-pixel-per-frame velocities, with only a handful extra bits per galpixel!

Let us look at this "counter" idea from a slightly different direction. Just say that we have told the video controller to count up to 16 before moving this particular galpixel one pixel to the right. Why not *pretend* that, on each count, the galpixel "really *is*" moving 1/16 of a pixel to the right—just that we don't actually see it move because our display isn't of a high enough resolution. Rather, the galpixel says to itself on each count, "Hmm, this display device doesn't have any fractional positions; I'll just throw away the fraction and stay here." But then, upon reaching the count of 16, the galpixel says, "Hey, now I'm supposed to be 16/16 pixels to the right—but that's one *whole* pixel, and I can do that!" Clearly, this is a better description for the counter than our original one—we now know what to do if counting, say, by 3s—namely, we count up $3, 6, 9, 12, 15, 18\ldots$, whoops! total is over 16—so move right one pixel and "clock over" to a count of $2, 5, 8, 11, \ldots$; and so on.

And so we come—by a rather roundabout route, to be sure—to the conclusion that the *simplest* way to allow sub-pixel-per-frame velocities is to ascribe to each galpixel two additional attributes: a *fractional position* in each of the x and y directions. The above roundabout explanation has, as a consolation prize, already told us how many bits of fractional positional information we require for a $N_{\text{prop}} = 16$ rated system, namely, 4, for each of the x and y directions. Clearly, the number of bits required in the general case is just equal to $(\log_2 N_{\text{prop}})$, *i.e.*, 3 bits for $N_{\text{prop}} = 8$; 5 bits for $N_{\text{prop}} = 32$, and so on.

There is, however, a slightly undesirable feature of the above specification of the action of fractional position, that we must now repair. In the example given, the galpixel "moved" 1/16 of a pixel each frame; on the 16th frame it moved to the right by one physical display pixel. Is this appropriate behaviour? Consider the situation if the galpixel had in fact been moving with an x-direction velocity of *minus* one-sixteenth of a pixel per frame. On the first frame, it would have *decremented* its fractional position—initially zero—and "reverse clocked" back to a count of 15, simultaneously moving to the *left* by one physical display pixel. But this is crazy—if it takes 16 frames to move one pixel right, why does it only take one frame to move one pixel left, if it is supposed to be moving at the *same speed* (*viz.* 1/16 pixels per frame)? Clearly, we have been careless about our arithmetic: we have been *truncating* the fractional part off when deciding where to put the pixel on the physical display; we should have been *rounding* the fraction off. Implementing this repair, then, we deem that, if a fractional position is greater than or equal to one-half, the physical position of the galpixel in the galpixmap matrix is incremented, and the fractional part is decremented by 1.0. (Of course, there is no need to *actually* do any decrementing in the fractional bits—one just proclaims that 1000_2 represents a fraction $-8/16$; 1001_2 represents $-7/16$; and so on, up to 1111_2 representing $-1/16$.) With this repair, a galpixel with a constant speed of 1/16 pixels per frame (and initial fractional position of zero, by definition!) will take 8 frames to move one pixel if moving to the right, and 9 frames (by reverse clocking over $-8/16 \rightarrow -9/16 \equiv +7/16$ if moving to the left—as symmetrical a treatment as we are going to get (or, indeed, care about).

There is one objection, however, that might be raised at this point: we originally constructed our fractional position carefully, with the correct number of bits, so that the galpixel would be able to wait up to 16 frames before having to move one pixel. Why have we now restricted this to only 8 (or 9) frames? The answer is that we haven't, really; the motion still *is* at a speed of 1/16 pixels per frame. To see this, one need only continue the motion on for a longer time period: the displayed pixel "jumps" at frames $8, 24, 40, \ldots$, and so on. It is only in order to assure a left–right (and up–down) symmetric treatment of the motion that the *first* eight or nine frames seem strange, at first sight. Looked at another way, consider *successive* display processor updates, spaced 16 frames apart, for a galpixel that is, in fact, moving at the constant velocity of 1/16 frames per second. On the first update, the galpixel is at $x = 0$ (say); on frame number 8 it moves one physical pixel to the right, to $x = 1$. For the remaining 7 extrapolated frames of the first update period, it doesn't move; at the end of this time, its fractional position is $-1/16$, with respect to its physical position of $x = 1$. Then the second update comes. The galpixel is now painted in at $x = 1$, with a fractional position of zero—exactly as it would have been if the previous frame has simply been propagated. *This* galpixel now waits 8 frames before moving to $x = 2$. Does this mean that it is actually moving at a rate of 1/8 pixels per frame, rather than 1/16 are desired? No—one must add the 7 frames of the *previous* update period, plus the update frame, to these 8 frames—making a total of 16 frames since it had last moved, just as we wanted! Of course, the display processor in some sense "destroys" the previous galpixel when it draws the new one in the update frame, and so in that sense it's not truly the "same" galpixel that accumulates the two halves of the 16-frame waiting period. However, the whole idea of a $G^{(2)}$ display system is precisely to make it *look* as if this is the same galpixel moving along—which, if the true motion is reasonably well approximated by uniform acceleration, and if the inter-update time is not too excessive, will indeed be the case. On the other hand, if the motion is *not* well approximated by uniform acceleration, then it must necessarily be "jerking around" a bit—and now we rely on the *psychological* observation that, in such circumstances, small details are difficult to discern! Of course, if the viewer puts herself in such a position to "take a better look" at the object represented by the galpixel, then, by the very *definition* of "taking a better look" (*viz.* stabilising the apparent motion of the object in one's field of view), the gross jerkiness has been transformed away, and the display is again accurate! It should be now apparent why psychological criteria played such an important rôle in the decisions made in the previous section—they effectively "save our bacon" when things get technically difficult!

We now turn to the question of determining how many bits of accuracy are required for the velocity and acceleration components themselves, for the example of a $N_{\text{prop}} = 16$

system. It might be thought that, since the position information is only accurate to four bits itself, then equation (12) means that any more than four bits in $\boldsymbol{v}_i$ or $\boldsymbol{a}_i$ would be wasted. However, this conclusion would be erroneous: one must take into account equation (13) as well. It is clear that $\boldsymbol{v}_i(t)$ or $\boldsymbol{a}_i$ might well continue to be propagated from frame to frame at a *higher* accuracy than four bits, which would then feed through, indirectly, to equation (12), through the velocity velocity $\boldsymbol{v}_i(t)$.

Let us, therefore, examine this question a little more closely. Consider, now, not the i-th galpixel travelling with constant *velocity*, but, rather, travelling under the effect of a constant *acceleration* $\boldsymbol{a}_i$. Imagine, for simplicity, that the initial velocity of the galpixel, $\boldsymbol{v}_i(0)$, is zero, as is its initial position $\boldsymbol{x}_i(0)$, and that its acceleration $\boldsymbol{a}_i$ is purely in the x-direction: $\boldsymbol{a}_i = (a, 0, 0)$. The x-motion of this galpixel will clearly then be given by

$$x(t) = \frac{1}{2}at^2. \tag{14}$$

Now consider how far this galpixel travels from the time $t = 0$ to the time $t = N_{\text{prop}} = 16$ frame periods: this distance is clearly $a(16)^2/2 = 128a$ pixels. If we want the minimum specifiable distance travelled after $t = N_{\text{prop}} = 16$ frame periods to be 1 pixel, we therefore require a *minimum specifiable acceleration* of 1/128 pixels per frame per frame, or, in other words, we require *seven fractional bits* for the acceleration, not four. In the general case, of arbitrary N_{prop}, the number of bits required for the acceleration is clearly $(2\log_2 N_{\text{prop}} - 1)$; the factor of 2 arises from the fact that the time is *squared* in equation (14) (since $\log a^2 \equiv 2\log a$), and the subtraction of 1 from the result arises from the fact that we divide the acceleration by 1/2 in equation (14) (since $\log_2(a/2) \equiv \log_2(a) - 1$). Thus, for example, for $N_{\text{prop}} = 8$ we would require 5 fractional bits for the acceleration; for $N_{\text{prop}} = 32$ we would require 9 bits; and so on.

What, then, does this requirement of 7 bits for the fractional part of each component of $\boldsymbol{a}_i$ (for our example of $N_{\text{prop}} = 16$) mean for the required accuracy of the *velocity* vector, $\boldsymbol{v}_i$? Clearly, (12) cannot carry more than four bits of information, since the position is only stored from frame to frame with this accuracy. The responsibility for propagating the information from the full 7 fractional bits of the acceleration *must* therefore be carried by equation (13). Thus, *the velocity also needs to have 7 bits of fractional accuracy*—or, in the general case, $\boldsymbol{v}_i$ must have the same number of bits as $\boldsymbol{a}_i$, namely, $(2\log_2 N_{\text{prop}} - 1)$.

We must now consider the problem of how we should add together the various differing-accuracy numbers in (12): $\boldsymbol{x}_i$ has 4 fractional bits, $\boldsymbol{v}_i$ has 7, and $\boldsymbol{a}_i/2$ has 8 (once we have multiplied it by the half, *i.e.*, shifted it right one bit). The naïve thing to do would be to simply compute it at the accuracy of $\boldsymbol{x}_i$—or 4 fractional bits in our example. However, this would *unnecessarily* throw away information that is already contained in the last three bits $\boldsymbol{v}_i$, and the last four bits of $\boldsymbol{a}_i/2$. The *correct* procedure is to add $\boldsymbol{v}_i$ and $\boldsymbol{a}_i/2$ together with *a full 8 bits* of accuracy; then to *round* this number off to the nearest 4-bit-fraction number. (This *rounding* can be effected in the same step by simply adding the number 0.00001000_2 to the sum of $\boldsymbol{v}_i$ and $\boldsymbol{a}_i/2$, and then *truncating* the result to four bits.) This four-bit number should then be added to the current fractional position, as in (12). In this way we utilise the information stored in the $G^{(2)}$ frame buffer optimally.

We now turn the question of the accuracy required for the z-buffer position, velocity and acceleration information. Clearly, there is no advantage to thinking in terms of "fractional" bits in the z direction *per se*—because the visual information is not matricised in that direction anyway. Rather, one must simply allocate a sufficient number of bits for the z-buffer to ensure that the finest movement in this direction that the application software requires can be accurately *propagated* over the rated propagation time N_{prop}. It should be noted that, in general, this will require a *greater* number of bits for the z-buffer than for the same system without Galilean Antialiasing, for the same reasons as applied to the use of fractional positional data in the x and y directions.

An interesting problem arises when an existing hardware z-buffer is in place which, for Galilean Antialiasing purposes, is not of a sufficiently high number of bits to meet the design specifications for the applications intended for use. In such a case, it *is* useful to think of adding "fractional bits" to the z-buffer; these extra bits are then stored in a new *physical* memory device, but in all respects are *logically* appended to the trailing end of the corresponding z-buffer values already implemented in hardware. The controlling software may then choose to either compute z values to the full accuracy of this extended z-buffer; or, for backwards compatibility with older applications, may choose to simply specify only integral z-buffer values, using the fractional bits purely to ensure rated performance under Galilean propagation.

We now turn to the question of how many *integral* bits are required for the velocity $\boldsymbol{v}_i$ and acceleration $\boldsymbol{a}_i$ of the i-th galpixel. This question is less straightforward than determining the number of fractional bits, as above, and to some extent depends on the experience and opinions of the designer of the Virtual Reality system. Clearly, the maximum velocity portrayable on the display device is equal to one display width or height in the period of one frame. Any higher velocity than this and the galpixel in question either wasn't visible on the previous frame, or else won't be on the subsequent frame. Since practical Virtual Reality displays are currently limited, in rough terms, to a maximum resolution of around 1000×1000 at best, this suggests that no more than 10 integral bits need be stored for each Cartesian component of $\boldsymbol{v}_i$ (since $2^{10} \approx 1000$). This estimate, however, is too generous, because the human visual system cannot even *see* an image that appears on a display for only a single frame, let alone recognise it. More useful would be to consider as a limiting velocity a traversal of the entire display *over a period of* N_{prop} *frames*. For $N_{\text{prop}} = 16$, this suggests that about six integral bits for each component of velocity may be sufficient to portray the maximum visualisable apparent velocity; in the general case of a display of linear dimension D pixels, this estimated number of integral bits is simply given by the formula $(\log_2 D - \log_2 N_{\text{prop}})$.

One must, however, also take in account the *acceleration*, $\boldsymbol{a}_i$, in these considerations, as may be seen in the following example: Imagine that there is a virtual projectile being displayed that is shot up from the bottom of the display, rises *just* to the top of the display under the simulated effect of gravitation, and then falls back to the bottom of the display. If we assume that the most rapid motion of this sort visualisable by the viewer should also take place over a time interval of N_{prop} frames, we can compute the corresponding maximum velocity and acceleration of the galpixel exactly. Using Galileo's equations of motion $y_i(t) = y_i(0) + v_i^y(0) + a_i^y t^2/2$ and $v_i^y(t) = v_i^y(0) + a_i^y t$, the above motion can be described by the constraints $y_i(0) = 0$ (say), $y_i(N_{\text{prop}}/2) = D$, and $v_i^y(N_{\text{prop}}/2) = 0$. The first constraint tells us that $y_i(0) = 0$ (obviously); the second that

$$D = v_i^y(0)\frac{N_{\text{prop}}}{2} + \frac{1}{2}a_i^y\left(\frac{N_{\text{prop}}}{2}\right)^2,$$

or, or rearranging,

$$a_i^y N_{\text{prop}}^2 + 4v_i^y(0)N_{\text{prop}} - 8D = 0; \tag{15}$$

and the third constraint tells us that

$$0 = v_i^y(0) + a_i^y\frac{N_{\text{prop}}}{2}. \tag{16}$$

Solving the pair of linear simultaneous equations (15) and (16) for the two unknowns $v_i^y(0)$ and $a_i^y(0)$, we find

$$v_i^y(0) = \frac{4D}{N}$$

and

$$a_i^y = -\frac{8D}{N^2}.$$

This analysis suggests that the maximum relevant velocity is in fact four times that which we computed for uniform motion, or, in other words, the number of required integral bits for $\boldsymbol{v}_i$ is approximately $(\log_2 D - \log_2 N_{\text{prop}} + 2)$ (since $\log_2(4b) \equiv \log_2(b) + 2)$. Similarly, the number of required integral bits for acceleration will then be $(\log_2 D - 2\log_2 N_{\text{prop}} + 3)$ (since $\log_2 b^2 \equiv 2\log_2 b$ and $\log_2(8b) \equiv \log_2(b) + 3)$.

We can now start to plug in some typical, real-life figures for D and N_{prop} to get a feel for how many bits, *in total*, we shall require for each of the x and y components of each galpixel's motion. For $D = 1024$ and $N_{\text{prop}} = 16$, we require 4 fractional position bits, 7 fractional velocity bits, 7 fractional acceleration bits, 8 integral velocity bits, and 5 integral acceleration bits: total, 31 bits. Thus, we can fit each of the x and y components of motional information for a given galpixel into less than four bytes (*i.e.*, eight bytes in total, for x and y combined; z motion information, which we have not yet considered in this respect, must also be added). If we take $D = 512$ to be closer to the mark of the resolution of our device, we save two bits for each Cartesian component; or, if $D = 256$ is roughly the resolution, we can save another two bits, bringing the total to 27 bits for each of the x and y components. On the other hand, *increasing* N_{prop} seems, according to our formulæ, to *decrease* the overall number of bits required. This does not seem right: after all, to propagate further forwards in time, do we not require greater accuracy, not less? The reason we have obtained this misleading result is that we have assumed N_{prop} to be the *minimum recognisability time* of the human visual system, as well as the rated propagation time, without any good reason for assuming these two quantities to be equal (other than the fact that, numerically, they seemed to be so for the example we chose). Clearly, a Virtual Reality system designer will want to consider these two parameters independently; we should replace N_{prop} where used above in its recognisability rôle by a new (psychological) parameter, the *recognisability time* N_{recog}, being the smallest time interval for which an object must be shown for it to be registered by the conscious brain (subliminal virtual advertising being ignored, for the moment—although this tactic might be worthwhile when funding agency bureaucrats come around "for a play"). The *total* number of bits required for *each* of the x and y components of positional information is then given by

$$N_{\text{bits}} = 2\log_2 D + 5\log_2 N_{\text{prop}} - 3\log_2 N_{\text{recog}} + 3.$$

Thus, the rules of thumb are the following: doubling the display resolution in each direction adds two bits to each component; doubling the propagation time adds five bits to each; *halving* the recognition time adds three bits to each.

We have not, so far, considered the z buffer motional information. As already noted, the z buffer comes in for different treatment, because it is not matricised or displayed as the x and y components are. It *may* prove convenient, from a hardware design point of view, to simply use the *same* motional structure for the z direction as is used for the x and y directions. However, in practice, if memory constraints are a concern, one can allocate fewer bits to the z buffer motional data. For example, for the 31-bit-per-component example given above (where $D = 1000$ and $N_{\text{prop}} = N_{\text{recog}} = 16$), quite a workable system can be developed using only sixteen bits for the z direction motional information and fractional position, in conjuction with an existing 16-bit z-buffer (thus yielding four bytes for z information in total). It should be noted, however, that *some* sort of z-motion information *must* be stored with the galpixmap, as will become clear shortly.

Thus, we find that, as a very rough estimate, we shall need about 12 bytes to store the motional and z-buffer data for each galpixel. This may seem high; but at 320×200 resolution this only amounts to three-quarters of a megabyte; for 512×512 it amounts to three megabytes. This is not a lot of memory even by today's standards; it will seem even less significant as time goes by. The important point to note is that memory demands are *not* a barrier to implementing Galilean Antialiasing immediately.

We have, of course, not yet considered the *colour* or *shading* information that must be stored with each galpixel. Clearly, in time, this will be universally stored as 24-bit RGB colour information, as display devices improve in their capabilities. This is, of course, another three bytes of data per galpixel that must be stored. But the colour–shading question is, in fact, more subtle than this. Consider what occurs as one walks past a wall in real life—which may, say, be represented as a Gouraud-shaded polygon in the virtual world: the light reflected from the wall *changes in intensity* as we view it from successively different angles. Now, in the spirit of Galilean Antialiasing, we should really provide a method for the video controller to keep up this "colour-changing inertia" as time progresses, until the next update arrives, so that our beautifully rendered or textured objects do not suddenly change colours, in a "flashing" manner, whenever a new display update arrives. How, then, should we encode this information? And how do we compute it in the first place? The answer to the latter question is relatively simple, at least in principle: we only need to compute the *time-derivatives* of the primitive-shading algorithm we are applying, and program this information into the display processor as well; colour temporal derivatives may then be generated at scan-conversion time. The former question, however—encoding the information efficiently—is a little more subtle. The naïve approach would be to simply store the instantaneous time derivatives of the red, green and blue components of the colour data for each pixel. However, this naïve approach ignores the fact that, *most* of the time, the change in colour of the object will be solely a change in *intensity*—hue and saturation will stay relatively constant. (Most

Real Reality violations of this statement—such as a red LED changing to green—are due to artificial man-made objects anyway; our visual systems do not respond well to such shifts, evolving as we have under the light from single star.) It is therefore prudent to encode our RGB information in such a way that *intensity* derivatives can be given a relatively generous number of bits (or, indeed, even second-derivative information), whereas the remaining, hue- and saturation-changing pieces of information can be allocated a very small number of bits. On the other hand, this information must be regenerated, at video scan-out speeds, into RGB information that can be added to the current galpixel's colour values; we should not, therefore, harbour any grand plans of implementing this encoding in a terribly clever, but in practice unimplementable, way. A good solution, with hardware in mind, would be to define three new signals, A, B and C, related to the r, g and b signals via the relations

$$\begin{aligned} A &= \frac{1}{3}(r+g+b), \\ B &= \frac{1}{3}(2r-g-b), \\ C &= \frac{1}{3}(r-2g+b). \end{aligned} \tag{17}$$

Clearly, A represents the intensity of the pixel in this coding scheme: a generous number of bits may be allocated to storing dA/dt—and maybe even a few for d^2A/dt^2, if deemed worthy. On the other hand, dB/dt and dC/dt would be encoded with a very small number of bits, allowing gross changes of colour to be reasonably interpolated, but otherwise not caring too much about inter-update hue and saturation shifts. The set of transformations (17), while not fundamentally optimal, is especially amenable to hardware decoding: the reverse transformations may be verified as being

$$\begin{aligned} r &= A+B, \\ g &= A-C, \\ b &= A-B+C, \end{aligned} \tag{18}$$

which, of course, also apply to the time derivatives of these quantities. In this way, we can efficiently encode colour derivative information for storage, while at the same time allowing a hard-wired reconstruction process in the video controller to be possible. Further technical specifications for the optimal number of bits of storage to be relegated to each of the quantities A, B and C will be left for subsequent researchers to determine; a full analysis of the situation requires a careful consideration of the algorithmics employed in the shading process, the capabilities of the particular display device employed, and, most importantly, the nature of the human colour visual system. (The author, being short of 33% of experiential information in the last regard, disqualifies himself from research on this topic, lest he be tempted to encode all hues most efficiently to his own isochromatic line in order to save on RAM requirements.)

We have now outlined, in broad terms, most of the hardware additions and modifications necessary to retrofit an existing Virtual Reality system with a Galilean Antialiasing display system. There are, however, two final, important questions that must be addressed, that are slightly more algorithmic in nature: What happens when two galpixels are propagated to the *same* pixel in the new frame? And what happens if a pixel in the new frame is not occupied by *any* galpixel propagated from the previous frame? There must, of course, be answers supplied to these questions for any Galilean Antialiasing system to work at all; however, the *best* answers depend on both how much "intelligence" can be reliably crammed into a high-speed video controller, as well as the nature of the images are being displayed. In section 4.2, suggested changes to image generation philosophy will subtly shift our viewpoint on this question yet again; however, general considerations, at least as far as Virtual Reality applications are concerned, will remain roughly the same.

We first consider the question of *galpixel clash*: when two galpixels are propagated forward to the same physical display pixel. Clearly, the video controller must know which galpixel should "win" in such a situation. This is the reason that we earlier insisted that hardware z-buffering *must* be employed: without this information, the video controller would be left with a dilemma. *With* z-buffer information, on the other hand, the video controller's task is simple: just employ the standard z-buffering algorithm.

Having dealt so effortlessly with our first question, let us now turn to the second: what happens when there is an unoccupied galpixel in the new frame buffer? It might appear that the video controller cannot do *anything* in such a situation: there simply is not enough information in the galpixmap; whatever *should* now be in view must have been obscured at the time of the last update. While this is indeed true when the unoccupied pixel *does*, in fact, correspond to an "unobscured" object—and will, of course, require some sort of acceptable treatment—it is *not* the only situation in which empty pixels can arise. Firstly, consider the finite nature of our arithmetic: it may well be that a particular galpixel just happens to "slip" onto its neighbour; where there were before two galpixels, there is now only one. This "galpixel fighting" leads to a mild attrition in the number of galpixels as the video controller propagates from frame to frame; this is not a serious problem, but it must nevertheless be kept in mind. Secondly, and more importantly, it must be remembered that we are here rendering true, three-dimensional perspective images—*not* simply $2\frac{1}{2}$-dimensional computer graphics—and must consider what this means for the apparent motion of the objects in the scene. Obviously, motion *towards* or *away from* the observer will lead to a *change in apparent size* of the object in question. If the object is moving *away* from the observer, the object's apparent image gets smaller; in this situation, some galpixels will "fight it off" in a z-buffer duel; visible detail will, of course, be reduced; but all of the pixels within the object's boundaries will *still* remain filled in the new frame. However, if the object is moving *towards* the observer, then it will "expand" in apparent size: "holes" will appear in the new frame, since we only have a constant number of galpixels to fill the ever-increasing number of pixels contained within the boundaries of the object.

How, then, are we to treat this latter case? Clearly, the best idea would be to simply "fill in" the holes, with some smooth interpolation of colour, so that we get some sort of consistent "magnification" of the (admittedly still low-resolution) object that is approaching the observer. But before we implement such a strategy, it is necessary to note that *we must have some way of distinguishing unobscuration and expansion.* Why? Because, in general, we would like to apply a different solution to each of these two cases. Is it not possible apply one general fix-all solution? After all, we have a fundamental deficit of information in the unobscuration case anyway—why not just use the expansion algorithm there too so that at least we have *something* to show? This sounds reasonable, and, indeed, might be the wisest course of attack in highly constrained retrofit situations. However, in general, we can do much better than this, for essentially no extra effort; we shall now outline these procedures.

We shall first take a nod at history, and consider the case of a *wire-frame* Virtual Reality display system. While destined to slip nostalgically into the long-term memories of workers in the field of computer graphics, and will only be able to be seen at all by the year 2001 by watching Stanley Kubrick's masterpiece of the same name, wire-frame graphics nevertheless provides a simple proof-of-concept test-bed for prototyping Galilean Antialiased displays, and is so simple to implement that it is worth including here. In a wire-frame display system, the vast majority of the display is covered by a suitable background colour (black, or dark blue); on top of this background, lines are drawn to represent the edges of objects. That such graphics can look realistic at all is a testament to our edge-detection and pattern-recognition visual senses: quite literally, almost of the visual information in the scene has been removed. This removal of information, however, is what makes unoccupied-pixel regeneration so simple in wire-frame graphics: the best solution is to simply *display the background colour* for such pixels. For sure, a part of a line might disappear if it happens to coincide with one that is closer to the observer in any given frame (although this problem can in fact be removed by using the enhancements of section 4.2); however, we assume that the rendering system is not too sluggish about provide updates anyway: the piece of line will only be AWOL for a short time. Overall, the accurate portrayal of *motion* far outweighs, in psychological terms, any problems to do with disappearing bits of lines.

Let us now turn to the case of *shaded*-image systems. Will the approach taken for the wire-frame display system work now? It is clear that it will not: in general, there is no such thing as a "background" colour: each primitive must be "coloured in" appropriately. Thus, *even for the unobscuration case*, we must come up with some reasonable image to put in the empty space: painting it with black or dark blue would, in most situations, look simply shocking. The solution that we propose is the following: *if a portion of the display is suddenly unobscured, just leave whatever image was there last frame*. Why should we do this? Well, for starters, we have nothing better to paint there. Secondly, if the display update is not too long in coming, this trick will tend to mimic CRT phosphor persistence: it looks as if that part of the image just took a little bit longer to decay away. This "no-change" approach thus fools the viewer into thinking nothing too bad as happened, since the fact that that part of the display "does not look up-to-date" will not truly register, consciously, for (say) five or ten frames anyway. Thirdly, and most revealingly, *this approach yields no worse a result than is already true for conventional sample-and-hold $G^{(0)}$ display devices!* If a participant can be convinced of the reality of the virtual world with a lousy $G^{(0)}$ display, she can *surely* be convinced of it when only an occasional, small piece of world defaults back to $G^{(0)}$ technology, with the overwhelming majority of the virtual world charging ahead with full $G^{(2)}$ persuasion!

Of course, this explanation is a tad more mischievous than it appears: mixing $G^{(0)}$ and $G^{(2)}$ algorithms together does *not quite* have the same effect as either algorithm on its own. There is a slight "edge-matching" inconsistency when they are used cojointly: an object propagates away, under the $G^{(2)}$ algorithm—yet a part of it may simultaneously be "left behind" on the display, under the $G^{(0)}$ "no-pixel" fallback provisions! Fortunately, however, the nature of the human visual system makes this seeming inconsistency quite mild: the object simply appears to be moving, even though bits of it keep getting left behind. That this, in fact, works quite well in pratice can be verified by observing the Mouse Trails that Microsoft Windows 3.1 can provide for laptop displays: they look a little funky, but nevertheless one does *not* become psychologically confused when seeing dozens of little mouse pointers left behind. Over longer time scales, but with lower pixel-per-frame apparent velocities, the effect resembles to some extent that of the NCC–1701 upon its violation of Einsteinian relativity. At these time scales, the effect is becoming noticeable, but, again, is not overwhelmingly disturbing. In any case, the author has not been able to think of any better way (apart from the wholesale changes outlined in section 4.2) to deal with the unobscuration problem.

We now consider in more detail how this mixing of $G^{(0)}$ and $G^{(2)}$ methodologies can be carried out in practice. The general idea is as follows: Consider the video controller's task as it updates a "new" frame buffer from an "old" frame buffer. The first task it does is to copy across the *static colour* information of each galpixel to the *same* position in the new frame buffer as it occupied in the old. This is the $G^{(0)}$ methodology at work. The velocities, accelerations, fractional positions, *etc.*, of these copied-across pixels are all zeroed, just as if they were inhabitants of a standard $G^{(0)}$ display. These copied-across pixels will be termed *debris*. Each pixel of debris must be further subjected to an additional operation: it must be stamped with a *debris indicator* to denote its status. The debris indicator may be a special z-buffer value reserved specifically for this purpose, say (although this will cause problems with the methods of section 4.2, and should probably be avoided), or it may instead be a *special value* for one of the lesser-used colour derivatives (such as dC/dt) that is reserved specifically for this purpose, and not considered to be a valid physical value for that derivative by the operating software. If, on the other hand, the Virtual Reality system designer is prepared to dedicate a *whole* bit of memory to this purpose, for each galpixel of each frame buffer, the debris indicator will then have its own exclusive *debris flag*; such an approach uses more memory, but it may be more efficient, in terms of scanning speed, if a simple flag must be tested, rather than an equality test being performed between a multi-bit number and an (admittedly hard-wired) debris code.

The next task for the video controller is to simply perform the $G^{(2)}$ propagation procedure from the old frame buffer to the new. Fully-fledged card-carrying galpixels *always* replace any debris that they encounter "sleeping in *my* bed"; if they clash with another galpixel, standard z-buffering takes place.

At the end of this two-pass process, the new frame buffer will contain as much true Galilean Antialiased information as possible; the remaining unoccupied galpixels simply let the debris "show through". (Of course, we have not yet considered the surfaces that are "expanding"; this will come shortly.)

The prospective Galilean-Antialiased display designer might, by this point, be worrying about the fact that we have introduced a *two-pass* process into the video controller's propagation procedure: nanoseconds are going to be tight anyway; do we really need to double the procedure just for the sake a copying across debris? Fortunately, the practical procedure need only be a *one*-pass one, when done carefully, as follows: Firstly, the new frame buffer must have all of its debris indicators set to true; nothing else need be cleared. This will be especially easy when the debris indicator is a dedicated set of debris flags that are all located on the same memory chip: flags can be hardware-set *en masse*. Next, the video controller scans through the old frame buffer, galpixel by galpixel, retrieving information from each in turn. Two parallel arms of the video controller's circuitry now come into play. The first proceeds to check to see whether the *debris* from that particular galpixel may be copied across: it checks the corresponding galpixel in the new frame buffer; if it is debris, it overwrites it (because debris is copied across one-to-one anyway, so that the debris that is there must be simply the cleared-buffer message); if it is a non-debris galpixel, it leaves it alone. Simultaneously, the second parallel arm of the video controller circuitry propagates the galpixel according to $G^{(2)}$ principles: it computes (among other things) the new position of the galpixel in the new frame buffer, and checks the corresponding galpixel that is currently there (after checking for out-of-view conditions, of course). If that galpixel is debris, it overwrites it; if it is a non-debris galpixel, it performs its usual z-buffer test to determine if it should be overwritten. Furthermore, there must be circuitry to coordiante between these two parallel processes in the case that they are trying to look at the *same* galpixel in the new frame: clearly, in this case, the old galpixel takes precedence over its debris image; the latter need not proceed further. Finally, in the case that the *original* pixel is itself debris, the galpixel-arm of the parallel circuitry should be disabled: debris may be copied across from frame to frame indefinitely (if so required), but they have lost all of their inertial properties.

As an aside, it is worth noting, at this point, that the prospects for implementing significant parallelism in the video controller circuitry *in general* are reasonably good, if a designer so wishes to—or needs to—proceed in that direction. One approach would be to slice the display up into smaller rectangular portions, and assign dedicated controller circuitry to each of these portions. Of course, such implementations need to correctly treat memory contention issues, since a galpixel in any part of the display can, in general, be propagated to any other part of the display by the next frame. The extent to which this kneecaps the parallel approach depends intimately on the nature of the memory hardware structure that is implemented. Furthermore, the video controller is subject to a rock-solid deadline: the new frame buffer *must* be completely updated before the next tick of the frame clock; this further complicates the design of a reliable parallel system. However, this is a line of research that may prove promising.

We now return to the problem of empty-pixel regeneration. We have, above, outlined the general plan of attack in *unobscuration* situations. But what about *expansion* situations? We should first check to see if using debris solves this problem already. Imagine an object undergoing (apparent) expansion on the display. Just say a pixel near the centre becomes a "hole". What will the debris method do? Well, it will simply fill it in with the *old* central pixel—which looks like a reasonably good match! But now consider a hole that appears near the boundary of the expanding object. The debris method will now fill in whatever was *behind* that position in the previous frame—which could be any other object. Not so good. Now consider an even worse situation: the object is apparently moving towards the viewer *as well as moving transversely*: in the new frame, it occupies a new position, but its own debris is left behind. Again, objects behind the one approaching will "show through" the holes. Not very satisfactory at all.

We must, therefore, devise some way in which the video controller can tell whether a missing pixel is due to unobscuration, of whether it is due to expansion of an object. But this test *must be lightning-fast*: it must essentially be hard-wired to be of any use in the high-speed frame buffer propagation process. Undaunted, however, let us consider, from first principles, how this test might be carried out. Firstly, even in abstract terms, how would one distinguish between unobscuration and expansion anyway? Well, we might look at the *surrounding galpixels*, and see what patterns they form. How does this help? Well, in an *expansion* situation, the surrounding pixels will all lie on a three-dimensional, roughly smooth surface. On the other hand, on the edge of a patch of *unobscured* galpixels, the surrounding pixels will be discontinuous, when looked at in three-space: the object moving in front of the one behind must be at least some *measurable* z-distance in front of the other (or else z-buffering would be screwed up anyway). So, in abstract terms, our task is to examine the surrounding pixels, and see if they all lie on the same surface.

How is this to be done in practice, much less in hardware? To investigate this question, we first note that we are only interested in a *small area* of the surface; in such a situation, one may approximate the surface as *approximately flat*, whether it really *is* flat (*e.g.*, a polygon) or not (*e.g.*, a sphere). (If the surface is not really flat, and only subtends a few pixels' worth on the display, in which area the curvature of the surface is significant, then this argument breaks down; the following algorithm will mistakenly think unobscuration is happening; but, in this situation, such a small rendering of the object is pretty well unrecognisable anyway, and a hole through it is probably not going to bring the world to an end.) Now, we can always approximate a sufficiently flat surface by a *plane*, at least over the section of it that we are interested in. Consider, now, an *arbitrary* surface in three-space, which we assume can be expressed in the form $z = z(x, y)$, *i.e.*, the z component at any point is specified according to some definite single-valued function of x and y. (Our display methodology automatically ensures only single-valued functions occur anyway, due to the hidden-surface removal property of z-buffering.) Now, if that surface is a *plane*, z will (by definition) be simply a linear function of x and y:

$$z(x, y) = \alpha x + \beta y + \gamma, \tag{19}$$

where α, β and γ are simply constants specifying the orientation and position of the plane. Now consider taking the *two-dimensional* gradient of this function $z(x, y)$, namely, for the case of the plane,

$$\nabla z(x, y) = \nabla(\alpha x + \beta y + \gamma) = \alpha \boldsymbol{i} + \beta \boldsymbol{j}, \tag{20}$$

where the two-dimensional gradient ∇ has components $(\partial/\partial x, \partial/\partial y)$, and $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the x and y directions respectively. Now consider taking the divergence of (20) itself, namely

$$\nabla \cdot \nabla z(x, y) \equiv \nabla^2 z(x, y) = \nabla \cdot (\alpha \boldsymbol{i} + \beta \boldsymbol{j}) = 0.$$

This tells us that, for any point on a smooth surface, the function $z = z(x, y)$ satisfies *Laplace's equation*,

$$\nabla^2 z(x, y) = 0. \tag{21}$$

Consider, on the other hand, what happens if we abut two *different* surfaces side-by-side, such as is done when we produce a z-buffered display showing one object partly obscuring another. Assume, for simplicity, that this "join" is along the y-axis of the display. To the right of the y-axis, we have one surface, which can be described by the function $z_R = z_R(x, y)$; to the left of the y-axis, we have a different surface, described by $z_L = z_L(x, y)$. We can describe the function $z = z(x, y)$ for *all* x simultaneously (*i.e.*, on *both* sides of the y-axis) by employing the *Heaviside step function*, $\theta(x)$, which is equal to $+1$ for $x > 0$, and is equal to 0 for $x < 0$ (*i.e.*, it "switches on" only to the right of the y-axis):

$$z(x, y) = \theta(x) z_R(x, y) + \theta(-x) z_L(x, y), \tag{22}$$

where the $\theta(-x)$ term similarly "switches on" the left-hand surface to the left of the y axis. Let us now proceed to take the Laplacian of (22), step by step, as was done in proceeding from (19) to (20) to (21). Taking the two-dimensional gradient of (22), we have

$$\begin{aligned}\nabla z(x, y) &\equiv \left\{ \boldsymbol{i}\frac{\partial}{\partial x} + \boldsymbol{j}\frac{\partial}{\partial y} \right\} z(x, y) \\ &= \boldsymbol{i} \left\{ \delta(x) \left[z_R(x, y) - z_L(x, y) \right] + \theta(x) \frac{\partial z_R(x, y)}{\partial x} + \theta(-x) \frac{\partial z_L(x, y)}{\partial x} \right\} \\ &\quad + \boldsymbol{j} \left\{ \theta(x) \frac{\partial z_R(x, y)}{\partial y} + \theta(-x) \frac{\partial z_L(x, y)}{\partial y} \right\} \\ &\equiv \theta(x) \nabla z_R(x, y) + \theta(-x) \nabla z_L(x, y) + \boldsymbol{i}\delta(x) \left[z_R(x, y) - z_L(x, y) \right]\end{aligned} \tag{23}$$

where we have used the product rule for differentiation, and where $\delta(x) \equiv d\theta(x)/dx$—the derivative of the Heaviside step function—is the *Dirac delta function*, which is an infinitely tall, infinitely thin spike at the origin, such that the area under the curve is equal to one. We now need to take the divergence of (23) itself. (Readers who are by this stage nauseated by the mathematics should persevere, if only skimmingly; things will be simplified again shortly.) Performing this divergence, we have

$$\begin{aligned}\nabla \cdot \nabla z(x, y) \equiv \nabla^2 z(x, y) &\equiv \left\{ \boldsymbol{i}\frac{\partial}{\partial x} + \boldsymbol{j}\frac{\partial}{\partial y} \right\} \cdot \nabla z(x, y) \\ &= \delta'(x) \left[z_R(x, y) - z_L(x, y) \right] + 2\delta(x) \left\{ \frac{\partial z_R(x, y)}{\partial x} - \frac{\partial z_L(x, y)}{\partial x} \right\} \\ &\quad + \theta(x) \nabla^2 z_R(x, y) + \theta(-x) \nabla^2 z_L(x, y),\end{aligned} \tag{24}$$

where $\delta'(x) \equiv d\delta(x)/dx$ is the derivative of the Dirac delta function (which looks like an infinite up-then-down double-spike). Now, if each of the surfaces on the right and left are smooth (or, in the case of the closer surface, *would* be smooth if continued on in space), then they will, by (21), satisfy $\nabla^2 z_R(x, y) = 0$ and $\nabla^2 z_L(x, y) = 0$; thus, the last line of equation (24) will vanish. Let us, therefore, consider the remaining terms in (24). Since both $\delta(x)$ and $\delta'(x)$ vanish everywhere except for the line $x = 0$ (*i.e.*, the y-axis), (24) tells us that the expression $\nabla^2 z(x, y)$ *also* vanishes everywhere except the y-axis; *i.e.*, away from the join, each surface separately satisfies equation (21)—this we could have guessed. What about *on and about* the y-axis? Well, in that vicinity, neither $\delta(x)$ nor $\delta'(x)$ vanish; the only way that the whole expression (24) can vanish is if the factors that these functions are respectively multiplied by *both* separately vanish. This would require

$$z_R(0, y) = z_L(0, y) \tag{25}$$

and

$$\frac{\partial z_R}{\partial x}(0, y) = \frac{\partial z_L}{\partial x}(0, y) \tag{26}$$

to both hold true. But (25) says that the two surfaces on either side of the y-axis must be at the *same depth* for the particular y value in question (and thus coincident in three-space); and, furthermore, (26) requires that their *derivatives* be equal across the y-axis. But if we had two surfaces next to each other, with moreover a smooth "matching" of the rate of change of depth across the join, then these two surfaces may just as well be called *one single* surface—they are smoothly joined together. Therefore, *for any two non-smoothly-joined surfaces apparently abutting each other in the display plane*, we conclude that the depth function $z = z(x, y)$ obeys the equation

$$\nabla^2 z(x, y) = \rho(x, y), \tag{27}$$

with some "source" function $\rho(x, y)$ *that is non-zero at the abutment, but zero everywhere else*. Equation (27) is, of course, *Poisson's equation.* It is clear that it is an ideal way to determine whether we are "inside" a smooth surface (*i.e.*, expansion is in order) or at the *edge* of two surfaces (which will allow us to detect the edges of unobscuration areas), simply by computing $\nabla^2 z(x, y)$ on the z-buffer information, and checking whether or not the answer is (reasonably close to) zero; we shall call such a test a *Poisson test.*

"OK, then," our recently-nauseated readers ask, "How on earth do we compute Poission's equation, (27), in our video controllers? What sort of complicated, slow, expensive, floating-point chip will we need for *that*?" The answer is, of course, that computing (27) is *especially* easy if we have z-buffer information on a rectangular grid (which is precisely what we *have* got!). Why is this so? Well, considered the Laplacian operator $\nabla^2 \equiv \nabla \cdot \nabla$. Performing this dot-product *before* having either of the ∇ operators act on anything, we have

$$\nabla^2 \equiv \nabla \cdot \nabla \equiv \left\{ \boldsymbol{i} \frac{\partial}{\partial x} + \boldsymbol{j} \frac{\partial}{\partial y} \right\} \cdot \left\{ \boldsymbol{i} \frac{\partial}{\partial x} + \boldsymbol{j} \frac{\partial}{\partial y} \right\} \equiv \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{28}$$

(hence the suggestive notation ∇^2); expressed like this, Poisson's equation (27) becomes

$$\frac{\partial^2 z(x, y)}{\partial x^2} + \frac{\partial^2 z(x, y)}{\partial y^2} = \rho(x, y).$$

Thus, if we can compute both $\partial^2 z/\partial x^2$ and $\partial^2 z/\partial y^2$, then we can perform our Poisson test! How, then, do we compute (say) $\partial^2 z/\partial x^2$? Well, using the information in the display buffer, we cannot do this *exactly*, of course; however, we *can* get quite a good estimate of it by using *finite differences*. To see this, let us first worry about simply computing the *first* derivative, $\partial z/\partial x$. The fundamental definition of this derivative is

$$\left. \frac{\partial z(x, y)}{\partial x} \right|_{x=x_0, y=y_0} \equiv \lim_{\varepsilon \to 0} \frac{z(x_0 + \varepsilon, y_0) - z(x_0, y_0)}{\varepsilon}, \tag{29}$$

where the subscript on the left hand side denotes the fact that we are evaluating the derivative at the point $(x = x_0, y = y_0)$. Now, we cannot get arbitrarily close to $\varepsilon \to 0$, as this definition specifies: we can only go down to $\varepsilon = 1$ pixel (which, if the surfaces are a significant number of pixels in size, *is* really a small distance, despite appearances). However, if we have a *finite* step size ε, as we have, then where would the approximate derivative, as computed via (29), "belong to"? In other words, is it the derivative evaluated at (x_0, y_0); or is it the derivative evaluated at (x_0+1, y_0); or is it something else again? Just as with our earlier considerations of such questions, a "democracy" principle is optimal here: we "split the difference", and say that the computed derivative "belongs" to the (admittedly undisplayable) point $(x_0+1/2, y_0)$. In other words,

$$\left.\frac{\partial z(x,y)}{\partial x}\right|_{x=x_0+1/2,y=y_0} \approx z(x_0+1,y_0) - z(x_0,y_0) \tag{30}$$

is a good estimate of the quantity $\partial z/\partial x$. But we want $\partial^2 z/\partial x^2$, not $\partial z/\partial x$; how do we get this? Well, we simply apply (29) a *second* time, since

$$\frac{\partial^2 z}{\partial x^2} \equiv \frac{\partial}{\partial x}\left\{\frac{\partial z}{\partial x}\right\}. \tag{31}$$

But, so far, we only have $\partial z/\partial x$ evaluated at $(x = x_0 + 1/2, y = y_0)$: we need *two* points to compute a discrete derivative using the approximate formula (30). Well, this is easy to arrange: we just use the points $(x = x_0, y = y_0)$ and $(x = x_0 - 1, y = y_0)$ to compute $\partial z/\partial x$ as evaluated at the position half a pixel to the *left*, $(x = x_0 - 1/2, y = y_0)$; in other words,

$$\left.\frac{\partial z(x,y)}{\partial x}\right|_{x=x_0-1/2,y=y_0} \approx z(x_0,y_0) - z(x_0-1,y_0). \tag{32}$$

We now use (31), with (30) and (32), to get a good estimate of $\partial^2 z/\partial x^2$:

$$\begin{aligned}\left.\frac{\partial^2 z(x,y)}{\partial x^2}\right|_{x=x_0,y=y_0} &\approx [z(x_0+1,y_0) - z(x_0,y_0)] - [z(x_0,y_0) - z(x_0-1,y_0)] \\ &= z(x_0+1,y_0) + z(x_0-1,y_0) - 2z(x_0,y_0).\end{aligned} \tag{33}$$

We now note that, not only has our "split the difference" philosophy brought us back bang on to the pixel $(x = x_0, y = y_0)$ that we wanted, *we can perform this computation using only an adder and by shifting bits*: it can be hard-wired into our video controller, no problems! To the result (33) we must add, of course, the corresponding second derivative in the y-direction, $\partial^2 z/\partial y^2$; this follows the same procedure as used to obtain (33), but now taken adjacent pixels in the y direction. Adding these quantities together, and collecting together terms, we thus have our final, hard-wireable form of the Poisson test:

$$\begin{aligned}\left.\nabla^2 z(x,y)\right|_{x_0,y_0} &\approx z(x_0+1,y_0) + z(x_0-1,y_0) \\ &\quad + z(x_0,y_0+1) + z(x_0,y_0-1) - 4z(x_0,y_0).\end{aligned} \tag{34}$$

We must therefore simply implement this hard-wired test in hardware, and check to see if the result is reasonably close to zero. (Just *how* close is something that must be checked for typical operating conditions against real-life objects that are rendered; no general formula can be given; research, with a modicum of commonsense, should prevail.) If the Poisson test reveals a non-zero "source" at that pixel, $\rho(x_0, y_0)$, then we are probably at the edge of two separated surfaces; conversely, if the test shows a "source" value compatible with zero, then we are, most likely, within the interior of a single surface.

OK, then, we have shown that computing the Poisson test for any pixel is something that we can definitely do in hardware. How do we use this information, in practice, to perform an "expansion" of the image of an object moving towards us? Well, this is an algorithmically straightforward task; it does, however, require some extra memory on the part of the video controller, as well as sufficient speed to be able to perform (yes, it must be admitted) *two* sweeps: once through the old frame buffer (as described above), and once to go through the new frame buffer. The general plan of attack is as follows. The first sweep, through the old frame buffer, follows the procedure outlined above: both debris and propagated galpixels are moved to the new frame buffer. To this first sweep, however, we add yet *another* piece of parallel circuitry (in addition to two circuits already there—that copy debris and propagate galpixels respectively), that computes the Laplacian of z-buffer information according to the approximation (34), as each galpixel is encountered in turn. For each galpixel, it determines whether the Poisson test is satisfied or not, and stores this information in a *Poisson flag map*. This 1-bit-deep matrix simply stores "yes" or "no" information for each galpixel: "yes" if there is a non-zero Poisson source, "no" if the computed source term is compatible with zero. This Poisson flag map is only a modest increase in memory requirements (only 120 kB for even a 1000×1000 display) compared to the memory that we have already allocated for Galilean Antialiasing. If *any* of the five z-buffer values used to test the Poisson equation are *debris* (*i.e.*, debris already left behind on a previous frame propagation), the value "yes" is automatically stored, since the pixel in question cannot then be inside a surface (because debris should never have been left inside the surface last time through). Pixels around the very edge of the display, however, *cannot* be tested; they may simply (arbitrarily) be also flagged as "yes". (The out-of-view buffering method of section 4.2 makes this *edge effect* much less important.)

Once the first sweep has been completed (and not before), a second sweep is made, this time of the *new* frame buffer. At the beginning of this sweep, the new frame buffer consists solely of propagated galpixels and, where they are absent, debris; and the Poisson flag map contains the "yes–no" Poisson test information about every galpixel in the *old* frame buffer. The video controller now scans through the new frame buffer, looking for any pixels marked as debris. When it finds one, it then looks at the *previous* galpixel the new frame buffer (*i.e.*, with a regular left-to-right, top-to-bottom scan, the pixel immediately to the left of the debris). Why? The reasoning is a little roundabout. First, assume that the unoccupied pixel of the new frame buffer *is*, in fact, within the boundaries of a smooth surface. If that *is* true, then the galpixel to its left must be inside the surface too. If we then *reverse-propagate* that galpixel (the one directly to the left) back to where it *originally* came from in the old frame buffer (as, of course, we *can* do—simply reversing the sign of t in the Galilean propagation equation, using the same hardware as before), then we can simply check its Poisson flag map entry to see if, in fact, it *is* within the boundaries of a surface. If it is, then all our assumptions are in fact true; the debris pixel in the new frame buffer is within the surface; we should now invoke our "expansion" algorithm (to be outlined shortly). On the other hand, if the Poisson flag map entry turns out to show that the reverse-propagated galpixel is *not* within the boundaries of a surface, then our original assumption is therefore false: we have (as best as we can determine) a *just-unobscured* pixel; for such a pixel, debris should simply be displayed—but since there already *is* debris there, we need actually do nothing more.

However, we must, in practice, add some "meat" to the above algorithm to make it a little more reliable. What happens when the galpixel to the left was *itself* on the left edge of the surface? In that situation, we *do* want to fill in the empty pixel as an "expansion"—but the Poisson test tells us (rightly, of course) that the pixel to the left is on an edge of something. In an ideal world, we could always "count up" the number of edges we encounter to figure out if we are entering or leaving a given expanding surface. But, even apart from the need to treat special cases in mathematical space, such a technique is not suitable at all in the approximate, noisy environment of a pixel-resolution *numerical* Poisson test. One solution to the problem is to leave such cases as simply not correctly done: holes may appear near the edges of approaching objects; we have, at least, covered the vast majority of the interior of the surface. A better approach, if at least one extra memory fetch cycle from the old frame buffer is possible (and, perhaps thinking wishfully, up to three extra fetch cycles), is to examine not only the galpixel in the new frame buffer to the *left* of the piece of debris found, but also the ones *above*, to the *right*, and *below*. If *any* of these galpixels are in a surface's interior, then the current pixel probably is too.

On the other hand, what is our procedure if *all* of the surrounding galpixels that we have time to fetch are debris? Then it means that, most likely, we are in the interior of a *just-unobscured* region of the new frame buffer, and we should leave the debris in the current pixel there. This is how the algorithm "bootstraps" its way up from purely edge-detection to actually filling in the *interior* of just-unobscured areas. In any case, the video controller, in such a case, would have no information to change the pixel in question anyway—the correct answer is forced on whether we like it or not!

Let us now return to the question of what our "expansion algorithm" should be, for pixels that we decide, by the above method, *should* be "blended" into the surrounding surface. There are many fancy algorithms that one might cook up to "interpolate" between points that have expanded; however, we are faced with the dual problems that we have very little time to perform such feats, and we do not really know for sure (without some complicated arithmetic) just where the new pixel *would* have been in the old frame buffer. Therefore, the simplest thing to do is simply to replicate the galpixel to the left—colour, derivatives, velocity, and all—except that its *fractional position* should be zeroed (to centre it on its new home). If we *did*, perchance, fetch *two* surrounding galpixels—say, the ones to the left and to the right—then a simple improvement is to *average* each component of RGB (which only requires an addition, and a division by half—*i.e.*, a shift right). Finally, if we have really been luxurious, and have fetched *all four* immediately-surrounding pixels, then again we can perform a simple averaging, by using a four-way adder, and shifting two bits to the right. Considering that it is a fairly subtle trick to "expand" surfaces in the first place, and that a display update will soon be needed to fill in more detail for this object approaching the viewer in any case, any of these interpolative shading approaches should be sufficient in practice—even if, from a computer graphics point of view, they are fairly primitive.

Now, the complete explanation above of this "second frame sweep" seems to involve a fair bit of "thinking" on the part of the video controller: it has to look for debris; *if* it finds it, it then has to fetch the galpixel to the left; it then reverse-propagates it back to the old frame buffer; it then fetches its Poission flag map entry; it then decides whether to replicate the galpixel to the left (if in a surface) or simply leave debris (if representing unobscuration);

all of these steps seem to add up to a pretty complicated procedure. What if it had to do this for almost every galpixel in the new frame buffer? Can it get it done in time? The mode of description used above, however, was presented from a "microprocessor", sequential point of view; on the other hand, all of the operations described are really very trivial memory fetch, add, subtract, and compare operations, which can be hard-wired trivially. Thus, *all* of the above steps should be done *in parallel*, for *every* galpixel in the second sweep, *regardless* of whether it is debris or not: the operations are so fast that it will not slow anything down to have all that information ready in essentially one "tick of the clock" anyway. Thus, the only overhead with this process is that one needs to have time to perform both the first and second sweeps—which, it should be noted, *cannot* be done in parallel.

There is, however, a more subtle speed problem that is somewhat hidden in the above description of the Poisson test. This speed problem is (perhaps surprisingly) not in the second sweep at all—but, rather, in the *first*. The problem arises when we are trying to compute the Poisson test, according to (34), for each galpixel: this requires fetching *five* entries of z-buffer information at the same time. "But," the reader asks, "you have described many times procedures that require the simultaneous fetching of several pieces of memory. Why is it only now a problem?" The subtle answer is that, in those other cases, the various pieces of information that needed to be simultaneously fetched were always *different* in nature. For example, to compute the propagation equation (12), we must fetch all three components of motional information simultaneously. The point is that *these pieces of information can quite naturally be placed on different physical RAM chips*—which can, of course, all be accessed at the same time. On the other hand, the five pieces of z-buffer information that we require to compute (34) will most efficiently be stored on the *one* set of RAM chips—which requires that *five separate memory-fetch cycles* be initiated (or, at the very least, something above three fetch cycles, if burst mode is used to retrieve the three value entries on the same row as the pixel being tested). This problem would, if left unchecked, ultimately slow down the whole first-sweep process unacceptably.

The answer to this problem is to have an extra five small memory chips, each with just enough memory to store a *single row* of z-buffer information. (In fact, to store the possible results from equation (34), they need three more bits per pixel than the z-buffer contains, because the overall computation of Poisson's equation has a range $[-4z_{\text{max}}, +4z_{\text{max}}]$, where the z-buffer's minimum and maximum values are assumed to be 0 and z_{max} respectively.) These five long, thin, independent memory structures will be termed *Poisson fingers*. As the video controller sweeps through the old frame buffer, it adds the (single-fetch) z-buffer information of the galpixel to the Poisson finger entry directly below, to the right, to the left, and above the current galpixel's location; and shifts the z-buffer value to the left by 2 bits (multiplying it by four) and subtracts it from the Poisson finger entry right at the current pixel position. Clearly, the Poisson fingers for the rows directly above and below the currently-scanned row are only required to be written to once for each pixel-scanning period. The remaining three Poisson fingers are arranged such that one takes column $0, 3, 6, 9, \ldots$ entries, the next takes columns $1, 4, 7, 10, \ldots$, and the third takes columns $2, 5, 8, 11, \ldots$. Thus, in any one pixel-scanning period, all of these Poisson fingers are being written into at the same time, and only one write to each is required in each such period. It is in this way that the five-way Poisson test bottleneck of equation (34) is avoided using the five Poisson fingers.

It now simply remains to be specified how the Poisson finger information is used. At any given pixel being processed during the first sweep, the information necessary to finish the computation of the Poisson test for the galpixel *directly above* the current pixel becomes available. In our previous description, we simply said that this Poisson finger is *written* to, like the other three surrounding pixels; in fact, all that is necessary is that it be *read* (and not rewritten), added to this final piece of z-buffer information, and passed through to the Poisson test checker (the circuitry that determines whether the source term in the Poisson equation is close enough to zero to be zero or not).

At the end of scanning each row in the first sweep, the video controller circuitry must multiplex the contents of the three centre-row Poisson fingers back into a single logical finger, and "scroll" them up to the top finger; likewise, the finger below must be demultiplexed into three sets, to be placed in the three central-row fingers; and the finger below must in turn be zeroed, ready to start a new row. These memory-copy operations can be performed in burst mode, and only need be done at the end of each scan-row. Alternatively, for even better performance, *nine* Poisson fingers may be employed, with three each for the row being scanned and the rows directly above and below; the chip-addressing multiplexers can then simply be permuted at the end of each scan-row, so that no bulk copying need take place at all. The only remaining task is then to *zero* the set of fingers corresponding to the row that is becoming the new bottom row; the chosen memory chip should have a line that clears all cells simultaneously. Of course, with the nine-finger option, each chip need only store the information for a third of a row, since it will always be allocated to the same position in the $(0, 1, 2)$ cyclic sequence.

We have now completed our considerations on what hardware additions and modifications are required for a minimal implementation of $G^{(2)}$ Antialiasing, as might be retrofitted to an existing Virtual Reality system. In summary, the hardware described, if implemented in full, will propagate forward in time, from frame to frame, moving images that are rendered by the display processor. If an object appears to expand on the display, the display hardware will, in most situations, detect this fact, and "fill in" the holes so produced. If objects that were previously obscured become unobscured, the system will fall back to conventional, $G^{(0)}$ methods, and will simply leave the previous contents of those areas unchanged, as debris.

These properties of the $G^{(2)}$ display system that we have outlined are achieveable today, on existing systems. Designers of brand new Virtual Reality systems, however, can do even better than this. We invite such designers to glance at the further enhancements suggested in section 4: implementing these additional suggestions promises to result in a display technology that will be truly powerful, and, it might be argued, for the first time in history worthy of a place in the new, exciting field of commercial Virtual Reality.

3.2. Software Modifications and Additions

In the previous section, the minimal additions to hardware necessary to implement Galilean Antialiasing immediately, on an existing Virtual Reality system, were described. In this section, we consider, in a briefer form, the software changes that are needed to drive such a system. Detailed formulas will, in the interests of space, not be listed; but, in any case, they are simple to derive from first principles, following the guidelines that will be given here.

Our first topic of consideration must be the *display processor*, which has been treated in fairly nebulous way up to now. The display processor must, at the very least, scan-convert all of the appropriately clipped, transformed primitives passed to it from further down the graphical pipeline. In $G^{(0)}$ display systems, this consists of several logical tasks (which may, of course, be merged together in practice), which, broadly speaking, are the following: determine the rasterised boundary of the primitive; scan through the pixels in the interior of the boundary (except in wire-frame systems); compute z-buffer information for each interior pixel, using its value to determine visibility; compute appropriate shading information for each visible interior pixel. A Galilean Antialiased system adds two more tasks to this procedure (which may, however, be carried out in parallel with the others): interpolation of velocity and acceleration information across the filled primitive; and a computation of colour-information time derivatives.

The first additional task, that of interpolating velocity and acceleration information, is particularly simple to implement if the logical display device is a planar viewport on the virtual world, and if all of the primitives to be rendered are planar (polygons, lines, *etc.*). In such systems, the linear nature of the perspective transformation means that lines in physical space correspond to lines in display space, and, as a consequence, planar polygons in physical space correspond to planar polygons in display space. But if we take successive time derivatives of these statements, we find that the velocities, accelerations, *etc.*, of the various points on a line, or the surface of a polygon, are simply *linear interpolations* of the velocities of the vertices of the line or polygon in question. Thus, the same algorithms that are used to linearly interpolate z-buffer values can be used for the velocity and acceleration data also; we simply need a few more bits of hardware to carry this task out.

Computing the time derivatives of the colour data, on the other hand, is a little more complicated. In general, one must simply write down the complete set of fundamental equations for the illumation model and shading model of choice, starting from the fundamental physical inputs of the virtual world (such as the position, angle, brightness, *etc.* of each light source; the direction of surface normals of the polygons and the positions of their vertices; and so on), and proceeding right through to the ultimate equations that yield the red, green and blue colour components that are to be applied to each single pixel. One must then carefully take the time-derivative of this complete set of equations, to compute the first derivative of the red, green and blue components in terms of the fundamental physical quantities (such as light source positions, *etc.*), and their derivatives. This is easier said than done; the resulting equations can look very nasty indeed. So far, however, the described procedure has been purely mathematical: no brains at all needed there, just a sufficiently patient computer (whether human or machine) to compute derivatives. The real intelligence comes in distilling, from this jungle of equations, those relations that have the most effect, *psychologically*, on the perceived temporal smoothness of the colour information presented in a real-life Virtual Reality session. There are innumerable questions that need to be investigated in this regard; for example, if our shading model interpolates intensities across a surface, should we take the time derivative of the interpolation, or should we interpolate the time derivatives? The former approach will clearly lead to a greater *consistency* between updates, but the latter is almost certainly easier to compute in real life. Such questions can only be answered with a sufficient amount of focused, quality research. The author will not illustrate his ignorance of this research any further, but will rather leave the field to the experts.

A more subtle problem arises when we consider *optically sophisticated models*. By this term, we mean the use of a rendering model that simulates, to a greater or lesser extent, some of the more subtle aspects of optics in real life: the use of shadows; the portrayal of transparent and translucent objects; the use of ray-tracing. While these techniques are (for want of gigaflops) only modelled crudely, if at all, in current Virtual Reality systems, there is nevertheless a more or less unstated feeling that these things will be catered for in the future, when processing power increases. The author, however, has serious doubts about this goal. For sure, spectacular simulations of the real world are vitally important proof-of-concept tools in conventional computer graphics; and there are a large number of potentially powerful applications that can make excellent use of such capabilities. These applications—and, indeed, new ones—will also be, in time, feasible targets for the Virtual Reality industry: simulating the real world will be a powerful domestic and commercial tool. But Virtual Reality will be a poor field indeed if it is limited to simply simluating Real Reality with ever-increasing levels of realism. Rather, it will be the imaginative and efficient programmes of *virtual world design* that will be the true cornerstone of the Virtual Reality industry in years to come. Do not replicate the mistakes of Pen Computing—otherwise known as How You Can Build a $5000 Electronic Gadget to Simulate a $1 Notepad—no one will be very interested.

To illustrate what a mistake it would be to pursue Real Reality simulation to the exclusion of everything else, it is sufficient to note that even the field of "sexy graphics" (that can produce images of teddy-bears and office interiors practically indistinguishable from the real thing) is usually only equipped to deal with a very small subset of the physical properties of the real world. Consider, as an example, what happens if one shines a green light off a large, flat, reflecting object that is coming towards you. What do you see? Obviously, a green light. What happens if the object is moving towards you very fast? So what; it still looks green. But what if it is moving *really* fast, from a universal point of view? What then? Ah, well, then the real world diverges from computer graphics. In the real world, the object looks blue; on the screen, however, it still looks green.

"But that's cheating," a computer graphics connoisseur argues, "Who cares about the Doppler effect for light in real life?" Excluding the fact that thousands of cosmologists would suddenly jump up and down shouting "Me! Me!", just imagine that we *do* want to worry about it: what then? Well, perhaps in the days of punch cards and computational High Priests, the answer would have been, "Too bad. Stop mucking around. We only provide Real Reality simulations here." But this attitude will not be tolerated for a minute in today's "computationally liberated" world.

Of course, the algorithms for ray-tracing may be modified, quite trivially, to take the Doppler effect into account. But what if we now wanted to look at a lensed image of a distant quasar (still a real-world situation here, not a virtual one); what then? Ah, well, yes, well we'd have to program General Relativity in as well. Well what about interference effects? Er..., OK, I think I can program in continuous fields. The photoelectric effect? Optical fibres? Umm..., yes, well..., well what on Earth do you want all this junk for anyway?!

The point, of course, is that, on Earth, we don't. But that doesn't mean that people want their brand-new fancy-pants ray-traced Virtual Reality system to be the late-1990s version of a Pen Computer! If someone wants to view a virtual world in the infra-red, or ultra-violet, or from the point of view of high-energy gamma rays for that matter, why stop them? What a Virtual Reality system must do, as best it can, is stimulate our senses in ways that we can comprehend, *but for the sole purpose of being information inputs to our brains.* If we want to simulate the ray-traced, sun-lit, ultra-low-gravity, ultra-low-velocity, ultra-low-temperature world that is our home on the third planet around an average-sized star, two-thirds of the way out of a pretty ordinary-looking galaxy, then that is fine. Such applications will, in many cases, be enormously beneficial (and profitable) to the participants. But it is a very small fragment of what *can*—and will—be done.

So where is all of this meaningless banter leading? The basic point is this: already, with wire-frame graphics, the early Virtual Reality systems were able to give a surprisingly good feeling of "presence" in the virtual world. Flat-shaded polygons make this world a lot "meatier". Interpolative shading, curved surfaces and textures make the virtual world look just that little bit nicer. However, we are rapidly reaching the saturation limit of how much information can be absorbed, via these particular senses, by our minds; making the *visible* virtual world even more "realistic" will not lead, in itself, to much more of an appreciation of the information we are trying to understand with the technology (although history already has places reserved for the many advances in stimulating our *other* senses that will heighten the experience—aural presence being excitingly close to being commercially viable; force and touch being investigated vigorously; smell and taste being in the exploratory phases). When you walk around a (Real Reality) office, for example, do you have to stop and stare in awe because someone turns another light on? Can you still recognise your car if it is parked in the shade? Does the world look hallucinogenically different under fluorescent lighting than under incandescent lighting? The simple fact is that *our brains have evolved to normalise out much of this lighting information as extraneous*; it makes sexy-looking demos, but so do Pen Computers.

It may be argued, by aficionados of the field of optically sophisticated modelling, that the author's opinions on this topic are merely sour grapes: machines can't do it in real-time right now, so I don't want it anyway. This could not be further from the truth: ideas on how these sophisticated approaches might be reconciled with the techniques outlined in this paper are already in the germination stage; but they are of a purely speculative nature, and will remain so until hardware systems capable of generating these effects in practical virtual-world-application situations become widespread. Implementing optically sophisticated techniques in Virtual Reality will, indeed, be an important commericial application for the technology in years to come. But this is all it will be: *an* application, not the whole field. It will not be part of general-purpose Virtual Reality hardware; it will be a special-purpose, but lucrative, niche market. It will be a subdiscipline.

All that remains is for Virtual Reality consumers to decide What on Virtual Earth they want to do with the rest of their new-found power.

4. Advanced Enhancements

The previous section of this paper outlined how a minimal modification of an exisiting Virtual Reality system might be made to incorporate Galilean Antialiasing. In this section, several more advanced topics, not required for such a minimal implementation, are contemplated. Firstly, a slight interlude away from the rigours of Galilean Antialiasing is taken:

in section 4.1, a simple-to-follow review of the fundamental problems facing the designers of *wrap-around head-mounted display systems* is offered; the technical material offered, however, is not in any way new. After this respite, section 4.2 renews the charge on Galilean Antialiasing proper, and suggests further ways in which Virtual Reality systems can be optimised to present visual information to the participant in the most psychologically convincing way that technological constraints will allow.

4.1. Wrap-Around Head-Mounted Display Design

In conventional computing environments, the display device is generally flat and rectangular, or, in light of technological constraints on CRT devices, as close as possible to this geometry as can be attained. The display device is considered to be a "window on the virtual world": it is a planar rectangular viewport embedded in the real world that the viewer can, in effect, "look through", much as one looks out a regular glass window at the scenery (or lack of) outside.

In a similar way, most of the mass-marketed *audio* reproduction equipment in use around the world aims to simply offer the aural equivalent of a window: a number of "speakers" (usually two, at the present time) reproduce the sounds that the author of the audio signal has created; but, with the exception of quadrophonic systems, and the relatively recent Dolby Surround Sound, have not attempted to portray to the listener any sense of *participation*: the listener is (quite literally) simply a part of the *audience*—even if the reproductive qualities of the system are of such a high standard that the listener may well believe, with eyes closed, that they are actually sitting in the audience of a live performer.

Linking these two technologies together, "multimedia" computing—which has only exploded commercially in the past twelve months—heightens the effectiveness of the "window on a virtual world" experience greatly, drawing together our two most informationally-important physical senses into a cohesive whole. High-fidelity television and video, in a similar way, present a "read-only" version of such experiences.

Virtual Reality, on the other hand, has as its primary goal *the removal of the window-frame, the walls, the ceiling, the floor*, that separate the virtual world from the real world. While the prospect of the demise of the virtual window might be slightly saddening to Microsoft Corporation (who will, however, no doubt release Microsoft Worlds 1.0 in due time), it is an extremely exciting prospect for the average man in the street. Ever since Edison's invention of the gramophone just over a century ago, the general public has been accustomed to being "listeners"—then, later, "viewers" (*à la* Paul Hogan's immortal greeting, "G'day viewers")—and finally, in the computer age, "operators"; now, for the first time in history, they have the opportunity of being completely-immersed *participants* in their virtual world, with the ability to mould and shape it to suit their own tastes, their own preferences, their own idiosyncracies.

Attaining this freedom, however, requires that the participant is convinced that they *are*, truly, immersed in the virtual world, and that they have the power to mould it. There are many, many challenging problems for Virtual Reality designers to consider to ensure that this goal is in fact fulfilled—many of them still not resolved satisfactorily, or in some cases not at all. In this paper, however, we are concerned primarily with the visual senses, to which we shall restrict our attention. Already, in sections 2 and 3, we have considered how best to match computer-generated images to our visual motion-detection senses. However, we have, to date, assumed that the “window” philosophy underlying traditional computer graphics is still appropriate. It is this unstated assumption that we shall now investigate more critically.

Traditionally, in computer graphics, one sits about half a metre in front of a rectangular graphics display of some kind. The controlling software generates images that are either inherently 2-dimensional in nature; are “$2\frac{1}{2}$-dimensional” (*i.e.*, emulating three dimensions in some ways, but which do not have all of the degrees of freedom of a true 3-dimensional image); or are true perspective views of 3-dimensional objects. Clearly, the former two cases are not relevant to immersive Virtual Reality discussions, and do not need to be considered further. The last case, however—perspective 3-dimensional viewing—forms the very basis of current Virtual Reality display methodology. By and large, the concepts, methods, algorithms, and vast experience gained in the field of traditional 3-dimensional computer graphics development have been carried across to the field of Virtual Reality unchanged. The reasons for such a strong linkage between these two fields are many—some historical, some simply practical: Sutherland’s pioneering efforts in both; the relative recency of large-scale Virtual Reality development following the emergence of enabling technologies; the many technical problems common to both; the common commercial parentage of many of the pivotal research groups in both; and so on. However, it is clear that, at some point in time, the field of Virtual Reality must eventually be weaned off its overwhelming dependency on the field of Computer Graphics, for the health of both. Of course, there will remain strong bonds between the two—we will rue the day when a Master of Virtual Reality student comes up with the “novel” idea of collaborating with his counterparts in the Computer Graphics department—but, nevertheless, the cord must be cut, and it must be cut soon.

The reader may, by now, be wondered why the author is pushing this point so strongly—after all, aren’t Virtual Reality displays just like traditional computer graphics displays? Well, as has already been shown in sections 2 and 3, this is not the case: new problems require new solutions; and, conversely, old dogs are not always even *interested* in new tricks (as the millions of still-contented DOS users can testify). But more than this, there is, *and always will be*, a fundamental difference between the field of *planar* Computer Graphics, and the subdiscipline of the field of Virtual Reality that will deal with visual displays: namely, *the human visual system is intrinsically curved*: even without moving our heads, each of our eyes can view almost a complete hemisphere of solid angle. Now, for the purposes of traditional computer graphics, this observation is irrelevant: the display device is *itself* flat; thus, images *must* be computed as if projected onto a planar viewing plane. Whether the display itself is large or small, high-resolution or low, it is always a planar “window” for the viewer. But the aims of Virtual Reality are completely different: we wish to *immerse* the participant, as completely as technically possible, in the virtual world. Now our terminology subtly shifts: since we ideally want to cover each of the participant’s eye’s *entire* almost-hemispherical field of view, the *size* and *physical viewing distance* of the display are irrelevant: all we care about are *solid angles*—not “diagonal inches”, not “pixels per inch”, but rather *steradians*, and *pixels per radian*. We are working in a new world; we are subject to new constraints; we have a gaggle of new problems—and, currently, only a handful of old solutions.

“Surely,” a sceptic might ask, “can’t one always cover one’s field of view using a planar display, by placing the eye sufficiently ‘close’ to it, and using optical devices to make the surface focusable?" The theoretical answer to this question is, of course, in the affirmative: since the angular range viewable by the human eye is less than 180 degrees in any direction (a violation of which would require considerable renovations to the human anatomy), it *is*, indeed, always possible to place a plane of finite area in front of the eye such that it covers the entire field of view. However, theoretical answers are not worth their salt in the real world; our ultimate goal in designing a Virtual Reality system is to *maximise the convinceability* of the virtual-world experience, given the hardware and software resources that are available to us.

How well, then, does the planar display configuration (as suggested by our sceptic above) perform in real life? To answer this question at all, quantitatively, requires consideration of the *human* side of the equation: What optical information is registrable by our eyes? The capabilities of the human visual system have, in fact, been investigated in meticulous detail; we may call on this research to give us precise quantitative answers to almost any question that we might wish to ask. It will, however, prove sufficient for our purposes to consider only a terribly simplistic subset of this information—not, it is hoped, offending too greatly those who have made such topics of research their life's work—to get a reasonable "feel" for what we must take into account. In order to do so, however, we shall need to have an accurate way of portraying *solid angles*. Unfortunately, it is intrinsically impossible to represent solid angles without distortion on a flat sheet of paper, much less by describing it in words. Seeing as that we are still far from having "Virtual Reality on every desk", it is also not currently possible to use the technology itself to portray this information. The reader is therefore asked to procure the following pieces of hardware so that a mathematical construction may be carried out: a mathematical compass—preferably one with a device that locks the arms after being set in position; one red and one blue ballpoint pen, that both fit in the compass; a texta (thick-tipped fibre marker), or a whiteboard marker; a ruler, marked in millimetres (or, in the US, down to at least sixteenths of an inch); a simple calculator; a pair of scissors, or a knife; an orange, or a bald tennis ball; a piece of string, long enough to circumnavigate the orange or tennis ball; and a roll of sticky tape. Readers that have had their recessionary research budgets cut so far that this hardware is prohibitively expensive should skip the next few pages.

The first task is to wrap a few turns of sticky tape around the sharp point of the compass's "pivot" arm, slightly extending past the point. This is to avoid puncturing the orange or tennis ball (as appropriate); orange-equipped readers that like orange juice, and do not have any objections to licking their apparatus, may omit this step.

The next task is to verify that the orange or tennis ball is as close to spherical as possible for objects of their type, and suitable for being written on by the ballpoint pens. If this proves to be the case, pick up the piece of string and wrap it around the orange or ball; do not worry if the string does not yet follow a great circle. Place your right thumb on top of the string near where it overlaps itself (but not on top of that point). With the fingers of your left hand, roll the far side of the string so that the string become more taut; let it slip under your right thumb only gradually; but make sure that no parts of the string "grab" the surface of the orange or ball (except where your right thumb is holding it!). After the rolled string passes through a great circle, it will become loose again (or may even roll right off). Without letting go with the right thumb, mark the point on the string where it crosses its other end with the texta. Now put the orange or ball down, cut the string at the mark, and dispose of the part that did not participate in the circumnavigation. Fold the string in half, pulling it taut to align the ends. At the half-way fold, mark it with the texta. Then fold it in half again, and mark the two unmarked folds with the texta. On unravelling the string, there should be three texta marks, indicating the quarter, half and three-quarter points along its length. Now pull the string taut along the ruler and measure its length. This is the circumference of the orange or ball; store its value in the calculator's memory (if it has one), or else write it down: we will use it later.

We now define some geographical names for places on our sphere, by analogy with the surface of the Earth. For orange-equipped readers, the North Pole of the orange will be defined as the point where the stem was attached. For tennis-ball-equipped readers, a mark should be made arbitrarily to denote the North Pole. Mark this pole with a small 'N' with the blue ballpoint pen. Similarly, the small mark 180° from the North Pole on an orange's surface will be termed the South Pole. Tennis-ballers, however, will need to use the piece of string to find this point, as follows: Wind the string around the ball, placing the two ends at the North Pole, and, as before, roll it carefully until it is taut; inspect it from the side to ensure that it appears to "dissect" the ball into equal halves. The half-way texta mark on the string is now over the South Pole; mark this spot on the orange with the blue ballpoint pen.

From this point, it will be assumed, for definiteness, that the object is an orange; possessors of tennis balls can project a mental image of the texture of an orange onto the surface of their ball, if so desired. Place the string around the orange, making sure the North and South Poles are aligned. (If possessors of oranges find, at this point, that the mark on the orange is not at the half-way point on the string, then either mark a new South Pole to agree with the string, or get another orange.) Now find the one-quarter and three-quarter points on the string, and use the texta to make marks on the orange at these two points. Put the string down. Choose one of the two points just marked—the one whose surrounding area is most amenable to being written on. This point shall be called *Singapore* (being a recognisable city, near the Equator, close to the centre of conventional planar maps of the world); write a small 'S' on the orange with the pen next to it. (This mark cannot be confused with the South Pole, since it is not diametrically opposite the North!) The marked point diametrically opposite Singapore will be called *Quito*; it may be labelled, but will not be used much in the following. Next, wind the string around the orange, so that is passes through all of the following points: the two Poles, Singapore and Quito. Use the blue ballpoint pen to trace around the circumference, completing a great circle through these points, taking care that the string is accurately aligned; this circle will be referred to henceforth as the *Central Meridian*. (If the pen does not write, wipe the orange's surface dry, get the ink flowing from the pen by scribbling on some paper, and try again. Two sets of ballpoint pens can make this procedure easier.) Now wrap the string around the orange, through the poles, but roughly 90° *around from* Singapore and Quito; *i.e.*, if viewing Singapore head-on, the string should now look like a circle surrounding the globe. This alignment need not be exact; the most important thing is that the string start and end on the North Pole, and pass over the South Pole. Make a small *ballpoint* mark at the one-quarter and three-quarter positions of the string. Now wind the string around the orange so that it passes over both of these two new points, as well as over both Singapore and Quito. Trace this line onto the orange. This is the Equator.

We are now in a position to start to relate this orangeography to our crude mapping of the human visual system. We shall imagine that the surface of the orange represents the solid angles seen by the viewer's eye, by imagining that the viewer's eye is located at the *centre* of the orange; the orange would (ideally) be a transparent sphere, fixed to the viewer's head, on which we would plot the extent of her view. Firstly, we shall consider the situation when the viewer is looking straight ahead at an object at infinity, with her head facing in the same direction. We shall, in this situation, define the direction of view (*i.e.*, the direction of *foveal view*—the most detailed vision in the centre of our vision) as being in the *Singapore* direction (with the North Pole towards the top of the head). One could imagine two oranges, side by side, one centred on each of the viewer's eyes, with both Singapores in the same direction; this would represent the viewer's direction of foveal view from each eye in this situation. Having defined a direction thus, the oranges should now be considered to be *fixed* to the viewer's head for the remainder of this section.

We now concentrate solely on the *left* eye of the viewer, and the corresponding orange surrounding it. We shall, in the following, be using the calculator to compute lengths on the ruler against which we shall set our compass; readers that will be using solar-powered credit-card-sized models, bearing in large fluorescent letters the words "ACME CORPORATION—FOR ALL YOUR COMPUTER NEEDS", should at this point relocate themselves to a suitably sunny location to avoid catastrophic system shut-downs. The compass, having been set using the ruler to the number spat out by the calculator, will be used to both measure "distances" between points inhabiting the curved surface of the orange, as well as to actually draw circles on the thing.

The first task is to compute how long 0.122 circumferences is. (For example, if the circumference of the orange was 247 mm, punch "$247 \times 0.122 =$" on the calculator, which yields the answer 30.134; ignore fractions of millimetres. Readers using Imperial units will need to convert back and forth between fractions of an inch.) Put the *red* ballpoint pen into the compass, and set its arms, with tips against the ruler, so that the tips are separated by this length of 0.122 circumferences (whatever the calculator gave for that answer). Now centre the *pivot* of the compass on Singapore, and draw a circle on the orange with the red pen (which is often easier said than done, but *is* possible). The solid angle enclosed by this red circle (subtended, as always, at the centre of the orange—where the viewer's eye is assumed to be located) indicates, in rough terms, all of the possible directions that our viewer can "look directly at"; in other words, the muscles of her eye can rotate her eyeball so that any direction in this solid angle is brought into central foveal view.

It will be noted that, all in all, this solid angle of foveal view is not too badly "curved", when looked at in three dimensions. Place a *flat plane* (such as a book) against the orange, so that it touches the orange at Singapore. One could imagine cutting out the peel of the orange around this red circle, and "flattening it" onto the plane of the book without too much trouble; the outer areas would be stretched (or, if dry and brittle, would fracture), but overall the correspondence between the section of peel and the planar surface is not too bad. (Readers possessing multiple oranges, who do not mind going through the above procedure a second time, might actually like to try this peel-cutting and -flattening exercise.) The surface of the plane corresponding to the flattened peel corresponds, roughly, to the maximum (apparent) size that a traditional, desk-mounted graphics screen can use: any larger than this and the user would need to *move her head* to be able to focus on all parts of the screen—a somewhat tiring requirement for everyday computing tasks. Thus, it can be seen that it is *the very structure of our eyes* that allows flat display devices to be so successful: any device subtending a small enough solid angle that all points can be "read" (*i.e.*, viewed in fine detail) without gross movements of one's head cannot have problems of "wrap-around" anyway.

Virtual Reality systems, of course, have completely different goals: the display device is not considered to be a "readable" object, as it is in traditional computing environments—rather, it is meant to be a convincing, *completely immersive* stimulation of our visual senses. In such an environment, *peripheral vision* is of vital importance, for two reasons. Firstly, and most obviously, the convinceability of the Virtual Reality session will suffer if the participant "has blinkers on" (except, of course, for the singular case in which one is trying to give normally-sighted people an idea of what various sight disabilities look like from the sufferer's point of view). Secondly, and quite possibly more importantly, is the fact that, although our peripheral vision is no good for *detailed* work, it is especially good at detecting *motion*. Such feats are not of very much use in traditional computing environments, but are vital for a Virtual Reality participant to (quite literally) "get one's bearings" in the spatially-extended immersive environment. We must therefore get some sort of feeling—using our orange-mapped model—for the range of human peripheral vision, so that we might cater for it satisfactorily in our hardware and software implementations.

The following construction will be a little more complicated than the earlier ones; a "check-list" will be presented at the end of it so that the reader can verify that it has been carried out correctly. Firstly, set the compass tip-separation to a distance (measured, as always, on the ruler) equal to 0.048 circumferences (a relatively small distance). Place the pivot on Singapore, and mark off the position to the *east* (right) of Singapore where the pen arm intersects the Equator. We are now somewhere near the Makassar Strait. Now put the *blue* ballpoint pen into the compass, and set its tip-to-tip distance to 0.2 circumferences; this is quite a large span. Now, *with the pivot on the new point marked in the Makassar Strait*, carefully draw a circle on the orange. The large portion of solid angle enclosed by this circle represents, in rough terms, the range of our peripheral vision. To check that the construction has been carried out correctly, measure the following distances, by placing the compass tips on the two points mentioned, and then measuring the tip-to-tip distance on the ruler: From Singapore to the point where this freshly-drawn circle cuts the Central Meridian (either to the north or south): about 0.18 circumferences; from Singapore to the point where the circle cuts the Equator to the *west*: about 0.16 circumferences; from Singapore to the point where the circle cuts the Equator to the *east*: about 0.23 circumferences. If these are roughly correct, one can, in fact, also check that the *earlier*, foveal construction is correct, by measuring the tip-to-tip distance between the red and blue circles where they cross the Equator and Central Meridian. To the west, this distance should be about 0.04 circumferences; to the east, about 0.13 circumferences; to the north and south, about 0.08 circumferences.

There are several features of the range of our peripheral vision that we can now note. Firstly, it can be seen that our eyes can actually look a little *behind* us—the left eye to the left, and the right eye to the right. (Wrap the string around the orange, through the poles, 90° east of Singapore; the peripheral field of view cuts behind the forward hemisphere.) To verify that this is, indeed, correct, it suffices to stand with one's face against a tall picket fence; with one's nose between the pickets, and looking straight ahead, you can still see things going on on *your* side of the fence! That our field of vision is configured in this way is is not too surprising when it is noted that one's *nose* blocks one's field of view in the other direction (but which is, of course, picked up by the other eye); hence, our eyes and faces have been appropriately angled so that our eyes' fields of vision are put to most use. But this also means that it is *not* possible to project the view from both eyes onto *one single* plane directly in front of our faces. Of course, we would intend to use two planar displays anyway, to provide stereopsis; but this observation is important when it comes to *clipping* objects for display (for which a single planar clip, to cover *both* eyes, is thus impossible).

Secondly, we note that a reasonable approximation is to consider the point near the Makassar Strait to be the "centre" of a circle of field of view for the left eye. (This unproved assertion by the author may be verified as approximately correct by an examination of ocular data.) The field of view extends about 75°–80° in each direction from this point, or about a 150°–160° "span" along any great circle. This is, of course, a little shy of the full 180° that a hemispherical view provides, but not by much (as an examination of the orange reveals). Now imagine that one was to cut the peel of the orange out around the large (blue) circle of peripheral vision, and were then to try and "flatten it" onto a plane. It would be a mess! An almost-hemisphere is something that does not map well to a flat plane.

Let us reconsider, nevertheless, our earlier idea of placing a planar display device in front of each eye (with appropriate optics for focusing). How would such a device perform? Let us ignore, for the moment, the actual size of the device, and merely consider the *density of pixels* in any direction of sight. For this purpose, let us assume that the pixels are laid out on a regular rectangular grid (as is the case for real-life display devices). Let the distance between the eye and the plane (or the effective plane, when employing optics) be R pixel widths (*i.e.*, we are using the pixel width as a unit of distance, not only in the display plane, but also in the orthogonal direction). Let us measure first along the x axis of the display, which we shall take to have its origin at the point on the display pierced by a ray going through the eye and the Makassar Strait point (which may be visualised by placing a plane, representing the display, against the orange at the Makassar Strait). Let us imagine that we have a head-mountable display of resolution (say) 512×512, which we want to cover the entire field of view (with a little wastage in the corners). Taking the maximum field of view span from the Makassar Strait to be about 80°, we can now compute R from simple trigonometry. Drawing a triangle connecting the eye, the Makassar Strait and the extremum pixel on the axis, and insisting that this extrumum pixel be just at the edge of the field of view (*viz.* 80° away), we obtain

$$\tan 80^\circ = \frac{256}{R},$$

where we have noted that half of the display will extend to the other side of the origin, and hence the centre-to-extremum distance in the plane is 256 pixels. Computing the solution to this equation, we have

$$R = \frac{256}{\tan 80^\circ} \approx 45.1 \text{ pixels.}$$

For a planar device of resolution 512×512 to just cover the entire field of view, this eye–plane distance is absolute; Virtual Reality displays, unlike traditional computing environments, by their nature specify *exactly* how far away the display plane must be; this is no longer a freedom of choice. The most important question then arises: how good a resolution does this represent, if we simply look straight ahead? Well, in this direction, we have the approximation $dx \approx R\,d\theta$, where $d\theta$ is the linear angle (in radians) subtended by the pixel of width dx. Since the pixel width dx is, by definition, 1 pixel-width (our unit of distance), we therefore find that

$$d\theta \approx \frac{dx}{R} \approx \frac{1}{45.1} \approx 0.0222 \text{ radians} \approx 1.27^\circ.$$

So how big is a 1.27°-wide pixel? Let us compare this with familiar quantities. The pixel at the centre of the 640×480 VGA display mode, viewed on a standard IBM 8513 monitor from a distance of half a metre, subtends an angle of about 2.2 arc-*minutes*, which is approaching the limit of our foveal vision. Thus, our planar-display wrap-around central pixel looks about the same as a 35×35 square does on a VGA 640×480 display—it is huge! But *why* is it so huge? After all, 512 pixels of a VGA display subtend about 18.6° of arc at the same half-metre viewing distance; stretch this out to cover 160° instead and each pixel should get bigger in each direction by a factor of about $160/18.6 = 8.6$. So why are we getting an answer that is four times worse in linear dimensions (or sixteen times worse in terms of area) than this argument suggests?

The answer comes from considering, as a simple example, the very *edgemost* pixel in the x-direction on the display—the one that is 80° away from the Makassar Strait point, just visible on the edge of our viewer's peripheral vision. How much angle does *this* pixel subtend? An out-of-hat answer would be "the same as the one in the middle"—after all, the rectangular grid is regularly spaced. But this reasoning relies, for its approximate truth, on the fact that *conventional* planar displays only subtend a small total angle anyway. On the contrary, we are now talking about a planar device subtending a *massive* angle; paraxial approximations are rubbish in this environment. Instead, consider the abovementioned relation,

$$x = R\tan\theta,$$

where $R = 45.1$ pixels in our example, and θ is the angle between the pixel's apparent position and the Makassar Strait. Inverting this relationship we have

$$\theta = \arctan\left(\frac{x}{R}\right),$$

and, on differentiation,

$$d\theta = \frac{dx}{\sqrt{x^2 + R^2}}.$$

For $dx = 1$ at $x = 0$ we find $d\theta \approx 1/45.1 \approx 0.0222$ radians, or 1.27°, as before; this only applies at $x = 0$. For $dx = 1$ at $x = 256$, on the other hand, we find that

$$d\theta = \frac{1}{\sqrt{256^2 + 45.1^2}} \approx 0.00385 \text{ radians} \approx 0.22^\circ.$$

Thus, the pixels at the outermost edges of the visible display are subtending a linear angle nearly six times *smaller* than at the centre of the display—or, in other words, it would take about 33 extremity-pixels to subtend the same solid angle as the *central* pixel! But this is crazy! We already know that the peripheral vision, outside the range of foveal tracking (the red circle on the orange), is sensitive to significantly *less* detail that the fovea! Why on earth are we putting such a high pixel density out there? Who ordered that? Sack him!

The reason for this waste of resolution, of course, is that we have tried to stretch a planar device across our field of view. What is perhaps not so obvious is the fact that no amount of technology, no amount of optical trickery, can remove this problem: *it is an inherent flaw in trying to use a planar display as a flat window on the virtual world.* This point is so important that it shall be repeated in a slightly different form: *Any rendering algorithm that assumes a regularly-spaced planar display device will create a central pixel 33 times the size of a peripheral pixel under Virtual Reality conditions.* This is not open for discussion; it is a mathematical fact.

Let us, however, consider whether we might not, ultimately, avoid this problem by simply producing a higher-resolution display. Of course, one can always compensate for the factor of four degradation by increasing the linear resolution of the device in each direction by this factor. However, there is a more disturbing psychological property of the planar display: pixels near the centre of view seem chunkier than those further out; it becomes psychologically preferable to *look askew* at objects, using the eye muscles together with the increased outer resolution to get a better view. This property is worrying, and could, potentially, cause eye-strain. To avoid this side-effect, we would need to have the *entire* display of such a high resolution that even the central pixel (the worst one) is below foveal resolution. We earlier showed that a 512-pixel-wide planar display produces a central pixel angular length of about 1.27°. Foveal resolution, however, is about 1 arc-minute at best—about what one can get from a Super-VGA display at normal viewing distance. To achieve this, we could simply "scale down" our pixel size, providing a greater number of total pixels to maintain the full-field-of-vision coverage; performing this calculation tells us that we would need

$$512 \times \frac{1.27}{1/60} \approx 39000$$

pixels in each direction for this purpose. No problems—just give me a 39000 × 39000 head-mountable display device, and I'll shake your hand (if I can find it, behind the 4.5 GB of display memory necessary for even 24-bit colour, with no z-buffering or Galilean Antialiasing. . .). And all this just to get roughly Super-VGA resolution. . . do you think you have a market?

The clear message from this line of thought is the following: *planar-display rendering has no future in Virtual Reality.* Assumption of a planar view-plane is, in traditional computer graphics applications, convenient: the perspective view of a line is again a line; the perspective view of a polygon is also a polygon. Everything is cosy for algorithm-inventors; procedures can be optimised to a great extent. But its performance is simply unacceptable for wrap-around display devices. We must farewell an old friend.

How, then, are we to proceed? Clearly, designers of current Virtual Reality systems have not been hamstrung by this problem: there must be a good trick or two that makes everything reasonable again, surely? And, of course, there is: one must map (using optics) the rectangular display devices that electronics manufacturers produce in an *intelligent* way onto the solid angle of the human visual field. One must not, however, be fooled into thinking that any sophisticated amount of optics will ever "solve" the problem by itself. The mapping will also transform the *logical* rectangular pixel grid that the *rasterisation software* uses, in such a way that (for example) polygons in physical space will *not* be polygons on the transformed display device. (The only way to have a polygon stay a polygon is to have a planar display, which we have already rejected.)

Let us now consider how we should like to map the physical display device onto the eye's field of vision. Our orange helps us here. All points on the surface within the red circle should be at *maximum solid-angle resolution*, as our foveal vision can point in any direction inside this circle. However, look at the skewed-strip of solid angle that this leaves behind (*i.e.*, those solid angles that are seen in peripheral vision, but not in foveal vision; the area between the red and blue circles): it is not overwhelming. Would there be much advantage in inventing a transformation that left a *lower*-resolution coverage in this out-lying area, to simulate more closely what we can actually perceive? Perhaps; but it is the opinion of the author that simply removing the distortions of planar display viewing should be one's first consideration. Let us therefore simply aim to achieve a *uniform solid-angle resolution in the entire field of view*. Note carefully that this is *not* at all the same as simply viewing a planar display directly that itself has a uniform resolution across its planar surface—as has been convincingly illustrated above. Rather, we must obtain some sort of smooth (and, hopefully, simple) *mapping* of the one to the other, which we will implement physically with optics, and which must be taken account mathematically in the rendering software.

How, then, does one map a planar surface onto a spherical one? Cartographers have, of course, been dealing with this problem for centuries, although usually with regard to the converse: how do you map the surface of the earth onto a flat piece of paper? Of the hundreds of cartographical projections that have been devised over the years, we can restrict our choices immediately, with some simple considerations. Firstly, we want the mapping to be smooth everywhere, out to the limits of the field of view, so that we can implement it with optical devices; thus, "interrupted" projections (*e.g.*, those with slice-marks that allow the continents to be shown with less distortion at the expense of the oceans) can be eliminated immediately. Secondly, we want the projection to be an *equal-area* one: equal areas on the sphere should map to equal areas on the plane. Why? Because this will then mean that *each* pixel on the planar display device will map to the *same* area of solid angle—precisely what we have decided that we want to achieve.

OK, then, the cartographers can provide us with a large choice of uninterrupted, equal-area projections. What's the catch? This seems too easy! The catch is, of course, that while all of the square pixels *will* map to an equal area of solid angle, they will *not* (and, indeed, mathematically *cannot*) all map to square-shapes. Rather, all but a small subset of these pixels will be distorted into diamond-like or rectangular-like shapes (the suffix *-like* being used here because the definition of these quantities on a curved surface is a little complicated; but for small objects like pixels one can always take the surface to be locally flat). Now, if our rendering software were to think that it was still rendering for a *planar* device, this distortion would indeed be real: objects would be seen by the Virtual Reality participant to be warped and twisted, and not really what would be seen in normal perspective vision. However, if we have suitably briefed our rendering software about the transformation, then the image *can* be rendered free of distortion, by simply "undoing" the effect of the mapping. Again we ask: what *is* the catch?

The catch—albeit a more subtle and less severe one now—is that the directional resolution of the device will not be homogeneous or isotropic. For example, if a portion of solid angle is covered by a stretched-out rectangular pixel, the local resolution in the direction of the shorter dimension is higher then that in the longer direction. We have, however, insisted on an equal-area projection; therefore, the "lengths" of the long and short dimensions must multiply together to the same product as any other pixel. This means that the *geometric mean* of the local resolutions in each of these two directions is a constant, independent of where we are in the solid angle almost-hemisphere, *i.e.*, the square-root of {the pixels-per-radian in one direction} times {the pixels-per-radian in the orthogonal direction}, evaluated at *any* angle of our field of view, will be some constant number, that characterises the resolution quality of our display system. This is what an equal-area projection gives us.

OK then, we ask, of all the uninterrupted equal-area projections that the cartographers have devised, is there any one that does *not* stretch shapes of pixels in this way? The cartographer's answer is, of course, no: that is the nature of mapping between surfaces of different intrinsic curvature; you can only get rid of some problems, but not all of them. However, while there is *no* way to obtain a distortion-free projection, there are, in fact, an *infinite* number of ways we could implement simply an equal-area projection. To see that, it is sufficient to consider the coordiantes of the physical planar display device, which we shall call X, Y and Z (where Z-buffering is employed), as functions of the spherical coordiantes r, θ and ϕ. For reasons that will become clear, we define the spherical coordinates in terms of a *physical* (virtual-world) Cartesian three-space set of coordiantes, u, v and w, in a slightly non-standard way, namely

$$\begin{aligned} u &= r\cos\theta\sin\phi, \\ v &= r\sin\theta, \\ w &= r\cos\theta\cos\phi. \end{aligned} \tag{35}$$

(We will specify precisely the meaning of the (u, v, w) coordinate system shortly.) Now, the equal-area criterion states that the area that an infinitesimally small object covers in (X, Y) space should be the *same* as that contained by the solid-angle area of its mapping in (θ, ϕ) space, up to some constant that is independent of position. The solid angle of an infinitesimally small object of extent $(dr, d\theta, d\phi)$ centred on the position (r, θ, ϕ) in spherical coordinates in given by

$$d\Omega = \cos\theta\, d\theta\, d\phi$$

(and is of course independent of r or dr); thus, using the Jacobian of the transformation between (X, Y) and (θ, ϕ), namely,

$$\frac{\partial\left[X(\theta,\phi), Y(\theta,\phi)\right]}{\partial\left[\theta,\phi\right]} \equiv \left| \det \begin{pmatrix} \dfrac{\partial X(\theta,\phi)}{\partial\theta} & \dfrac{\partial X(\theta,\phi)}{\partial\phi} \\ \dfrac{\partial Y(\theta,\phi)}{\partial\theta} & \dfrac{\partial Y(\theta,\phi)}{\partial\phi} \end{pmatrix} \right|$$

(which relates infinitesimal areas in (X, Y) space to their counterparts in (θ, ϕ) space), the equal-area criterion can be written mathematically as

$$\left| \frac{1}{\cos\theta} \left\{ \frac{\partial X(\theta,\phi)}{\partial\theta} \frac{\partial Y(\theta,\phi)}{\partial\phi} - \frac{\partial X(\theta,\phi)}{\partial\phi} \frac{\partial Y(\theta,\phi)}{\partial\theta} \right\} \right| = \text{const.} \tag{36}$$

This is, of course, only *one* relation between the two functions $X(\theta, \phi)$ and $Y(\theta, \phi)$; it is for this reason that we are still free to choose, from an infinite number of projections, the one that we would like to use.

Let us, therefore, consider again the human side of the equation. Our visual systems have evolved in an environment somewhat unrepresentative of the Universe as a whole: gravity pins us to the surface of the planet, and drags everything not otherwise held up downwards; our evolved anatomy requires that we are, most of the time, in the same "upright" position against gravity; our primary sources of illumination (Sun, Moon, planets) were always in the "up" direction. It is therefore not surprising that our visual senses do not interpret the three spatial directions in the same way. In fact, we often tend to view things in a somewhat "$2\frac{1}{2}$-dimensional" way: the two horizontal directions are treated symmetrically, but logically distinct from the vertical direction; we effectively "slice up" the world, in our heads, into horizontal planes.

Consider, furthermore, the motion of our head and eyes. The muscles in our eyes are attached to pull in either the horizontal or vertical directions; of course, two may pull at once, providing a "diagonal" motion of the eyeball, but we most frequently look either up–down *or* left–right, as a rough rule. Furthermore, our necks have been designed to allow easy rotation around the vertical axis (to look around) and an axis parallel to our shoulders (to look up or down); we can also cock our heads of course, but this is less often used; and we can combine all three rotations together, although this may be a bone-creaking (and, according to current medical science, dangerous) pastime.

Thus, our *primary* modes of visual movement are left–right (in which we expect to see a planar-symmetrical world) and up–down (to effectively "scan along" the planes). Although this is a terribly simplified description, it gives us enough information to make at least a reasonable choice of equal-area projection for Virtual Reality use. Consider building up such a mapping pixel-by-pixel. Let us start in the *centre* of our almost-hemispherical solid area of vision (*i.e.*, the Makassar Strait on our orange), which is close to—but not coincident with—the direction of straight-ahead view (Singapore on our orange). Imagine that we place a "central pixel" there, *i.e.*, in the Makassar Strait. (By "placing a pixel" we mean placing the mapping of the corresponding square pixel of the planar display.) In accordance with our horizontal-importance philosophy, let us simply continue to place pixels around the Equator, side by side, so that there is *an equal, undistorted density of pixels* around it. This is what our participant would see if looking directly left or right from the central viewing position; it would seem (at least along that line) nice and regular. (We need only stack them as far as 80° in either direction, of course, but it will be convenient to carry this around a full 90° to cover a full hemisphere, to simplify the description.) On the (X, Y) display device, this corresponds to using the pixels along the X-axis for the Equator, *with equal longitudinal spacing* (which was *not* the case for the flat planar display).

Now let us do the same thing in the *vertical* direction, stacking up a single-pixel wide column starting at the Makassar Strait, and heading towards the North Pole; and similarly towards the South Pole. (Again, we can stop short 10° from either Pole in practice, but consider for the moment continuing right to the Poles.) This corresponds to the pixels along the Y-axis of the physical planar display device, *at equally spaced latitudes*; such equal angular spacing was *also* not true for the planar device. The Equator and Central Meridian have therefore been mapped linearly to the X and Y axes respectively; pixels along these lines are *distortion-free*.

Can we continue to place any more pixels in a distortion-free way? Unfortunately, we cannot; we have used up all of our choices of distortion-free lines. How then do we proceed now? Let us try, at least, to maintain our philosophy of splitting the field of view into *horizontal planes*. Consider going around from Makassar Strait, placing square pixels, as best we can, in a "second row" above the row at the Equator (and, symmetrically, a row below it also). This will not, of course, be 100% successful: the curvature of the surface means there must be gaps, but let us try as best we can. How many pixels will be needed in this row? Well, to compute this roughly, let us approximate the field of view, for the moment, by a complete hemisphere; we can cut off the extra bits later. Placing a second row of pixels on top of the first amounts to traversing the globe at a *constant latitude*, *i.e.*, travelling along a Parallel of latitude. This Parallel is *not* the shortest surface distance between two points, of course; it is rather the intersection between the spherical surface and a *horizontal plane*—precisely the object we are trying to emulate. Now, how long, in terms of surface distance, is this Parallel that our pixels are traversing? Well, some simple solid geometry and trigonometry shows that the length of the (circular) perimeter of the Parallel of latitude θ is simply $C\cos\theta$, where C is the circumference of the sphere. Thus, in some sort of "average" way, the number of pixels we need for the second row of pixels will be $\cos\theta$ times smaller than for the Equator, if we are looking at a full hemisphere of view. This corresponds, on the (X, Y) device, to only extending a distance roughly $\cos\theta$ *shorter* in the X direction for the horizontal line of pixels cutting the Y-axis at the value $Y = 1$ pixel, than was the case for the Equatorial pixels (which mapped to the line $Y = 0$). It is clear that the shape we are filling up on the (X, Y) device is *not* a rectangle; this point will be returned to shortly.

We can now repeat the above procedure again and again, placing a new row of pixels (as best will fit) above and below the Equator at successively polar latitudes; eventually, we reach the Poles, and only need a single pixel on each Pole itself. We have now completely covered the entire forward hemisphere of field of view, with pixels smoothly mapped from a regular physical display, according to our chosen design principles. What is the precise mathematical relationship between (X, Y) and (θ, ϕ)? It is clear that the relations are simply

$$\begin{aligned} X &= \frac{N_{\text{pixels}}}{\pi}\phi\cos\theta, \\ Y &= \frac{N_{\text{pixels}}}{\pi}\theta, \end{aligned} \tag{37}$$

where N_{pixels} is simply the number of pixels the display device possesses in the shorter of its dimensions. The relation for Y follows immediately from our construction: lines parallel to the X-axis simply correspond to Parallels of latitude; they are equally spaced (as we have filled the latitudes with pixels one plane at a time); θ is simply the latitude in spherical coordiantes (which should always be measured in *radians*, not degrees, *except* perhaps when computing a final "quotable" number); and the scaling factor N_{pixels}/π simply ensures that the poles are mapped to the edges of the display device: $Y(\theta = \pm\pi/2) = \pm N_{\text{pixels}}/2$. (To later "slice off" the small part of the hemisphere not visible, we will simply increase the *effective* N_{pixels} of the device by the factor $90/80 = 1.125$, rather than trying to change relations (37) themselves; this practice will maintain conformity between different designers who choose a slightly different maximum field of view—say, 75° rather than 80°.) The relation for X follows by noting that the ϕ-th Meridian cuts the θ-th Parallel a surface distance $R\phi\cos\theta$ from the Central Meridian (where R is the radius of the sphere, and the distance is measured along the Parallel itself); since we have stacked pixels along the Parallel, this surface distance measures X on the display device; and the normalisation constant ensures that, at the Equator, $X(\theta = 0, \phi = \pm\pi/2) = \pm N_{\text{pixels}}/2$.

Now, our method above *should* have produced an equal-area mapping—after all, we built it up by notionally placing display pixels directly on the curved surface! But let us nevertheless verify mathematically that the equal-area criterion, equation (36), *is* indeed satisfied by the transformation equations (37). Clearly, on partial-differentiating equations (37), we obtain $\partial X/\partial\theta = -\gamma\phi\sin\theta$, $\partial X/\partial\phi = \gamma\cos\theta$, $\partial Y/\partial\theta = \gamma$, and $\partial Y/\partial\phi = 0$, where we have defined the convenient constant $\gamma \equiv N_{\text{pixels}}/\pi$. The equal-area criterion, (36), then becomes

$$\left|\frac{1}{\cos\theta}\left\{-\gamma\phi\sin\theta\cdot 0 - \gamma\cos\theta\cdot\gamma\right\}\right| \equiv \gamma^2 = \text{const.},$$

which is, indeed, satisfied. Thus, we have not made any fundamental blunders in our construction.

The transformation (37) is a relatively simple one. The forward hemisphere of solid angle of view is mapped to a portion of the display device whose shape is simply a *pair of back-to-back sinusoids* about the Y-axis, as may be verified by plotting all of the points in the (X, Y) plane corresponding to the edge of the hemisphere, namely, those corresponding to $\phi = \pm 90^\circ$, for all θ. And, as could have been predicted, our cartographer colleagues have used this projection for a long time: it is known as the *Sinusoidal* or *Sanson–Flamsteed projection*. Now, considering that the term "sinusoidal" is used relatively abundantly in Virtual Reality applications, we shall, for sanity, actively avoid it in this particular context, and instead give Messrs Sanson and Flamsteed their due credit whenever discussing this projection.

The astute reader may, by now, have asked the question, "Isn't it silly to just use a sinusoidal swath of our display device—we're wasting more than 36% of the pixels on the display!" (or possibly more, if the physical device is rectangular, rather than square). Such wastage does, indeed, on the surface of it, seem like a bad idea. However, it is necessary to recall that we are here balancing between various evils. We have now, indeed, corrected for the anomalously bad central resolution of a planar device: resolution is constant over all solid angle, rather than bad where we want it and good where we don't; the central pixel *will* now be four times smaller in each direction (or sixteen times smaller in area) than before, as our rough "stretch the display around" estimate suggested should be the case. We are further *insisting* on full coverage of the field of view as a fundamental Virtual Reality design principle; cutting off "edges" of solid angle to squeeze use out of the extra 36% of unused pixels (which requires a slice at roughly 55° in each direction, rather than the full 75°–80° that we want) is likely to cause a greater negative psychological effect than those extra pixels could possibly portray by slightly increasing the linear resolution (only by about 40%, in fact) of the remaining, mutilated solid angle.

There is, however, a more subtle reason why, in fact, not using 36% of the display can be a *good* thing. Consider a consumer electronics manufacturer fabricating small, light, colour LCD displays, for such objects as camcorders. Great advances are being made in this field regularly; it is likely that ever more sophisticated displays will be the norm for some time. Consider what happens when a *single LCD pixel* is faulty upon manufacture (more likely with new, ground-breaking technology): the device is of no further commerical use, because all consumer applications for small displays need the full rectangular area. There is, however, roughly a 36% chance that this faulty pixel falls *outside* the area necessary for a Virtual Reality head-mounted display—and is thus again a viable device! If the electronics manufacturer is simultaneously developing commerical Virtual Reality systems, here is a source of essentially free LCD displays: the overall bottom line of the corporation is improved. Alternatively, other Virtual Reality hardware manufacturers may negotiate a reasonable price with the manufacturer for purchasing these faulty displays; this both cuts costs of the Virtual Reality display hardware (especially when using the newest, most expensive high-resolution display devices—that will most likely have the highest pixel-fault rate), as well as providing income to the manufacturer for devices that would otherwise have been scrapped. Of course, this mutually beneficial arrangement relies on the supply-and-demand fact that the consumer market for small LCD display devices is huge compared to that of the current Virtual Reality industry, and, as such, will not remain so lucrative when Virtual Reality hardware outstrips the camcorder market; nevertheless, it may be a powerful fillip to the Virtual Reality industry in the fledgling form that it is in today.

It is now necessary to consider the full transformation from the physical space of the virtual world, measured in the Cartesian coordinate system (x, y, z), to the space of the physical planar display device, (X, Y, Z) (where Z will be now be used for the Z-buffering of the physical display and its video controller). In fact, the only people who will ever care about the intermediate spherical coordinate system, (r, θ, ϕ), are the designers of the optical lensing systems necessary to bring the light from the planar display into the eye at the appropriate angles. (It should be noted, in passing, that even a *reasonably* accurate physical replication of the mapping (37) in optical devices would be sufficient to convince the viewer of reality; however, considering the time-honoured and highly advanced status of field of optics [Hubble Space Telescope notwithstanding], there is no doubt that the optics will not be a serious problem.)

What, then, is the precise relationship between (X, Y, Z) space and (x, y, z) space? To determine this, we need to have defined conventions for the (physically fixed) (x, y, z) axes themselves in the first place. But axes that are *fixed* in space are not very convenient for *head-mounted* systems; let us, therefore, define a *second* set of Cartesian axes (u, v, w), whose (linear) transformation from the (X, Y, Z) space consists of the translation and rotation from the participant's head position to the fixed coordinate system.

At this point, however, we note a considerable complication: the line of vertical symmetry in the (X, Y) plane has (necessarily) been taken to be through the Makassar Strait direction—which is in a *different* physical direction for each eye. Therefore, let us define, not one, but *two* new intermediate sets of Cartesian axes in (virtual) physical space, (u_L, v_L, w_L) and (u_R, v_R, w_R), with the following properties: (u_L, v_L, w_L) is used for the left eye, (u_R, v_R, w_R) for the right; the origins of these coordinate systems lie at the effective optical centres of the respective eyes of the participant; when the head is vertically fixed, the u_L–v_L plane is normal to the line connecting the centre of the left eye and its Makassar-Strait direction; w_L measures positions in this normal direction, increasing towards the *rear* of the participant (so that all visible w_L values are in fact negative—chosen for historical reasons); u_L measures "horizontally" in the u_L–v_L plane with respect to the viewer's head, increasing to the right; v_L measures "vertically" in the u_L–v_L plane in the same sense, increasing as one moves upwards; and the (u_R, v_R, w_R) axes are defined in exactly the same way but with respect to the right eye's position and Makassar Strait direction.

With these conventions, the (u_L, v_L, w_L) and (u_R, v_R, w_R) coordinate systems are Cartesian systems in (virtual) physical three-space, whilst still being rigidly attached to the participant's head (and, as a consequence, the display devices themselves). We can now use the definitions (35) directly, for each eye separately, namely

$$\begin{aligned} u_e &= r_e \cos\theta_e \sin\phi_e, \\ v_e &= r_e \sin\theta_e, \\ w_e &= r_e \cos\theta_e \cos\phi_e, \end{aligned} \tag{38}$$

where e, the *eye index*, is equal to L or R as appropriate. Similarly, we need an (X, Y, Z) coordinate system for each eye, which will thus be subscripted by $e = L$ or R as appropriate.

There is still, however, the question of deciding what functional form Z_e will take, in terms of the spherical coordinates (r_e, θ_e, ϕ_e). It should be clear that this should only be a function of r_e, so that the Z_e-buffer of the physical display device *does* actually refer to the physical distance from the (virtual) object in question to the e-th eye. It will prove convenient to define this distance *reciprocally*, so that

$$Z_e = \frac{\beta}{r_e},$$

where β is a constant chosen so that the (usually integral) values of the Z_e-buffer are best distributed for the objects in the virtual world in question; for instance, objects closer than about 50 mm in distance cannot be focused anyway, so the maximum Z_e may as well be clamped to this value. This reciprocal definition carries with it the great advantage that it does not break down for objects very far away (although it may, of course, round off sufficiently far distances to $Z = 0$); and it takes account of the fact that a given change in distance is, in fact, more important visually the *closer* that that distance is to the observer. We then have, using (37),

$$\begin{aligned} X_e &= \frac{N_{\text{pixels}}}{\pi}\phi_e \cos\theta_e, \\ Y_e &= \frac{N_{\text{pixels}}}{\pi}\theta_e, \\ Z_e &= \frac{\beta}{r_e}. \end{aligned} \tag{39}$$

We can now use the relations (38) and (39) to eliminate the intermediate spherical coordinates (r_e, θ_e, ϕ_e) completely. Inversion of relations (38) yields

$$\begin{aligned} r_e &= \sqrt{u_e^2 + v_e^2 + w_e^2}, \\ \theta_e &= \arcsin\left(\frac{v_e}{\sqrt{u_e^2 + v_e^2 + w_e^2}}\right), \\ \phi_e &= \arctan\left(\frac{u_e}{w_e}\right). \end{aligned} \tag{40}$$

Inserting these results into (39), and noting the identity

$$\cos[\arcsin(a)] \equiv \sqrt{1-a^2},$$

we thus obtain the desired one-step transformations

$$\begin{aligned} X_e &= \frac{N_{\text{pixels}}}{\pi} \sqrt{\frac{u_e^2 + w_e^2}{u_e^2 + v_e^2 + w_e^2}} \cdot \arctan\left(\frac{u_e}{w_e}\right), \\ Y_e &= \frac{N_{\text{pixels}}}{\pi} \arcsin\left(\frac{v_e}{\sqrt{u_e^2 + v_e^2 + w_e^2}}\right), \\ Z_e &= \frac{\beta}{\sqrt{u_e^2 + v_e^2 + w_e^2}}. \end{aligned} \tag{41}$$

Of course, in practice, there are not as many operations involved here as there appear at first sight: many quantities are used more than once, but only need to be computed once. Nevertheless, this transformation from physical space to display space is much more computationally expensive than is the case for conventional planar computer graphics—an unavoidable price that must be paid, however, if equal rendering time and display resolution are to be devoted to all equivalent portions of solid area in the field of view.

Finally, linking the transformations (41) to the virtual-world *fixed* physical coordinate system, (x, y, z), requires a transformation from the Makassar-centred coordinate system (u_e, v_e, w_e). This transformation is, however, a standard one, consisting of an arbitrary three-displacement coupled with the three Euler angles specifying the rotational orientation of the head; as there is nothing new to be added to this transformation, we shall not go into further explicit and voluminous details here. By carefully taking the temporal derivatives of these transformations, one may obtain the relationships between the *true* physical motional derivatives of objects in virtual three-space, and the corresponding time derivatives of the *apparent* motion—*i.e.*, the derivatives of the motion as described in terms of the physical display device coordianetes, (X_e, Y_e, Z_e); this information is needed for Galilean Antialiasing to be implemented. Again, these formulas may simply be obtained by direct differentiation; we shall not derive them here.

A final concern for the use of the Sanson–Flamsteed projection (or, indeed any other projection) in Virtual Reality is to devise efficient rendering algorithms for everyday primitives such as lines and polygons. Performing such algorithmic optimisation is an artform; the author would not presume to intrude on this intricate field. However, a rough idea of how such non-linear mappings of lines and polygons might be handled is to note that *sufficiently small* sections of such objects can always be reasonably approximated by lines, parabolas, cubics, and so on. A detailed investigation of the most appropriate and efficient approximations, for the various parts of the solid angle mapped by the projection in question (which would, incidentally, become almost as geographically "unique", in the minds of algorithmicists, as places on the real earth), would only need be done once, in the research phase, for a given practical range of display resolutions; rendering algorithms could then be implemented that have this information either hard-wired or hard-coded. It may well be useful to slice long lines and polygons into smaller components, so that each component can be handled to pixel-resolution approximation accurately, yet simply. All in all, the problems of projective, perspective rendering are not insurmountable; they simply require sufficient (preferably non-proprietary-restricted) research and development.

If you thought the science of computer graphics was a little warped before, then you ain't seen nothing yet.

4.2. Local-Update Display Philosophy

Having diverted ourselves for a brief and fruitful (sorry) interlude on head-mounted display devices, we now return to the principal topic of this paper: Galilean Antialiasing, and its implementation in practical systems. We shall, in this final section, critically investigate the *basic philosophy* underlying current Virtual Reality image generation—which was accepted unquestioningly in the minimal implementation described in section 3, but which we must expect to be *itself* fundamentally influenced by the use of Galilean Antialiasing.

Traditionally, perspective 3-dimensional computer graphics has been performed on the "clean slate" principle: one erases the frame buffer, draws the image with whatever sophistication is summonable from the unfathomable depths of ingenuity, and then projects the result onto a physical device, evoking immediate spontaneous applause and calls of "Encore! Encore!". This approach has, to date, been carried across largely unmodified into the Virtual Reality environment, but with the added imperative: get the bloody image out within 100 milliseconds! This is a particularly important yet onerous requirement: if the Virtual Reality participant is continually moving around (the general case), the view is continually changing, and it must be updated regularly if the participant is to function effectively in the virtual world at all.

With Galilean Antialiasing, however, our image generation philosophy may be profitably shifted a few pixels askew. As outlined in section 3, a *Galilean* update of the image on the display gives the video controller sufficient information to move that object reasonably accurately for a certain period of time. This has at least one vitally important software ramification: *the display processor no longer needs to worry about churning out complete images simply to simulate the effect of motion*; to a greater or lesser extent, objects will "move themselves" around on the display, "unsupervised" by the display processor. This suggests that the whole philosophy of the image generation procedure be subtly changed, but changed all the way to its roots: *Only objects whose self-propagating displayed images are significantly out-of-date should be updated.* Put another way, we can now organise the image generation procedure in the same way that we (usually!) organise our own lives: urgent tasks should be done NOW; important tasks should be done *soon*; to-do-list tasks should be picked off when other things aren't so hectic.

How, then, would one go about implementing such a philosophy, if one were building a brand-new Virtual Reality system from the ground up? Firstly, the display-processor–video-controller interface should be designed so that updates of only *portions* of the display can be cleanly *grafted* onto the existing self-propagating image; in other words, *local updates* must be supported. Secondly, this interface between the display processor and the video controller—and, indeed, the whole software side of the image-generation process—must have a reliable method of ensuring *timing and synchronisation.* Thirdly, the display processor must be redesigned for *reliable rendering* of local-update views: the laxity of the conventional "clean the slate, draw the lot" computer graphics philosophy must be weeded out. Fourthly, it would be most useful if updates for *simple motions of the participant herself* could be catered for automatically, by specialised additions to the video controller hardware, so that entire views need not be regenerated simply because the participant has started jerking around a bit. Fifthly, some sort of *caching* should be employed on the Galilean pixelated display image, to further reduce strain on the process of generating fresh images. Finally, and ultimately most challengingly, the core Virtual Reality operating system, and the applications it supports, must be structured to fully exploit this new hardware philosophy maximally.

Let us deal with these aspects in turn. We shall not, in the following discussion, proscribe solutions to these problems in excessive technical detail, as performing this task optimally can only be done by the Virtual Reality designer of each particular implementation.

First on our list of requirements is that *local updates* of the display be possible. To illustrate the general problem most clearly, imagine the following scenario: A Virtual Reality participant is walking down the footpath of a virtual street, past virtual buildings, looking down the alley-ways between them as she goes. Imagine that she is walking down the left-hand footpath of the street, *i.e.*, on the same side of the road as a car would (in civilised countries, at least). Now imagine that she is passing the front door of a large, red-brick building. She does not enter, however; rather, she continues to stroll past. As she approaches the edge of the building's facade, she starts to turn her head to the left, in anticipation of looking down the alley-way next to the building. At the precise instant the edge of the facade passes her... press an imaginary "pause" button, and consider the session to date from the Virtual Reality system's point of view.

Clearly, as our participant was passing the front of the building, its facade was slipping past her view smoothly; its apparent motion was not very violent at all; we could quite easily redraw most of it at a fairly leisurely rate, relying on Galilean Antialiasing to make it move smoothly—and the extra time could be used to apply a particularly convincing set of Victorian period textures to the surfaces of the building (which would propagate along with the surfaces they are moulded to). We are, of course, here relying on the relatively mundane motion of the virtual objects in view as a *realism lever*: these objects can be rendered less frequently, but more realistically. And this is, indeed, precisely what one *does* want from the system: a casual stroll is usually a good opportunity to "take a good look at the scenery" (although mankind must pray that, for the good of all, no one ever creates a Virtual Melbourne Weather module).

Now consider what happens at the point in time at which we freeze-framed our session. Our participant is just about to look down an alley-way: she doesn't know what is down there; passing by the edge of the facade will let her have a look. The only problem is that *the video-controller doesn't know what's down there either*: the facade of the building Galileanly moves out of the way, leaving... well, leaving nothing; the video controller, for want of something better, falls back to $G^{(0)}$ technology, and simply leaves the debris of the *previous* frame on the display—hoping (if a video controller is capable of such emotions) that the display controller gets its proverbial into gear and gives it a picture to show quick smart.

So what *should* be visible when our participant looks down the alley? Well, a whole gaggle of objects may just be coming into view: the side wall of the building; the windows in the building; the pot-plants on the windowsills; a Japanese colleague of our participant who has virtu-commuted to her virtual building for a later meeting, who is right now sipping a coffee and happily waving out the window at her; and so on. But none of this is yet known to the video controller: a massive number of objects simply do not exist in the frame buffer at all. We shall say that these objects *have gone information-critical*: they are Priority One: something needs to be rendered, and rendered NOW.

How does the Virtual Reality system carry out this task? To answer this, it is necessary to examine, in a high-level form, how the entire system will work as a whole. To see what would occur in a *well-designed* system at the moment we have freeze-framed, we need to wind back the clock by about a quarter of a second. At that earlier time, the operating system, constantly projecting the participant's trajectory forward in time, had realised that the right side wall of building #147 would probably go critical in approximately 275 milliseconds. It immediately instigated a Critical Warning sequence, informing all objects in the system that they may shortly lose a significant fraction of display processing power, and should take immediate action to ensure that their visual images are put into stable, conservative motion as soon as possible. The right wall of building #147 is informed of its Critical Warning status, as well as the amount of extra processing power allocated to it; the wall, in turn, proceeds to carry out its pre-critical tasks: a careful monitoring of the participant's extrapolated motion and critical time estimate; a determination of just precisely which direction of approach has triggered this Critical Warning; a computation of estimates of the trajectories of the key control points of the wall and its associated objects; and so on. By the time 150 milliseconds have passed, most objects in the system have stabilised their image trajectories. The right wall of building #147 has decided that it will now definitely go critical in 108 milliseconds, and requests the operating system for Critical Response status. The operating system concurs, and informs the wall that there are no other objects undergoing Critical Reponse, and only one other object on low-priority Warning status; massive display-processing power is authorised for the wall's use, distributed over the next 500 milliseconds. The wall immediately refers to the adaptive system performance table, and makes a conservative estimate of how much visual information about the wall and associated objects it can generate in less than 108 milliseconds. It decides that it can comfortably render the gross features of all visible objects with cosine-shaded polygons; and immediately proceeds to instruct the display controller with the relevant information—not the positions, velocities, accelerations, colours and colour derivatives of the objects as they are *now*, but rather where they will be *at the critical time*. It time-stamps this image composition information with the projected critical time, which is by this time 93 milliseconds into the future; and then goes on to consider how best it can use its authorised *post-critical* resources to render a more realistic view. While it is doing so—and while the other objects in the system monitor their own status, mindful of the Critical Response in progress—the critical time arrives. The pixelated wall—with simple shaded polygons for windows, windowsills, pot-plants, colleagues—which the display processor completed rendering about 25 milliseconds ago, is instantaneously grafted onto the building by the video controller; the participant looks around the corner and sees... a wall! 200 milliseconds later—just as she is getting a good look—the video controller grafts on a new rendering of the wall and its associated objects: important fine details are now present; objects are now Gouraud-shaded; her colleague is recognisable; the coffee cup has a handle. And so she strolls on... what an uneventful and relaxing day this is, she thinks.

Let us now return to our original question: namely, what are the additional features our display processor and video controller need to possess to make the above scenario possible. Clearly, we need to have a way of *grafting* a galpixmap frame buffer onto the existing image being propagated by the video controller. This is a similar problem (as, indeed, much of the above scenario is) to that encountered in *windowed* operating systems. There, however, all objects to be grafted are simply rectangles, or portions thereof. In such a situation, one can code very efficiently the shape of the area to be grafted by specifying a coded sequence of corner-points. However, our Virtual Reality scenario is much more complex: how do you encode the shape of a wall? The answer is: you don't; rather, you (or, more precisely, your display processor) uses the following more subtle procedure: Firstly, the current frame buffer that has been allocated to the display processor for rendering purposes is cleared. How do we want to "clear" it? Simple: set the *debris indicator* of each galpixel in the frame buffer to be true. Secondly, the display processor proceeds to render only those objects that it is instructed to—clearing the debris indicators of the galpixels it writes; leaving the other debris indicators alone. Thirdly, when the rendering has been completed, the display processor informs the video controller that its pizza is ready, and when it should deliver it; the display processor goes on to cook another meal. When the stated time arrives, the video controller grafts the image onto its current version of the world, simply *ignoring* any pixels in the new image that are marked as debris. It is in this way that a wall can be "grafted" onto a virtual building without having to bulldoze the whole building and construct it from scratch.

It should be obvious that it is necessary to come up with some sort of terms that portray the difference between the frame buffers that the video controller uses to Galileanly-propagate the display from frame to frame, and the frame buffers that the display processor uses to generate images that will be "grafted on" at the appropriate time. To this end, we shall simply continue to refer to the video controller's propagating frame buffers as "frame buffers"—or, if a distinction is vital, as "Galilean frame buffers". Buffers that the display processor uses to compose its grafted images will, on the other hand, be referred to as *Flinders Street buffers*. ("Meet you under the clocks at Flinders Street Station at 3 o'clock" being the standard rendezvous arrangement in this city.) Clearly, for the display processor to be able to run at full speed, it should have at least two—and preferably more—Flinders Street buffers, so that once it has finished one it can immediately get working on another, even if the rendezvous time of the first has not yet arrived.

It is also worthwhile considering the parallel nature of the Virtual Reality system when designing the display controller and the operating system that drives it. At any one point time, there will in general be a number of objects (possibly a very large number) all passing image-generation information to the display processor for displaying. Clearly, this cannot occur directly: the display processor would not know whether it was Arthur or Martha, with conflicting signals coming from all directions. Rather, the operating system must handle image-generation requests in an organised manner. In general, the operating system will coordinate the requests of various objects, using its intelligence to decide on when the "next train from Flinders Street" will be leaving. Just as with the real Flinders Street Station, image generation requests will be pooled, and definite display rendezvous times scheduled; the operating system then informs each requesting object of the on-the-fly timetable, and each object must compute its control information *as projected to the rendezvous time of* *the most suitable scheduled time*. Critical Warning and Critical Response situations are, however, a little different, being much like the Melbourne Cup and AFL Grand Final days: the whole timetable revolves around these events; other objects may be told that, regrettably, there is now no longer any room for them; they may be forced to re-compute their control points and board a later train.

These deliberations bring us to the second of our listed points of consideration for our new image-generation philosophy: *timing and synchronisation*. The following phrase may be repeated over and over by Virtual Reality designers while practising Transcendental Meditation: "Latency is my enemy. Latency is my enemy. Latency is my enemy. . . ." The human mind is simply not constructed to deal with latency. Echo a time-delayed version of one's words into one's own ears and you'll end up in a terrible tongue-tied tangle (as prominently illustrated by the otherwise-eloquent former Prime Minister Bob Hawke's experience with a faulty satellite link in an interview with a US network). Move your head around to a visual world that lags behind you by half a second and you'll end up sick as a dog. Try to smack a virtual wall with your hand, and have it smack you back a second later, and you'll probably feel like you're fighting with an animal, not testing out architecture. Latency simply doesn't go down well.

It is for this reason that the above scenario (and, indeed, the Galilean Antialiasing technique itself) is rooted very firmly in the philosophy of *predictive* control. We are not generating the sterile, static world of traditional computer graphics: one *must* extrapolate in order to be believable. If it takes 100 milliseconds to do something, then you should find a good predictive "trigger" for that event that is reasonably accurate 100 milliseconds into the future. Optimising such triggers for a particular piece of hardware may, of course, involve significant research and testing. But if a suitably reliable trigger for an event *cannot* be found with the hardware at hand, then either get yourself some better hardware, or else think up a less complicated (read: quicker) response; otherwise, you're pushing the proverbial uphill. "Latency is my enemy. . . ."

With this principle in mind, the above description of a Flinders Street buffer involves the inclusion of a *rendezvous time*. This is the precise time (measured in frames periods) at which the video controller is to graft the new image from the Flinders Street buffer to the appropriate Galilean buffer. As noted above, some adaptive system performance analysis must be carried out by the operating system for this to work at all—so that objects have a reasonably good idea of just *what* they can get done in the time allocated to them. Granted this approximate information, image-generation instructions sent to the display processor should then be such that it *can*, in fact, generate the desired image before the rendezvous time. The time allowed the display processor should be conservative; after all, it can always start rendering another image, into another of its Flinders Street buffers, if it finishes the first one early. But it is clear that being *too* conservative is not wise: objects will unnecessarily underestimate the amount of detail renderable in the available time; overall performance will suffer. There must be an experienced balance between these two considerations.

In any practical system, however, Murphy's Law will always hold true: somewhere, some time, most probably while the boss is inspecting your magnificent creation, the display processor will not finish rendering an image before the rendezvous time. It is important that the system be designed to handle this situation gracefully. Most disastrous of all would be for the operating system to think the complete image *was* in fact generated successfully:

accurate information about the display's status is crucial in the control process. Equally disastrous would be for the display processor to not pass forward the image at all, or to pass it through "late": the former for the same reason as before; the latter becuase images of objects would then continue to propagate "out-of-sync" with their surroundings.

One simple resolution of this scenario is for the display processor to *finish what it can* before the rendezvous time; it then relinquishes control of the Flinders Street buffer (with a call of "Stand clear, please; stand clear"), and the partial graft is applied by the video controller. The display processor must then *pass a high-priority message back to the operating system that it did not complete in time*, with the unfulfilled, or partially-fulfilled, instructions simultaneously passed back in a stack. The operating system must then process this message with the highest priority, calling on the objects in question for a Critical Response; or, if these objects subsequently indicate that the omission is not critical, a regular re-paint operation can be queued at the appropriate level of priority. Of course, Incomplete Display Output events should in practice be a fairly rare occurrence, if the adaptive performance analysis system is functioning correctly; nevertheless, their non-fatal treatment by the operating system means that performance can be "tweaked" closer to the limits than would otherwise be prudent.

There is another side to the question of timing and synchronisation that we must now address. In section 3, we assumed that the apparent motion of an object is reasonably well described by a quadratic expression in time. This is, of course, a reasonable approximation for global-update systems (in which the inter-update time cannot be left too long anyway)—and is, of course, a vast improvement over current $G^{(0)}$ technology. However, with the introduction of *local* update systems we must be more careful. To see why this is the case, consider an object that is undergoing *rotation*. Now, ignoring that fact that more rotating objects seem to populate Virtual Reality demonstrations than exist in the entire visible Universe, it is clear that such objects pose a potential problem. This is because a quadratic expression only *approximately* describes the motion of any point in such an object. As a rule of thumb, once an object rotates by more than 45° about a non-trivial axis, this parabolic approximation starts to break down badly. It is therefore important that the object in question *knows* about this limited life of its Galilean image: it is useful for such objects to label their visual images with a *use-by date*. The operating system should treat use-by date expirations as a *high priority* event: rotating objects can rapidly fly apart if left any longer, scattering junk all over the display like an out of control Skylab—junk that may *not* be fully removed by simply re-rendering the object. As a *bare minimum*, in situations of Critical Warning, such objects should replace their visual images by a rough representation of the object, significantly blurred, and with an assigned velocity and acceleration close to zero. This will avoid the object from "self-destructing" while the system copes with its critical period; once the crisis is over, a more accurate (but again less stable) view can be regenerated. Of course, such intelligent actions must be programmed into the operating system and the object itself; this is one reason that the Critical Warning period was specified above, so that such evasive actions can be taken while there is still time.

This bring us most naturally to the general question of just *how* each object in a virtual world can decide how badly out-of-date its image is. This is a question that must, ultimately, be answered by extensive research and experience; a simple method will, however, now be proposed as a starting point. Each object knows, naturally, what information it sends to the display processor—typically, this consists of *control information* (such as polygon vertices, velocities, *etc.*), rather than a pixel-by-pixel description of the object. The object also knows just *how* the $G^{(2)}$ display system will propagate these control points forward in time—namely, via the propagation equations derived in sections 2 and 3 (and, importantly, with finite-accuracy arithmetic). On the other hand, the object can also compute where these control points *should* be, based on the exact virtual-world equations of motion, and also taking into account the relatives jerk of the object and the participant since the last update (but subtracting off the effects of specialised global updates, which shall be discussed shortly). By simply taking the "error" between the displayed and correct positions, and making an intelligent estimate, based on the projection system in used, about just how "bad" the visual effects of each of these errors are, an object can fairly simply come up with a relative estimate of how badly it is in need of an update. If all of the objects in the virtual world can agree on a protocol for quantifying this relative need for an update, then the operating system can use this information intelligently when prioritising the image update scheduling process.

Another exciting prospect for the algorithmics of image update priority computation is the technique of *foveal tracking*, whereby the direction of foveal view is tracked by a physical transducer. A Virtual Reality system might employ this information most fruitfully by having a *foveal input device driver* (like a mouse driver on a conventional computer) which gathers information about the foveal motion and relays it to the operating system to use it as it sees fit. Our foveal view, of course, tends to flick quickly between the several most "interesting" areas of our view, regularly returning to previously-visitied objects to get a better look. By extracting a suitable overview of this "grazing" information and time-stamping it, the foveal device driver can leave it up to the operating system to decipher the information if it sees fit. In times of Critical Response, of course, such information will simply be ignored (or stored for later use) by the operating system; much more important processes are taking place. However, at other times, where the participant is "having a good look around", this foveal information may (with a little detective work) be traced back to the objects that occupied those positions at the corresponding times; these objects may be boosted in priority over others for the purposes of an improved rendering or texturing process; however, one must be careful not to play "texture tag" with the participant by relying too exclusively on this (fundamentally historical) information.

We now turn to our third topic of consideration listed above, namely, ensuring that the display processor can reliably generate Flinders Street buffer images in the first place. This is *not* a trivial property; it requires a careful cooperation between the display processor and the image-generation software. This is because, in traditional global-update environments, the display processor can be sure that *all* visible objects will be rendered in any update; z-buffering can therefore be employed to ensure correct hidden surface removal. The same is *not* true, however, in local-update systems: only *some* of the objects may be in the process of being updated. If there are un-updated objects partially *obscuring* the objects that *are* being updated, hidden surface removal must still somehow be done.

One approach to this problem might be to define one or more "clip areas", that completely surround the objects requiring updating, and simply render all objects in these (smaller than full-screen) areas. This approach is unacceptable on a number of counts. Firstly, it violates the principles of local updating: we do *not* wish to re-render all objects in the area (even if it is, admittedly, smaller than full-screen); rather, we only want to re-render the objects that have requested it. Secondly, it carries with it the problem of *intra-object mismatch*: if one part of an object is rendered with a low-quality technique, and another part of the same object (which may "poke into" some arbitrary clip area) with a *high*-quality technique, then the strange and unnatural "divide" between the two areas will be a more intriguing visual feature than the object itself; the participant's consciousness will start to wander back towards the world of Real Reality.

Our approach, therefore, shall be a little subtler. In generating an image, we will consider two classes of objects. The first class will be those objects that have actually been scheduled for re-rendering. The second class of objects will consist of all of those objects, not of the first class, which are known to *obscure* one or more of the objects of the first class (or, in practice, *may* obscure an object of the first class, judged by whatever simpler informational subset that optimises the speed of the overall rendering procedure). Objects of the first class shall be rendered in the normal fashion. Objects of the *second* class, on the other hand, will be *shadow-rendered*: their z-buffer information will be stored, where appropriate, *but their corresponding colour information will flag them as debris*. "Clear-frame" debris (the type used to wipe clear the Flinders Street buffer in the first place), on the other hand, will both be marked as debris, and z-depth-encoded to be as far away as possible. Shadow-rendering is clearly vastly quicker than full rendering: no shading or texturing is required; no Galilean motion information need be generated; all that is required are the pixels of obscuration of the object: if the galpixel currently at one such position in the Flinders Street buffer is currently "further away" than the corresponding point of the second class object, then that pixel is turned to debris status, and its z-buffer value updated to that of the second-class object; otherwise, it is left alone. In this way, we can graft complete objects into an image, *without* affecting any other objects, with much smaller overheads than are required to re-render the entire display—or even, for that matter, an arbitrary selected "clip window" that surrounds the requesting objects.

We now turn to the fourth topic of consideration above: namely, the inclusion of specialised hardware, over and above that needed for Galilean Antialiasing, that can modify the *entire* contents of a Galilean frame buffer to take into account the perspectival effects of the motion of the participant herself (as distinct from the *proper motion—i.e.*, with respect to the laboratory—of any of the virtual objects). Such hardware is, in a sense, at the other end of the spectrum from the local updating just performed: we now wish to be able to perform *global updates* of the galpixmap—but only of its motional information. This goal is based on the fact that small changes of acceleration (both linear and rotational) of the participant's head will of themselves jerk the entire display; and all of the information necessary to compute these shifts (relying on the z-buffer information that is required for Galilean Antialiasing anyway) is already there in the Galilean frame buffer. Against this, however, is the fact that the mathematical relations describing such transformations are nowhere near as strikingly simple as those required for Galilean Antialiasing itself (which can, recall, be hard-wired with great ease); rather, we would need some sort of maths coprocessor (or a number of them) to effect these computations. The problem is that these computations *must* be performed during a *single* frame scan, and added to the acceleration of each galpixel as it is computed (where, recall, standard $G^{(2)}$ Antialiasing simply copies across a constant acceleration for each galpixel in the absence of more information); whether such a high-speed computational system will be feasible is questionable. However, were such systems to become feasible (even if only being able to account for suitably small jerks, say), the performance of Virtual Reality systems employing this technique will be further boosted as the display processor is relieved of the need to re-render objects simply because their overall acceleration is now out of date because the participant has jerked her head a little.

We now turn to the fifth and penultimate topic of consideration posed above, that of *obscuration caching*. The principles behind this idea are precisely the same as those behind the highly successful technique of *memory-caching* that is employed in many processor environments today. The basic idea is simple: one can make good use of galpixels that have been recently obscured if they again become unobscured. The principle situation in which this occurs is where an object close to the participant "passes in front of" a more distance one, due to parallax. Without obscuration caching, the closer object "cuts a swath" through the background as it goes, which thus requires regular display processor updates in order to be regenerated. On the other hand, if, when two galpixels come to occupy the same display position, the closer one is displayed, and the farther one is not discarded, but rather relegated to a *cache Galilean frame buffer*, this galpixel can be "promoted" back from the cache to the main display frame buffer if the current position in the main frame buffer becomes unoccupied. This requires, of course, that the cache have its own "propagator" circuitry to propel it from frame to frame, in accordance with the principles of $G^{(2)}$ antialiasing, and thus requires at least two frame buffers of its own; and it requires an increased complexity in memory access and comparison procedures between the cache and main displays; nevertheless, the increased performance that an obscuration cache would provide may make this a viable proposition.

Another, simpler form of caching, *out-of-view buffering*, may also be of use in Virtual Reality systems. With this approach, one builds *more memory* into each frame buffer than is required to hold the galpixel information for the corresponding display device: the extra memory is used for the logical galpixels *directly surrounding* the displayable area. An out-of-view buffer may be used in one of two ways. In *cache-only out-of-view buffering*, the display processor still renders only to the displayable area of memory; the out-of-view buffer acts in a similar way to an obscuration cache. Thus, if the participant rotates her head slightly to the right, the display galpixels move to the left; those galpixels that would have otherwise "fallen off the edge" of the memory array are instead shifted to the out-of-view buffer, up to the extent of this buffer. If the viewer is actually in the process of *accelerating* back to the *left* when this sweep begins, then in a short time these out-of-view galpixels will again come back into view (as long as they are propagated along with the rest of the display), and thus magically "reappear" by themselves, without having to be regenerated by the display processor.

On the other hand, in *full out-of-view buffering*, the entire out-of-view buffer is considered to be a part of the "logical display", of which the physical display device only displays a smaller subset. Objects are rendered into the out-of-view buffer just as to any other part of the display device. This approach can be useful for *preparative buffering*, especially when specialised head-motion-implementing hardware is present: views of those parts of the virtual world *just outside* the display area may be rendered in advance, so that if the participant happens to quickly move her head in that direction, then at least *something* (usually a low-quality rendition) is already there, and the response by the operating system need not be so critical. The relative benefits of out-of-view buffering depend to a great extent on the specific configuration of the system and the virtual worlds that it intends to portray; however, at least a *modest* surrounding area of out-of-view buffer is prudent on any Virtual Reality system: as the participant rotates her head, this small buffer area can be used to consecutively load "scrolling area" grafts from the Flinders Street buffers a chunk at a time, so that, at least for modest rotation rates, the edge of the display device proper never goes highly critical.

Finally, we must consider in a fundamental way the operating system and application software itself in a Virtual Reality system, if we wish to apply a local-update philosophy at all. Building the appropriate operating environments, and powerful applications to suit, will be an enormously complicated task—but one that will, ultimately, yield spectacular riches. And herein will lie the flesh and blood of every virtual world, whether it be large or small; sophisticated or simple; a simulation of Real Reality, or a completely fictitious fantasy world. Regrettably, the author must decline any impulse to speculate further on the direction that this development will take, and will leave this task to those many experts that are infinitely more able to do so. He would, nevertheless, be most interested in visiting any such virtual worlds that may be offered for his sampling.

And thus, in conclusion, we come to one small request by the author—the ultimate goal, it may be argued, of the work presented in this paper: Could the software masters at ORIGIN please make a Virtual Reality version of *Wing Commander*? I can never look out the side windows of my ship without fumbling my fingers and sliding head-first into the Kilrathi. . . .

5. Acknowledgments

Many helpful discussions with A. R. Petty and R. E. Behrend are gratefully acknowledged. This paper, and the supporting software developed to assist in this work, was written on an IBM PS/2 Model 70-386 equipped with an Intel i387 maths coprocessor, running MS-DOS 5.0 and Microsoft Windows 3.1, and employing VGA graphics. The software was developed using the Microsoft C/C++ 7.0 Compiler. The patient assistance rendered by Microsoft Australia Product Support is greatly appreciated.

This work was supported in part by an Australian Postgraduate Research Allowance, provided by the Australian Commonwealth Government.

Historical phrases used in this document that have a sexist bias syntactically, but which are commonly understood by English-speakers to refer unprejudicially to members of either sex, have not been considered by the author to be necessary of linguistic mutiliation. This approach in no way reflects the views of the University of Melbourne, its office-bearers, or the Australian Government. The University of Melbourne is an Affirmative Action *(sic)* and Equal Opportunity employer. Choice of gender for hypothetical participants in described thought experiments has been made arbitrarily, and may be changed globally using a search-and-replace text editor if so desired, without affecting the intent of the text in any way.

Queries or suggestions are welcome, and should be addressed to the author; preferably via the electronic mail address `jpc@physics.unimelb.edu.au`; or, failing this, to the postal address listed at the beginning of this paper.

Printed in Australia on acid-free unbleached recycled virtual paper.