The Rotating Planets in the Classic Game: *Elite*

Michael Fairbank
School of Computer Science and Electronic Engineering
University of Essex
Colchester, UK
m.fairbank@essex.ac.uk

Mark Moxon
mark@moxon.net

***Abstract*—This paper examines the graphical techniques used to generate rotating 3D planets in the classic 1984 space simulation game, *Elite*. Specifically, it describes the algorithms and geometric principles used, focusing on the drawing of planetary craters, meridians and equators. Emphasis is placed on the fast circle, ellipse and line-drawing algorithms, and on the game's orthographic projections that are used to display the elliptical features on the planet surfaces. We also describe the way the game rotates 3D objects at sufficient speed, by avoiding floating-point multiplications. The analysis was made by manually reverse engineering the game's 6502 assembler code. The findings contribute to an understanding of the programming tricks necessary to make early 3D video games viable, and illustrate how minimalist mathematical models can achieve compelling visual realism, all while running on a 2MHz 8-bit home computer.**

***Index Terms*—Gaming history, Elite, 3D graphics, orthographic projections, parametric ellipses, software archaeology**

I. Introduction

The video game *Elite* is widely regarded as a landmark in the history of computer games, particularly in terms of its hidden-line 3D graphics and pioneering use of procedural generation. Originally released for the BBC Micro in 1984, the game was ported to almost all contemporary home computer platforms and inspired a number of sequels, making *Elite* the longest-running space simulation series of all time [1].

Originally developed for 8-bit home computers with notable memory and processing constraints, *Elite* achieves a remarkable level of visual sophistication through innovative mathematical techniques. Among its most striking graphical features are the rotating 3D planets, which provide players with a sense of scale and realism, despite the limitations of monochrome wireframe rendering.

Each of the game's 2048 star systems has one planet. Procedural generation is used to generate a large amount of data for each system, such as population, economy, technology level, and galactic coordinates [2]. The planet type is determined by bit 1 of the system's technology level (which is an integer in the range 1 to 15), leading to two distinct planetary styles:

- Planets with craters, which employ elliptical geometry to simulate surface irregularities and depth; the crater rotates coherently with the body to give curvature to the planet's surface.
- Planets with meridians and equators, which use systematic line placement to suggest rotational axes and geographic segmentation.

This paper explores the key aspects of rendering these two planet types, and their elliptical features. The elliptical features drawn onto the planet (i.e. the craters and meridians) create visual cues that enhance the perception of the planet being a rotating sphere, as opposed to being a simple wireframe circle. These elliptical features exist on the BBC Micro version of *Elite*, but are omitted from most other home-computer versions of *Elite*, for speed.

Rather than relying on pre-rendered assets, *Elite* uses algorithmic methods to construct planetary bodies dynamically. By analysing these methods, we aim to demonstrate how simple mathematical constructs are leveraged in *Elite* to achieve compelling visual effects, offering insights into the ingenuity of early game graphics design.

The remainder of this paper is organised as follows. Section II reviews related work on *Elite*. Section III describes the fast line and circle algorithms used in the game, while Sections IV and V examine the mathematics of the orthographic projections used for the two main planet types (craters and meridians, respectively). Section VI discusses the fixed-point rotation algorithms employed by the game, and Section VII concludes the paper.

II. Related Work

As an ultimate primary source on the way *Elite* was programmed, Ian Bell (one of the co-authors of the game) hosts a website that includes the original BBC Micro source code for *Elite* from 1984. His website also includes additional documentation and design data such as ship blueprints, which form a durable foundation for technical inquiry [3]. However, because the game was built using a BBC Micro, the source code has to fit into the limited amount of memory on that machine, so it is stripped of spaces, with only a handful of extremely terse comments. Also, because David Braben (the other co-author) developed his 3D code on the even more restricted Acorn Atom, most label names are in a letter/number style like "XX21" or "QQ18", as required by that machine. As a result, the original source is extremely hard to follow and the algorithmic and mathematical details behind the game's inner workings are not at all obvious.

This source code was analysed more recently by the authors of this paper, with the results published on the software archaeology site at *bbcelite.com* [4]. This site contains the original source code from Ian Bell's site, but laid out in a fully documented and cross-referenced form that covers all the 6502-based platforms for which the game was released (BBC Micro cassette/disc, Acorn Electron, BBC Master, 6502 Second Processor, Commodore 64, Apple II and NES). The site also contains over 130 deep dives into all aspects of the game, and provides buildable repositories of all the game's variants on the 6502.

The influence of the game *Elite* has been discussed a lot since its original publication. In the broader historical context, Spufford's *Backroom Boys* situates the game's algorithmic ambition within the 1980s Cambridge/Acornsoft culture and the UK's "boffin" tradition [5]. Gazzard [6] examines *Elite* through the lens of platform studies and media archaeology, situating the game within the specific cultural, material, and technological conditions of British home computing in the 1980s. Her analysis draws on archival media to emphasise the role of hardware constraints, distribution practices, and player communities in shaping the game's historical significance.

Public discourse also includes creator talks and post-mortems, notably David Braben's GDC 2011 classic post-mortem on *Elite*, subsequent talks on procedural generation, and live design/exploration sessions around *Elite: Dangerous* (EGX 2014), while interviews with Ian Bell provide firsthand technical and design recollections [7]–[10].

Reverse engineering as a historical and technical method has been formalised in Aycock's *Retrogame Archeology* [11], which examines how early 8-bit and microcomputer games achieved sophisticated behaviour despite extreme hardware limitations. Through close analysis of machine-level code across multiple platforms, the book identifies recurring implementation strategies such as aggressive memory management, fixed-point arithmetic, procedural generation, and low-level optimisation. *Elite* is discussed as a notable example of ambitious 8-bit design, particularly with respect to its procedurally generated universe, though Aycock does not detail any of the specific graphical algorithms used within *Elite*.

From an 8-bit era gaming-graphics perspective, Collins [12] provides a technical survey of rendering techniques used in 8-bit games, emphasising raster-based graphics, integer arithmetic, and hardware-specific optimisations driven by limited CPU and memory resources. Focusing primarily on the Commodore 64, with comparative discussion of Atari 8-bit and Sinclair Spectrum systems, the article documents both standard and idiosyncratic approaches to achieving visual complexity on early home computers, and makes a mention of *Elite* being one of the earliest wireframe 3D games on these platforms.

Taken together, these works establish both the historical importance of *Elite* and the feasibility of algorithmic archaeology, while leaving key elements of its graphics mathematics and rendering algorithms unexplored: gaps that the present paper addresses.

III. Fast circles and straight lines on a 2MHz 8-bit machine

The planets in *Elite* are drawn in real time, at a typical frame rate of about 5 to 10 fps (for example, the equatorial meridian planet shown in Fig. 7 runs at around 8 fps on a standard BBC Micro). This is very impressive, considering the processing power of the machine (a 6502 processor clocked at 2MHz). In this era of 8-bit computing, it was more typical of home computers of the day to draw curves and circles at about 1-2 fps[1], which was a rate at which the user could see the curve gradually extending. So the fact that *Elite* could animate a moving circle, with elliptical features moving on them too (see Fig. 1), with a near-instantaneous appearance of each frame of the animation (albeit with some flickering), was quite remarkable at the time.

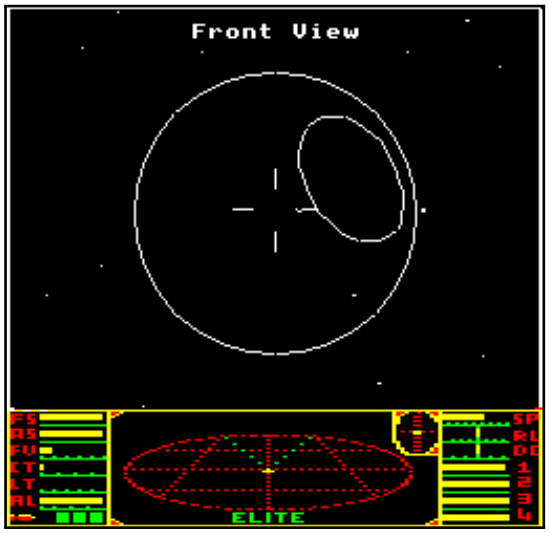

Fig. 1. The crater planet type.

In this section we describe some of the key methods *Elite* uses for circle drawing, straight line drawing, and trigonometric calculations.

A. Drawing the planet's circular outline

The 3D coordinates used in *Elite* are first-person camera-centric. The camera is at the coordinate origin, with its view looking down the positive z-axis (Fig. 2).

Consider a planet with centre $C = (x_c, y_c, z_c)$, and radius R_c.

Hence, in 2D screen coordinates, this needs to be drawn as a circle with centre (x_0, y_0) and radius r_0, where:

$$x_0 = \frac{x_c}{z_c}, \qquad y_0 = \frac{y_c}{z_c}, \qquad r_0 = \frac{R_c}{z_c}. \tag{1}$$

The $1/z_c$ factors are there to apply the perspective scaling. The screen resolution of the space view in the BBC Micro version

[1]For example, the ZX Spectrum's built-in circle routine can draw a circle of radius 75 pixels in 0.9s. The ZX Spectrum version of *Elite* can draw them much faster.

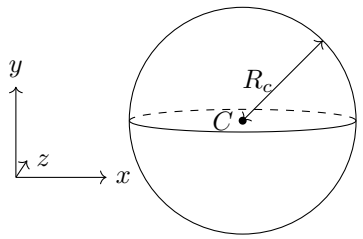

Fig. 2. Planetary Sphere and Coordinate axes

of *Elite* is 192×256 pixels, and r_0 is a one-byte quantity, and x_0 and y_0 are each two bytes.

Instead of using the familiar Bresenham circle algorithm [13], circles in *Elite* are drawn more simply, by using the parametric circle equation:

$$\mathbf{p}(\theta) = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + r_0 \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}, \qquad \theta \in [0, 2\pi). \tag{2}$$

The sine and cosine values required for this are stored in pre-computed lookup tables, with an angular resolution of 64 intervals around the unit circle; i.e. $360°/64 \approx 5.6°$ per unit of angle. For each angular step taken, the engine draws a straight line from the previous point to the current one. The angular steps move clockwise around the full 2π radians of the circle, so the routine does not exploit either the vertical or horizontal symmetry of a circle to speed this up (which seems like a very simple optimisation they omitted; presumably to save memory).

The product with r_0 in (2) requires some fairly costly 8-bit by 8-bit multiplications for each step of θ evaluated.[2] These multiplications seem to be the most significant bottleneck in the algorithm's execution speed. However, the following optimisation is made: when drawing this circle, instead of using all 64 angles, a tunable "step size" is used (defined by (3)), which trades angular resolution for execution time. This means that planets which are further away (and hence having a smaller radius r_c) are drawn with fewer steps, and hence more quickly. For example, there is no point in drawing 64 pixels if the circumference is much less than this.

$$\text{step\_size} = \begin{cases} 2 & \text{if } 60 \le r_0 \\ 4 & \text{if } 8 \le r_0 < 60 \\ 8 & \text{if } r_0 < 8 \end{cases} \tag{3}$$

This is one of the key tricks to *Elite*'s graphical speed - minimising the amount of computation needed, as early as possible in the algorithmic pipeline.

B. Sine/cosine lookup table

Angles in *Elite* are approximated in units with 64 intervals representing 2π radians. Trigonometric values are stored in a 32-byte lookup table of one-byte sine values. The nth value in the table stores $\text{round}(256 \sin(n/64 \times 2\pi))$, for $n = 0, \ldots, 31$.

[2]On the 8-bit 6502 processor, there is no multiply instruction. *Elite*'s 8-bit by 8-bit multiplication uses a shift-and-add algorithm, requiring a loop with 8 passes (see, e.g., [14]).

Cosines are recovered from sines using the identity $\cos(\theta) \equiv \sin(\theta + \pi/2)$. This is implemented by looking up the table's sine entry at an index offset by +16, modulo 32.

Angles beyond π are calculated from the same lookup table, using the trigonometric identity $\sin(\theta + \pi) \equiv -\sin(\theta)$. This allows the entire codebase of *Elite* to access the cosine or sine of any of the full range of 64 angles representing 2π radians, using only byte-sized table fetches and trivial modular arithmetic.

C. Drawing straight line segments at speed

The circles and ellipses in *Elite* are broken down into straight line segments, which need plotting as quickly as possible. The straight-line drawing code used by the game is a very fast implementation, around 5 times faster than the BBC Micro's built-in operating system's line drawing routine [15]. It is used for all of the wireframe objects in the game, and of course the planets too. It is tailored to take advantage of the game's custom screen resolution, which makes the screen exactly 256 pixels wide.

The *Elite* line drawing algorithm can be described as an integer slope-and-error stepper. It first categorises lines as either shallower than 45 degrees or steeper than 45 degrees. Alg. 1 illustrates the case of shallow sloping lines (i.e. where $|\Delta y| < |\Delta x|$). It increments x by one pixel per iteration, maintains an accumulated error proportional to $\Delta y/\Delta x$, and conditionally adjusts y by one line when the error exceeds a threshold. This algorithm is an instance of a digital differential analyser (DDA) rasterizer [16].

Algorithm 1 *Elite* straight-line drawing algorithm; for $|\Delta y| < |\Delta x|$

1: **Input:** endpoints (x_1, y_1), (x_2, y_2)
2: **Output:** pixels approximating the line segment
3: **if** $x_1 > x_2$ **then**
4: swap(x_1, x_2); swap(y_1, y_2)
5: **end if**
6: $\Delta x \leftarrow x_2 - x_1$; $\Delta y \leftarrow y_2 - y_1$
7: $m \leftarrow \left\lfloor 256 \cdot \dfrac{|\Delta y|}{|\Delta x|} \right\rfloor$ {fixed-point step}
8: $error \leftarrow 128$ {half-pixel start error (=0.5)}
9: $x \leftarrow x_1$; $y \leftarrow y_1$
10: **while** $x \le x_2$ **do**
11: PLOT(x, y) {plots with XOR}
12: $error \leftarrow (error + m) \bmod 256$
13: **if** addition in line 12 exceeded 255 **then**
14: $y \leftarrow y + \text{sign}(\Delta y)$
15: **end if**
16: $x \leftarrow x + 1$
17: **end while**

The algorithm requires one up-front division calculation per line, to calculate the 8-bit fractional part (multiplied by 256 to enable it to be stored in 1 byte) of the gradient $m = \Delta y/\Delta x$ (which is less than 1, since $\Delta y < \Delta x$). To do this, it uses an 8-bit shift-and-subtract algorithm, which requires a loop with

8 passes. Once this fraction m is found, the line-drawing loop only needs one 8-bit quantity to keep track of the cumulative error, and the main loop just needs an 8-bit addition to update this error term. Overall, there are only four variables needed in the line drawing loop: the current x-coordinate, the current y-coordinate, the accumulated error term, and the constant m. This avoids floating-point arithmetic entirely, and reduces everything to purely one-byte additions, carry-over checks, and occasional increments (see lines 10-16 of Alg. 1).

In contrast, the more well-known Bresenham straight-line drawing algorithm [17] does not require any division at all, so it might seem a better choice. However, on a 6502 processor this advantage is outweighed by the higher cost of the main drawing loop: Bresenham requires maintaining five variables, namely (x, y), a 16-bit accumulated error term, and the constants Δx and Δy, which entail multi-byte arithmetic and carry propagation on each iteration. By comparison, the *Elite* routine keeps its error accumulator in a single byte, and updates it using one 8-bit addition per pixel, reusing the resulting carry flag directly to determine when the y-coordinate must be incremented. Hence the main loop of the *Elite* line algorithm is significantly simpler and faster than the Bresenham loop, and the initial cost of the one-off division is paid back after only a short run of pixels. As a result, the *Elite* algorithm is faster for any line longer than just a few pixels.

There are four variants of the line-drawing algorithm included in *Elite*, each programmed for a different case of line angles. These include two algorithms for shallow sloping lines (where $|\Delta y| < |\Delta x|$); one for Δy positive, and another for Δy negative. And another two analogous versions for steeply sloping lines ($|\Delta y| \geq |\Delta x|$). It is interesting to note that the *Elite* authors spared the memory to include these four specialised variants of the line-drawing code (instead of a single flexible variant to handle all four cases). This memory-sacrifice was presumably made because speed of wireframe animations was such a priority for the game.

To animate the wireframe objects in *Elite*, before the next frame can be drawn, the old frame needs erasing. There is no such luxury of clearing the whole screen between each frame, as clearing a screen would require writing 6KB of memory, which would be too time consuming on the 6502 CPU. Instead, *Elite* erases each line one-by-one from the old wireframe position, by drawing on top of the old image with an XOR drawing mode. This is faster than screen-clearing, but does produce some flickering. In later versions of 6502 *Elite*, this flickering is significantly reduced, by erasing and replacing lines on a line-by-line basis, as opposed to on a spaceship-by-spaceship basis.

IV. Drawing Planets with Craters

In this section, we describe how *Elite* calculates and draws the planets with a crater feature on their surface (as in Fig. 1).

A. Orientation frame and coordinate conventions

Each planet carries an orthonormal frame of three unit vectors, $\{\vec{\mathbf{r}}, \vec{\mathbf{n}}, \vec{\mathbf{s}}\} \subset \mathbb{R}^3$, where $\vec{\mathbf{r}}$ ("roof") is the local outward normal, and $\vec{\mathbf{n}}$ ("nose") and $\vec{\mathbf{s}}$ ("side") span the tangent plane. These are illustrated in Fig. 3.

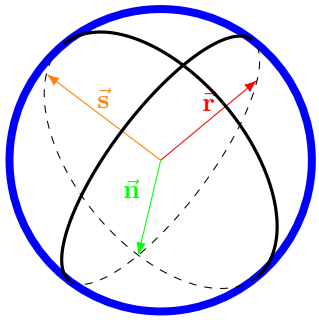

Fig. 3. Orthonormal axis vectors, $\vec{\mathbf{r}}$, $\vec{\mathbf{s}}$ and $\mathbf{u}$. Planet with two great circles (meridians) shown.

B. 3D representation of the crater circle

In 3D, the crater is represented as a circle embedded in the planet's surface. In 3D coordinates, let the planet centre be the origin. Then the crater's 3D circle, parametrically, is defined by:

$$\mathbf{p}(\varphi) = R_c \left(\frac{222}{256}\vec{\mathbf{r}} + \frac{\vec{\mathbf{n}}}{2}\cos\varphi + \frac{\vec{\mathbf{s}}}{2}\sin\varphi \right), \qquad \varphi \in [0, 2\pi). \tag{4}$$

Here the first term, $222\vec{\mathbf{r}}/256$, gives the centre of the crater's circle (relative to the centre of the planet), and $\vec{\mathbf{n}}/2$ and $\vec{\mathbf{s}}/2$ define an orthogonal set of basis vectors to define the plane of the crater's circle. This is illustrated in Fig. 4.

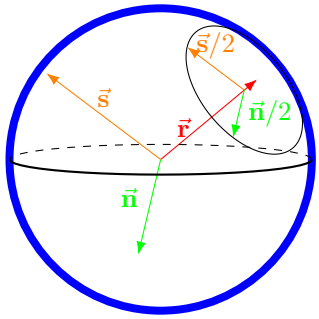

Fig. 4. Planet crater construction

The presence of the coefficient of $222/256$ appears explicitly in the game's code, and is there to ensure the rim of the crater lies at the surface level of planet. This makes the crater's centre slightly inset towards the planet's centre along $\vec{\mathbf{r}}$. (Note that if this coefficient was 1, then the rim of the crater would lie in the planet surface's tangent plane, and hence it would hover above the planet's surface). The coefficient $222/256$ is presumably an approximation to $\cos(\pi/6)$ (their actual values, to 3 decimal places, are 0.867 and 0.866 respectively). This is the natural distance the crater's centre would be from the centre of the planet's sphere, if you were to just create the crater by slicing off a chunk of the planet such that the radius of the crater created was $R_c/2$ (Fig. 5).

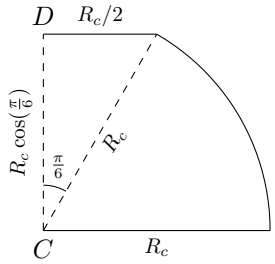

Fig. 5. Quarter cross-section of planet centre C, radius R_c, showing crater of radius $R_c/2$ and centre D at top.

C. Projection to 2D screen coordinates. Conjugate-radius parameterisation of the crater ellipse

The planets in *Elite* are projected to 2D screen coordinates using a scaled orthographic projection. This simply means that the z coordinate of the 3D equation (4) is removed, and the x and y coordinates are scaled by a constant scaling factor $1/z_c$. This combination of scaling and orthographic projection can be combined into a single matrix transformation, defined by:

$$M = \begin{pmatrix} 1/z_c & 0 & 0 \\ 0 & 1/z_c & 0 \end{pmatrix} \tag{5}$$

The scaling $1/z_c$ is the correct perspective for the planetary circumference, but it is not the correct perspective scaling for the features on the planet's surface that may be closer or further away to the viewer than z_c. Hence, by choosing this simplified orthographic projection instead of true perspective, the engine is deliberately ignoring the small depth variations between the front and back of the planet. Intuitively, the orthographic projection flattens the planetary sphere (and the crater feature on its surface) into a pancake in the xy plane. Because the variation of z around the crater is small compared with the planet-centre depth, this "orthographic + single-scale" approximation is visually faithful, yet computationally efficient.

If we define $\vec{\mathbf{r}}$, $\vec{\mathbf{s}}$ and $\vec{\mathbf{n}}$ to have components given by:

$$\vec{\mathbf{r}} = \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix}, \qquad \vec{\mathbf{n}} = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}, \qquad \vec{\mathbf{s}} = \begin{pmatrix} s_x \\ s_y \\ s_z \end{pmatrix}, \tag{6}$$

then applying the matrix transformation M to the 3D locus of the crater (4) gives:

$$\begin{aligned} \mathbf{x}(\varphi) =& M\mathbf{p}(\varphi) \\ =& \frac{R_c}{z_c}\left(\frac{222}{256}\begin{pmatrix} r_x \\ r_y \end{pmatrix} + \frac{1}{2}\begin{pmatrix} n_x \\ n_y \end{pmatrix}\cos\varphi \right. \\ & \left. + \frac{1}{2}\begin{pmatrix} s_x \\ s_y \end{pmatrix}\sin\varphi\right), \qquad \varphi \in [0, 2\pi). \end{aligned} \tag{7}$$

Adding on the screen coordinates (x_0, y_0) of the centre of the planet (defined by (1)), and reusing r_0 from (1), and then defining:

$$\mathbf{c} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \frac{222 r_0}{256}\begin{pmatrix} r_x \\ r_y \end{pmatrix}, \tag{8a}$$

$$\mathbf{u} = \frac{r_0}{2}\begin{pmatrix} n_x \\ n_y \end{pmatrix}, \qquad \mathbf{v} = \frac{r_0}{2}\begin{pmatrix} s_x \\ s_y \end{pmatrix}, \tag{8b}$$

means that (7) can be rewritten as:

$$\mathbf{x}(\varphi) = \mathbf{c} + \mathbf{u}\cos\varphi + \mathbf{v}\sin\varphi, \qquad \varphi \in [0, 2\pi). \tag{9}$$

This is the equation of the ellipse that is drawn to the screen. It is a standard parametric equation for an ellipse, where the centre is parameterised by $\mathbf{c}$, and the two vectors $\mathbf{u}$ and $\mathbf{v}$ act as *conjugate radii* [18]. Note that $\mathbf{u}$ and $\mathbf{v}$ need not be orthogonal: indeed, when they are oblique, the locus (9) is still an ellipse. See Fig. 6 for an illustration of this.

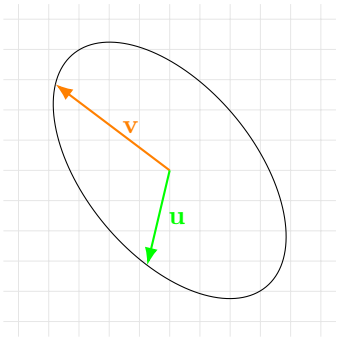

Fig. 6. Parametric 2D ellipse, with conjugate radii $\mathbf{u}$ and $\mathbf{v}$.

Using equation (9), and the sine and cosine lookup tables, the ellipse can be drawn fairly quickly, by *Elite*'s dedicated ellipse plotting routine. As with the circle algorithm used to draw the planet's circumference, the values of φ plotted step through the 64 degree angles used in the sine lookup table, in steps of 2, 4, or 8, decided by (3) (i.e. with more distant planets acquiring a larger angular step size, for speed).

To implement the illusion of the planet being solid, the crater is only drawn when it is on the far-side of the planet, i.e. it is only drawn when $r_z > 0$.[3] Thus, as the planet rotates, the crater disappears while it is behind the planet, and reappears as it moves around to the front again. When this rotating system is viewed, it provides the viewer with a strong sense of the spherical nature of the planet, with the circular crater hugging the planet's rotating spherical surface.

Note that from a reverse-engineering perspective, we had to start with the 6502 implementation of (9), i.e. an unknown 2D curve, with two non-orthogonal basis vectors and an obscure 222/256 coefficient in (8), and work backwards to discover the underlying 3D model (4); after which the nature of the 2D projection being orthographic becomes rather obvious. But initially it was quite challenging to deduce that unlike all

[3]It would make more sense to draw the crater on the front-side of the planet! But $r_z > 0$ is what the code does. From the viewer's point of view, they can't tell the difference whether the ellipse is at the front or back of the planet, due to the orthographic projection.

the wireframe spaceships in the game, the projections for the planets are not true perspective, and also to deduce how the non-perpendicular basis vectors for the parametric ellipse work and give what the human brain perceives as a circular crater on the planet's surface.

Also note that there is a further mathematical curiosity, in that there appears to be an extra degree of freedom when defining the ellipse: the vectors $\vec{\mathbf{s}}/2$ and $\vec{\mathbf{n}}/2$, which define the plane of the crater in Fig. 4, can be rotated about the axis of $\vec{\mathbf{r}}$, while still remaining perpendicular to each other and to $\vec{\mathbf{r}}$. This extra degree of freedom clearly does not distort the 3D circular crater defined in (4) (since a parametric circle can be defined using *any* two perpendicular basis vectors that lie in the plane of the circle); and hence, under the orthographic projection which transforms that 3D circle into a 2D ellipse in (7), this degree of freedom cannot distort the final ellipse drawn in 2D. Hence the final ellipse drawn is invariant to this extra degree of freedom.

V. Planets with Meridian and Equator

The second planetary style in *Elite* depicts a rotating globe with a longitudinal line (a meridian) and an equatorial line (Fig. 7). These features provide a strong visual cue of the planet's spherical nature, despite the absence of shading or texture.

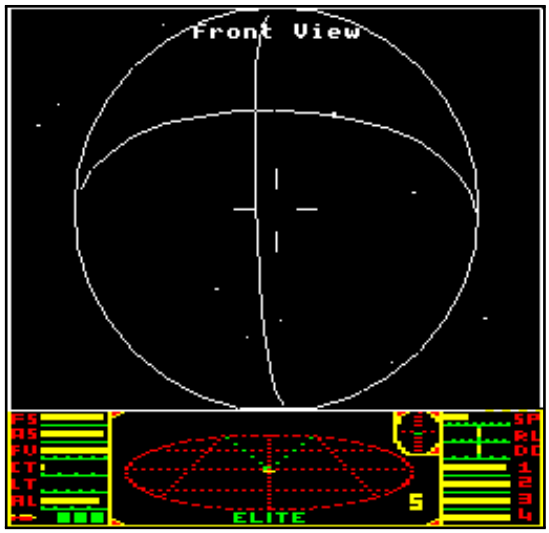

Fig. 7. The equatorial meridian planet type.

In wondering how this effect is achieved, the first question one might ask is, "What kind of shapes are the curved equatorial and meridian lines shown in Fig. 7?" Comparing this figure with the two partly-dashed ellipses drawn in Fig. 3, we see that the two curves drawn on the planet surface in Fig. 7 are indeed *half ellipses*. The game implementation re-uses the same ellipse-drawing routine used for craters, but applies it to draw the half-ellipses necessary for the meridian and equator.

A. 3D representation of the meridian and equator great circles

As before, the planet's orientation is represented by an orthonormal frame $\{\vec{\mathbf{r}}, \vec{\mathbf{n}}, \vec{\mathbf{s}}\}$. In three dimensions, the meridian and equator form great circles of the sphere. The construction of these great circles is illustrated in Fig. 3. For example, the equation of the great circle spanned by the $\vec{\mathbf{n}}$ and $\vec{\mathbf{s}}$ vectors shown in Fig. 3, relative to the planet's centre, is defined in 3D by:

$$\mathbf{p}(\varphi) = R_c \left(\vec{\mathbf{n}} \cos\varphi + \vec{\mathbf{s}} \sin\varphi\right), \qquad \varphi \in [0, 2\pi). \tag{10}$$

Note that this construction is very similar to the 3D crater construction (4), but for the great circles, the offset of $222\vec{\mathbf{r}}/256$ is omitted, and so are the scalings of $1/2$.

Equation (10) gives the great circle in the $\vec{\mathbf{n}}$ and $\vec{\mathbf{s}}$ plane. The construction of the great circle spanned by $\vec{\mathbf{n}}$ and $\vec{\mathbf{r}}$ proceeds in an identical way (hence, we omit that description from this paper).

B. Projection of great circles into two 2D half-ellipses.

To project the great circles into 2D ellipses, we again apply the orthographic transformation M (defined by (5)) to (10), to obtain:

$$\begin{aligned}\mathbf{x}(\varphi) =& M\mathbf{p}(\varphi)\\ =& \frac{R_c}{z_c}\left(\begin{pmatrix} n_x \\ n_y \end{pmatrix}\cos\varphi + \begin{pmatrix} s_x \\ s_y \end{pmatrix}\sin\varphi\right), \quad \varphi \in [0, 2\pi).\end{aligned} \tag{11}$$

Again, adding on the screen coordinates (x_0, y_0) of the centre of the planet (defined by (1)), and now defining:

$$\mathbf{c} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}, \qquad \mathbf{u} = r_0 \begin{pmatrix} n_x \\ n_y \end{pmatrix}, \qquad \mathbf{v} = r_0 \begin{pmatrix} s_x \\ s_y \end{pmatrix}. \tag{12}$$

means that (11) can be written in conjugate-radii parametric ellipse form, but modified to only draw half of the ellipse:

$$\mathbf{x}(\varphi) = \mathbf{c} + \mathbf{u}\cos\varphi + \mathbf{v}\sin\varphi, \qquad \varphi \in [\varphi_0, \varphi_0 + \pi). \tag{13}$$

This curve starts at the parameter value φ_0, which is described further in the next subsection.

C. Drawing the Correct Ellipse Half

Each ellipse is drawn as a polyline using *Elite*'s same dedicated ellipse plotting routine. But since the planet is opaque, and hence only half of the great circle is visible, it is only required to draw half of the ellipse. This is achieved by drawing only through angular range $\varphi \in [\varphi_0, \varphi_0 + \pi]$, where φ_0 is the parameter defined in Fig. 8.

We can identify which half of the ellipse we need to draw by considering the 3D equation of its great circle, (10), and then only drawing the half of it which is viewer-facing. Hence we can just solve which side of that circle has a negative z-coordinate. Expanding the z-coordinate of (10) for this viewer-facing criterion gives:

$$R_c \left(n_z \cos\varphi + s_z \sin\varphi\right) < 0 \tag{14}$$

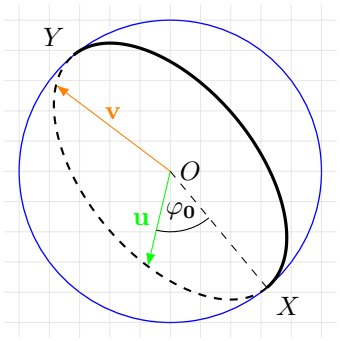

Fig. 8. Finding the start parameter, φ_0, at which to start drawing the visible half of the ellipse. X and Y are the points of intersection between the ellipse and the circle, and φ_0 is the "angle" between $\mathbf{u}$ and $\overrightarrow{OX}$. (Note that in this 2D image, φ_0 is not a true angle. It is merely the parameter in (13).) Note that the half-ellipse is drawn anti-clockwise, starting at φ_0.

The solutions to this inequality are given by:

$$\operatorname{atan2}(-n_z, s_z) + \pi < \varphi + 2k\pi < \operatorname{atan2}(-n_z, s_z) + 2\pi$$

for arbitrary integer k; and so, choosing $k = 0$, the starting parameter φ_0 is given by:

$$\varphi_0 = \operatorname{atan2}(-n_z, s_z) + \pi \tag{15}$$

The half-ellipse is drawn as a polyline using (13) and (15), with step-size increments given by (3). Perhaps strangely, the *Elite* code omits the $+\pi$ from (15); but in practice, because the projection of the planet is orthographic, the player cannot tell whether they are looking at the back-side of an ellipse or the front side of one.

D. Arctangent Lookup Table

Because the 8-bit 6502 processor lacks the hardware support for floating-point arithmetic or transcendental functions, *Elite* does not compute the arctangent directly at runtime. Instead, it relies on a compact, precomputed arctangent lookup table. This table stores quantised values of $\arctan(y/x)$ over a limited domain sufficient for use with the orientation vectors involved in planet rendering. The table occupies only 32 bytes in memory, reflecting both the severe memory constraints of the platform and the deliberately coarse angular resolution required for visually convincing results.

Interestingly, the only place in the game's entire codebase where this arctan table is used is in (15), i.e. for the purpose of drawing equators and meridians. It is not used anywhere else in the game. So the authors clearly prioritised having this kind of planet in the game; using up precious bytes of memory for it. They could instead have skipped this kind of planet, to save space, but thankfully that wasn't necessary.

VI. Planet Rotations

The planets in *Elite* are constantly rotating. On each frame of the game loop, the planets are rotated by $\alpha = 1/16$ radians about the planet's $\vec{\mathbf{s}}$ vector; and also $\alpha = 1/16$ radians about the planet's $\vec{\mathbf{n}}$ vector (so the planet is constantly undergoing a simultaneous pitch and roll).

Modern game engines typically implement rotations with rotation matrices or unit quaternions, performing arbitrary-angle updates with at least 32-bit floating-point precision; trigonometric functions, matrix–vector products, and quaternion normalisation are accelerated by contemporary CPUs and massively parallel GPUs, making millions of such operations per frame routine. In contrast, the BBC Micro version of *Elite* runs on a 2MHz 8-bit processor, and uses fixed-point 16-bit accuracy for its vector components, with no hardware multiplication, and sparse look-up tables for trigonometry; and so it needs a number of mathematical tricks to make the rotations of 3D objects such as planets sufficiently fast and accurate.

In the game, the changing orientations of planets and enemy spacecraft are treated in the same way as each other. Each object's 3D orientation is represented by its own three $\vec{\mathbf{n}}$, $\vec{\mathbf{r}}$ and $\vec{\mathbf{s}}$ orthogonal basis vectors, as described earlier. Rotations of all enemy spacecraft and planets are implemented by *rotating these basis vectors themselves* by a small, fixed angle $\alpha = \frac{1}{16}$ radians on each iteration of the main loop. This choice of angle means all visible objects in the game can only change their orientations at the same rate as each other. But this limitation allows *Elite* to use optimised multiplication routines and small-angle approximations to perform the rotation calculations; replacing expensive multiplications by efficient bit shifts, and replacing trigonometric evaluations by simple constant approximations.

For example, to implement an object's roll about the forward axis $\vec{\mathbf{n}}$, its forward vector is unchanged, while its $\{\vec{\mathbf{r}}, \vec{\mathbf{s}}\}$ pair is rotated within their plane. A matrix transformation to achieve this rotation is as follows:

$$\begin{bmatrix} \vec{\mathbf{r}}' & \vec{\mathbf{s}}' & \vec{\mathbf{n}}' \end{bmatrix}^T \approx \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{\mathbf{r}} & \vec{\mathbf{s}} & \vec{\mathbf{n}} \end{bmatrix}^T. \tag{16}$$

The choice of $\alpha = \frac{1}{16}$ enables the following small-angle approximations:

$$\sin\alpha \approx \alpha = \frac{1}{16},$$
$$\cos\alpha \approx 1 - \frac{\alpha^2}{2} = 1 - \frac{1}{2}\left(\frac{1}{16}\right)^2 = 1 - \frac{1}{512}.$$

Using these approximations, (16) can be implemented as:

$$\vec{\mathbf{r}}' \approx \left(1 - \frac{1}{512}\right)\vec{\mathbf{r}} + \frac{1}{16}\vec{\mathbf{s}}, \tag{17a}$$
$$\vec{\mathbf{s}}' \approx \left(1 - \frac{1}{512}\right)\vec{\mathbf{s}} - \frac{1}{16}\vec{\mathbf{r}}, \tag{17b}$$
$$\vec{\mathbf{n}}' = \vec{\mathbf{n}}. \tag{17c}$$

These constants make the matrix multiplication in (16) reducible to adds/subtracts and bit shifts. Note that a division by 16 can be implemented as 4 shifts to the right, and division by 512 can be implemented as 9 shifts to the right. On 8-bit machines, shifting by 9 bits is actually faster than shifting 4 bits for a 16-bit quantity (which consists of a high and a low

byte) - just replace the low byte by the high-byte right-shifted once, and then set the high byte to zero.

Equation (17) gives a roll rotation. An exactly analogous construction is used for pitch, rotating the $(\vec{\mathbf{r}}, \vec{\mathbf{n}})$ pair about $\vec{\mathbf{s}}$; and similarly for yaw (rotation about $\vec{\mathbf{r}}$). The algorithm can perform each rotation in either a clockwise or an anticlockwise direction, as needed; by providing an optional call argument which flips the signs of the 1/16 coefficients in equations (17a) and (17b).

By updating the orientation bases incrementally in this way, *Elite* achieves computationally inexpensive three-dimensional rotation, using only additions and power-of-two shifts, avoiding explicit trigonometric evaluation or general multiplication. However, because the game applies repeated small-angle approximations when rotating ships every frame, and only uses 16-bits of accuracy to represent each component of $\vec{\mathbf{n}}$, $\vec{\mathbf{r}}$ and $\vec{\mathbf{s}}$, numerical errors gradually accumulate, causing these vectors to drift away from orthonormality and leading to visible geometric distortion if left uncorrected. Modern 3D games generally avoid this problem entirely by using quaternions for rotations. However, to address this, *Elite* periodically re-orthogonalises the vectors using its "TIDY" routine: first normalising $\vec{\mathbf{n}}$, then making $\vec{\mathbf{r}}$ orthogonal to $\vec{\mathbf{n}}$ by subtracting its projection onto $\vec{\mathbf{n}}$, and finally recomputing $\vec{\mathbf{s}}$ as the cross product of the first two before normalising all vectors. The first two steps of this procedure are mathematically equivalent to a Gram–Schmidt orthonormalisation process [19]; but the final step of using the cross product is an enhancement over the regular Gram-Schmidt algorithm, improving speed and numerical stability. The outcome is to ensure that accumulated floating-point errors do not corrupt the local coordinate frame of the ship (or planet) over time.

VII. Conclusion

By reverse engineering the BBC Micro version of *Elite*, this paper has shown that the game's rotating planets arise from a compact and internally consistent mathematical model, rather than from ad-hoc graphical tricks. The implementation reflects an uncompromising dedication to speed: orthographic projection replaces true perspective, trigonometric evaluation is reduced to small lookup tables, and all geometry is expressed in fixed-point arithmetic amenable to fast shifts and adds on a 2 MHz 8-bit processor.

A key insight from this analysis is that *Elite* sometimes trades geometric exactness for perceptual coherence. Depth variation across the planet is deliberately ignored, yet temporally consistent rotation of surface features produces a strong impression of solidity. Reverse engineering reveals that seemingly opaque implementation details—most notably the use of the constant $222/256$ to offset the crater's location—encode precise geometric intent. Such intent is not apparent from the rendered output alone, but becomes clear only through reconstruction of the underlying mathematics.

Taken together, these findings illustrate how the authors of *Elite* transformed extreme hardware constraints into an effective and distinctive rendering style, and demonstrate the value of software archaeology for recovering design decisions and performance-driven insights embedded in historical code.

Acknowledgments

This article is an expanded and formalised version of work at [4]. Microsoft Copilot was used during parts of this document's preparation. The authors are grateful to Andrew Whinnett and Toby Nelson for advice on early versions of this paper. The game *Elite* was written by David Braben and Ian Bell. The BBC Micro version analysed here is copyright Acornsoft (1984) and the analysis is based on the original sources released by Ian Bell. The purpose of this article is to understand and explain some of the game's impressive programming tricks, from both a computer science perspective and as an appreciation of this historical game.

References

[1] "Longest-running space simulation series," https://www.guinnessworldrecords.com/world-records/397258-longest-running-space-simulation-series, 2014, accessed 2026-02-04.

[2] M. Moxon, "Deep dive: Galaxy and system seeds," https://elite.bbcelite.com/deep_dives/galaxy_and_system_seeds.html, n.d., BBC Elite Deep Dives. Accessed 04 February 2026.

[3] I. C. G. Bell, "The Elite Home Page," http://www.elitehomepage.org, 2026, accessed 2026-01-31.

[4] M. Moxon, "Elite on the 6502: Fully documented source code for *Elite* on the BBC Micro, Acorn Electron, Commodore 64, Apple II and NES," https://elite.bbcelite.com/, 2026, accessed 2026-01-31.

[5] F. Spufford, "The universe in a bottle," in *Backroom Boys: The Secret Return of the British Boffin*. London: Faber & Faber, 2003, ch. 3, pp. 71–120.

[6] A. Gazzard, "The platform and the player: exploring the (hi) stories of elite," *Game Studies*, vol. 13, no. 2, 2013.

[7] D. Braben, "Classic Game Postmortem – *ELITE* (GDC 2011)," https://gdcvault.com/play/1014628/Classic-Game-Postmortem, 2011, gDC Vault; Accessed 2026-01-31.

[8] ——, "On Procedural Generation in the *Elite* series," https://www.youtube.com/watch?v=f-AShRzPgOY, 2013, youTube; Accessed 2026-01-31.

[9] GameSpot and D. Braben, "*Elite: Dangerous* – Beta 2 Gameplay (EGX 2014 presentation)," https://www.youtube.com/watch?v=ZI6D-kQzyDo, 2014, youTube/GameSpot; Accessed 2026-01-31.

[10] I. Bell, "Elite's Ian Bell: Designing a Groundbreaking 3D Space Game (interview)," https://ohcast.info/retro/elite-creator-ian-bell/, 2024, podcast post; Accessed 2026-01-31.

[11] J. Aycock, *Retrogame Archeology: Exploring Old Computer Games*. Springer International Publishing, 2016.

[12] S. Collins, "Computer graphics during the 8-bit computer game era," Department of Computer Science, Trinity College Dublin, Tech. Rep. TCD-CS-1998-15, 1998. [Online]. Available: https://publications.scss.tcd.ie/tech-reports/reports.98/TCD-CS-1998-15.pdf

[13] J. Bresenham, "A linear algorithm for incremental digital display of circular arcs," *Communications of the ACM*, vol. 20, no. 2, pp. 100–106, 1977.

[14] L. A. Leventhal, *6502 assembly language programming*. McGraw-Hill, Inc., 1986.

[15] Toby Nelson, "6502 bresenham sheared ellipse," https://github.com/TobyLobster/ellipse, 2021, section "Theoretical Timings", Accessed: 2026-02-04.

[16] A. H. Watt, *3d Computer Graphics with Cdrom*. Addison-Wesley Longman Publishing Co., Inc., 1999, sec. 6.4.1.

[17] J. E. Bresenham, "Algorithm for computer control of a digital plotter," in *Seminal graphics: pioneering efforts that shaped the field*, 1998, pp. 1–6.

[18] Wikipedia contributors, "Conjugate diameters," https://en.wikipedia.org/wiki/Conjugate_diameters, 2026, accessed: 2026-02-04.

[19] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU press, 1996, Section 5.2.8.