(Published at IEEE Conference on Games, Madrid, 1–4 September, 2026)

Analysing the Saturn Loading Screen from the Classic Computer 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 draw the loading screen in the classic 1984 space-simulation game, *Elite*, which features a 3D rendering of Saturn, its rings, and a background starfield. Having reverse-engineered the game's 6502 assembler code, we describe how the image is constructed using pseudo-randomly distributed dots that are selectively filtered to form the planet's disk, its concentric rings, and the surrounding starfield. The most interesting aspect of this code is a single square root operation which we unpack and explain. We reveal that within that operation lies a 3D lighting model of a sphere illuminated from the side, including a texture-mapping effect for the random scattering of pixels that illuminate the planet's surface. This part of the algorithm works by passing uniformly distributed random numbers through an inverse-transform sampling function, such that the output distribution matches the desired lighting intensity with texture mapping. This process produces the illusion of curvature, depth, and directional lighting, despite the absence of conventional polygonal shading. By analysing the algorithmic construction of the Saturn image, this paper highlights how minimal computational resources were used to achieve a visually rich and dramatically anticipatory loading sequence.**

***Index Terms*—Gaming history, Elite, 3D graphics, inverse transform sampling, software archaeology**

I. Introduction

The 1984 release of *Elite*, written by Ian Bell and David Braben and published by Acornsoft, marked a defining moment in computer-game history, establishing many of the core principles of the modern space-simulation and open-world genres [1], [2]. Its influence stems from an unprecedented combination of real-time 3D wireframe graphics, an open-ended trading and combat engine, and a procedurally generated universe containing eight galaxies and 2,048 explorable planetary systems—an almost unimaginable scale on home hardware of that era. Built originally for the BBC Micro, and then ported to many contemporary 8-bit home computers including the Commodore 64 and the ZX Spectrum, *Elite* required extraordinary programming ingenuity to compress its flight simulator, economy model, and universe generator into a machine with only 32K of RAM. Its technical achievements and design innovations established *Elite* as one of the most influential computer games of the 1980s, shaping decades of successors in both space simulation and open-world design.

Fig. 1. The *Elite* loading screen, showing the game's loading progress.

Acornsoft understood this, right from the start, and *Elite* came bundled with an impressive number of extras that were designed to add even more immersion [3]. Squeezed into the game's inch-thick box were the game cassette (or disk), a 64-page flight manual, a 48-page novella, a ship reference card, a poster, a function key strip, loading instructions and a competition entry postcard. This lavish attention to detail even extended to the game's loading screen (Fig. 1), which is the subject of this paper.

The Saturn image on the game's loading screen is a notable example of *Elite*'s graphical and mathematical sophistication. The game was released on both cassette and floppy disk, but disk drives were relatively expensive in 1984, so most players had the cassette version. The tape-loading time on the BBC Micro was just over eight minutes (compared to about eight seconds for those users lucky enough to have a disk drive), so most players had quite a bit of time to stare at the loading screen, in awe and anticipation of the game that awaited.

At the core of the loading screen, behind the dashboard and

loading message, is the iconic Saturn image in Fig. 2. The planet is drawn dynamically by the loader and is not a static bitmap; watching the planet being drawn dot-by-dot is all part of the build-up inherent in the game's loading experience.

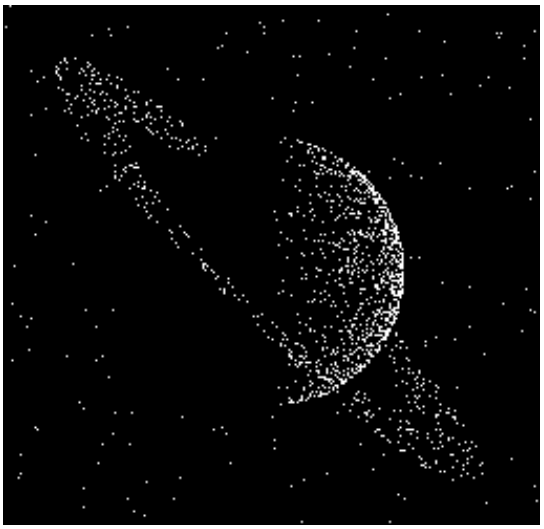


Fig. 2. The Saturn planet of the *Elite* loading screen (with the loading messages and dashboard removed).

Driving both the loading screen and *Elite*'s groundbreaking gameplay is a comprehensive set of mathematical routines that implement the game's varied arithmetic and trigonometric functions in 8-bit 6502 assembly language. *Elite*'s wireframe 3D graphics, complete with hidden-line removal, were revolutionary for home computers and helped redefine expectations for visual immersion in games, and they relied heavily on fast and compact algorithms. The authors clearly understood the importance of these routines and were "very protective of their code in *Elite*, particularly the math[s] routines" [4]. Indeed, programmers who were commissioned to convert *Elite* to non-6502 platforms had to write their own maths routines from scratch; almost nothing was passed on, even to those working on the game professionally [5].

The Saturn planet image is constructed in three stages, sequentially rendering the planet's disk, the starfield, and the surrounding rings. The routine employs a pseudo-random number generator, seeded by a hardware timer, to produce a slightly different dot distribution with each load. These dots are then filtered through geometric masks that determine their inclusion in the planet's silhouette, illuminated regions, or ring structures, thereby achieving curvature, depth, and shading without floating-point arithmetic or polygonal rendering. Note that generating the image dynamically like this helps improve the loading speed of the game, since the code to generate the image is smaller than an equivalent static bitmap would be. The loading screen exemplifies the efficiency and expressiveness of the low-level graphical techniques that are characteristic of *Elite*'s design.

The remainder of this paper is organised as follows. Section II reviews related work. Section III describes the algorithm used to render the background stars and planetary rings, while Section IV explains the construction and lighting of the planetary sphere. Finally, Section V presents our conclusions.

II. Related Work

The source code for *Elite* has been released by Ian Bell [6]. However, this original source is in 6502 assembler, and due to the memory limitations of having to build the game on a BBC Micro, the source is dense, largely comment-free, and uses obscure variable names. Derivations for the mathematics and algorithms are not provided there. Our work expands this original source code, by fully reverse-engineering and annotating it by hand, with the outputs fully documented at [7]; and we expand upon the aspects relating to the loading screen in this paper.

Reverse-engineering and algorithmic archaeology of early video games has precedent in both platform studies and computer graphics history. Aycock's *Retrogame Archeology* systematically examines how 8-bit and early microcomputer games achieved complex behaviour under severe hardware constraints, using close reading of machine-level implementation to recover lost programming techniques [8]. The book surveys several early games to illustrate common strategies in memory management, integer arithmetic, procedural generation, and optimisation, framing reverse engineering as a legitimate method of historical and technical analysis. *Elite* is discussed as a representative example of ambitious 8-bit design, with a discussion of its procedurally generated universe.

Collins [9] surveys the graphical techniques used by 8-bit computer games, with particular emphasis on raster-based graphics, integer arithmetic, and hardware-specific optimisations necessitated by severe memory and CPU constraints. It focuses primarily on the Commodore 64, Atari 8-bit and Sinclair Spectrum machines. *Elite* is explicitly mentioned as one of the first wireframe 3D games, noted for its innovative use of back-face culling.

Previous work has also reverse-engineered *Elite* to build new and extended versions of the game, including Angus Duggan's *Elite-A* (released in 1997, [10]); Christian Pinder's *Elite: The New Kind* (released in 2001, [11]); and *Oolite*, an open-source modernised version of *Elite* developed from 2003 onwards [12]. These projects focus on playability, extension, or modernisation of the original game, but none explain the inner workings of the original game itself.

The historical importance of *Elite* is underscored by the critical and scholarly attention it has received. Francis Spufford [2] describes how *Elite* broke the mould for computer games of the day by moving beyond short-form arcade gameplay and into open-world simulation, inventing new technical methods to make an unprecedented 3D universe fit into a few kilobytes, and redefining what a home computer game could be. At the time, most computer games were limited to only a few minutes of play, with a simple score and three lives.

Gazzard [13] situates *Elite* within the context of British home-computer culture of the 1980s, analysing historical media to understand how the game emerged from specific

hardware constraints and player practices. Her work focuses on the cultural and historical context of *Elite*, rather than its technical implementation.

III. Rendering the Background Stars and Rings

In this section, we cover the two simpler aspects of the *Elite* loading screen, i.e. the plain scattered stars, and ring system. We save describing the shaded planet until Section IV.

The planet and rings are bounded by a circle, and two ellipses, respectively. The equations for these, and the coordinate system used, are shown in Fig. 3.

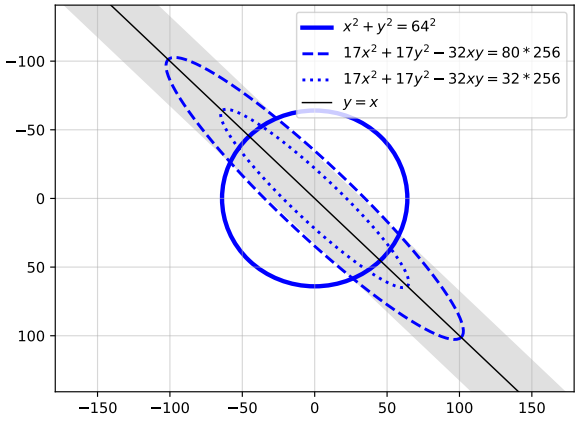


Fig. 3. Bounding circles and ellipses of planet and rings. Screen coordinates cover the range $-128 \le x \le 127$ and $-128 \le y \le 127$, i.e. the screen in the game *Elite* is 256×256 pixels.

A. Plotting the Background Stars

The background stars are plotted in a loop. For each star, consider a pixel at (x, y) where:

$$x \sim \text{Uniform}\{-128, \ldots, 127\},$$
$$y \sim \text{Uniform}\{-128, \ldots, 127\}.$$

The random numbers used here come from a bespoke additive lagged Fibonacci pseudo-random generator [14], which is seeded from the 6522 System VIA T1C-L hardware timer. Use of this timer means the random seed, and random distribution of stars, differ each time the game is loaded.

For each candidate star at (x, y), only pixels that pass a *circular exclusion* filter are plotted, ensuring stars appear outside the planetary disk. The integer-friendly filter is:

$$\frac{x^2 + y^2}{256} > 17, \tag{1}$$

which rearranges to:

$$x^2 + y^2 > (65.96)^2,$$

(to two decimal places). This means stars are excluded from a circle of radius 65.96, which slightly exceeds the planet's radius of 64, producing a thin two-pixel wide black circle absent of stars. It's as if this forms a 2-pixel wide "atmosphere" surrounding the planet.

Note that equation (1) contains a division by 256. This deliberate design choice makes the equation extremely easy and compact to implement in 8-bit machine code (where division by 256 just means ignoring the low byte of a 2-byte quantity); and extremely fast at execution time (because no actual division is required). We observe that many of the equations in this paper also have factors of 256 within them, for the same efficiency reasons. Even the custom screen size used in the game is deliberately chosen to be 256 pixels wide, to speed up translation of (x, y) coordinates into the corresponding memory address in video RAM.

Pseudocode for the star-plotting process is summarised in the first 7 lines of Alg. 1.

Algorithm 1 Plot Stars and Rings

```
{— Plot Stars —}
for count in range(477) do
    x ← random(−128, 127)
    y ← random(−128, 127)
    p ← ⌊(x² + y²)/256⌋
    if p > 17 then plot point at (x + 128, y + 128)
end for
{— Plot Rings —}
for count in range(1280) do
    r5 ← random(−128, 127)
    r6 ← random(−128, 127)
    r7 ← ⌊r5/4⌋
    x ← r6 + r7
    y ← r6
    p ← ⌊(x² + y²)/256⌋
    e ← ⌊((r6 + r7)² + r5² + r6²)/256⌋
    if (32 ≤ e < 80) and (r5 < 0 or p > 16) then
        plot point at (x + 128, y + 128)
    end if
end for
```

Before plotting a candidate point, (x, y), its coordinates need shifting to actual screen coordinates (which has its origin $(0, 0)$ in the top-left; unlike the centred coordinate system illustrated in Fig. 3). Hence coordinates of each point (x, y) are transformed to $(x + 128,\ y + 128)$ before finally plotting.

B. Plotting the Planetary Rings

The rings routine concentrates candidate points into a diagonal band, applies a two-ellipse (annulus) filter, and then culls pixels that would lie behind the planet.

For each candidate pixel in the planet rings, the algorithm first chooses two signed random values:

$$r_5, r_6 \sim \text{Uniform}\{-128, \ldots, 127\}.$$

Then define:

$$r_7 = \left\lfloor \frac{r_5}{4} \right\rfloor, \qquad x = r_6 + r_7, \qquad y = r_6. \tag{2}$$

Combining these gives:

$$y = x - \left\lfloor \frac{r_5}{4} \right\rfloor. \tag{3}$$

This concentrates (x, y) within a wide diagonal band from top-left to bottom-right (indicated by the light grey stripe in Fig. 3), increasing the likelihood that samples fall within the ring region (i.e. fewer wasted samples). This automatic restriction of candidate points to the grey band greatly speeds up the image generation, but even so, on the 2MHz 6502 processor, you can see the pixels arrive one-by-one on the screen.

a) Elliptical annulus filter.: Before any potential ring pixel is plotted, the candidate coordinates (x, y) must lie between two concentric ellipses. This is checked in integer arithmetic by the following test:

$$32 \;\le\; \frac{(r_6 + r_7)^2 + r_5^2 + r_6^2}{256} \;<\; 80, \tag{4}$$

which, combining with (2), rearranges to the quadratic form:

$$32 \cdot 256 \;\le\; 17x^2 - 32xy + 17y^2 \;<\; 80 \cdot 256. \tag{5}$$

Note that the bounds of these two inequalities in (5) represent two angled ellipses, each centred on the origin. In general, an ellipse centred on the origin can be described by the equation $ax^2 + bxy + cy^2 = 1$, with $a > 0$, $c > 0$, and $b^2 - 4ac < 0$. In (5), we have $a = c = 17$ and $b = -32$, and so the lower and upper bounds in (5) correspond to the inner and outer ring ellipses (plotted in Fig. 3), which the candidate pixel must lie between.

b) Planet occlusion test.: In addition to the elliptical tests of (5), to avoid drawing ring pixels where the rings pass behind the planet, the routine only plots a pixel if either:

$$x^2 + y^2 > 64^2 \quad \text{(outside of planetary disk)}, \tag{6a}$$

or:

$$r_5 < 0 \quad \text{(on front arc of rings)}. \tag{6b}$$

If neither (6b) nor (6a) holds, then the ring pixel is hidden behind the planet, and the sample is culled. Note that to understand (6b), the sign of r_5 (according to (3)) indicates whether a pixel is above or below the line $y = x$ (shown as a black line within the grey stripe in Fig. 3). Those pixels with $r_5 < 0$ are below $y = x$, and hence are in front of the planet (so plotted), and those with $r_5 > 0$ are above $y = x$, and hence are behind the planet (so not plotted).

Pseudocode for the rings-plotting process is summarised in the latter half of Alg. 1. Note that, unlike the starfield routine, the rings are not obscured by the two-pixel atmospheric gap, yielding a minor inconsistency between the filters.

IV. Shading of the Planetary Disk

The planet-drawing routine described here shades the planetary disk, by plotting dots scattered with a non-uniform probability distribution, corresponding to a visually convincingly lit hemisphere (Fig. 2). The method begins with uniform random sampling, applies a circular in-region test, and then transforms the accepted samples via a square-root mapping into a non-uniform distribution which corresponds to the desired 3D-lighting model.

A. Uniform Sampling and Circular In-Region Test

To generate each pixel on the planet disk, the algorithm generates two signed 8-bit uniform random numbers:

$$r_1, r_2 \sim \text{Uniform}\{-128, \ldots, 127\}. \tag{7}$$

This candidate point is accepted for plotting if:

$$(r_1^2 + r_2^2) < 128^2. \tag{8}$$

The above equation implies a circle of radius 128, however the coordinates are eventually halved before plotting, so the actual radius of planet plotted is 64.

The use of the expanded radius in (8) serves two practical purposes. Firstly, since the algorithm works by rejection sampling, by testing membership of a disk of radius 128 in (r_1, r_2) space rather than radius 64, the acceptance region occupies a four times larger area, increasing the probability that a random sample by (7) is accepted. This therefore reduces the expected number of rejection iterations, which directly improves plotting speed. Secondly, performing the in-disk test (8) at twice the final spatial scale allows all intermediate calculations to remain in integer arithmetic while retaining one additional bit of precision before the subsequent division by two. From a numerical perspective, this means that subsequent processing of the lighting model can be interpreted as operating in Q7.1 fixed-point arithmetic (i.e. with one bit after the binary point), before converting to final integer pixel coordinates.

B. Coordinate Mapping and Plotting

For an accepted sample (r_1, r_2), the routine chooses screen coordinates:

$$y \;=\; r_2, \tag{9a}$$

$$x \;=\; \sqrt{128^2 - (r_1^2 + r_2^2)}, \tag{9b}$$

and then plots at $(x/2 + 128,\ y/2 + 128)$.

The 8-bit processors of the time did not generally have in-built multiplication or square-root instructions, and the 6502 in the BBC Micro is no exception. For a multiplication algorithm, e.g. in order to compute the squared values appearing in (9b), *Elite* uses a shift-and-add algorithm, requiring a loop with 8 passes (see, e.g., [15]). For the square-root function, it appears (according to [16]) that *Elite* uses a 6502-assembler routine published in contemporary computer magazines [17], [18] (where the square-root algorithm itself was credited to a reader, John Kerr).

Equations (7)-(9b) fully describe the code used for the entire 3D lighting model, and are summarised in Alg. 2. This algorithm results in the shaded planet seen in the loading screen.[1]

The key equation of interest for the plotting of the planet is (9b), i.e.:

$$x \;=\; \sqrt{128^2 - (r_1^2 + r_2^2)}.$$

[1] Verifiable BBC-Basic implementations of the pseudocode are available at [19].

Algorithm 2 Plot Illuminated Planet

for count in range(1280) **do**
  $r_1 \leftarrow \text{random}(-128, 127)$
  $r_2 \leftarrow \text{random}(-128, 127)$
  $p \leftarrow r_1^2 + r_2^2$
  if $p < 128^2$ **then**
    plot point at $\left(\frac{\sqrt{128^2 - p}}{2} + 128,\ r_2/2 + 128\right)$
  end if
end for

This single line performs a transformation that redistributes sample density across the disk so that points are concentrated toward the lit rim on the right-hand side, implementing the intended lighting model without floating-point shading (Fig. 4). We spend the following subsections expanding upon how these equations work, and what lighting model they hide within.

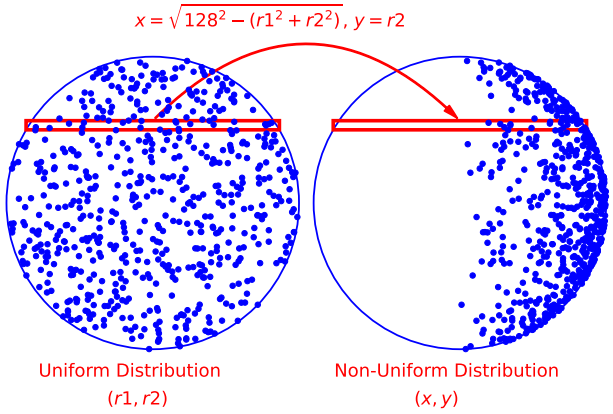


Fig. 4. Effect of Equations (9a) and (9b), which turn a uniformly distributed cluster of dots into a distribution corresponding to a 3D lighting model.

C. Probability Density Function for the Illumination Model

In this subsection, we try to discover what underlying probability density function (PDF) the transformation (9b) creates for x.

Consider a row of pixels at height y through the planetary disk (e.g. as illustrated by the red rectangle in Fig. 4). The right-most coordinate of this rectangle is given by:

$$r_x \;=\; \sqrt{128^2 - y^2} \;=\; \sqrt{128^2 - (r_2)^2}$$

Let:

$$u \;=\; \big|r_1/128\big|.$$

and note that $0 \le u \le 1$ is a uniform random variable (following from (7)).

Then the transformation (9b) can be written as:

$$x \;=\; \sqrt{(r_x)^2 - (128u)^2}. \tag{10}$$

Note that equation (10) takes as input a uniform random variable u (in the range 0 to 1) and applies a deterministic transformation that "squashes" this uniform distribution to produce a new random variable x with a *non-uniform* distribution. This kind of squashing mechanism is common in random number generation: whenever programmers call library routines that directly generate non-uniform distributions (such as Gaussian or normal variates), such routines typically begin with one or more uniform random numbers, and transform them by this kind of squashing function, to obtain the desired output distribution.

The formal theory underlying such transformations is known as *inverse transform sampling* [20], [21]. In this framework, a uniform random variable $u \in [0, 1]$ is mapped to an output variable x by applying the inverse of the cumulative distribution function (CDF) corresponding to the target probability density. That is:

$$x = \text{CDF}^{-1}(u),$$

where the CDF is defined as the integral of the associated probability density function (PDF). Interpreted in this light, equation (10) is acting as a CDF^{-1} operator for x along a fixed scanline at height y. Hence, inverting (10) (i.e. solving it for u) yields the underlying CDF:

$$u \;=\; \frac{\sqrt{(r_x)^2 - x^2}}{128} \;=\; \text{CDF}(x). \tag{11}$$

And since a CDF is defined to be the integral of its corresponding PDF, therefore differentiating (11) with respect to x gives the PDF for x:

$$\text{pdf}(x) \;=\; \begin{cases} \frac{x}{\sqrt{128^2 - y^2 - x^2} \cdot 128} & \text{for } x > 0 \\ 0 & \text{for } x \le 0. \end{cases} \tag{12}$$

This PDF precisely describes how the shading density increases toward the right-hand boundary of the planetary disk, and this equation is illustrated in Fig. 5.

The next step is to try to understand this equation geometrically.

D. Geometric Interpretation on a Sphere

Using the sphere relation:

$$x^2 + y^2 + z^2 \;=\; 128^2,$$

which rearranges into:

$$z \;=\; \sqrt{128^2 - y^2 - x^2},$$

the PDF (12) can be rewritten as:

$$\text{pdf}(x) \;=\; \frac{x}{128\, z} \qquad (\text{for } x > 0). \tag{13}$$

Let P be a point on the surface of the planet, with coordinates (x, y, z); and let $\hat{\mathbf{n}} = (n_x, n_y, n_z)$ be the unit surface normal at P, as illustrated in Fig. 6.

Observe that $\overrightarrow{OP} = (x, y, z)$ is parallel to $\hat{\mathbf{n}}$, so $x/z = n_x/n_z$, giving:

$$\text{pdf}(x) \;=\; \frac{n_x}{128\, n_z}, \qquad (\text{for } x > 0).$$

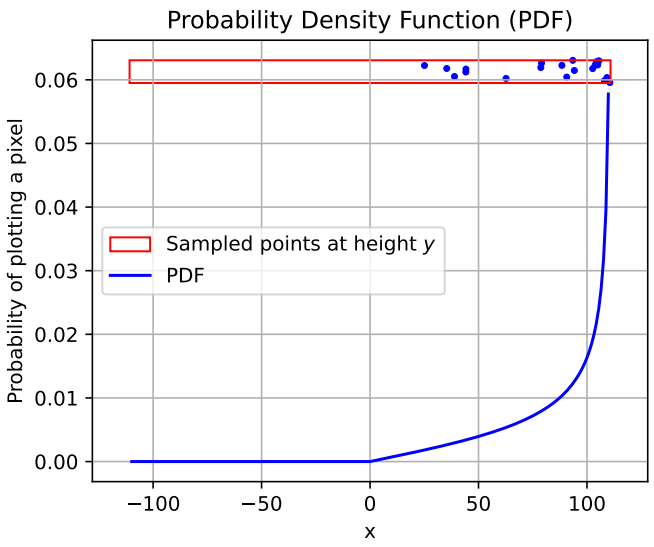


Fig. 5. Obtained PDF (Equation (12)). Here the red rectangle is representative of the distribution of plotted points in a horizontal slice of the planet image. This rectangle represents the same red slice as that highlighted in Fig. 4 (right).

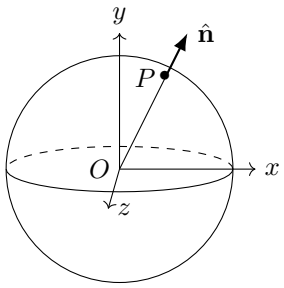


Fig. 6. Planetary Sphere

With Cartesian unit vectors $\mathbf{i}$ and $\mathbf{k}$ along x and z, and θ_x and ϕ_z as the angles between $\hat{\mathbf{n}}$ and the x and z axes respectively, we have $n_x = \hat{\mathbf{n}}\cdot\mathbf{i} = \cos\theta_x$ and $n_z = \hat{\mathbf{n}}\cdot\mathbf{k} = \cos\phi_z$, so:

$$\text{pdf}(x) \;=\; k\,\frac{\cos\theta_x}{\cos\phi_z}, \qquad (\text{for } x > 0), \tag{14}$$

with $k = 1/128$.

Hence the distribution is proportional to the cosine of the angle to the light direction (numerator) divided by the cosine of the angle to the viewing direction (denominator), i.e. a lighting term modulated by a foreshortening term.

E. Lighting model for the planetary sphere, and a texture-mapping effect

The numerator in (14) is $\cos\theta_x$. This factor represents how much light is collected by the surface: if the surface is oriented square-on to the light ($\theta_x = 0$, so $\cos\theta_x = 1$), then the surface collects maximal light. And if it is side on to the light source ($\theta_x = \pi/2$, so $\cos\theta_x = 0$), then it collects zero light. This proportionality of the incident light to $\cos\theta_x$ is Lambert's cosine law (for incident light); see Fig. 7 and [22].

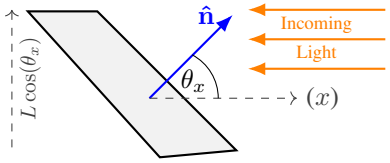


Fig. 7. Lambert's Cosine Law for incident light. The radiant energy collected by a surface patch is proportional to $\cos(\theta_x)$, where θ_x is the angle between the light source and the unit surface normal. As θ_x increases, the effective area facing the source decreases.

The numerator has accounted for the light incident to the surface. Next, we consider the light emitted from the surface. In computer graphics, a perfectly matte surface is a surface that reflects light perfectly diffusely; this is known as a Lambertian surface [22]. For a Lambertian surface, no matter where the camera or eye is located, the surface appears equally bright, as long as the relative angle between the incoming light and surface orientation remains the same [22]. This is not the case for the loading screen's planet surface, since the denominator of (14) makes an explicit dependence on the viewing angle ϕ_z. This denominator means that the brightness the viewer sees *increases*, the more side-on the surface is to their line-of-sight. This extra brightening requires that the surface reflects more light at oblique angles, rather than perpendicularly to its surface. The name for this kind of surface is a *grazing-boost microfacet surface* [23].

However, we feel that having a microfacet planet surface which generates exactly the required $\cos\phi_z$ denominator might be rather a stretch as a physical interpretation. So an alternative way to understand the denominator in (14) is to see the dot pattern as an actual spray of radiant dots on the planet's surface, and then the $\cos\phi_z$ denominator represents a geometric foreshortening of the dot pattern itself. An inclined dot pattern appears compressed by a factor of $\cos\phi_z$, due to foreshortening; increasing the apparent density of the dot pattern by $1/\cos\phi_z$ (see Fig. 8). This is analogous to *texture mapping* in computer graphics, where the width of an angled bitmap image is compressed by a factor $1/\cos\phi_z$ [24], [25].

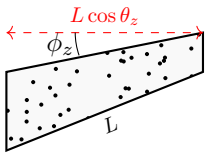


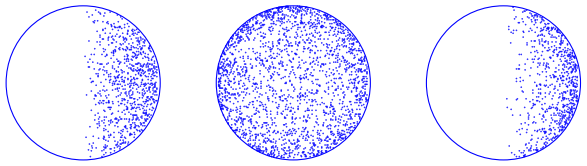
Fig. 8. A foreshortening effect compresses displayed dots into a smaller area, raising the final dot density. As the surface tilts away from the line of sight ($\phi_z \to 90°$), the dot pattern is foreshortened (by factor $\cos\phi_z$). However the same number of dots need plotting in this smaller screen area. This increases the dot density on screen, by a factor $1/\cos\phi_z$.

Hence we can understand the planet as a sphere sprayed with radiant dots, where the density of the dots on the planet's surface is proportional to the amount of light incident to the surface (i.e. proportional to $\cos\theta_x$); and then that distribution of dots is viewed using texture mapping. Together, these

factors reproduce the speckled distribution of points appearing in the loading screen's planet (Fig. 2).

F. Variations

We can investigate the relative importance of the numerator and denominator in (14) as follows. For example, if we only have the numerator present, then the plot looks like Fig. 9a (a true Lambertian surface).



(a) (b) (c)

Fig. 9. Different variations on probability distributions for the lighting model. a) pdf=$k\cos(\theta_x)$ b) pdf=$k/\cos(\phi_z)$ c) pdf=$k\cos(\theta_x)/\cos(\phi_z)$

Whereas if we only have the denominator present, then we only get the texture-mapping effect, shown in Fig. 9(b). Here you can see the points are more clustered towards the outside of the circle, which (with the assistance of a little imagination) shows a slight 3D curvature.

Finally, if we have both the numerator and denominator present, then we combine the above two effects, and get the shading pattern used in the *Elite* planet, giving a more three-dimensional looking sphere (Fig. 9(c).), with a higher concentration of pixels near the circle boundary.

It is not clear whether the *Elite* authors chose the texture-mapped image (Fig. 9(c)) over the more usual Lambertian surface (Fig. 9(a)) as their original goal, or whether it was a side-effect of the convenience of being able to use the single square-root function in (9b). As programming efficiency and final effect is what ultimately mattered, we speculate that convenience is what won out.

V. Conclusion

This paper has examined the Saturn loading screen in *Elite* as a self-contained case study in efficient 8-bit graphics programming. We have shown that the image is generated dynamically—rather than drawn from static bitmaps—through a sequence of modular routines that operate entirely with integer-friendly arithmetic and a centre-shifted screen coordinate system. The background stars are produced by uniform random sampling and a disk-exclusion filter that also introduces a thin atmospheric gap; the rings are constructed by concentrating samples into a diagonal band, testing membership in an elliptical annulus, and applying an occlusion rule to suppress rear arcs. Finally, the planet's disk is shaded via a single square-root transformation that re-shapes a uniform distribution into one consistent with Lambert's cosine law for incident light, and texture mapping to wrap the texture of random dots around the spherical surface, yielding the characteristic bright rim and convincing hemispheric shading.

While our analysis necessarily reflects an interpretation of low-level code rather than a direct record of the original design intent, the resulting model is mathematically consistent and visually explanatory.

Elite was a seminal game in computing history, and the loading screen produced a wow moment, enjoyed by a lot of gamers of that era; and in this paper we have explained the inner workings of that image. Impressively, a full 3D-lighting model, including probabilistic scattering of light, a 3D sphere, and texture mapping, are all packed into one square-root equation: line 6 of Alg. 2. We believe this single line of code is the highlight of this implementation, and it perhaps rivals the brilliance of the infamous inverse-square root method which appeared in early 3D games such as Quake 3 [26].

Methodologically, the loader's design demonstrates how careful formulation of integer inequalities (for circles and general ellipses) and judicious sampling transformations can deliver recognisable, high-impact visuals within severe resource constraints. The approach minimises computation by rejecting samples early, aligning candidate generation with target geometry, and exploiting simple vector interpretations of probability distributions to encode lighting and viewing effects without floating-point operations or polygonal rasterisation.

Acknowledgments

This article is an expanded version of our work at [19]. Microsoft Copilot was used during parts of this document's preparation. 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] The Guardian, "The 10 most influential video games of all time," https://www.theguardian.com/technology/gallery/2017/may/16/the-10-most-influential-video-games-of-all-time, May 2017, accessed 06 January 2026.

[2] 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.

[3] Frontier Astro, "BBC Elite - Acornsoft," https://www.frontierastro.co.uk/Elite/bbc_tape.html, accessed 13 February 2026.

[4] Codetapper/Action!, "An interview with Rob Northen," https://zakalwe.fi/~shd/amiga-cracking/rob_northen_interview.txt, 2011, accessed 13 February 2026.

[5] Ricardo Pinto, "An interview about elite z80 conversions," https://www.ricardopinto.com/2022/09/19/interview-about-elite-z80-conversions/, 2022, accessed 13 February 2026.

[6] "Ian Bell's Elite Pages," http://www.iancgbell.clara.net/elite/, n.d., accessed 06 January 2026.

[7] M. Moxon, "Elite on the 6502," https://elite.bbcelite.com, 2022, accessed 06 January 2026.

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

[9] 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

[10] M. Moxon and A. Duggan, "Fully documented source code for Angus Duggan's *Elite-A* on the BBC Micro," https://github.com/markmoxon/elite-a-source-code-bbc-micro, 2023, gitHub repository; Accessed 2026-01-31.
[11] C. Pinder, "Elite – the new kind (project background and downloads)," https://www.christianpinder.com/games/, 2024, accessed 2026-01-31.
[12] OoliteProject, "Oolite (main repository)," https://github.com/OoliteProject/oolite, 2026, gitHub repository; Accessed 2026-01-31.
[13] A. Gazzard, "The platform and the player: exploring the (hi) stories of elite," *Game Studies*, vol. 13, no. 2, 2013.
[14] D. E. Knuth, *The art of computer programming: Seminumerical algorithms, volume 2*. Addison-Wesley Professional, 2014, section 3.2.2.
[15] L. A. Leventhal, *6502 assembly language programming*. McGraw-Hill, Inc., 1986.
[16] Stardot Forums, "Forum post #349165," https://stardot.org.uk/forums/viewtopic.php?p=349165#p349165, 2026, accessed: 2026-02-04.
[17] D. Barrow, *Assembler Routines for the 6502*. London: Century Communications Ltd., 1985.
[18] "Personal computer world," *Personal Computer World*, vol. 6, no. 3, p. 186, Mar. 1983, march 1983 issue.
[19] "Drawing saturn on the loading screen," https://elite.bbcelite.com/deep_dives/drawing_saturn_on_the_loading_screen.html, 2023, BBC Elite Deep Dives. Accessed 06 January 2026.
[20] L. Devroye, "Nonuniform random variate generation," *Handbooks in operations research and management science*, vol. 13, pp. 83–121, 2006.
[21] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C (2nd ed.): the art of scientific computing*. New York, NY, USA: Cambridge University Press, 1992.
[22] J. H. Lambert, *Photometria sive de mensura et gradibus luminis, colorum et umbrae*. Augsburg: Christoph Peter Detleffsen, 1760. [Online]. Available: https://archive.org/details/lambertsphotome00lambgoog
[23] K. E. Torrance and E. M. Sparrow, "Theory for off-specular reflection from roughened surfaces," *Journal of the Optical society of America*, vol. 57, no. 9, pp. 1105–1114, 1967.
[24] E. Catmull, "Computer display of curved surfaces," in *Seminal graphics: pioneering efforts that shaped the field*, 1998, pp. 35–41.
[25] N. Greene and P. S. Heckbert, "Creating raster omnimax images," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86)*. ACM, 1986, pp. 65–68.
[26] C. Lomont, "Fast inverse square root," Tech. Rep., 2003, technical report analyzing the Quake III fast inverse square root algorithm. [Online]. Available: http://lomont.org/papers/2003/InvSqrt.pdf