Abstract

This paper explores the possibility of designing and programming games in higher-dimensional spaces, building on top of existing 4D rendering systems to create a novel ray-based implementation that attempts to make first-person manipulation of a 4D environment semi-intuitive. This implementation has the ability to render a variety of scenes in a relatively instinctive manner. Future work may be needed to expand to focus on more efficiently representing scene data to be passed to shaders and mapping 3D pixel solids onto 2D screens.

Interactive Visualizations in Higher-Dimensional Spaces

More than ever before, games and interactive simulations represent a growing part of people's lives globally, with more than 3 billion people actively playing video games in 2022 (Clement, 2022). Video games are a strong medium for the communication of otherwise complex ideas because of their widespread use, interactivity, and immersiveness. Mathematics, in particular, benefits largely from this idea, because topics such as non-Euclidean geometry are often non-intuitive in nature and yet are relatively easy to simulate programmatically. The nature of higher-dimensional space is one of these fields.

Several different implementations of higher-dimensional space in game design already have been implemented by game designers. One of the most popular, *Miegakure,* displays 3D cross sections of 4D space in order to create 4D puzzles that would be impossible to complete in 3D space, such as gathering objects from completely enclosed boxes (Miegakure). In terms of rendering techniques, however, the field still remains largely unexplored.

This paper combines and generalizes existing 3D ray-marching techniques developed by Inigo Quilez with other 4D rendering techniques in order to develop a novel 4D rendering approach for first-person graphics that could be used to create intuitive interactive programs.

A Gentle Introduction to 3D Rendering

The majority of 3D rendering systems used in games can be divided into ray-based systems and rasterized graphics. In both, scene geometry is defined in 3D space, which is then

projected directly or indirectly onto the camera plane. Different algorithms take different approaches to projection and lighting, which creates a variety of quality and speed in the rendering system.

Rasterized 3D Rendering

Rasterized rendering is one of the most common types of rendering implemented in real-time games. Rasterization relies upon describing a scene with 2D shapes such as triangles and then drawing individual triangles in the scene in sequence. This is useful because it allows the scene to be easily broken down into a number of different basic forms that can be relatively quickly rendered. However, it fails to often capture the realism of ray-based systems and fails to easily describe curved geometry.

Ray-Based 3D Rendering

In comparison to rasterized graphics, which loop through objects in the scene and then calculate which pixels they affect, ray-based graphics first loop through pixels to determine which objects should contribute to the resulting color.

To begin rendering in a ray-based system, a ‘ray’ is defined for each pixel on the screen. This vector represents the direction that a particular pixel is ‘pointing’ in much the same way that a particular point on a camera image was once a direction from the camera lens in 3D space. Each ray is checked for collisions with the scene and then rendered according to what object is hit. In the case of more realistic rendering, the ray may bounce multiple times to allow the simulation of effects like reflection and non-direct illumination. Ray-based rendering systems are

largely dependent on the type of system that is utilized to allow the ray-collision algorithm to calculate the position of a collision with the scene.

Sphere tracing (Hart, 1995) represents a robust method for rendering complex implicit 3D scenes in linear time. The algorithm makes use of a signed distance function (SDF) that is utilized to calculate the distance between a ray and the closest object in the scene. This distance is then repeatedly calculated and used to step the ray along its path the given distance. In this way, sphere tracing quickly converges to a collision with the scene while allowing the scene to be represented in a complex fashion.

Proposed Implementations of 4D Raymarching

The fundamentals of this proposed 4D rendering system builds upon the basics of sphere-tracing and generalizes the algorithm to the 4th dimension by simply adding additional components to all camera and ray vectors. The camera itself is represented by a three dimensional solid of ‘pixels’ that can be rendered. In order to actually create the display for the scene, one or multiple slices of the 3D pixel solid can be displayed to a computer's 2D display.

Representing Objects in the Fourth Dimension

In terms of points, representing higher-dimensional positions and vectors only requires adding an additional coordinate to traditional 3D points, w. The w axis extends perpendicular to the x, y, and the z axis. Rendering with ray-marching requires a function to estimate the distance between points and solids. Since sphere-tracing is most simply applied in 3-dimensions to spherical and cubic solids, it is natural that scenes in four dimensions can be represented with higher-dimensional equivalents.

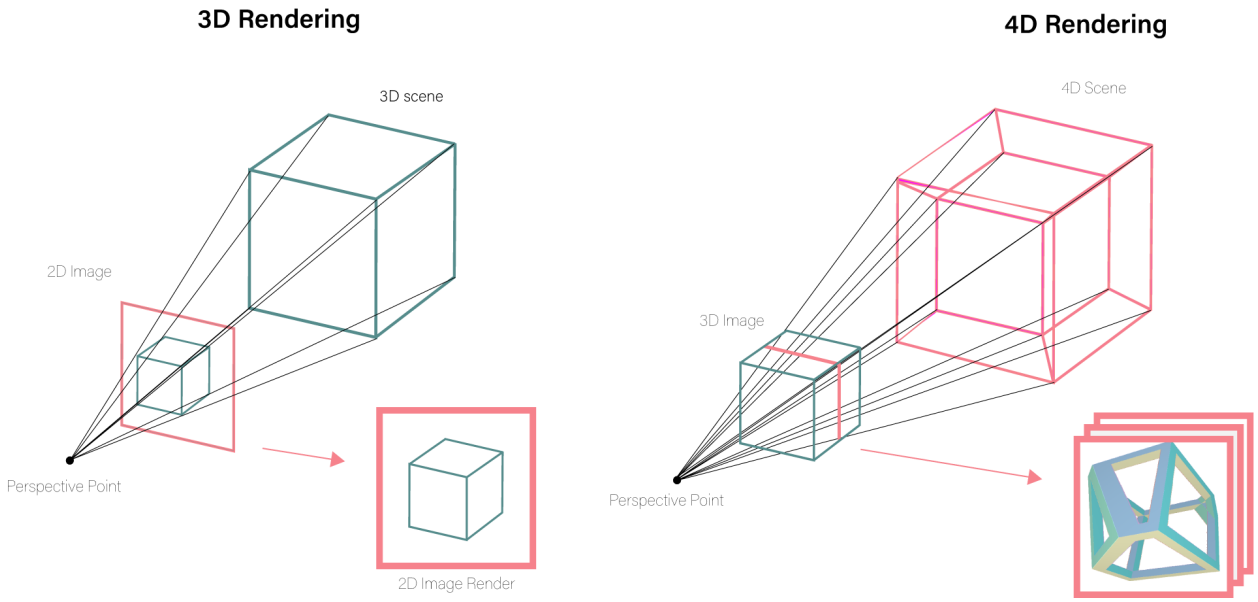
The distance estimators of simple 4D solids can be easily constructed based on known distance estimators for 2D and 3D solids. Using the definition of a unit sphere as a locus of points equidistant from a single point, the distance to a hypersphere can be calculated by simply calculating the length between a point and the center of the sphere and subtracting the square of the radius of the sphere. Since the *w* dimension is perpendicular to the other three dimensions, the distance from a point to an extended solid such as a cube or cuboid can be calculated by first finding the distance between the point and the solid at $w = 0$, and then using the pythagorean theorem to calculate the hypotenuse of the triangle with base lengths equal to that distance and the closest end of the solid pointing in the positive or negative *w* directions.

Representing a Camera in the Fourth Dimension

Accurately rendering the 4th dimension through a ray-based approach requires adding extra-dimensionality to the simulation of a camera system. In Flatland (Abbott, 2008), the main protagonist, a two dimensional creature, can observe the environment around him only through the forms of lines with the implied perception of depth. When introduced into the third dimension, he is suddenly able to view his surroundings with an additional axis of view. In either of these dimensions, a flat camera can ‘observe’ the world in exactly one less dimension than the physical space it is observing.

Likewise, the move to rendering 4D scenes requires the transition to 3D ‘screens’ to render this, as is shown in Figure 1. Similarly to how a the points in a 2D screen vary across two dimensions but remain constant in the other, the points in a 3D screen span across the first three dimension but must stay constant in the *w* dimension. Ray directions can be calculated by tracing normal vectors between the perspective point and the position of each pixel on the 3D ‘screen.’

Figure 1.



A true 4D rendering using this technique would result in a dense 3D object that represented a projection of the 4D scene. However, since 3D point-clouds are difficult or impossible to display on modern hardware, an additional step is needed to make this render practical for use in interactive simulations. Most simply, a single or multiple 2D slices can be taken from the resulting render and displayed on a normal computer screen. This does neglect parts of the render, but this can be mitigated by taking an increasing number of slices and displaying them concurrently.

Scene Estimation and Lighting

Once the camera rays have been generated for a particular pixel in a 3D viewport, a collision between the ray and scene is needed to generate an actual pixel color. The scene

collision itself can be calculated through the application of signed distance functions as previously mentioned, trivially generalized to the 4th dimension.

Many typical lighting techniques used in ray-marching, such as taking the dot product between the light vector and a surface normal to determine total luminosity, do not require explicit dimensionality to function, and therefore can be easily applied to light and color 4D surfaces. Additional techniques can be applied to enhance realism using the position of the collision, the normal of the collision surface, and typical ray-marching and ray-tracing techniques.

Movement and Rotation in the Fourth Dimension

Like any interactive simulation, four-dimensional simulations must include intuitive controls in order to be educational and usable. Although there are only three rotational directions in traditional 3D space, the fourth dimension has six separate simple rotational directions. Three of these correspond similarly to rotation in 3D, and the controls can be treated as such. If one is to assume gravity and disregard the two rotations that misalign the camera from perpendicularity to the ground plane, then the four remaining rotations can be divided into two rotations on each of two distinct track pads or joysticks.

Alternatively, on a laptop or computer, scrolling or a set of buttons can be allocated to one of the remaining non-traditional turning directions while two of the traditional ones are allocated to mouse controls in the form of standard 3D game controls. This allows more intuitive control of the player when rotation outside of the XYZ dimensions is not required and yet preserves most of the rotations needed to navigate in a four dimensional space.

Algorithmic Analysis

The rendering algorithm performs relatively well for simple scenes when implemented into a GLSL shader, using JavaScript for game inputs and scene states, allowing real-time rendering and interaction with four dimensional platforms and blocks. Although only displaying a single slice from the resulting 3D render to the screen disregards a large amount of potential data in the camera's field of view, it affords an image that is most similar to a 3D render, and therefore allows interactions with 4th dimensional simulations to occur similarity to interactions in 3D computing environments.

Practical Issues

One large issue for the proposed algorithm is the lack of algorithms to adequately represent a 3D solid of colors on a 2D screen. Although slicing the image is the most intuitive, it becomes unwieldy and unintuitive for users when a large number of slices is utilized. Meanwhile, a small number of slices means that the physical screen only displays a very small portion of the viewport. Potential implementations of 4D games in the future may be able to use virtual reality displays or other forms of three dimensional physical displays to truly exhibit true 3D data as resulting from a 4D render.

Conclusions

This paper explores one potential novel combination of ray marching and first-person projections for a four dimensional display and finds that visualizations of four dimensional spaces for games and simulations on modern equipment is feasible and has the potential to be semi-intuitive. Although issues persist with the rendering technique and its completeness, it

represents a step towards the adoption of more non-Euclidean computer explorations as norms in the game industry.

References

Abbott, E. A., & Jann, R. (2006). Flatland: a romance of many dimensions. Oxford, Oxford University Press.

Clement, J. (2022). Number of video game users worldwide from 2017 to 2027. *Statistica*. Retrieved from www.statista.com/statistics/748044/number-video-gamers-world.

Inigo, Q. (2022). Signed Distance Functions. Retrieved from iquilezles.org/articles/distfunctions.

Jiarathanakul, P. (n.d). Ray Marching Distance Fields in Real-time on WebGL. *University of Pennsylvania*. Retrieved from citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a964750aa212bd490d258221bc9756e7e58c5317.

Miegakure. Miegakure [Hide and Reveal]. Retrieved from miegakure.com.