2017-08-01

A General-Purpose Animation System for 4D

Justin Alain Jensen
Brigham Young University

A General-Purpose Animation System for 4D

Justin Alain Jensen

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Robert P. Burton, Chair
Parris K. Egbert
Seth R. Holladay

Department of Computer Science

Brigham Young University

ABSTRACT

A General-Purpose Animation System for 4D

Justin Alain Jensen
Department of Computer Science, BYU
Master of Science

Computer animation has been limited almost exclusively to 2D and 3D. The tools for 3D computer animation have been largely in place for decades and are well-understood.

Existing tools for visualizing 4D geometry include minimal animation features. Few tools have been designed specifically for animation of higher-dimensional objects, phenomena, or spaces. None have been designed to be familiar to 3D animators. A general-purpose 4D animation system can be expected to facilitate more widespread understanding of 4D geometry and space, can become the basis for creating unique 3D visual effects, and may offer new insight into 3D animation concepts.

We have developed a software package that facilitates general-purpose animation in four spatial dimensions. Standard features from popular 3D animation software have been included and adapted, where appropriate. Many adaptations are trivial; some have required novel solutions. Several features that are possible only in four or more dimensions have been included. The graphical user interface has been designed to be familiar to experienced 3D animators. Keyframe animation is provided by using a set of curves that defines movement in each dimension or rotation plane. An interactive viewport offers multiple visualization methods including slicing and projection. The viewport allows for both manipulation of 4D objects and navigation through virtual 4D space.

Keywords: 4d animation, four-dimensional geometry, general-purpose animation system

ACKNOWLEDGMENTS

I thank Dr. Robert P. Burton for his guidance and enthusiasm during this process. His method of teaching helped me grow and his praise inspired me to aim higher. He helped remove unnecessary obstacles. I also thank him for opening my eyes to the 4th dimension.

I am very grateful for the funding provided by the Department of Computer Science, the College of Physical & Mathematical Sciences, and the generous donation from an anonymous former member of the Hyperdimensional Research Group at BYU.

Finally, I thank my wife, Shelly, for her unending support throughout this adventure. She kept me going when I became discouraged and constantly reminded me of why we chose to do this. I would not have made it here without her.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 History of Animation

Animation has been a medium for storytelling since the early 1800s, beginning with the work of Joseph Plateau and his phenakistoscope [46]. His device, a round piece of cardboard with a series of pictures and thin slits, was extremely crude even when compared to the technology of the early 1900s. As crude as it was, it helped to spark the imagination of scores of individuals who helped develop the art and technology of animation. Animation is well-suited for communicating ideas, both concrete and abstract. Even from the very earliest days of traditional animation, artists have endeavored to tell stories in a manner often not possible with any other medium. Abstract forms, metamorphosis, and fictitious creatures have been frequent themes. Animation is not limited by real-world facts and laws. Because of this, an animator has the freedom to tell almost any story or to convey almost any information.

1.2 History of Popular Interest in 4D

The exploration of four spatial dimensions was a topic of interest even before 1884 when Edwin A. Abbot published his satirical novella, Flatland. Interest continued through the turn of the century when in 1909 Scientific American published an essay, The Fourth Dimension Simply Explained, which was the result of a worldwide contest to produce the best article to explain four-dimensional space quickly and accurately to a lay person [37]. In pursuit of the $500 prize, essays from around the world were submitted and reviewed. This contest and subsequent publication bolstered the public's interest in the topic. This was followed by a 1913 book entitled A Primer of Higher Space in which Claude Bragdon explains the plausibility of a higher dimensional universe and some possible implications [13]. In the intervening period, substantial work has been done to render objects in 4 spatial dimensions using computer graphics [3, 15, 17, 25, 28, 29, 35, 51, 52]. A. Michael Noll at AT&T Bell Laboratories was the first to animate 4D objects [42]. Since Noll's original work, many attempts have been made to continue developing techniques for 4D animation [8, 9, 14, 21]. However, all attempts have yielded simplistic, perfectly geometric results. Conspicuously absent is a general-purpose tool for animation of higher-dimensional objects, phenomena, and spaces.

1.3 Modern Interest in 4D

Individuals from the film industry have expressed interest in hyperdimensional phenomena. Michael Wadleigh, a director and cinematographer for Lorimar, approached the Hyperdimensional Research Group at Brigham Young University to request its assistance in identifying the capabilities of the 4D protagonist and antagonist for the prospective feature-length film, The Mirror Man. The Group responded by producing a stereo motion picture depicting several hypothesized 4D phenomena. Allen Hall, a recipient of an Oscar award for special effects, approached the Hyperdimensional Research Group for clarification of 4D phenomena for a scene in a feature-length motion picture.

In 2014 the film Interstellar was released which included concepts of relativity and 4- and 5-dimensional spaces. This film was nominated for five Academy Awards and was the winner of the Visual Effects award in 2015 [1]. A forthcoming video game, Miegakure, is being developed by independent game developer Marc ten Bosch. This platformer game involves a 3D protagonist who has the ability to travel through four spatial dimensions to solve puzzles that would otherwise be unsolvable in 3D. The game became widely-known in 2010 when the popular webcomic xkcd made reference to it [40]. The game, though not yet released, has generated a large following of people who are excited to interact with a hypothetical 4D world. These two recent developments involving hyperdimensional concepts are an indication of a growing fascination with the subject.

1.4 Categories of 4D Animation

1.4.1 Space-time Animation

4D spacetime animation is a technique that already is found in several software packages. It involves three-dimensional geometric shapes that vary over time. It is different from conventional 3D animation in some key ways. In 3D animation, an object's form and topology generally remain fixed; the only way to change an object over time is to apply a transformation to it or to impose some sort of deformation on it. Changing the topology is usually a difficult process and can involve continually replacing the object from frame to frame, giving an illusion of a topological change. In 4D spacetime animation, an object can change over time, even topologically [4]. The most common application of this technique is found in construction planning software. In planning the construction of a building, a modeller creates a series of 3D shapes which represent the building in its various stages of construction. He can assign a specific time value to each stage. When viewing the animation, a viewer sees what the building looks like as the project progresses from beginning to completion.

In spacetime animation, the fourth dimension is temporal, not spatial. David Banks points out why this technique cannot be classified as true 4D animation. To help understand, he uses the following lower-dimensional analogy. A flipbook is a series of 2D drawings arranged in a stack and ordered chronologically. One could think of this as a 3D spacetime animation where time is the third dimension. As a consequence of rotating the animation so that the temporal dimension is swapped with a spatial dimension, the resulting animation would neither make sense nor resemble the original [9].

1.4.2 Cross-secting 4D Datasets

Many interesting problems can be solved with the help of high-dimensional data sets. Each point of data can be represented by any number of values, each value being a dimension of the set. Information can be gleaned from a dataset by analyzing it with the help of various visualization techniques. These techniques typically reduce the data to a two-dimensional form so that it can be displayed on a conventional 2D computer screen. Each new visualization technique differs from all others by providing features for gaining insight into the data that the others cannot. These applications are not tied to a specific number of dimensions. It is equally plausible to process a 4D dataset as it is to process a 7D or 11D dataset.

Animation in this context is generally a secondary feature. With certain visualization techniques, animation helps to enhance understanding of the grouping of data points and the overall shape of the data. Seeing the data from multiple perspectives, especially if the data is in continuous motion, is helpful for obtaining additional insight from the visualization [36]. In other cases, animation serves only as a visually-appealing transition from one set of data to the next; its purpose is purely aesthetic. Further, a problem arises when trying to interpolate between certain values. Some of these dimensions are not part of a continuous spectrum; a dimension may be textual information. Interpolation and animation of textual data is not well-defined and would not yield useful information. For these reasons, this animation technique cannot be called true 4D animation.

1.4.3 Animation of 4D Geometric Shapes

The animation of 4D geometric shapes can be considered true 4D animation since it deals with four spatial dimensions. It is the most natural extension of the concepts and techniques of modern 3D animation. Each four-dimensional object is described by a set of vertices, edges, faces, and cells. Each vertex has four numerical values to indicate its position in space: X, Y, Z, and W. Arbitrary affine transformations can be applied to objects in order to translate, scale, rotate, or otherwise transform them. Animation is achieved by transforming over time.

Similar to the animation of 4D data sets, a visualization technique is required to represent a 4D scene with a 2D image. Several visualization techniques attempt to solve this problem and are described later. This definition of 4D animation is the one that is taken here.

1.5 4D Space

In section 1.4.3, we narrowed the definition of 4D animation to the animation of 4D geometric shapes. This section includes additional discussion about the 4th spatial dimension, a method for gaining intuition about 4D space, as well as the explanations of a several concepts that are referred to throughout this thesis.

1.5.1 Some 4D Terminology

A concept that is referred to frequently in this thesis is the 3D Hyperplane. The 3D hyperplane is the infinite 3D space such as the one in which we live. Just as a plane can intersect a 3D object and produce a 2D cross-section, a 3D hyperplane can intersect a 4D object and produce a 3D cross-section. Multiple 3D hyperplanes can be stacked in the 4th dimension. Compared to 4D space, a 3D hyperplane is flat. They can be arbitrarily close in the 4th dimension, yet objects in one 3D hyperplane are restricted to their hyperplane and cannot interact with or observe objects in another hyperplane if they do not intersect.

Often when discussing a 4th spatial dimension, we use the words up or down to denote relative position in the 4th dimension. However, using these terms introduces ambiguity to the discussion; without context, it is possible to interpret up or down as directions in either 3D space or in a 4th spatial dimension. To remove this ambiguity, Charles Hinton coined the words Ana and Kata, which correspond to the concepts of up and down respectively, but in the 4th dimension [27]. These terms are used in the remainder of this thesis.

1.5.2 Using Lower-Dimensional Analogies to Understand 4D

4D space is notoriously difficult to understand. As far as we know, the universe has only three spatial dimensions, so 4D space is purely theoretical and is completely outside of our experience. With our experience limited to only three spatial dimensions, it is nearly impossible to reason in 4D. There is one way of thinking, however, which can aid an inquisitive explorer in gaining some intuition for 4D space. The method is that of using lower-dimensional analogies. When faced with a difficult question about 4D space, a reader might be able reduce the number of dimensions, discover a solution there, and attempt to extend the solution to 4D. This approach is used by Edwin Abbott in [2], Paul Isaacson in [30], Cliff Pickover in [43], and many others.

1.5.3 Hypothetical Interaction Between 3D and 4D Objects

An animator may want to depict the interactions between 3D and 4D objects. This is a common theme in fictional stories that feature 4D objects [10, 43, 49]. Reasoning about the hypothetical interaction between 3D and 4D objects is useful because it can aid us in designing plausible stories about these interactions. Since 4D space is purely theoretical, a few reasonable assumptions must be made. First, we assume that 3D objects can physically touch 4D objects. If this is the case, the 4D object may feel the 3D object's touch, but it would be nothing more than a pinprick since the 3D shape is perfectly flat compared to the 4D shape. Second, we assume that 3D objects can apply sufficient force to move or rotate 4D objects. In this case, the 3D object can apply only 3D transformations to the 4D object. Because it is restricted to its 3D hyperplane, there is no way for it to push a 4D object Ana or Kata. Third, we assume that 4D shapes can physically touch 3D objects. If this is the case, the 4D object can see and touch every part of the 3D object simultaneously, both inside and out. Also, the 4D object can apply any 4D transformation to a 3D object, including one that would remove it from its 3D hyperplane or rotate it in some manner not possible in 3D. If this happens, the appearance of a 3D object can change dramatically, at least from the perspective of a 3D observer. For example, if the 3D object is pulled straight Ana or Kata, it will instantly disappear from view. If it is then moved parallel to the hyperplane, then put back onto the hyperplane, it will instantly reappear somewhere else. As another example, if the 3D object is able to bend in the 4th dimension and a 4D object grabs it on one end and pulls it Ana or Kata, the 3D object will disappear gradually, starting from one end and progressing to the other end. Finally, if the 3D object is rotated 180 degrees in either the XW, YW, or ZW rotation planes (see section 3.2.1), it will appear as if the object has transformed into a mirror image of itself. These phenomena and others are described in more detail in [30, 43].

1.6 Value of 4D Animation

1.6.1 Basis for Creating 3D Visual Effects

A practical application of 4D animation is the creation of unique 3D visual effects. After reviewing several 4D animations, two things become clear: 1) their movement and appearance are intriguing, and 2) mimicking this movement would be extremely difficult with existing 3D methods. These visual effects can be created using 4D techniques and then incorporated into traditional 3D-animated films, adding value to an animation studio's toolkit.

1.6.2 New Insight Into 3D Animation Concepts

In the process of creating animations in four dimensions, an investigator may gain facility or insight into observed, but heretofore unexplained phenomena in 3D. Consider, as a simple example, the translation of a point through 3D space using a composable transformation matrix. This is not possible with a 3x3 matrix; it requires a 4x4 matrix for transformation of 3D data represented in 4D. This 3D problem is solved elegantly in 4D. Similarly, the rearrangement of the sine and cosine rotation matrix elements, in particular the migration of the minus sign, commonly are perplexing to students new to computer graphics, is clarified when the same problem is solved in 4D (see Appendix A.3). Similar insights reasonably remain undiscovered.

1.6.3 Visualization of the 4th Dimension

4D animation provides the opportunity to achieve an intuitive sense of a 4D environment. This sense may include an intuitive feeling for navigation through a 4D space [55], what simple, animated 4D hyperobjects look like, an ability to perceive rigid objects as rigid under rotation and projection, an exploration and discovery of what is possible in a 4D space that is not possible in 3D and vice versa, and how a 3D being might perceive or experience such phenomena. The existence of 4D phenomena can only be hypothesized. It has in fact been hypothesized, it lends itself to convenient modeling of lower-dimensional phenomena, and the existence of such phenomena cannot be dismissed out of hand. Some aspects of each of these activities have a trivial extension from three dimensions to four. However others, such as rotation and the vector cross product and certain 4D phenomena such as Double Rotation, and certain concepts of Knot Theory, either fail to extend in an obvious way or simply don't exist in 3D. Being able to animate a 4D environment will help clarify and refine these concepts.

Being able to comprehend higher-dimensional problems offers advantages. It allows individuals to think about problems in new ways, often accommodating the dimensionality of the problem space. There are countless problems that involve a high number of dimensions. Having an intuitive sense for high-dimensional space will aid in solving these problems. Michio Kaku, world-renowned physicist, postulates that most of the basic laws of physics, which are seemingly unrelated and independent of each other, could be unified when observed in the context of higher-dimensional spaces [31].

Chapter 2

Related Work

2.1 Techniques for Interaction with 4D Geometry

2.1.1 Rotation

Unlike 3-space where there are 3 mutually perpendicular principal axes and 3 principal planes of rotation, 4-space has 4 mutually perpendicular principal axes and 6 principal planes of rotation. Because of this, creating a graphical interface for setting rotation in four dimensions is not a trivial extension of the conventional methods for doing so in three dimensions.

Explicit setting of rotation value: A simplistic approach to this interface is implemented by Paul Bourke in HyperSpace [12]. In HyperSpace, rotations are set using a series of six sliders, one for each rotation plane. The slider values range from -180 to 180 degrees, encompassing a full rotation of the plane. While this interface allows for full customization of a rotation, it is cumbersome because there are six individual sliders to manipulate. Also, this method does not promote kinesthetic correspondence, or the correspondence between a hand motion and the motion of a virtual object, that would otherwise aid users in more quickly gaining an intuition of rotations in 4-space [16].

Input mapping: A second approach is to map the rotation value in each of the six rotation planes to 2D mouse movement. Since there are two degrees of freedom with a conventional mouse, the user can control two rotation planes at any given moment. He can select which rotation planes to manipulate pressing and holding specific keyboard or mouse buttons while moving the mouse. This method is used by D'Zmura et al. [20] in an interactive 4D game. The Rolling Ball technique is a more significant variation to input

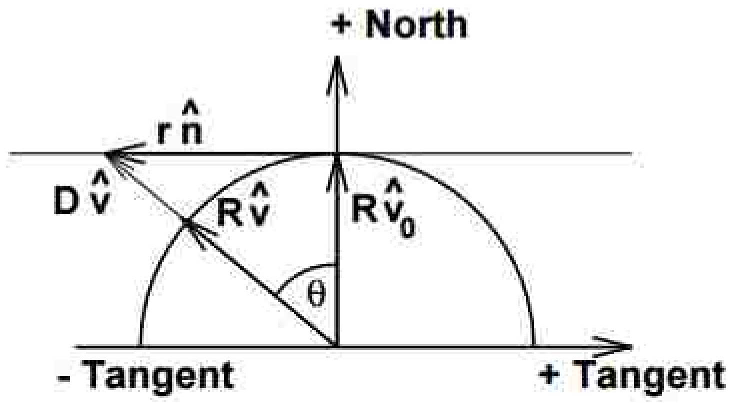


Figure 2.1: A diagram of the 2D version of the Rolling Ball technique [24]. The vector $r\hat{n}$ is an n-1 dimensional unit vector which points in the direction of rotation of the nD hypersphere.

mapping. This technique was originally developed for 3D rotation, and subsequently extended to n-dimensional rotation by Hanson. The method's name describes the method itself. To rotate an nD ball, the user envisions an n-1 dimensional plane tangent to the "north pole" of the ball. He then chooses a unit vector in the plane. The ball rotates in the direction of that unit vector. The 4D version requires three input parameters which can be provided by either a 3D mouse or a 2D mouse by using a mouse button to switch between the XY and XZ planes. The three parameters can then be manipulated and combined to create a 4x4 rotation matrix which is then applied to the 4D object [24].

Tetrahedral control space: A novel approach to interactively setting 4D rotations is using a tetrahedral control space method. The four vertices of a regular tetrahedron are used to represent the four principal axes of four-dimensional space. Each edge represents a rotation plane of the two axes whose vertices are the endpoints of the edge. For example, the edge between the X and W vertices represents the XW rotation plane. A user may select a vertex, a point on an edge, or a point on any of the four faces of the tetrahedron. The position of the selected point relative to each edge determines the amount of rotation in the plane the edge represents. If the point lies on an edge, the rotation occurs only in that plane. However, if the selected point lies on a triangular face, the rotation occurs in the three planes represented by the edges that surround the face. For example, a point selected in the YZW triangle will apply a rotation in the YZ, YW, and ZW planes. David Prabhat implemented this technique in both a desktop application and a CAVE environment. In the desktop version, the user uses a standard mouse to select a point on an on-screen tetrahedron. In the CAVE, the user uses a 3D input device to select the point [45]. This technique allows a user to quickly and accurately specify a rotation using standard computer equipment. However, since only one point on the surface of the tetrahedron may be selected, a disadvantage of this technique is that rotation is limited to either one or three planes, meaning that the user selects a point on an edge or a face, respectively.

2.1.2 Translation

Translation factors generally are set explicitly. Often they are not directly accessible since one of the limitations of many applications is that the object is always centered in the screen.

2.1.3 Scaling

Scaling is usually performed uniformly in all dimensions. Because of this assumption, extending interactive scaling to any number of dimensions is trivial. A one-dimensional input (e.g. moving a mouse left and right) is sufficient to apply a uniform scaling.

2.1.4 Shearing

Shearing transformations are not usually easily-accessible in software if implemented at all. Often, the only way to apply a uniform shearing transformation is to create a custom transformation matrix, assuming custom matrices are supported by the software.

2.1.5 Reflection

Reflection is more commonly used as a modeling tool in the context of animation software. Reflection occurs across an n-1 dimensional hyperplane. The reflection operation ensures that the transformed shape and all of its components are the same distance from the hyperplane as they were before the transformation. Additionally, the hyperplane separates the reflected shape from the original shape [47].

2.2 Existing 4D Animation Software

2.2.1 Criteria for 4D Animation Systems Reviewed

For a software system to qualify as a valid 4D animation system, it needs to meet the following criteria:

- Cartesian visualization
  - Rationale: We are concerned solely with animating four-dimensional geometric shapes in Cartesian four-space, not visualizing other higher-dimensional spaces, multivariate datasets, or three-dimensional spacetime objects.
- Programmable
  - The user needs to be able to provide input to the system. Data could be in the form of 4D shapes, custom scripts, or saved files (created by the system for future use).
  - Rationale: The ability to handle user-provided input is evidence of a non-trivial system. It is relatively easy to create a visualization of a rotating hypercube, but that doesn't qualify as an animation system.
- Visualize 4D geometry
  - The software needs to be able to render a visualization either to the screen or to an image file.
  - Rationale: Visualization is a critical component of animation. Animation without visualization is just simulation.
- Sufficient information is available

  - Either the software/source code must be publicly available for download (and it needs to run on a modern computer) or there must be a publicly available detailed description of the features of the system and other relevant details (such as a user's manual or a detailed description of the software)
  - Rationale: Without sufficient information about the system, it is not possible to make fair and accurate comparisons with other systems.

- Reasonable general-purpose capability

  - Rationale: Several simple applications are able to present 4D geometry, animate transformations, and even provide interactivity. However, these applications are highly specialized, providing only enough 4D capability to satisfy the needs of the application, with little or no thought for generalizability.

2.2.2 Stella4D

Stella4D is a geometry exploration tool. It was designed mainly for mathematicians and others interested in 3D and 4D geometry. It claims to have animation capabilities. However, these are limited to animated transitions between shapes. Very little customization of animation is possible. It does, however, have good real-time visualization capabilities. Another useful feature is the automatic generation of 4D nets, or 4D shapes 'unfolded' into 3D space.

4D-to-3D Perspective: Stella4D includes a useful variation of the 4D-to-3D perspective projection technique. Its purpose is to allow solid shading of a polychoron while retaining some ability to view internal geometry. The polychoron is projected to 3D and displayed as a wireframe object. Each cell is also rendered with opaque surfaces, but individually scaled down so that there are gaps between each. The user can control the scaling factor to allow either more or less of the internal geometry to be shown.

4D Unfolding: Stella4D is capable of 4D unfolding, a concept described in [30]. The 4D nets are automatically generated without the aid of a user. However, the user may alter the net by selecting two cells that were adjacent in the polychoron and indicating that they should remain adjacent in the 4D net. This action recalculates the 4D net and preserves the adjacency the user desires.

4D Tumbling (rotation): Interactive rotation is achieved using a combination of keyboard keys and mouse movement. All rotation occurs relative to screen space instead of the principal axes. By pressing Shift and left-dragging (holding the left mouse button while moving the mouse), rotation occurs in the XW plane and the YW plane in response to right-left and forward-backward movements of the mouse, respectively. By pressing Shift and right-dragging, rotation occurs in the ZW plane. Rotation can occur in the XZ and YZ planes by left-dragging while not pressing any keyboard keys. Rotation in the XY plane occurs by middle-dragging.

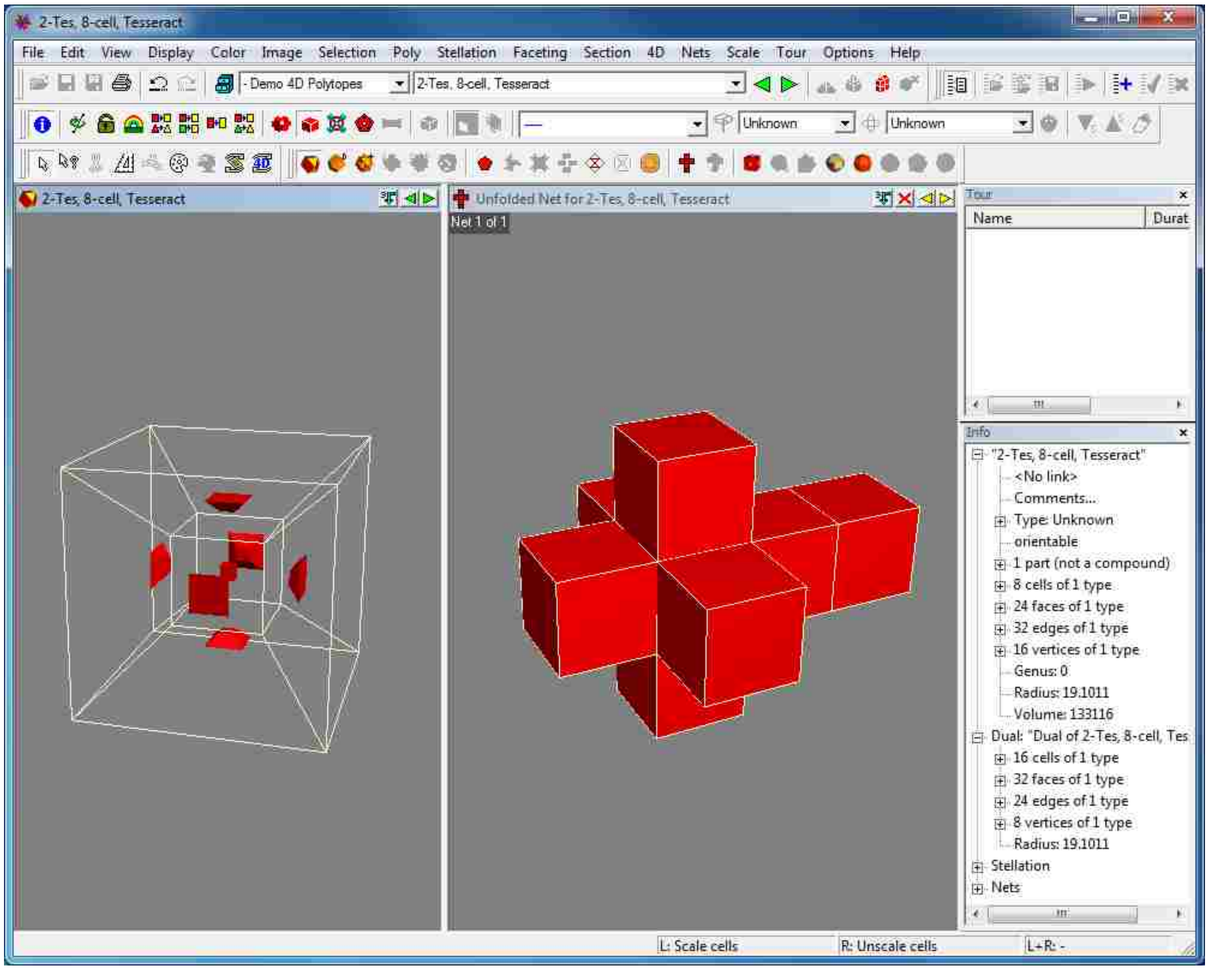
2.2.3 HyperSpace

Hyperspace[1] is a 4D geometry exploration tool similar to Stella4D but with a smaller feature set. It is a simple tool that is designed to teach the user about 4D geometry via interaction with several visualizations. It does not claim to possess any animation features and is not suited for animation. It does, however, have visualization features not found in any other 4D software reviewed in this proposal.

Multi-view visualization: The greatest strength of Hyperspace is its variety of visualization methods. There are two main modes: projection and cross-sections.

In projection mode, the shape is rotated according to a user-defined rotation, then projected directly from 4D to 2D. The user may either select one of the six projection planes or he may view all six projection planes simultaneously. When viewed simultaneously, the view is split into six equally-sized regions and the shape is projected onto each plane. For convenience, the axes of each region are labeled XY, XZ, XW, etc. The shapes are displayed

[1]Due to the age of the software and the dearth of documentation on this software, we are unable to discover the full details of this software. The information included above has been obtained from the user's manual and from an email conversation with the creator of HyperSpace.



```
"2-Tes, 8-cell, Tesseract"
  <No link>
  Comments...
  Type: Unknown
  orientable
  1 part (not a compound)
  8 cells of 1 type
  24 faces of 1 type
  32 edges of 1 type
  16 vertices of 1 type
  Genus: 0
  Radius: 19.1011
  Volume: 133116
Dual: "Dual of 2-Tes, 8-cell, Tes
  16 cells of 1 type
  32 faces of 1 type
  24 edges of 1 type
  8 vertices of 1 type
  Radius: 19.1011
Stellation
Nets
```

Figure 2.2: A screenshot of Stella4D. Notice the hybrid wireframe/solid shading. The interface is mostly devoted to exploring one shape at a time and the animation features are limited, making it insufficient for general-purpose animation.

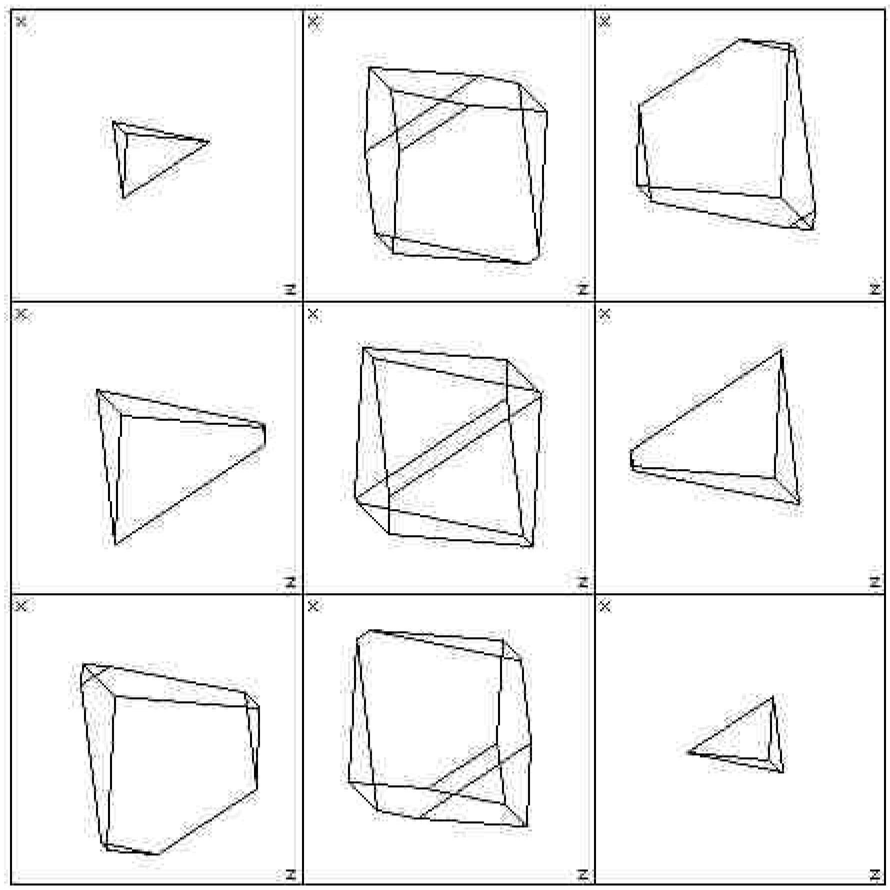


Figure 2.3: A screenshot of HyperSpace simultaneously displaying nine parallel slices of a hypercube. The software is devoted to geometry exploration and lacks many animation features.

as either a wireframe mesh or flat-shaded. If displayed as a wireframe, the user has the option to enable a depth cue which shades each edge according to its distance along a selected axis.

In cross-section mode, the shape is once again rotated according to a user-defined rotation. The view is partitioned into nine equally-sized regions. Nine cross-sections are taken along the W axis of the object, each of which is equally-spaced along that dimension. The cross-sections are then projected onto the XZ plane. This method allows a user to gain a better understanding of the true form of the shape, while avoiding the problem of hidden internal geometry, but without clear correlation among the nine figures.

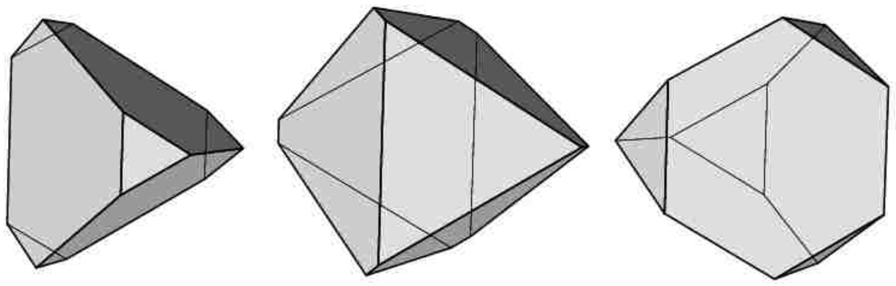


Figure 2.4: A series of cross-sections of a hypercube, sliced at different depths. Note the presence of the line thickness depth cue.

2.2.4 Peek

Peek was a student project developed by Gordon Kindlmann at the University of Utah. Similar to HyperSpace, Peek is a geometry exploration tool designed to aid the user in understanding 4D geometric shapes. It has simple animation capabilities but offers no interactivity. When the program is executed, it loads a file containing the geometry definition and renders the visualization to a PostScript file for later viewing. It features two depth cues not found in any of the other 4D systems listed: Hue: When the geometry is shaded using flat shading (as opposed to wireframe), each face is color-coded based on its distance from the viewpoint in the W axis. Hues are taken from the spectrum of visible colors ranging from red to blue. Red faces are nearest to the viewpoint in the W axis while blue faces are furthest. Green faces are halfway between the nearest and furthest points of the object on the W axis. The hues of all intermediate positions are linearly-interpolated between red and green or green and blue if they are between the nearest and midway points or midway and furthest points, respectively. Line thickness: Line thickness as a depth cue is used only in wireframe visualizations of the geometry. Hidden edges are drawn thinner than visible edges. Thickness for each edge is constant from one endpoint to the other.

2.2.5 Meshview

Meshview was developed under the direction of Andrew Hanson. Because of its feature set and intended purpose, we consider it a true 4D animation package. Its main purpose

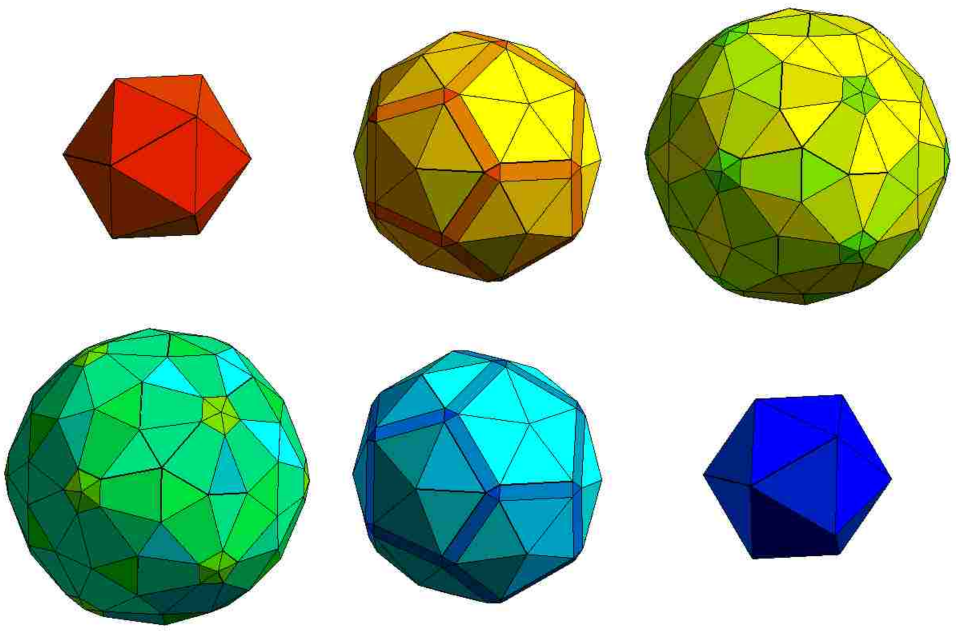


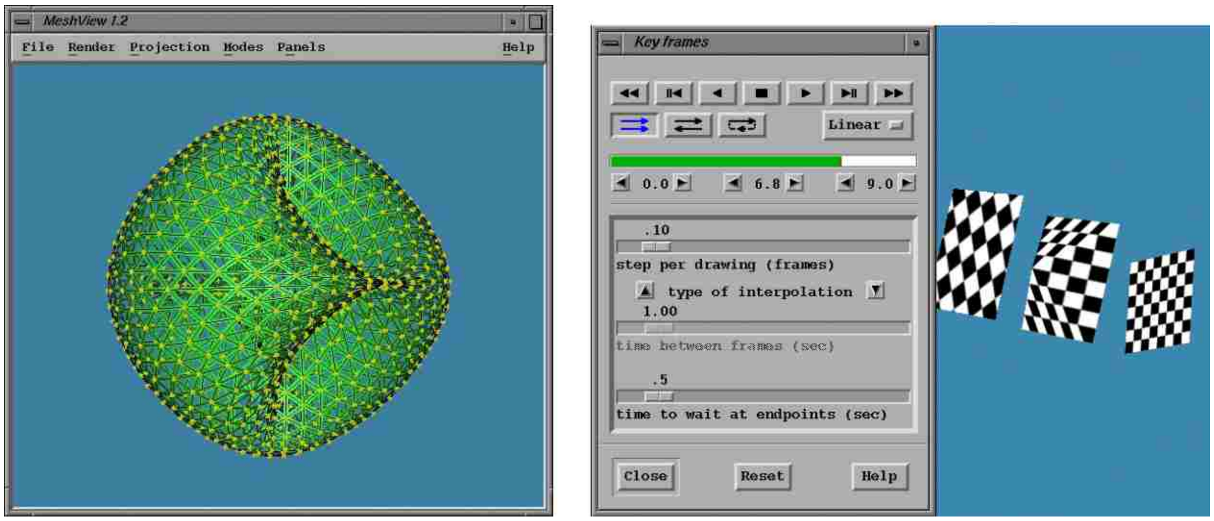
Figure 2.5: A series of cross-sections of a 600-cell, sliced at different depths. Note the varying hue of each face, indicating depth in the W axis.

is twofold: first, it is used visualize four-dimensional geometric shapes and mathematical functions, and second, it is used to create animations of 4D geometric shapes which are used to assess a viewer's ability to gain an intuitive understanding of the transformations being depicted.

Real time visualization: This software includes a useful shading method dubbed Screen Door Transparency. The geometric shape appears as a wireframe mesh but the edges retain the smooth shading of the shape's surface. Vertices are also rendered on top of the wireframe. This shading method combines both wireframe and smooth shading so that the internal geometry is visible but the curvature of the surface is also discernable. Conventional shading methods, including smooth shading and texture mapping, are also available.

4D-to-3D Projection: Meshview supports both perspective and parallel projection from 4D to 3D.

Keyframe Interpolation: Keyframes are stored as individual shapes. The user may load each shape as a different keyframe and interpolate between each. Interpolation occurs



Linear

0.0 6.8 9.0

.10
step per drawing (frames)

type of interpolation

1.00
time between frames (sec)

.5
time to wait at endpoints (sec)

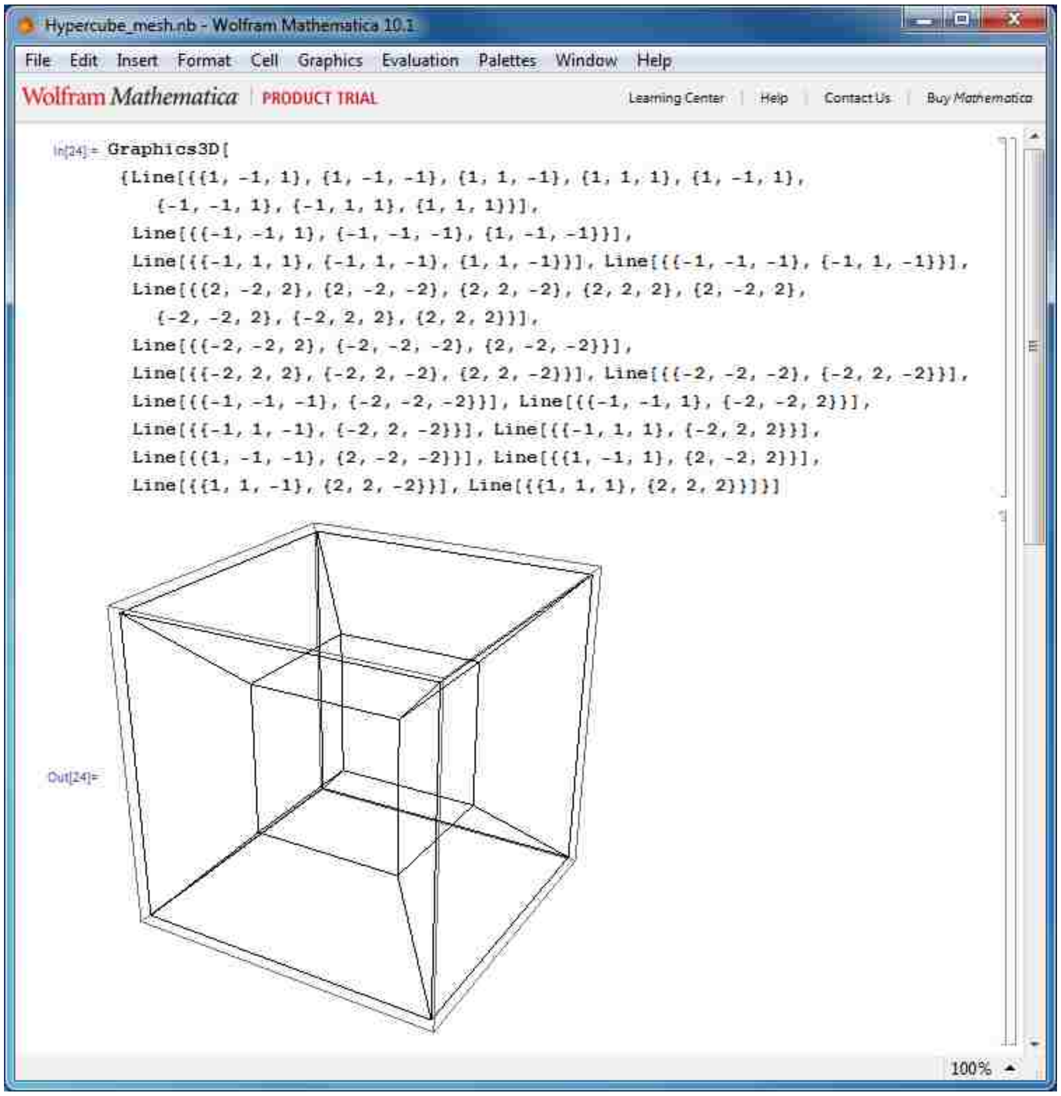
Figure 2.6: A screenshot of Meshview. The image on the left shows the screen-door transparency shading mode. The image on the right shows the key frame animation interface.

on individual vertices. This is not an animation of an object's position, rotation, scale, etc., but an interpolation of the position of each vertex in the shape.

While this software represents a true 4D animation package, its intended audience is mathematicians and computer scientists, not animators and artists. This fact is readily apparent when the software's feature list is compared against that of a conventional 3D animation package. While users are offered a meaningful degree of control over keyframe interpolation, the software lacks a curve editor and a dope sheet for fine-tuning the keyframe values and adjusting keyframe timing, respectively. It is able to load only one shape at a time. Additionally, it lacks a scene hierarchy for object parenting. Finally, there is no way to export the projected geometry into a 3D animation package for further artistic development.

2.2.6 Wolfram Mathematica

Wolfram Mathematica is a technical computing tool with extensive visualization features. It is used extensively by mathematicians, data analysts, and engineers, among others. The interface to this software is mainly through its own scripting language. While visualizing 4D geometric shapes is possible, it can be accomplished only via the built-in scripting language. The user needs to have a strong grasp on 4D geometric shapes and must provide code to


Hypercube_mesh.nb - Wolfram Mathematica 10.1
File Edit Insert Format Cell Graphics Evaluation Palettes Window Help
Wolfram Mathematica | PRODUCT TRIAL
Learning Center | Help | Contact Us | Buy Mathematica
Out[24]=
100%

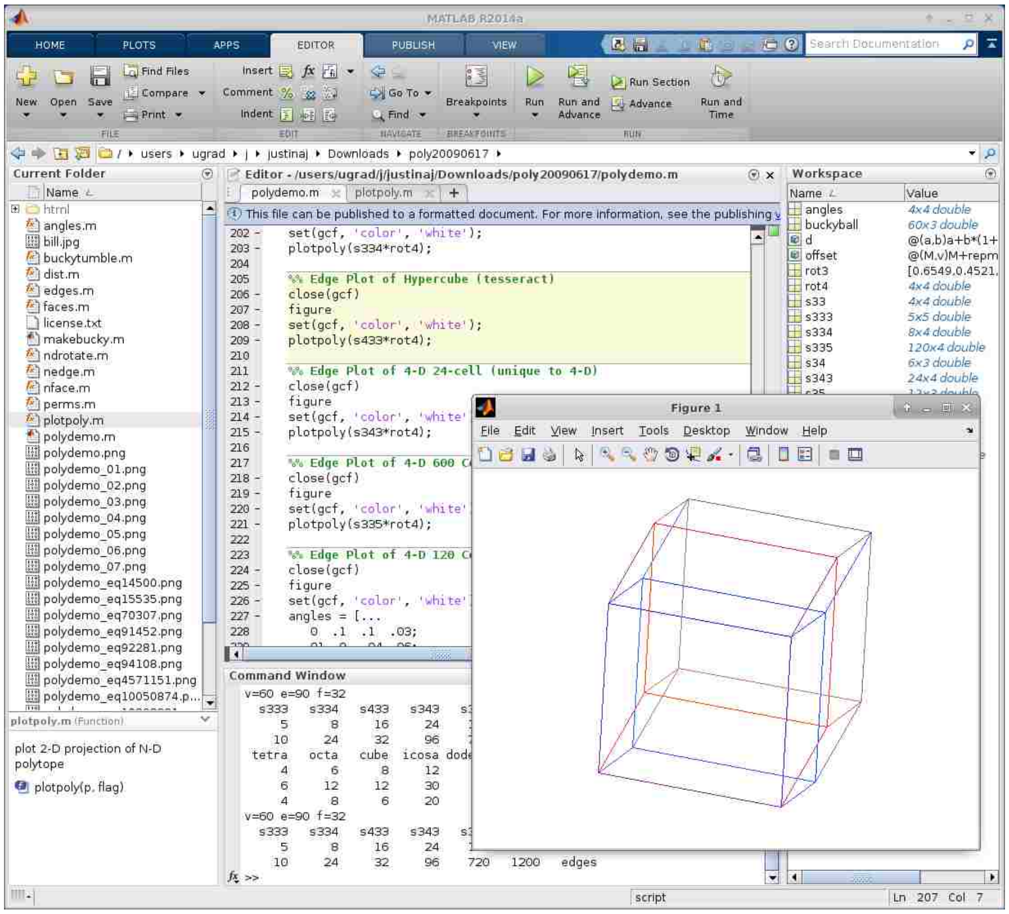
```
In[24]:= Graphics3D[
       {Line[{{1, -1, 1}, {1, -1, -1}, {1, 1, -1}, {1, 1, 1}, {1, -1, 1},
          {-1, -1, 1}, {-1, 1, 1}, {1, 1, 1}}],
        Line[{{-1, -1, 1}, {-1, -1, -1}, {1, -1, -1}}],
        Line[{{-1, 1, 1}, {-1, 1, -1}, {1, 1, -1}}], Line[{{-1, -1, -1}, {-1, 1, -1}}],
        Line[{{2, -2, 2}, {2, -2, -2}, {2, 2, -2}, {2, 2, 2}, {2, -2, 2},
          {-2, -2, 2}, {-2, 2, 2}, {2, 2, 2}}],
        Line[{{-2, -2, 2}, {-2, -2, -2}, {2, -2, -2}}],
        Line[{{-2, 2, 2}, {-2, 2, -2}, {2, 2, -2}}], Line[{{-2, -2, -2}, {-2, 2, -2}}],
        Line[{{-1, -1, -1}, {-2, -2, -2}}], Line[{{-1, -1, 1}, {-2, -2, 2}}],
        Line[{{-1, 1, -1}, {-2, 2, -2}}], Line[{{-1, 1, 1}, {-2, 2, 2}}],
        Line[{{1, -1, -1}, {2, -2, -2}}], Line[{{1, -1, 1}, {2, -2, 2}}],
        Line[{{1, 1, -1}, {2, 2, -2}}], Line[{{1, 1, 1}, {2, 2, 2}}]}]
```

Figure 2.7: A screenshot of Mathematica with a wireframe drawing of a hypercube.

project from 4D to 3D or 2D. Additionally, he must write additional code that defines any graphical interface that might be desired. It is capable of general-purpose 4D animation but almost all functionality must be coded by the user.

2.2.7 Mathworks Matlab

Similar to Mathematica, Matlab is a technical computing tool with visualization features. The only interface to the software is via the built-in scripting language. The language includes several features which allow a user to quickly manipulate hyperdimensional geometry. Like



This file can be published to a formatted document. For more information, see the publishing

```
    set(gcf, 'color', 'white');
    plotpoly(s334*rot4);

    %% Edge Plot of Hypercube (tesseract)
    close(gcf)
    figure
    set(gcf, 'color', 'white');
    plotpoly(s433*rot4);

    %% Edge Plot of 4-D 24-cell (unique to 4-D)
    close(gcf)
    figure
    set(gcf, 'color', 'white'
    plotpoly(s343*rot4);

    %% Edge Plot of 4-D 600 C
    close(gcf)
    figure
    set(gcf, 'color', 'white'
    plotpoly(s335*rot4);

    %% Edge Plot of 4-D 120 C
    close(gcf)
    figure
    set(gcf, 'color', 'white'
    angles = [...
       0  .1  .1  .03;
```

```
v=60 e=90 f=32
  s333   s334   s433   s343   s
     5      8     16     24
    10     24     32     96
 tetra   octa   cube  icosa dode
     4      6      8     12
     6     12     12     30
     4      8      6     20
v=60 e=90 f=32
  s333   s334   s433   s343   s
     5      8     16     24
    10     24     32     96    720   1200   edges
>>
```

Current Folder:

```
html
angles.m
bill.jpg
buckytumble.m
dist.m
edges.m
faces.m
license.txt
makebucky.m
ndrotate.m
nedge.m
nface.m
perms.m
plotpoly.m
polydemo.m
polydemo.png
polydemo_01.png
polydemo_02.png
polydemo_03.png
polydemo_04.png
polydemo_05.png
polydemo_06.png
polydemo_07.png
polydemo_eq14500.png
polydemo_eq15535.png
polydemo_eq70307.png
polydemo_eq91452.png
polydemo_eq92281.png
polydemo_eq94108.png
polydemo_eq4571151.png
polydemo_eq10050874.p...
```

plotpoly.m (Function)

plot 2-D projection of N-D polytope

plotpoly(p, flag)

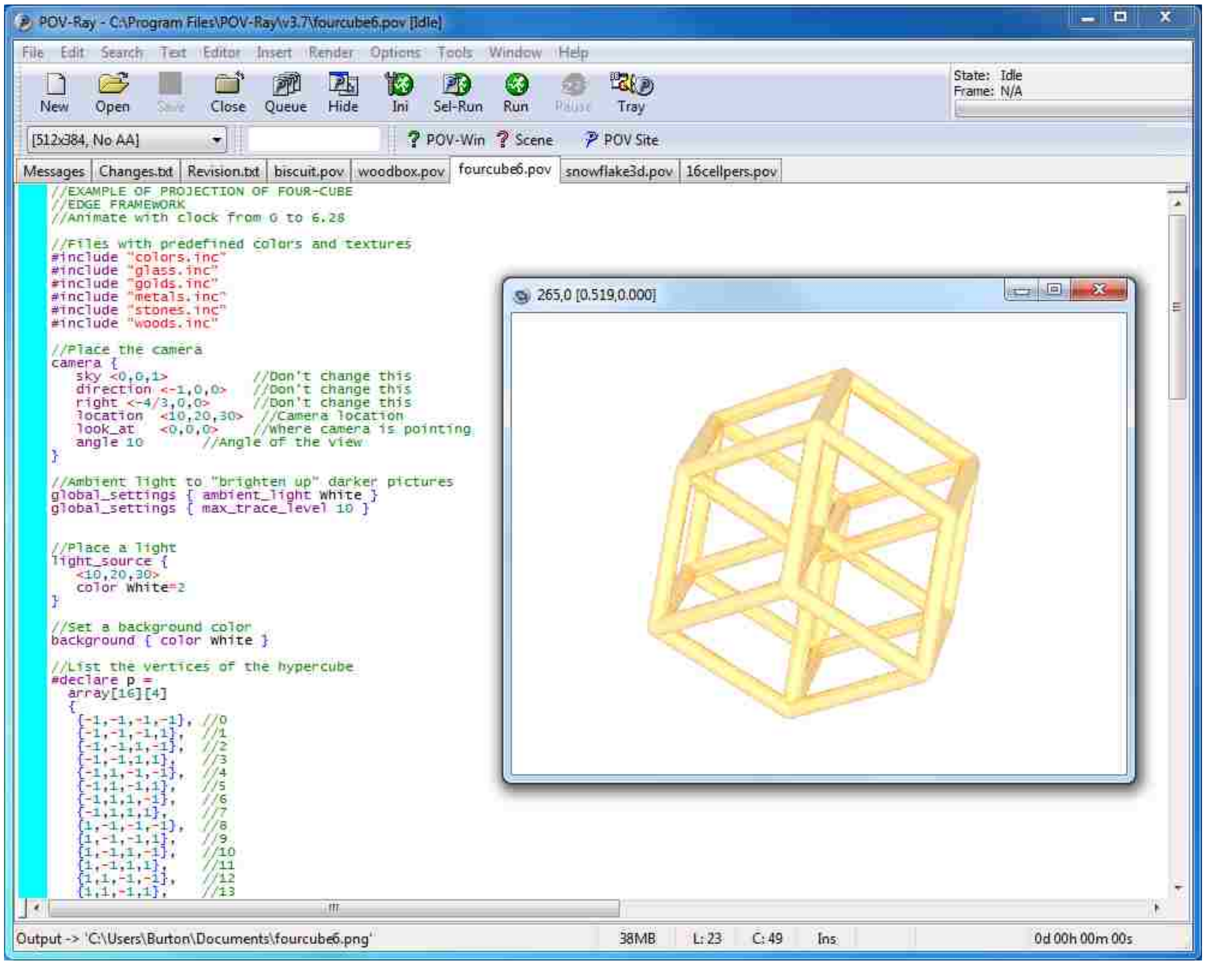
Name	Value
angles	4x4 double
buckyball	60x3 double
d	@(a,b)a+b*(1+
offset	@(M,v)M+repm
rot3	[0.6549,0.4521,
rot4	4x4 double
s33	4x4 double
s333	5x5 double
s334	8x4 double
s335	120x4 double
s34	6x3 double
s343	24x4 double

Figure 2.8: A screenshot of Matlab with an edge plot of a rotated hypercube.

Mathematica, the user needs to have a deep understanding of 4D geometric shapes and computer graphics. He also needs to provide code to project from 4D to 2D. Additionally, realtime animation is possible (again, via scripting), but interactivity is not possible.

2.2.8 POV-Ray

POV-Ray is both a 3D ray tracing engine and an animation scripting system. It has no built-in 4D features, but because of its flexible scripting system, the software has been used to generate animations of 4D hyperobjects. Because it is strictly a 3D package, the user must write scripts that will create a three-dimensional visualization of the 4D shapes. While 4D animation with POV-Ray is possible, it requires extensive knowledge of both 4D geometry and linear algebra which precludes most users from being able to use it for 4D animation.


POV-Ray - C:\Program Files\POV-Ray\v3.7\fourcube6.pov [Idle]
File Edit Search Text Editor Insert Render Options Tools Window Help
New Open Save Close Queue Hide Ini Sel-Run Run Pause Tray
State: Idle
Frame: N/A
[512x384, No AA]
POV-Win Scene POV Site
Messages Changes.txt Revision.txt biscuit.pov woodbox.pov fourcube6.pov snowflake3d.pov 16cellpers.pov
265,0 [0.519,0.000]
Output -> 'C:\Users\Burton\Documents\fourcube6.png'
38MB L: 23 C: 49 Ins
0d 00h 00m 00s

```
//EXAMPLE OF PROJECTION OF FOUR-CUBE
//EDGE FRAMEWORK
//Animate with clock from 0 to 6.28

//Files with predefined colors and textures
#include "colors.inc"
#include "glass.inc"
#include "golds.inc"
#include "metals.inc"
#include "stones.inc"
#include "woods.inc"

//Place the camera
camera {
   sky <0,0,1>            //Don't change this
   direction <-1,0,0>     //Don't change this
   right <-4/3,0,0>       //Don't change this
   location  <10,20,30>   //Camera location
   look_at   <0,0,0>      //Where camera is pointing
   angle 10         //Angle of the view
}

//Ambient light to "brighten up" darker pictures
global_settings { ambient_light White }
global_settings { max_trace_level 10 }


//Place a light
light_source {
   <10,20,30>
   color White*2
}

//Set a background color
background { color White }

//List the vertices of the hypercube
#declare p =
  array[16][4]
  {
   {-1,-1,-1,-1}, //0
   {-1,-1,-1,1},   //1
   {-1,-1,1,-1},   //2
   {-1,-1,1,1},    //3
   {-1,1,-1,-1},   //4
   {-1,1,-1,1},    //5
   {-1,1,1,-1},    //6
   {-1,1,1,1},     //7
   {1,-1,-1,-1},   //8
   {1,-1,-1,1},    //9
   {1,-1,1,-1},    //10
   {1,-1,1,1},     //11
   {1,1,-1,-1},    //12
   {1,1,-1,1},     //13
```

Figure 2.9: A screenshot of POV-Ray with a rendered hypercube. The user is required to write code that handles projection from 4D to 3D.

2.3 Deficiencies of Existing 4D Animation Software

Incomplete feature sets: As seen in table 2.1, none of the systems reported contains all the features required for a competent 4D animation system. Many features are available only via user-provided scripting. Some features, such as an animation graph editor and a dope sheet, are not found in any existing system.

Steep learning curve: Four of the systems reported (Peek, Mathematica, MatLab, and POV-Ray) require substantial knowledge of computer graphics, scripting, and 4D geometry. They are suitable for those who are already well-versed in these areas, but are foreign to much of the animation community. Because of this, 4D animation remains generally inaccessible to the animation community and generally difficult to comprehend.

Lack of control or too much control: Three of the systems (Stella4D, HyperSpace, and Meshview) have interfaces that are moderately user-friendly. Unfortunately, this comes at the cost of severely-limited control over the final animation. Four other systems (Peek, Matlab, Mathematica, and POV-Ray) lie at the other end of the spectrum; they offer substantial control but require substantial familiarity with 4D geometry to be useful. In addition, they require a user to learn a new scripting language that is unique to the system.

Restricted to a single object: With the exception of Matlab and Mathematica, none of the systems allow for the simultaneous animation of multiple 4D objects. There is no concept of a 'scene' in which multiple objects exist and move. One implication is the absence of a hierarchy in which one object can inherit the transformation of another.

Absence of focus on animation: In six of the systems (Stella4D, HyperSpace, Peek, Matlab, Mathematica, and POV-Ray), animation is difficult to perform. It is not the main purpose of the system, so it requires significant or overwhelming effort to create. In other systems animation is present, but cannot be adjusted, so the animator's control is severely limited.

		Stella4D	HyperSpace	Peek	Meshview	Matlab	Mathematica	POV-Ray
Animation Techniques	Translate	S		S	S	*	*	*
	Rotate	S	S	S	S	*	*	*
	Scale	S				*	*	*
	Shear					*	*	*
	Reflect					*	*	*
	Camera movement					*	*	*
Interface for Creating Animation	Keyframe interpolation			S	S	*	*	*
	Animation graph editor							
	Dope Sheet							
	Scripted Animation					S	S	S
Realtime Visualization	3D cross-section	S	S	S		*	*	
	4D to 3D Perspective	S		S	S	*	*	
	4D to 3D Parallel	S	S	S	S	*	*	
	4D to 2D projection		S	S	S	*	*	
	Wireframe	S	S	S	S	*	*	
	Solid	S	S	S	S	*	*	
	3D Stereoscopic	S			S			
Scene Hierarchy	Parenting					*	*	*
	Multiple objects					*	*	*
Playback	Realtime Playback	S			S	*	*	
	Non-linear editing				S			

Table 2.1: Features of existing 4D animation software. S = supported, * = not supported, but possible via scripting [12, 26, 32, 38, 44, 54, 57].

Chapter 3

Foundational Material

This chapter contains relevant algorithms, concepts, and conventions that enable the creation of a general-purpose 4D animation system.

3.1 Conventions and Characteristics of 3D Animation Software

Modern 3D animation software packages have achieved a relatively stable feature set. These features are expected to be found in almost any 3D animation software package. By including and extending these features to 4D, we preserve familiar conventions and characteristics of 3D animation software. Animation features include Affine Transformations (Translation, Rotation, Scaling, Shearing), Keyframe Animation, an Animation Graph Editor, a Scene Hierarchy, and Real-time Interactive Visualization (see 3D Studio Max [5], Maya [6], Softimage [7], Blender [22], Modo [23], Cinema 4D [39], Houdini [50]). All of these features are extended to 4D in this and the next chapter.

3.2 Transformations in 4D

Several transformations commonly are found in 3D animation software, including both affine and non-affine transformations. These 3D transformations have been extended to 4D for use in 4D-capable software. The extensions of transformations in this section are drawn from [3], [30], and [29]. A transformation in 4D is represented by a 5x5 matrix and a homogeneous point in 4D is represented by a 5D vector. For a thorough treatment on 4D vector operations, see section 2.1 of [29].

3.2.1 Affine Transformations

The Translation and Scaling transformations extend to 4D in a straightforward manner. The added dimension simply adds another coordinate value for the W axis. Likewise, the transformation matrices extend easily to 4D. The matrices simply have an extra row and column for the added W axis. See Appendix A.1 and Appendix A.2 for details on constructing the Translation and Scaling matrices for 4D.

Rotation commonly is thought to happen around an axis, but actually happens in perpendicular planes. The confusion exists because in 3D there are three mutually perpendicular axes and three planes of rotation, each of which is perpendicular to exactly one axis. However, in 4D, by listing all possible combinations of any two axes, we find that there are six planes of rotation. The rotation transformation, therefore, must contain orientation values for each of the six planes of rotation. See Appendix A.3 for details on constructing the Rotation matrices for 4D.

Uniform shearing, while not generally used as an animation technique in 3D animation software, is included for the sake of completeness. Shearing in 4D is identical to shearing in 3D, except that we have more dimensions from which to choose. See Appendix A.4 for details on constructing a shearing matrix for 4D.

Reflection is not included as an animation technique in this system. The rationale is that it is generally used as a modeling technique, not an animation technique in 3D animation software. The other affine transformations generally are sufficient to create complex animations.

3.2.2 Projection Transformations

In projection from 4D to 3D, we have constrained the problem purely to projection; we are not concerned with viewing orientation, hidden surface elimination, or frustum culling. These activities are handled by OpenGL when projecting from 3D to 2D. We have also limited the projection methods to parallel orthographic and perspective orthographic projections.

Oblique projections are left as options for future work. Given these constraints, perspective and parallel projection have straightforward extensions from 3D to 4D [3]. To perform parallel orthographic projection, we simply apply the object's transformation matrix and drop the W coordinate. See Appendix A.6 for more details on the parallel projection matrix for 4D.

Perspective projection extends in a straightforward manner to 4D and was described first by [41]. The focal length, f, of the camera is calculated in the same manner as it is in 3D computer graphics. See Appendix A.5 for details on constructing the perspective projection matrix for 4D.

3.2.3 Nested Transformations

The method for applying nested transformations to 4D objects in a hierarchical structure is nearly identical to that of 3D hierarchical structures and is the same method used by [3]. As in 3D computer graphics, nested transformations in 4D allow for parent-child relationships between 4D objects. Figure 3.1 demonstrates a nested transformation.

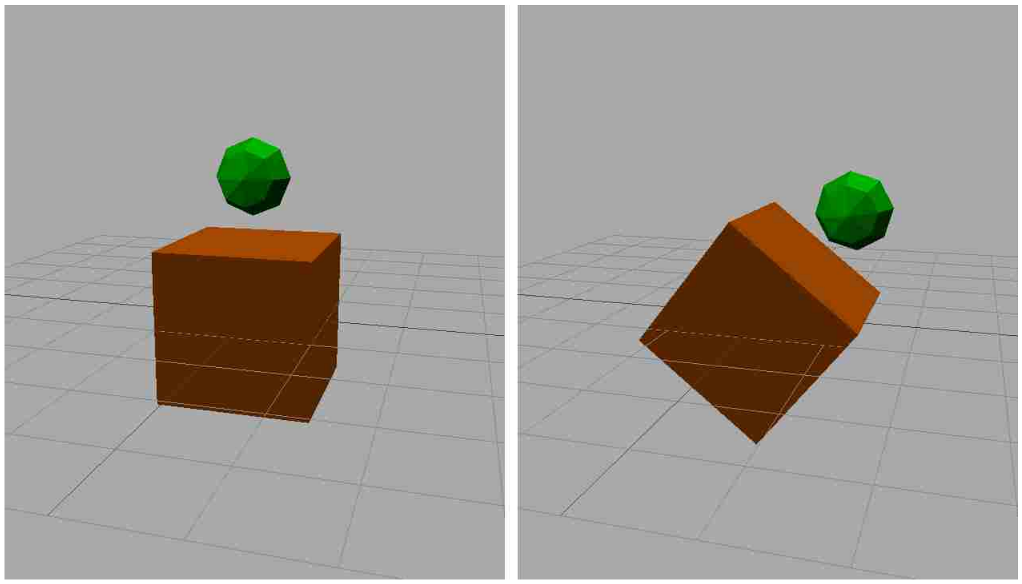


Figure 3.1: A parent hypercube with a child hypersphere. As the hypercube rotates, the hypersphere rotates relative to the hypercube's centroid.

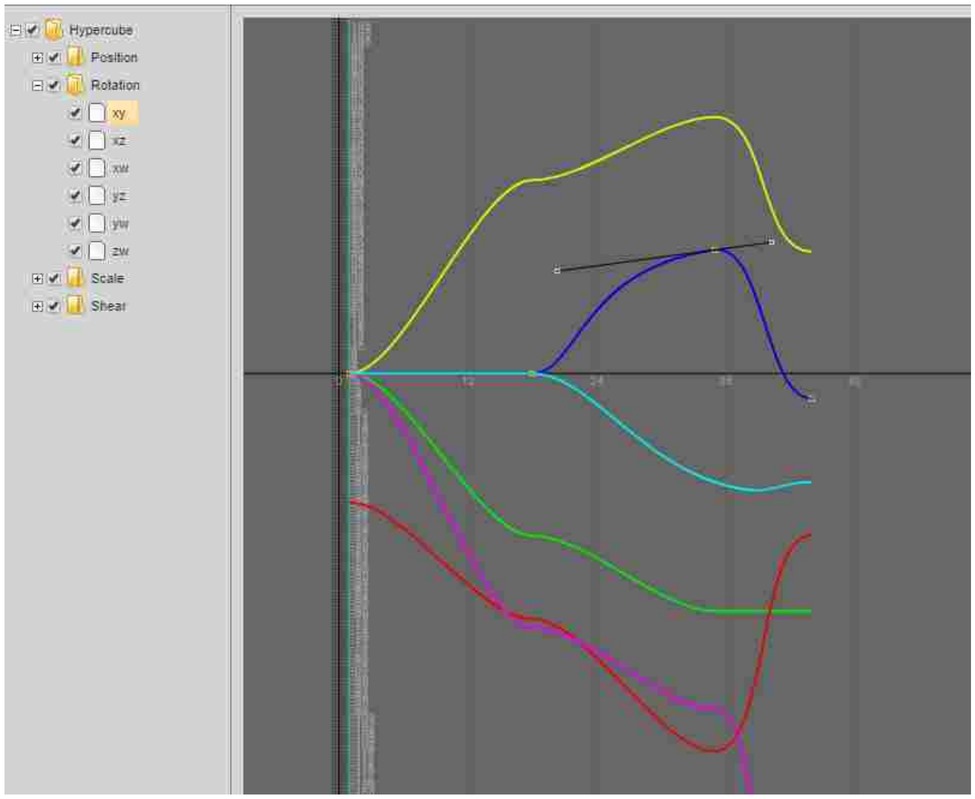


Figure 3.2: A screenshot of the extended animation curve editor. In this image, each curve represents one of the six 4D planes of rotation.

3.3 Keyframe Animation in 4D

Keyframe animation is extended to 4D by [26]. The extension of keyframe animation from 3D to 4D simply adds an extra principal axis, W, to Translation, Scaling, and Shearing, and adds three planes of rotation: XW, YW, and ZW. Our system also extends the animation curve editor commonly found in conventional 3D animation software. Figure 3.2 shows a screenshot of the curve editor extended to 4D.

3.4 Visualization Techniques for 4D Geometry

Since we inhabit a 3D world and typically use 2D display devices, visualization of 4D geometry in our lower-dimensional world becomes one of the major challenges of 4D animation. In some way, the visualization needs to encode, compress, or remove two dimensions from the data. There are several computer-aided methods for accomplishing this task. The two most commonly-used ones are reviewed below.

3.4.1 3D Cross-Section

In the same way that a two-dimensional being might view a three-dimensional object as it passes through the being's 2D plane of existence, we might similarly view a 4D object passing through three-space. It would appear as an ordinary 3D object, a cross-section of the 4D object, but would appear to change shape and/or topology as it passes through our space. Generating 3D cross-sections is an appealing visualization method because it depicts hyperdimensional objects as they would appear if they intersected the 3D hyperplane.

An algorithm for generating 3D cross-sections involves directly calculating the geometry of the cross-sections and rendering them using any conventional 3D rendering technique. Paul Isaacson [30] presents an algorithm for generating these cross-sections for any number of dimensions, provided that the geometry is stored in a hierarchical format (i.e. vertices, edges, and faces). In the 4D version, the algorithm is performed on each one-dimensional edge imbedded in four-space. If the endpoints of an edge are on opposite sides of the 3D hyperplane, the point at which the edge crosses the hyperplane is added to a list. This test is performed on all edges of a given 3D cell. Once all the intersection points for the cell's edges have been collected, they are connected to form a 2D face. This process is repeated for all 3D cells in the 4D object. The resulting faces form the mesh of the 3D cross-section (see figure 3.3.

3.4.2 Projection

Projection is another way to compress 4D geometry into 2D space. After applying any desired transformations to the object (any combination of translation, rotation, scaling, etc.), two of the dimensions can simply be dropped. There are multiple combinations of transformation

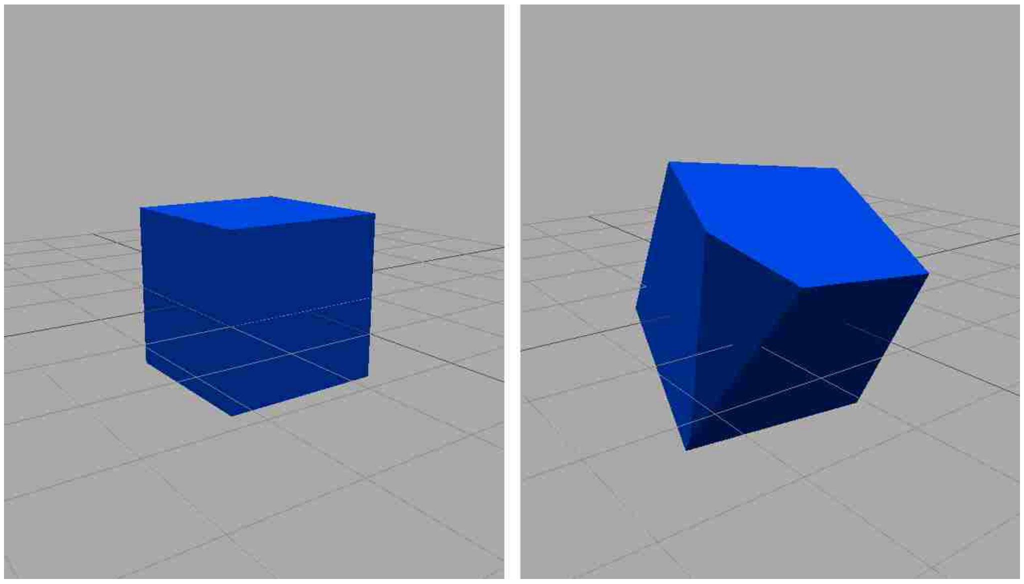


Figure 3.3: The cross-section of a hypercube, one with no rotation, and one with some rotation on several planes of rotation.

and dimension-dropping that are worth mentioning. One class of them involves a direct projection from 4D to 2D. Another class includes an intermediate 3D step.

4D-to-2D Projection: The simplest way to project four-dimensional geometry to a 2D plane is to simply drop two of the four dimensions. This projection method effectively gives a side-on view using a parallel orthographic projection. In three dimensions, there are only three ways to drop a single dimension. In four dimensions, however, we need to drop two distinct dimensions and so there are six different ways to do so. These six ways correlate directly to each of the six rotation planes. For example, by dropping the X and Z coordinates of all vertices, we project the geometry onto the YW plane using a parallel orthographic projection. We can do the same for the XY, XZ, XW, YZ, and ZW planes. If desired, the object can be rotated before applying the projection, either by its own local transform or by a camera transform that is applied to it, which are mathematically equivalent. This is one approach taken by Paul Bourke in his software HyperSpace [12], and by Hüseyin Koçak in his visualization of Linear Hamiltonian Systems [33]. See figure 3.4.

4D-to-3D-to-2D Projection: There is a second method for projecting 4D geometric shapes to a 2D screen. Unlike the previous method, it involves an intermediate step which projects the shape into 3D, then a second projection from 3D to 2D. A. Michael Noll extended the traditional 3D projection algorithm to any n-dimensional projection to n-1 dimensional space, a crucial tool for this second projection method [41]. The algorithm places the viewpoint along the nth axis and transforms the scene's geometry accordingly. An n-1 dimensional projection plane centered on the W axis is placed between the viewpoint and the scene. Each vertex in the scene is then scaled up proportionate to the distance between the viewpoint and the projection plane. This allows a variable field of view. The vertex is then divided by its distance from the viewpoint in the nth axis. This provides the perspective effect (see figure 3.4). Finally, the n-component of the vertex is set to some arbitrary constant, ensuring that the projection lies completely within the n-1 dimensional projection plane. In this method, the two projections (4D-to-3D and 3D-to-2D) are independent of each other. For each projection, a distinct nD viewpoint and n-1 D projection plane are required. For example, we could project from 4D to 3D using a perspective projection, then project from 3D to 2D using a parallel projection, even from a different viewpoint. Because of the various possible combinations of projections and viewpoints, it becomes important to know which methods were chosen for a given image in order to better comprehend the true four-dimensional nature of the shape being depicted.

A principle advantage of the projection method is that all of the lower-dimensional geometric components of the shape are present in the scene; all vertices, edges, and faces are present, albeit with fewer dimensions. However, due to the nature of projection to lower dimensions, much of the geometry can be hidden internally, a problem that is not unique to 4D. This problem can be mitigated in several ways. The simplest way is to render a wireframe mesh of the object so that only edges are visible instead of rendering a solid, shaded shape. However, without animation or other depth cues, it can be difficult to discern the depth and orientation of each edge in 3D space on the 2D image. Another way to expose internal geometry is to slice the shape into a series of ribbons or bands, so that the internal form is visible in the gaps between ribbons [33]. Yet another way is to render the objects is to display them as semi-transparent volumes. This method allows some semblance of a solid surface but also makes the internal geometry visible [53]. A second disadvantage of the projection method is that rigid rotation in four dimensions produces projections that do not appear rigid. Once again, this problem is not unique to 4D, but it is more difficult for the viewer to comprehend initially because of his lack of intuition concerning 4D rotation. With some experience, however, an intuition for high-dimensional rotation may be gained [9].

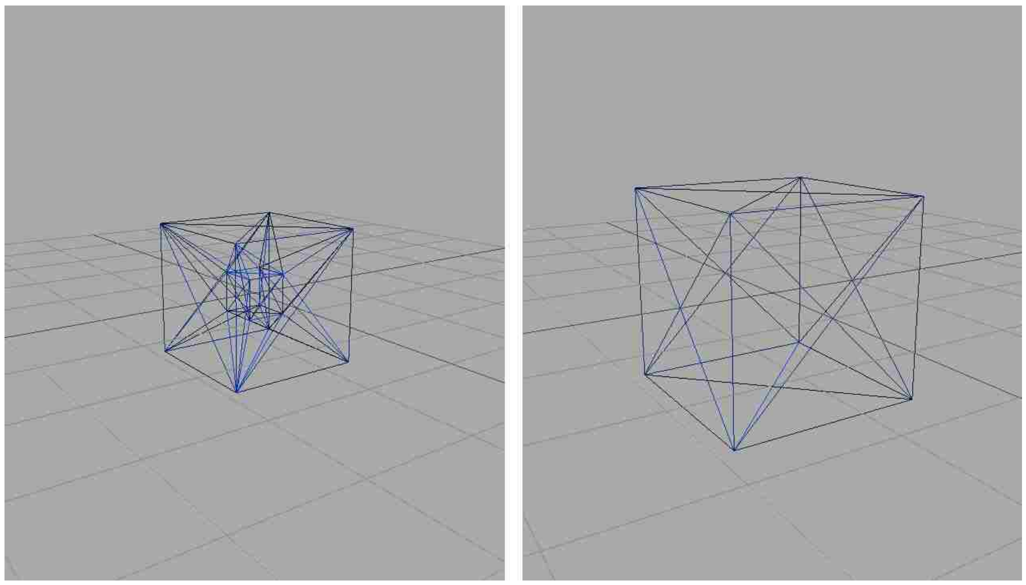


Figure 3.4: Two projections of a hypercube. A perspective projected hypercube (left) and a parallel projected hypercube (right). Neither hypercube has any rotation applied.

3.5 4D Geometry

3.5.1 Tetrahedral Meshes

Since the tetrahedron is the 3D counterpart to the triangle, a mesh composed of tetrahedra is a natural extension of the triangular mesh model commonly found in 3D computer graphics. A 4D object can be approximated by a tetrahedral mesh that represents the boundary of the hypersurface. While this method is only an approximation of the hypersurface, the simplicity of the model lends itself to relatively simple algorithms for visualizing 4D shapes. As proposed

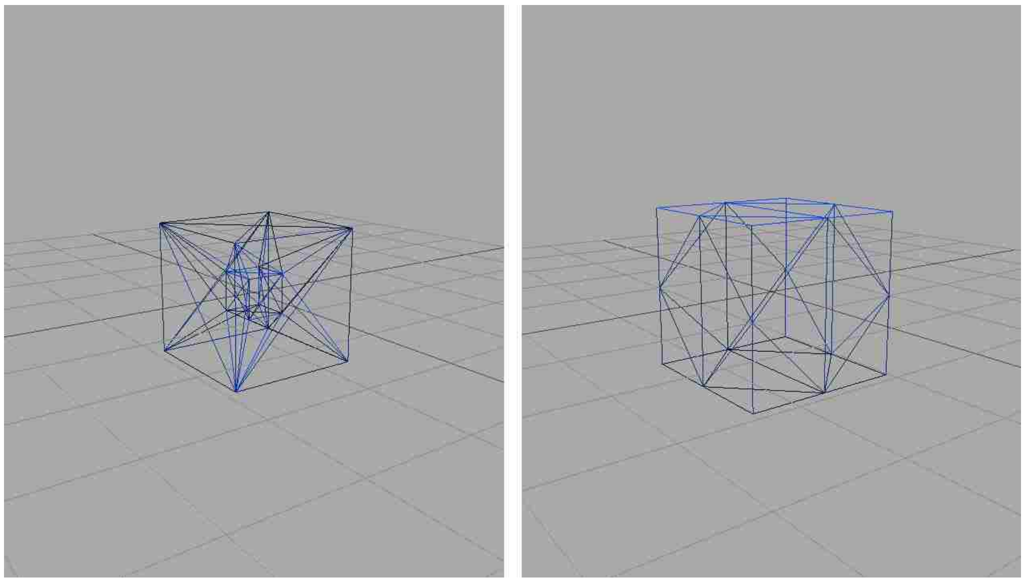


Figure 3.5: A wireframe view of a perspective projected hypercube (left) and a wireframe view of the cross-section of a hypercube (right). There is a subtle difference between this figure and figure 3.4. The hypercubes on the right half of each figure have different triangular meshes. This figure features a cross-section while the other figure features a parallel projection.

in [19], we can calculate a 4D tetrahedron normal which is a vector that is perpendicular to the tetrahedron, a fact that we make use of in section 4.2.3.

3.5.2 Simple 4D Solids

Software for creating 4D geometric shapes is meager and there is none that allows for arbitrary shapes. The shapes most often featured by 4D-capable software, including those featured in this section, are generated procedurally. This limits us to a small set of simple geometric shapes that are extensions of simple 3D solids. We include three 4D geometric shapes: the Hypercube, the (tessellated) Hypersphere, and the Hypercrystal.

Hypercube: A hypercube is the 4D analogue of a cube. It consists of eight cubical cells. A unit hypercube is constructed by placing two unit cubes 1 unit apart along the W axis and connecting the corresponding vertices of each cube. We create the eight cubical cells of the hypercube, then split each cube into five tetrahedra, as done by [19]. See figure 3.5 for a hypercube rendered with a perspective projection and as a cross-section.

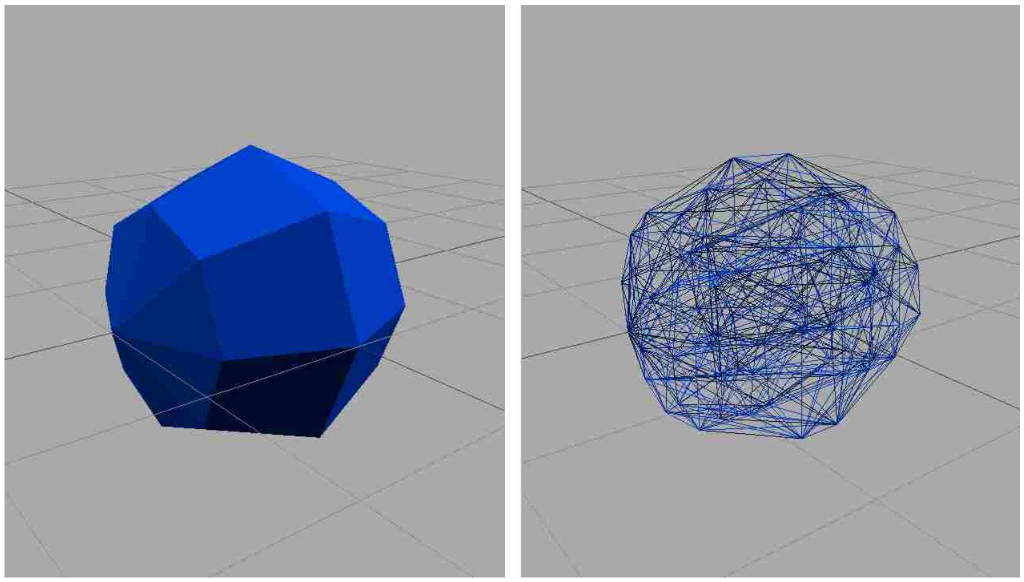


Figure 3.6: A cross-section of a hypersphere (left) and a wireframe view of the perspective projection of a hypersphere (right).

Tessellated Hypersphere: A hypersphere is the 4D analogue of a sphere. The surface of a hypersphere of radius r is the set of points in 4D space that are distance r from the center of the hypersphere. In 3D graphics libraries, a sphere is usually approximated by tessellating it into a quadrilateral mesh, then converting each quadrilateral to a pair of triangles. We have extended this technique to 4D by tessellating the hypersphere into a cuboid mesh [56] and converting each cuboid into a set of tetrahedra. See figure 3.6 for a hypersphere rendered with a perspective projection and as a cross-section.

Hypercrystal: The hypercrystal is a 4D shape theorized by Marc ten Bosch in [11]. Bosch explains the construction of a hypercrystal by first describing the construction of a 3D crystal-like shape. The 3D crystal is constructed by placing two hexagons some distance apart along an axis that is perpendicular to each hexagon. The vertices of each hexagon are connected. Then each hexagon is extruded to points further along the axis, as in figure 3.7. The construction of a hypercrystal is an extension of this idea. It is constructed by placing two dodecahedra some distance apart along the W axis and connecting the corresponding vertices of each dodecahedron. Each dodecahedron is then extruded to a point further along the W axis. Bosch chose a dodecahedron because the cross-sections of the resulting hypercrystal

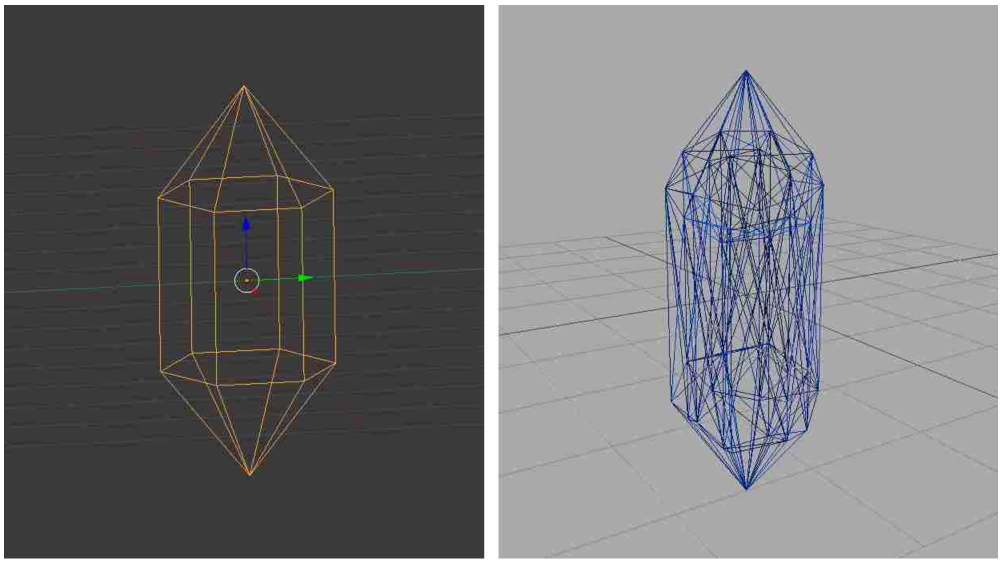


Figure 3.7: A 3D crystal-like shape, created from two parallel hexagons (left). A wireframe view of a perspective projected hypercrystal (right).

often resemble a 3D crystal-like shape. The projection of a hypercrystal is shown in figure 3.7. With only these simple shapes at our disposal, we are still able to create moderately complex geometric shapes. Using the provided affine transformations as well as nested transformations, an animator is able to combine multiple simple shapes to form larger, more complex shapes. For example, figure 3.8 depicts a 4D grid of hypercubes which pivot together around a central point. The individual pieces of this aggregate shape are very simple, but the resulting group of shapes is quite complex.

3.6 A 4D Camera Model

The 4D camera model proposed by [29] closely mirrors the camera model found in many 3D graphics libraries. The camera is defined by two points: "From" and "To", and two vectors: "Up" and "Over". The Over vector is an extra vector which is required to create an orthonormal basis in 4D. The viewing matrix is created with the help of a 4D version of the Cross Product vector operation, also described by [29].

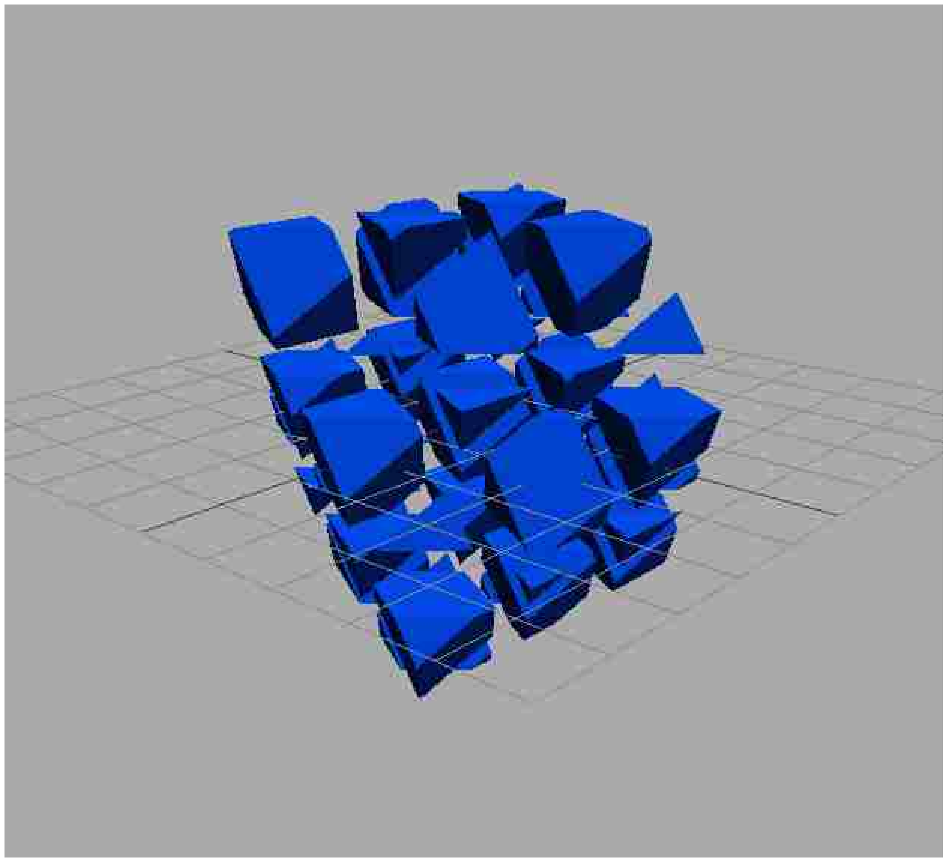


Figure 3.8: The cross-section of a 3x3x3x3 grid of hypercubes

3.7 3D Stereoscopic Rendering

Many 3D animation packages have built-in support for 3D stereoscopic rendering in the viewport. Though the implementations are varied (crossed-eyes, frame sequential, red-cyan anaglyph, interlaced, etc.) the purpose is the same: to preview the stereoscopic effect and make adjustments interactively. We have selected red-cyan anaglyph rendering because it is relatively easy to implement and the viewing glasses required for this technique are usually very affordable.

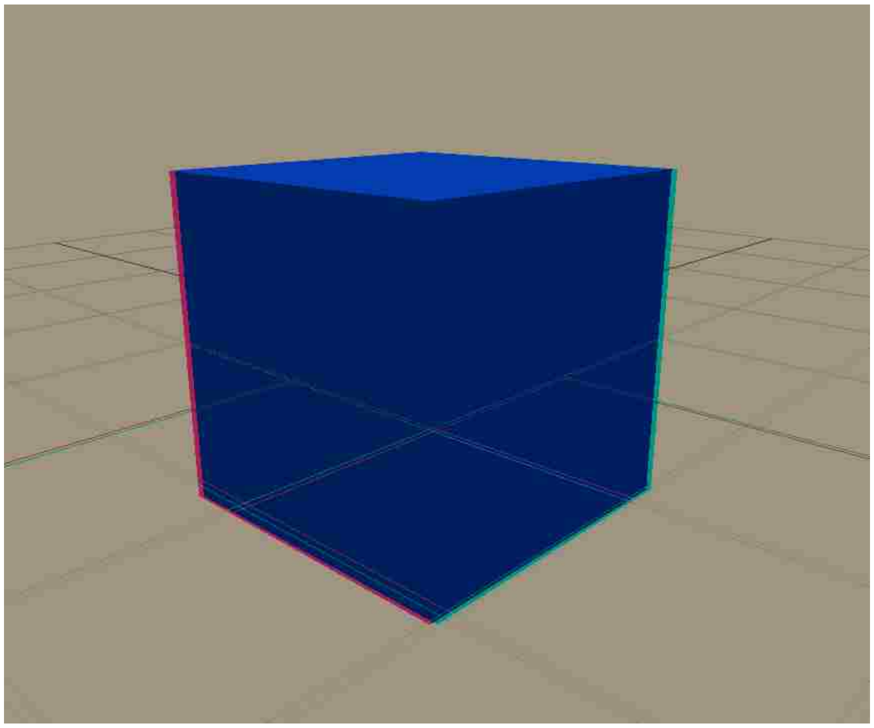


Figure 3.9: The cross-section of a hypercube, rendered with red-cyan stereoscopic rendering.

3.8 Deficiencies in Foundational Techniques

These foundational techniques provide many of the desired features of a general-purpose 4D animation system, but there are several techniques that are not found in the current literature.

Interactive Transformation: existing interactive transformation techniques either require special-purpose hardware, or do not resemble the conventional techniques found in 3D animation software.

Interoperability: Since the visualizations remain in existing 4D animation software, with no support for exporting them to conventional 3D formats, there is no need for interoperability between existing 4D and 3D animation software. This is a problem if the animator wants to animate 4D shapes and use the animation in conventional 3D animation software.

4D Tetrahedral Mesh File Format: There are published file formats for 4D geometry, but they exhibit several drawbacks which can limit the robustness of an animation system that uses them. Some are designed for wireframe mesh rendering, but not for solid rendering. Others allow arbitrary cell shapes which complicates visualization algorithms. There is no published file format for 4D tetrahedral meshes.

Chapter 4

Innovations

We have created a 4D animation system which enables general-purpose 4D animation. We call this system *Fourveo*. This system addresses the deficiencies of existing 4D -partially-capable systems and is designed to feel familiar to conventional 3D animators. Several innovations were necessary to create this system, including:

- Projected Markers, for locating 4D shapes in the absence of a visible cross-section
- Consistent vertex winding order in cross-sections, which enables the animator to export cross-section animations to 3D animation software
- An interactive 4D transformation widget, which creates a tight feedback loop for the animator and facilitates 4D intuition
- The extension of the 3D animation interface to 4D, so that it feels familiar to 3D animators
- The specification for a 4D tetrahedral mesh geometry file format

4.1 Projected Markers

When visualizing 4D geometry with the cross-section visualization method, a visible cross-section exists only if the 4D hypershape intersects the 3D hyperplane; if none of the elements of a 4D shape intersect the hyperplane, there is no visible cross-section. Without a visible cross-section, there is no way for an animator to visually determine where the hypershape is located or where a cross-section will appear once the hypershape returns to the hyperplane.

See figure 4.1 for the analogue of this problem in 3D. While this disappearing and reappearing phenomenon is a characteristic of interacting with lower-dimensional spaces, the animator is at a distinct disadvantage without some other visual indicator of the 4D position of the hypershape he is animating.

To alleviate this problem, and to bring more intuition to 4D animation, we introduce the concept of Projected Markers. A projected marker represents the centroid of a 4D shape, projected onto a 3D hyperplane. A point is placed at the centroid of a 4D shape. This point is then projected parallel to the 3D hyperplane. At this projected point in 3D space, we place a small sphere to mark its position. The sphere's radius is 0.15 units; it is large enough to be seen easily, but small enough so as not to obscure most cross-sections. If the animator desires, he can hide these markers. The position of the 4D centroid relative to the hyperplane is color coded. If the centroid is above the hyperplane (i.e. its W coordinate is greater than 0), the marker is colored red. If it is below the hyperplane (i.e. its W coordinate is less than 0), the marker is colored blue. If it is on the hyperplane (i.e. its W coordinate is equal to 0), the marker is colored white. To ensure that the marker is visible regardless of the color of the object, the marker has a dark outline. As an object moves about 4D space, the projected marker moves correspondingly on the 3D hyperplane (see figure 4.2). The colored portion of the marker is scaled according to the distance between the centroid and the hyperplane. The dark outline does not scale, which allows the animator to see the difference in size between the outline and the marker, thereby allowing him to estimate the centroid's distance from the hyperplane. As the centroid of an object moves away from the hyperplane, the marker scales according to an exponential decay function so that the scaling effect is most apparent when the object's centroid is near the hyperplane. In most cases the cross-section appears close to the projected marker[1]. Additionally, the animator can quickly determine in which direction

[1]There is an exception. If the 4D shape is significantly larger along some axis than along its other axes and the object is oriented such that the larger extension is nearly (but not exactly) parallel to the hyperplane, its first cross-section will appear far from its centroid. As a lower-dimensional analogy, imagine holding a pencil above a pool of water, nearly parallel to the water's surface, and dropping it. The point where the pencil first intersects the water's surface may be relatively far from the pencil's centroid.

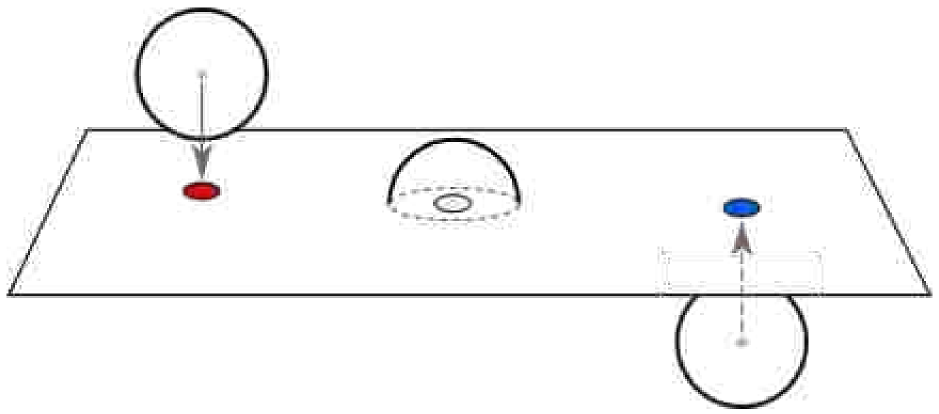


Figure 4.1: A 3D analogue of Projected Markers. The centroid of the sphere on the left is above the plane, so its marker appears below it on the plane in red. The centroid of the middle sphere is on the plane, so its marker appears in white. The centroid of the sphere on the right is below the plane, so its marker appears above it on the plane in blue.

to translate a 4D shape to cause it to intersect the hyperplane and create a cross-section. This gives the animator more confidence in animating 4D shapes, especially when the shapes move away from the hyperplane.

4.2 Consistent Vertex Winding Order in Cross-Section Geometry

In the process of creating a 3D animated sequence, several independent pieces of software commonly are used to achieve the final result. It often becomes necessary to export data from one piece of software for use by another. Rendering the 4D animation to a sequence of mesh files (.OBJ in this case) allows a user to create a 4D animation, and then transfer the projected or sliced geometry to a conventional 3D package for subsequent work. This subsequent work includes, but is not limited to, procedural shading, photorealistic rendering, and integration into traditional 3D animated sequences.

The cross-section of a tetrahedron embedded in 4D space can be a point, an edge, a triangle, a quadrilateral, or the entire tetrahedron itself. Points and lines are discarded when producing solid, opaque cross-sections (as opposed to a wireframe mesh which lacks solid faces). Some 3D graphics libraries are able to draw triangles, but not quadrilaterals. The quadrilaterals need to be triangulated into two non-intersecting triangles. By imposing a

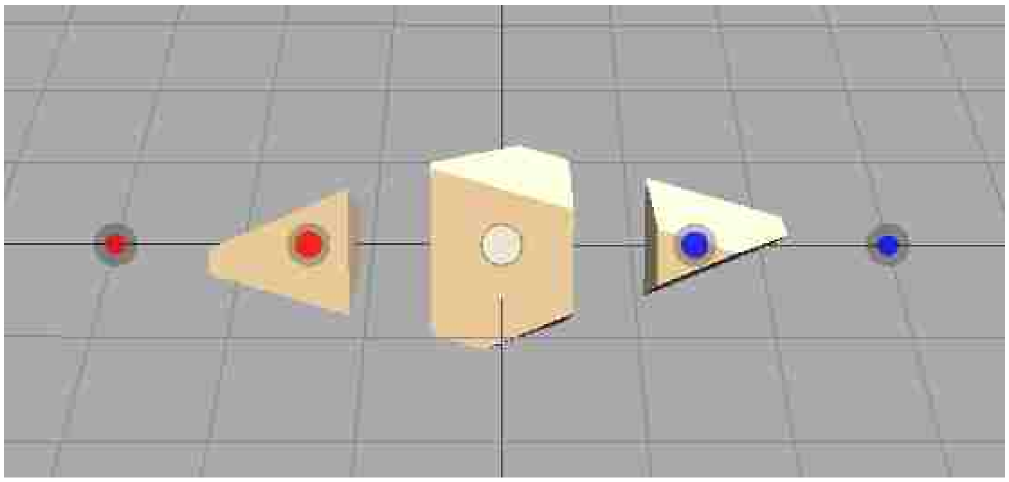


Figure 4.2: A screenshot of five projected markers. Each marker represents the centroid of a hypercube. The two red markers on the left indicate that the centroids are above the 3D hyperplane (Ana). The white marker in the middle indicates that the centroid is on the 3D hyperplane. The two blue markers indicate that the centroids are below the 3D hyperplane (Kata). The two outer markers correspond to hypercubes which do not intersect the hyperplane. Notice the relative sizes of the colored portion of the markers, which indicate the distances from their corresponding centroids to the hyperplane.

vertex ordering for the tetrahedron, an ordered list of vertices of the quadrilateral can be generated such that: 1) the quadrilateral is not a complex polygon and 2) the quadrilateral is convex. Such a quadrilateral is then trivial to triangulate.

Another issue to consider is front- and back-facing polygons. Some existing algorithms for generating cross-sections are designed for generating wireframe meshes, not solid polygonal objects. Because the [30] slicing algorithm did not feature hidden-removal, no thought was taken previously to enforce a specific winding order of the vertices in the resulting polygons. Other algorithms sidestep this issue by rendering both sides of the polygon and therefore do not take advantage of backface culling. Since we intend to transfer scene geometry to other animation software and cannot depend on the rendering algorithms used in each software package to provide all the needed characteristics in an appropriate form, we need to enforce a consistent vertex winding order. This winding order is needed not only for rendering, but also for saving the cross-sections to secondary storage since other software will expect a consistent winding order.

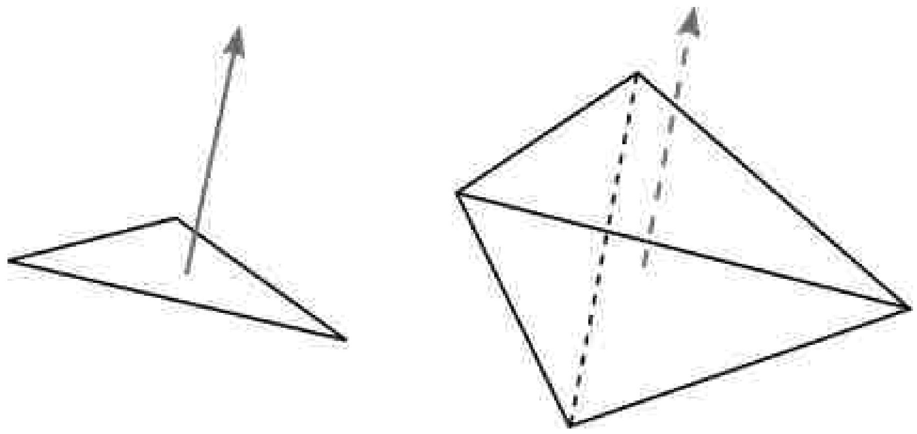


Figure 4.3: A triangle with its surface normal (left) and a tetrahedron with its 4D tetrahedron normal, projected to 2D (right).

4.2.1 4D Tetrahedron Normals

The basis for achieving a consistent vertex winding order is generating a 4D Tetrahedron Normal for each tetrahedron. This is a vector which is normal to the entire tetrahedron, and therefore mutually perpendicular to every edge in the tetrahedron. As an analogy, consider a triangular face. We can generate a face normal that is mutually perpendicular to each edge in the triangle. Note that this normal vector must exist in at least 3 spatial dimensions. Likewise, the tetrahedron normal vector must exist in at least 4 spatial dimensions (see figure 4.3). To generate a 4D tetrahedron normal, we first get three vectors from the tetrahedron:

$$E1 = v1 - v0, E2 = v2 - v0, E3 = v3 - v0$$

Using the strategy implemented by [19, 29], we can calculate the tetrahedron normal by finding the determinant of the following matrix:

$$tetrahedron\ normal = \begin{vmatrix} i & j & k & l \\ E1_x & E1_y & E1_z & E1_w \\ E2_x & E2_y & E2_z & E2_w \\ E3_x & E3_y & E3_z & E3_w \end{vmatrix}$$

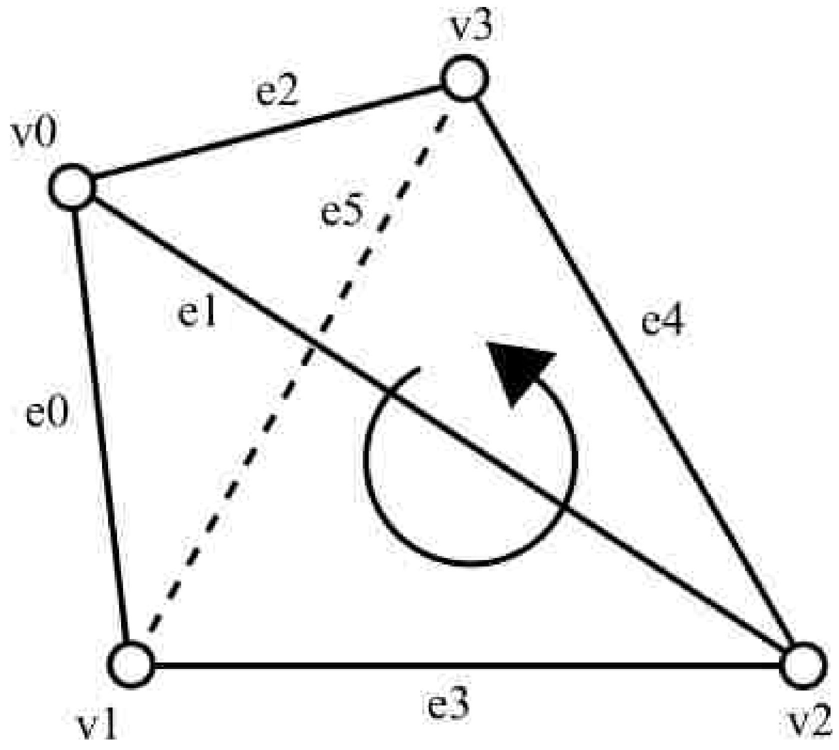


Figure 4.4: The vertex ordering convention used in Fourveo. Vertices 1, 2, and 3 form the base of the tetrahedron and are arranged counterclockwise when viewed from above. Vertex 0 is the apex of the tetrahedron. The edge ordering convention is also listed.

We normalize the resulting determinant to get the tetrahedron normal (see figure 4.3). That normalized vector is our tetrahedron normal. By providing consistent tetrahedron vertex ordering (section 4.2.2), these normals will consistently point away from the interior of the 4D hypershape.

4.2.2 Tetrahedron Vertex Ordering

In order to generate consistent tetrahedron normals, we need to enforce a consistent vertex order in the mesh's tetrahedra. The particular ordering is arbitrary, but must be consistent. Referring to figure 4.4, vertices 1, 2, and 3 form a triangular face that is parallel to the page and vertex 0 is located above the page. Vertices 1, 2, and 3 are arranged in counterclockwise order when viewed from above the page. This ordering allows us to generate consistent tetrahedron normals. There are two methods for ensuring consistent tetrahedron vertex ordering: 1) If the 4D geometric shape is imported from a file, we assume that the tetrahedron vertices are already ordered consistently. Discovering inconsistent vertex ordering for arbitrary meshes is a difficult problem that is left to future research. In this case, the burden of enforcing consistent tetrahedron vertex ordering falls on the software which generated the geometry. 2) If the shape is generated algorithmically and is a convex shape, we can ensure consistent tetrahedron vertex ordering in the following manner: For each tetrahedron: Generate its 4D tetrahedron normal. Get the vector from the centroid of the 4D shape to the centroid of the tetrahedron. Calculate the dot product of these two vectors. If the dot product is negative, we reverse the order of vertices 1, 2, and 3 (see Algorithm 8).

4.2.3 Enforcing a Consistent Vertex Winding Order

Once the tetrahedra have been sliced, we enforce a consistent vertex winding order in the polygons contained in the cross-section. To do this, we consider a technique that is used in CNC milling and 3D printing, and we extend the technique to 4D. When converting a 3D polygonal mesh to a set of toolpaths, the algorithm proposed by [18] slices the mesh into a stack of 2D slices. Each of these slices is a closed polygon with an interior and an exterior denoted by the orientation of the edges in the polygon. A slice whose edges form a counterclockwise path is an exterior edge whereas a clockwise path denotes an interior edge. Each edge in the slice corresponds to a triangle in the 3D mesh. For each triangle, they find the cross-section at the slicing depth which yields a line segment. To do this, they check each edge of the triangle in order: $v0 \rightarrow v1$, $v1 \rightarrow v2$, then $v2 \rightarrow v0$. By intersecting the edges in this order for all triangles in the mesh, and provided the 3D mesh adheres to a consistent winding order, this algorithm ensures that the resulting edges in the slice will be ordered head-to-tail in a counterclockwise direction (see figure 4.5).

We extend this technique to 4D. Given a 4D tetrahedral mesh with consistent outward-facing tetrahedron normals, we can enforce a consistent vertex winding order in the resulting slice polygons. For each tetrahedron that is intersected by the 3D hyperplane, we calculate its

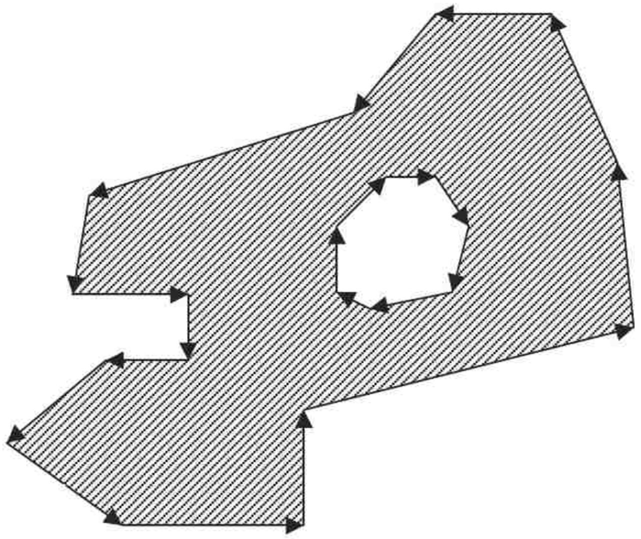


Figure 4.5: A figure from [18] depicting a slice of a 3D mesh. Edges are arranged head-to-tail such that counterclockwise paths represent exterior edges and clockwise paths represent interior edges.

4D tetrahedron normal and project it to the 3D hyperplane (in practice, we simply drop the W coordinate). We take the intersection of the tetrahedron (a triangle or quadrilateral) and generate a 3D surface normal assuming a counterclockwise vertex order. We then calculate the dot product of this surface normal and the projected 4D tetrahedron normal. If the dot product is negative, the vertex order is incorrect and we need to reverse it. Once reversed, the vertices will be stored in counterclockwise order, the de facto standard winding order in most 3D animation software.

4.2.4 Tetrahedron Slicing Algorithm

With tetrahedron vertex ordering, 4D tetrahedron normals, and a mechanism for enforcing a consistent vertex winding order in place, we developed an algorithm for slicing 4D tetrahedral meshes. Some foundational information facilitates an understanding of the specific details of the algorithm.

4D Shape representation: Each 4D shape is stored in memory as a complete list of unique vertices and a list of tetrahedra. Each tetrahedron is represented by two arrays: 1) an array of four vertex indices and 2) an array of six edges, each of which is a pair of vertex indices. See figure 4.4 for the vertex and edge order convention used by Fourveo.

Binary representation of edges sliced: To find the intersection of a tetrahedron with the 3D hyperplane, we check each edge of the tetrahedron to learn if it intersects the 3D hyperplane and we keep track of which edges are sliced. This enables us to generate a cross-section of the tetrahedron. We determine which edges are sliced and encode this information in a 6-bit binary number. Each bit represents one edge of the tetrahedron. A bit is set to 1 if the edge is intersected by the 3D hyperplane and 0 otherwise. The edge order is described in figure 4.4.

Edge intersection order to ensure non-overlapping quadrilateral triangulation: When the cross-section of a tetrahedron yields a quadrilateral, we need to order the four vertices such that the quadrilateral can be split into two non-overlapping triangles. To accomplish this, we have enumerated all possible configurations of edges sliced, and have created an ordered list of edges from which to generate the vertices of the quadrilateral for each configuration. Each ordered list represents the edge indices from which to calculate the intersection points of the tetrahedron. Each ordered list of edges is not the only possible solution (e.g. each ordered list can be rotated and still yield a correct solution), but only one solution is needed for non-overlapping triangles. See figure 4.6 for more details and table 4.1 for the complete list of slice configurations. These configurations are stored in an array and are initialized in Algorithm 1 on line 1.

Algorithm 1 (pg. 49) slices a tetrahedral mesh and returns a set of triangles which define the intersection of the mesh with the 3D hyperplane. For clarity and readability, the algorithm is separated into several logical components: the main algorithm and several algorithm snippets that are referenced by the main algorithm. Several functions have been separated and named for clarity.

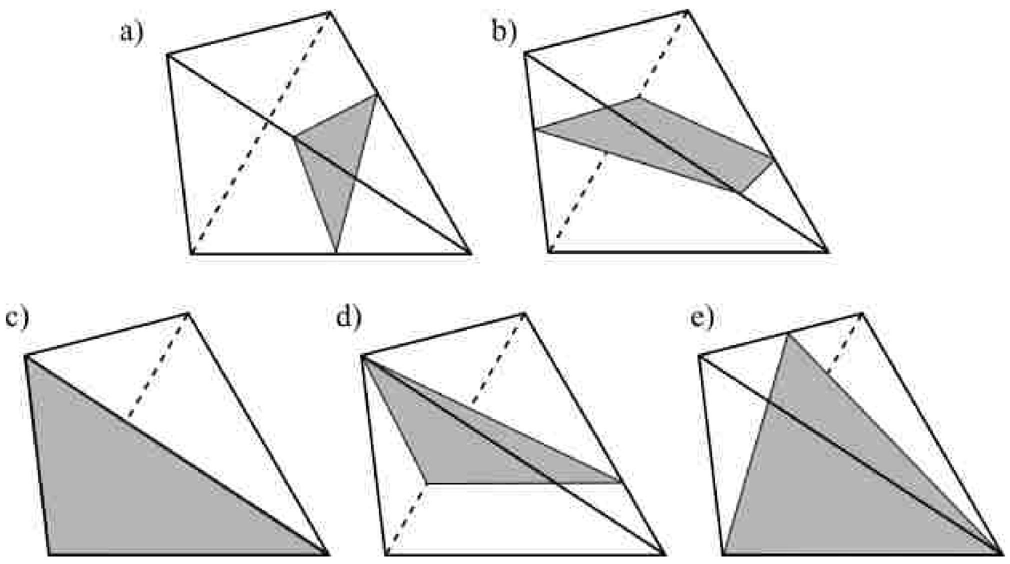


Figure 4.6: The common tetrahedron slice configurations. a) a triangular slice which intersects three edges. b) a quadrilateral slice which intersects four edges. c) a triangular slice that perfectly intersects a single face of the tetrahedron. d) a triangular slice that intersects the tetrahedron vertex-first. e) a triangular slice that intersects the tetrahedron edge-first. An additional configuration, where all four triangular faces intersect the 3D hyperplane, is not pictured here.

e0	e1	e2	e3	e4	e5	Decimal	Edge Order
1	1	1	0	0	0	56	[0,1,2]
1	0	0	1	0	1	37	[0,3,5]
0	1	0	1	1	0	22	[1,3,4]
0	0	1	0	1	1	11	[2,4,5]
0	1	1	1	0	1	29	[1,2,5,3]
1	0	1	1	1	0	46	[0,2,4,3]
1	1	0	0	1	1	51	[0,1,4,5]

Table 4.1: Edge intersections and an edge intersect order for creating a tetrahedron slice. For each edge, a '1' means the edge is intersected while a '0' means it is not intersected. The intersected edges are encoded as a binary number with the decimal value listed. Each edge order array is one of several correct edge orderings which yield quadrilaterals that triangulate to two non-overlapping triangles.

Algorithm 1: Intersect 4D object with the 3D hyperplane

```
Input: A finite set of vertices in object space $verts$, a finite set of tetrahedron edges
       $allTetEdges$, a finite set of tetrahedron vertex indices $allTetVerts$, the view
       matrix $viewMat$
Output: The intersection of the object with the 3D hyperplane
1  initialize $intersectEdgeOrders$ (Table 4.1)
2  $transVerts \leftarrow \emptyset$
3  $intersectTris \leftarrow \emptyset$
4  $objVertsOnPlane \leftarrow \emptyset$
5  $tetVertIsOnPlane \leftarrow \emptyset$
6  for $idxv \leftarrow 0$ to $length(verts) - 1$ do
7      $transVerts[idxv] \leftarrow verts[idxv] * viewMat$
8      $objVertsOnPlane[idxv] \leftarrow transVerts[idxv].w = 0?1 : 0$
9  end
10 $intersectionTriangles \leftarrow \emptyset$
11 for $idxv \leftarrow 0$ to $length(allTetVerts) - 1$ do
12     $tetVerts \leftarrow allTetVerts[idxt]$
13     $v0, v1, v2, v3 \leftarrow transVerts[tetVerts[0, 1, 2, 3]]$
14     $slicePoints \leftarrow \emptyset$
15     $sliceTriangles \leftarrow \emptyset$
16     $numVertsOnPlane \leftarrow 0$
17     for $idxv \leftarrow 0$ to $3$ do
18         $tetVertIsOnPlane[idxv] \leftarrow objVertsOnPlane[allTetVerts[idxt][idxv]]$
19         if $tetVertIsOnPlane[idxv] = 1$ then
20             $tetVertsOnPlane.push(transVerts[allTetVerts[idxt][idxv]].xyz)$
21             $numVertsOnPlane \leftarrow numVertsOnPlane + 1$
22     end
23     if $numVertsOnPlane < 3$ then
24         Intersect Tetrahedron Edges (2)
25         if $numVertsOnPlane = 0 \wedge numEdgesSliced > 0$ then
26             Get Tetrahedron Slice (3)
27         else if $numVertsOnPlane = 1 \wedge numEdgesSliced = 2$ then
28             Get Vertex-on Slice (4)
29         else if $numVertsOnPlane = 2 \wedge numEdgesSliced = 1$ then
30             Get Edge-on Slice (5)
31     else if $numVertsOnPlane = 3$ then
32         Get Tetrahedron Face (6)
33     else if $numVertsOnPlane = 4$ then
34         Get Full Tetrahedron (7)
35     Add Vertices CCW (8)
36 end
37 return $intersectionTriangles$
```

Algorithm 2: Intersect Tetrahedron Edges

```
$edgesSliced \leftarrow 0b000000$
$edgeBit \leftarrow 0b100000$
$tetIntersectPoints \leftarrow \emptyset$
$tetEdges \leftarrow allTetEdges[idxt]$
$edgeIsSliced \leftarrow \emptyset$
for $idxe \leftarrow 0$ to $5$ do
    $ev0 \leftarrow transVerts[tetEdges[idxe][0]]$
    $ev1 \leftarrow transVerts[tetEdges[idxe][1]]$
    $edgeIsSliced[idxe] \leftarrow 0$
    if $(ev0.w > 0 \wedge ev1.w < 0) \vee (ev0.w < 0 \wedge ev1.w > 0)$ then
        $edgesSliced \leftarrow edgesSliced | edgeBit$
        $tetIntersectPoints[idxe] \leftarrow intersectHyperplane(ev0, ev1)$ (9)
        $edgeIsSliced[idxe] \leftarrow 1$
        $numEdgesSliced \leftarrow numEdgesSliced + 1$
    $edgeBit \leftarrow edgeBit/2$
end
```

Algorithm 3: Get Tetrahedron Slice

```
$tmpIntersectOrder \leftarrow edgeIntersectOrders[edgesSliced]$
for $idxv \leftarrow 0$ to $length(tmpIntersectOrder) - 1$ do
    $slicePoints.push(edgeIntersectPoints[tmpIntersectOrder[idxv]])$
end
$sliceTriangles.push(slicePoints[0, 1, 2])$
if $length(tmpIntersectOrder) = 4$ then
    $sliceTriangles.push(slicePoints[0, 2, 3])$
```

Algorithm 4: Get Vertex-on Slice

```
$slicePoints.push(tetVertsOnPlane[0])$
for $idxe \leftarrow 0$ to $5$ do
    if $edgeIsSliced[idxe] = 1$ then
        $slicePoints.push(edgeIntersectPoints[idxe])$
end
$sliceTriangles.push(slicePoints[0, 1, 2])$
```

Algorithm 5: Get Edge-on Slice

```
$slicePoints.push(tetVertsOnPlane[0])$
$slicePoints.push(tetVertsOnPlane[1])$
for $idxe \leftarrow 0$ to $5$ do
    if $edgeIsSliced[idxe] = 1$ then
        $slicePoints.push(edgeIntersectPoints[idxe])$
end
```

Algorithm 6: Get Tetrahedron Face

```
1 $sliceTriangles.push(tetVertsOnPlane[0, 1, 2])$
```

Algorithm 7: Get Full Tetrahedron

```
1 $sliceTriangles.push(tetVertsOnPlane[0, 1, 2])$
2 $sliceTriangles.push(tetVertsOnPlane[0, 2, 3])$
3 $sliceTriangles.push(tetVertsOnPlane[0, 3, 1])$
4 $sliceTriangles.push(tetVertsOnPlane[1, 2, 3])$
```

Algorithm 8: Add Vertices CCW

```
1  $tetNormal4D \leftarrow calculate4DTetrahedronNormal(v0, v1, v2, v3)$ (10)
2  $tetNormal3D \leftarrow tetNormal4D.xyz$
3  for $idxtri \leftarrow 0$ to $length(sliceTriangles) - 1$ do
4      $tedge1 \leftarrow sliceTriangles[idxtri][1] - sliceTriangles[idxtri][0]$
5      $tedge2 \leftarrow sliceTriangles[idxtri][2] - sliceTriangles[idxtri][1]$
6      $sliceNormal3D \leftarrow tedge1 \times tedge2$
7      if $sliceNormal3D \cdot tetNormal3D < 0$ then
8          $sliceTriangles[idxtri].reverse()$
9      $intersectionTriangles.push(sliceTriangles[idxtri])$
10 end
```

Algorithm 9: intersectHyperplane

```
  Input: two vertices $tv1$ and $tv2$
  Output: The point where the edge formed by $tv1$ and $tv2$ intersects the 3D hyperplane
1 $factor \leftarrow |tv1.w|/|tv1.w - tv2.w|$
2 $new4DPt \leftarrow (tv1 * (1 - factor)) + (tv2 * factor)$
3 return $new4DPt.xyz$
```

Algorithm 10: calculate4DTetrahedronNormal

```
  Input: four vertices $v0, v1, v2, v3$
  Output: A 4D vector that is perpendicular to the tetrahedron
1 $v1\_v0 \leftarrow v1 - v0$
2 $v2\_v0 \leftarrow v2 - v0$
3 $v3\_v0 \leftarrow v3 - v0$
4 return $cross4D(v1\_v0, v2\_v0, v3\_v0)$ (see [29])
```

4.3 Interactive 4D Transformation Widget

Interactive transformation allows an animator to translate, rotate, or scale an object quickly without having to enter its value manually. This is a desirable feature when it is more important to set multiple values quickly than it is to set exact values. Most 3D animation packages implement a comparable feature [5–7, 22, 23, 39, 50]. Interactivity also facilitates the development of intuition [16]. We have developed an interactive transformation widget which allows the animator to adjust the position, orientation, and size of a 4D object in 4D space. The widget adapts the conventions of 3D transformation widgets, namely, color-coded control handles with specific shapes to indicate the type of transformation.

4.3.1 Translation and Scaling

Since the appearance and use of the translation and scaling widgets are nearly identical, they are combined in this section with their differences explained. As seen in figure 4.7 and figure 4.8, the widgets include four color-coded handles which correspond to the four principal axes: red for X, green for Y, blue for Z, and yellow for W. Each handle has a shape which indicates both the direction of the handle and the type of transformation. Conventionally, a cone denotes translation and a cube denotes scaling. The direction and length of each handle is determined as described next: each handle points along a principal axis in 4D. The handles are rotated to match the orientation of the 4D object, then projected to the 3D hyperplane. If the length of the projected handle is greater than 0.05, it is normalized. However, if it is less than or equal to 0.05, the length is retained. A short handle is a visual cue that the axis is perpendicular or nearly perpendicular to the 3D hyperplane. When the animator clicks and drags a handle, the selected transformation is applied to the 4D object.

4.3.2 Rotation

The rotation widget follows a similar color-coding scheme, but the handles are circles. Recall from section 3.2.1 that there are six rotation planes in 4D. The color-coded rotation planes

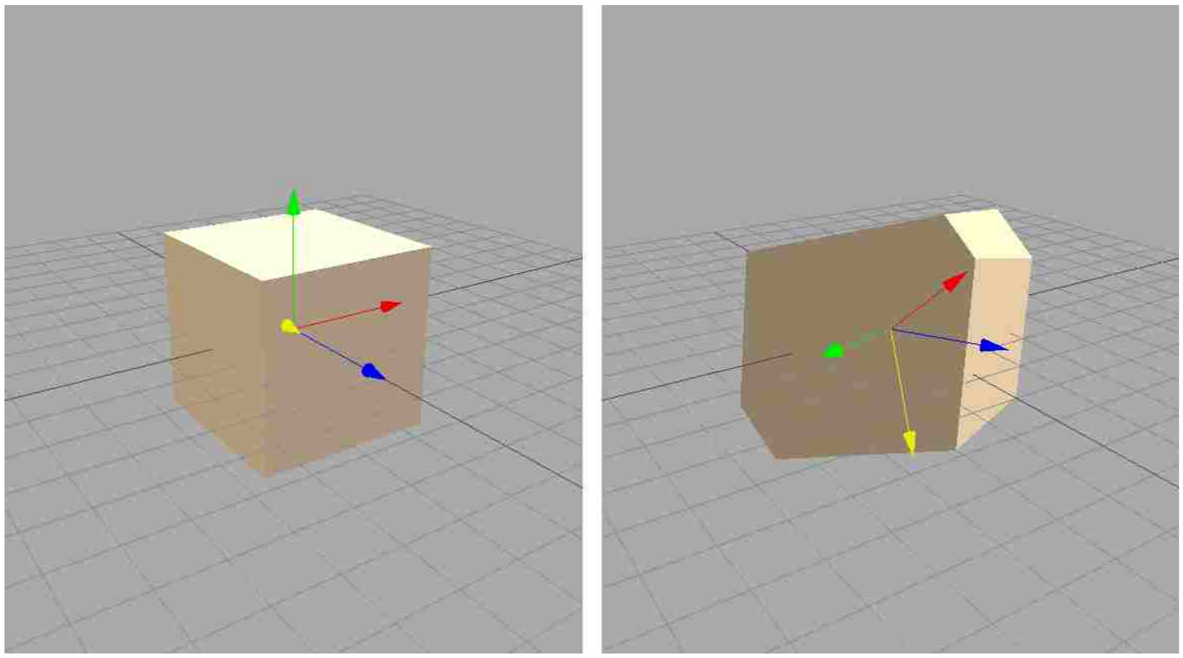


Figure 4.7: The 4D translation widget with no rotation (left) and some rotation (right).

are as follows: red for YZ, green for XZ, blue for XY, yellow for XW, cyan for ZW, and magenta for YW. Each rotation plane is represented by a circle. For each handle, 32 vertices are arranged in a circle on the corresponding rotation plane. The vertices are rotated to match the orientation of the 4D object, and then are projected to the 3D hyperplane. The projection of the circle in 3D will be either a perfect circle, an ellipse, or a line segment, depending on its orientation. A perfect circle indicates that the particular rotation plane is parallel to the 3D hyperplane while a line segment indicates that the particular rotation plane is perpendicular to the 3D hyperplane. This visual cue informs the animator of the orientation of the object in 4D (see figure 4.9). When the animator clicks and drags a handle, the selected rotation is applied to the 4D object.

4.4 Preserving Conventions and Characteristics of 3D Animation Systems

A primary goal of Fourveo is to make 4D animation accessible to 3D animators. Fourveo shares several core features of mainstream 3D animation software including affine transformations, keyframe interpolation, real time visualization, and scene hierarchy, among others. The 4D

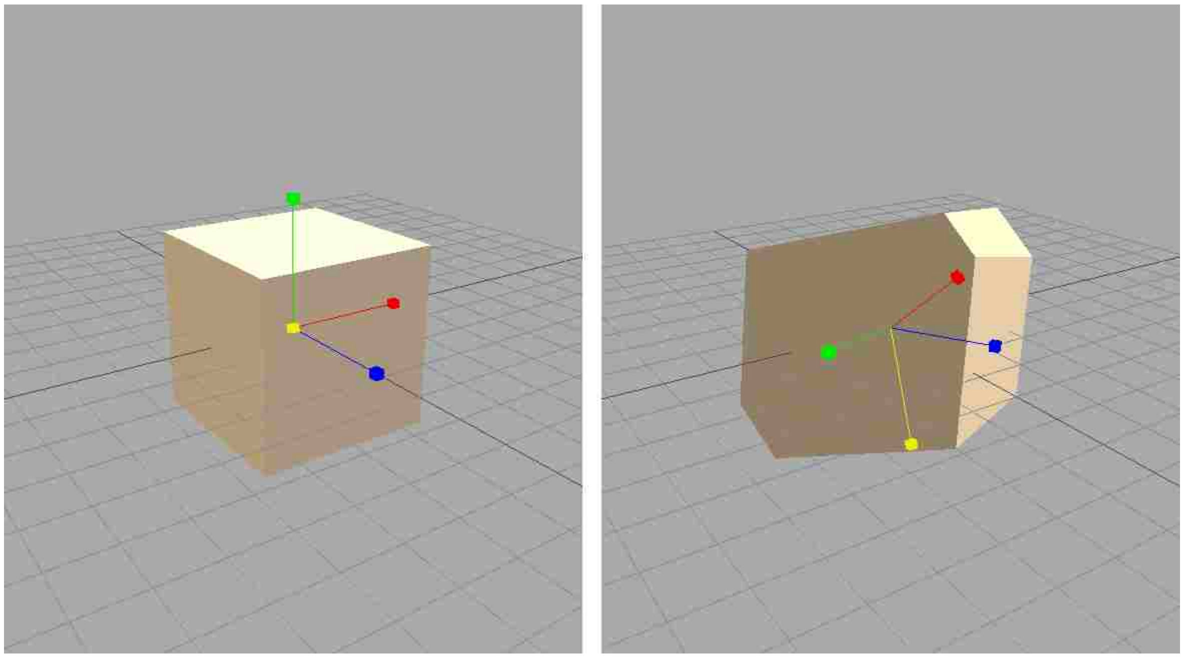


Figure 4.8: The 4D scaling widget with no rotation (left) and some rotation (right).

objects supported by Fourveo are hierarchical data structures (made up of vertices, edges, faces, etc.), similar to the polygonal mesh representation of 3D objects in 3D animation software. Finally, the GUI offers the same basic functionality as that of 3D animation software. It is important that we make the learning curve as gentle as possible since understanding 4D objects and 4D space is difficult. Preserving key conventions and characteristics of 3D animation software helps alleviate this difficulty.

4.4.1 Conventions and Characteristics Borrowed from 3D Animation Systems

As in conventional 3D animation software, the animator is given full control over the transformation of scene objects and the camera. Transformations may be adjusted in three ways: 1) interactively in the viewport (section 4.3), 2) explicitly in the object's properties panel, and 3) by manipulating keyframes in the keyframe curve editor. Fourveo also allows the animator to create a scene graph in which he can define parent-child relationships between scene objects. Because of these parent-child relationships, each object inherits the

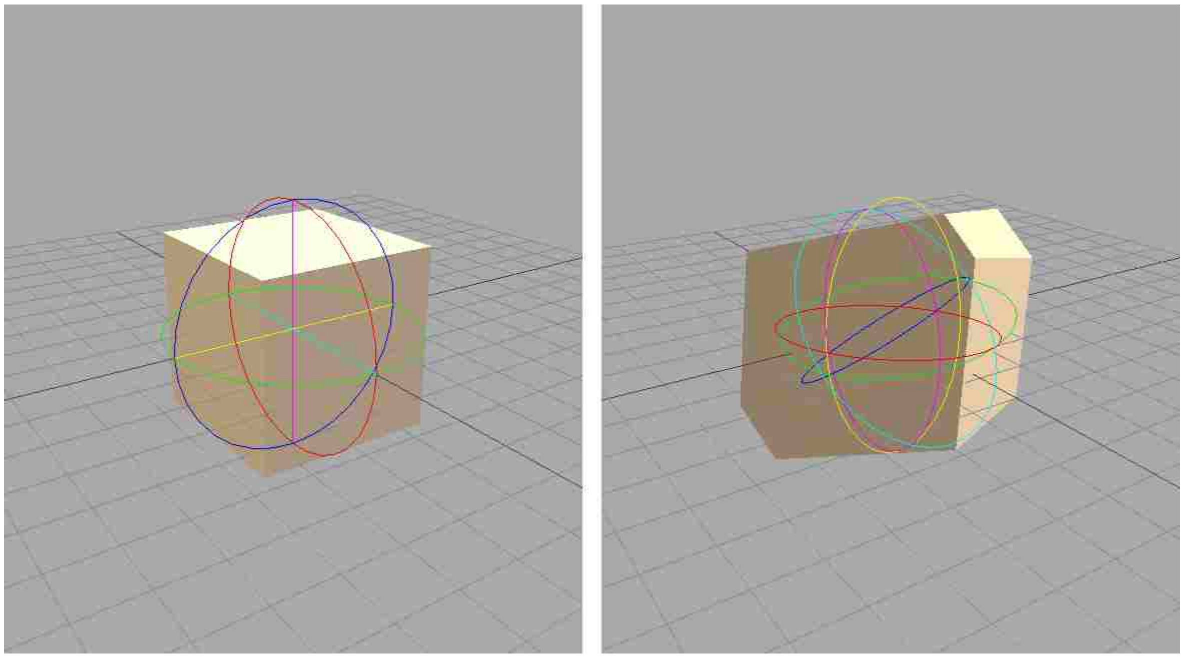


Figure 4.9: The 4D rotation widget with no rotation (left) and some rotation (right). In the left image, three of the circular handles are projected to lines which visually indicate that those rotation planes are oriented perpendicular to the 3D hyperplane.

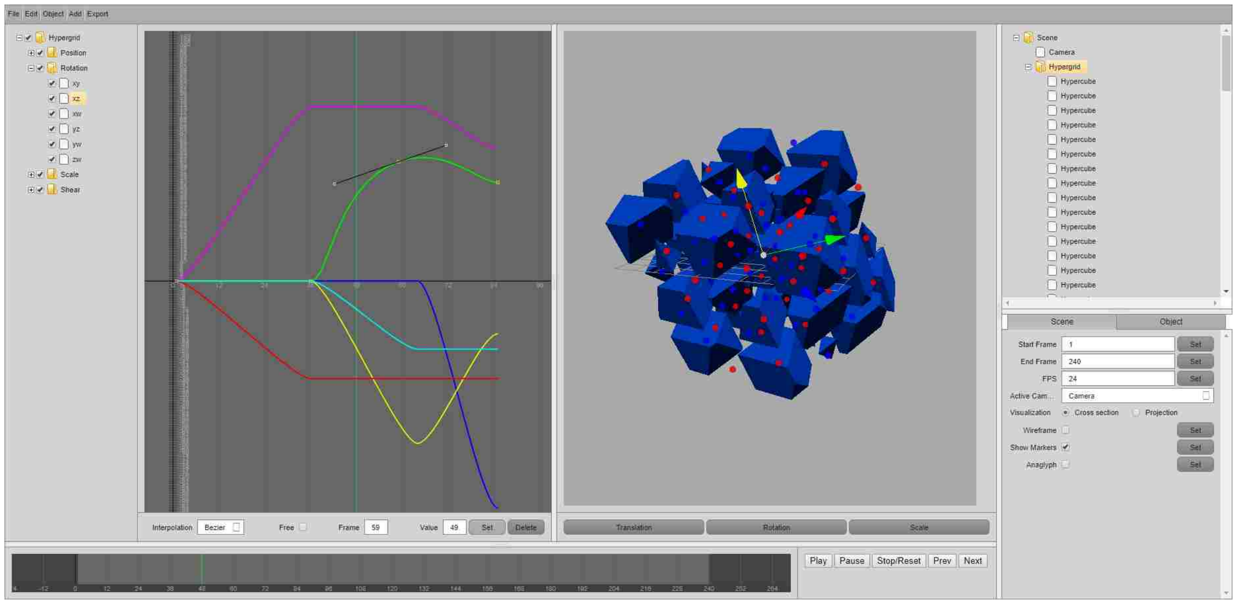
transformation of its parent. Each object also may have its own transformation, always relative to the transformation of the parent.

The graphical user interface of Fourveo also resembles the graphical user interface of conventional 3D animation software packages. The interface includes an interactive viewport, a scene graph, an object properties panel, a timeline, an animation curve editor, and a file menu (see figure 4.10).

4.4.2 Enabling Workflow to Span Several Software Packages

We give the animator the ability to include Fourveo into a conventional 3D animation workflow. We do so with two features: 1) Exporting 4D Animation into 3D, and 2) Importing 3D Animation into 4D.

Exporting 4D Animation into 3D: Most 3D animation software allows some sort of data exchange via geometry export/import operations. We allow the animator to export the animation to a series of 3D Wavefront .OBJ files, where each frame in the timeline is



Interpolation: Bezier
Free: ☐
Frame: 59
Value: 49

Scene | Object

Start Frame: 1
End Frame: 240
FPS: 24
Active Cam...: Camera
Visualization: (● Cross section) ○ Projection
Wireframe: ☐
Show Markers: ✓
Anaglyph: ☐

Figure 4.10: The Fourveo GUI, consisting of the interactive viewport (right middle), the scene graph (top-right), the object and scene properties panels (bottom-right), the timeline (bottom), the animation curve editor (left), and the file menu (top).

represented by a single .OBJ file. The Wavefront OBJ file format is a nearly ubiquitous format and often is used for exchanging geometry between 3D software packages. By exporting a 4D animation into this format, the animator is able to import it to nearly any 3D animation package and apply further effects such as shading, lighting, texture, and photorealistic rendering. It also enables him to combine 4D animated objects with traditional 3D animation, where 3D objects react to 4D objects (see section 1.5.3 for details on 3D-4D interaction).

Importing 3D Animation into 4D: We have included a 3D scene importer which reads 3D animation data stored in the COLLADA format. This format is a fairly universal scene description format which includes both geometry and keyframe animation, among other things. With this feature, the animator is able to animate an object in 3D and have a 4D object react to the animation. See section 1.5.3 for details on 3D-4D interaction, and see section 5.2 for a detailed explanation of the use of this feature in 3D/4D animation.

4.4.3 A 4D Geometry File Format

While Fourveo can generate several simple 4D shapes (see section 3.5.2), the animator may want to import different arbitrary 4D shapes for use in his animations. We describe a simple 4D geometry file format for data exchange between this and future 4D systems. The file format is inspired by the Wavefront OBJ format for 3D geometry. The format assumes a mesh composed of tetrahedra and allows 3D texture coordinates and 4D tetrahedron normals. It also allows tetrahedron-to-cell mapping, which allows the animator to indicate that several adjacent tetrahedra form a single co-hyperplanar solid.

A sample file with clarifying comments is included here (see Appendix B for a sample file containing a hypercube):

```
# Comments are allowed, and start with '#'


# List of geometric vertices, with (x,y,z,w) coordinates.
v 0.123 0.234 0.345 0.456
v ...
...
# (Optional) List of texture coordinates, in (u,v,w) coordinates,
# these will vary between 0 and 1.
vt 0.500 1 0.250
vt ...
...
# (Optional) List of tetrahedron normals in (x,y,z,w) form;
# normals might not be unit vectors.
tn 0.000 0.000 0.000 1.0
tn ...
...
# Tetrahedron cell element (see below)
```

```
t 1 2 3 4
t 3/1 4/2 5/3 6/4
t 6/4/1 3/5/3 7/6/5 4/7/7
t 7//1 8//2 9//3 10//4
t ...
...
# Vertex Indices
# t v1 v2 v3 v4

# Vertex Texture Coordinate Indices
# t v1/vt1 v2/vt2 v3/vt3 v4/vt4

# Tetrahedron Normal Indices
# t v1/vt1/tn1 v2/vt2/tn2 v3/vt3/tn3 v4/vt4/tn4

# Tetrahedron Normals without Texture Coordinate Indices
# t v1//tn1 v2//tn2 v3//tn3 v4//tn4

# (Optional) Cell Indices
# If two or more tetrahedra are part of a more complex cell (a cube,
# for example), you can group them together. This allows their
# cross-sections to be simplified since they will be coplanar
# (assuming the tetrahedra in each cell are cohyperplanar).
# There should be one entry in this list for each tetrahedron
# listed earlier in the file.
# They should be in the same order as the tetrahedron list
c 1
```

c 2

c 2

c . . .

. . .

Chapter 5

Animation Workflow

5.1 Suggested 3D/4D Animation Workflow

Simple 4D modeling and 4D animation can be done entirely in Fourveo where the animator may also want to depict various interactions between 4D and 3D objects. There are several possible scenarios for 3D/4D animation (see section 1.5.3 on 3D/4D interaction):

- Pure 4D animation
- 3D/4D animation, where the 3D and 4D objects interact with each other
- 3D/4D animation, where there is no interaction between different-dimensioned objects

These scenarios and their respective workflows are described in the following sections.

5.1.1 Pure 4D Animation

The animator may want to create an animation that contains only 4D objects. In this scenario, every object is free to transform in 4D space. The workflow for this scenario is fairly simple because there is no interaction between 3D and 4D objects, so all modeling and animation is created within Fourveo. Once the animation is completed, the animator can export it to 3D once for final rendering. Since there is no interaction between 3D and 4D objects to fine-tune, the number of exports to 3D and imports from 3D is minimal.

5.1.2 3D and 4D Objects Interacting Simultaneously

The animator may want to create an animation which contains both 3D and 4D objects and in which the 3D and 4D objects interact with each other, both directly and indirectly. In this scenario, the 4D objects are free to transform in 4D space while the 3D objects are restricted to transformation in 3D space. As suggested in section 1.5.3, a 4D object can apply forces on a 3D object such that the 3D object transforms in 4D space. The 3D object, however, can apply forces on a 4D object such that the 4D object transforms only in 3D space. The workflow for this scenario can be more complex than that of the scenario described in section 5.1.1. The animator can begin both 3D and 4D modeling and animation simultaneously if they are independent of each other. Once he begins animating the interactions, his workflow may involve animating in 4D, exporting to 3D, animating in 3D, evaluating the result, importing from 3D, and making adjustments to both 3D and 4D animation. He may need to repeat this part of the workflow several times to reach the desired result. Once the animation is completed, he may apply lighting and shading before doing the final rendering.

5.1.3 3D and 4D Objects Without Interaction

In a third scenario, both 3D and 4D objects exist in the animation, but neither affects the other. A 3D object is not affected by any 4D objects in any way. Likewise, a 4D object is not affected by any 3D objects in any way. While these scenarios are less interesting to investigate, they are included for the sake of completeness. The workflow for these scenarios is similar to that of pure 4D animation, but with the addition of 3D animation. The animator models and animates 3D and 4D objects simultaneously in their respective animation systems. Once the 4D animation is complete, he may export it to 3D. With the 3D and 4D animation combined, he may apply lighting and shading before the final render.

5.2 3D Animation Import

Physical interaction between 3D and 4D objects may be wanted by an animator. This requires coordination between the objects to determine the precise timing of physical contact. A simple method for creating such an animation is to create the 4D animation in Fourveo, export it to 3D animation software, check to see if the position and timing of the physical contact is adequate, adjust the 4D animation in Fourveo, and repeat the process until the desired result is achieved. Unfortunately, this method can be time-consuming because the creation of animation happens in Fourveo but the timing of the interaction must be verified in 3D software.

To reduce the time required to animate physical interaction between 3D and 4D objects, we have implemented a feature in Fourveo which imports 3D animation into the 4D animation workspace. This allows the animator to determine whether the positioning and timing of a physical interaction is satisfactory without multiple time-consuming OBJ sequence export steps. In animations where there is substantial 3D/4D interaction, this feature greatly reduces the time required to produce the animations. Fourveo accepts the COLLADA 3D interchange file format, which is implemented by many 3D animation packages, including all of the packages mentioned in section 3.1.

5.3 3D/4D Animation Workflow Case Studies

Several animations were created with Fourveo to demonstrate that it can be included in a 3D animation workflow. This section describes four of those animations. Blender is the 3D animation software used in these examples. All 4D objects were animated in Fourveo, exported as OBJ sequences, then shaded, illuminated, and rendered in Blender. For each animation listed, we include a brief description of the animation, a detailed listing of workflow steps taken, a single rendered frame from the animation, and a link to the animation hosted on YouTube.

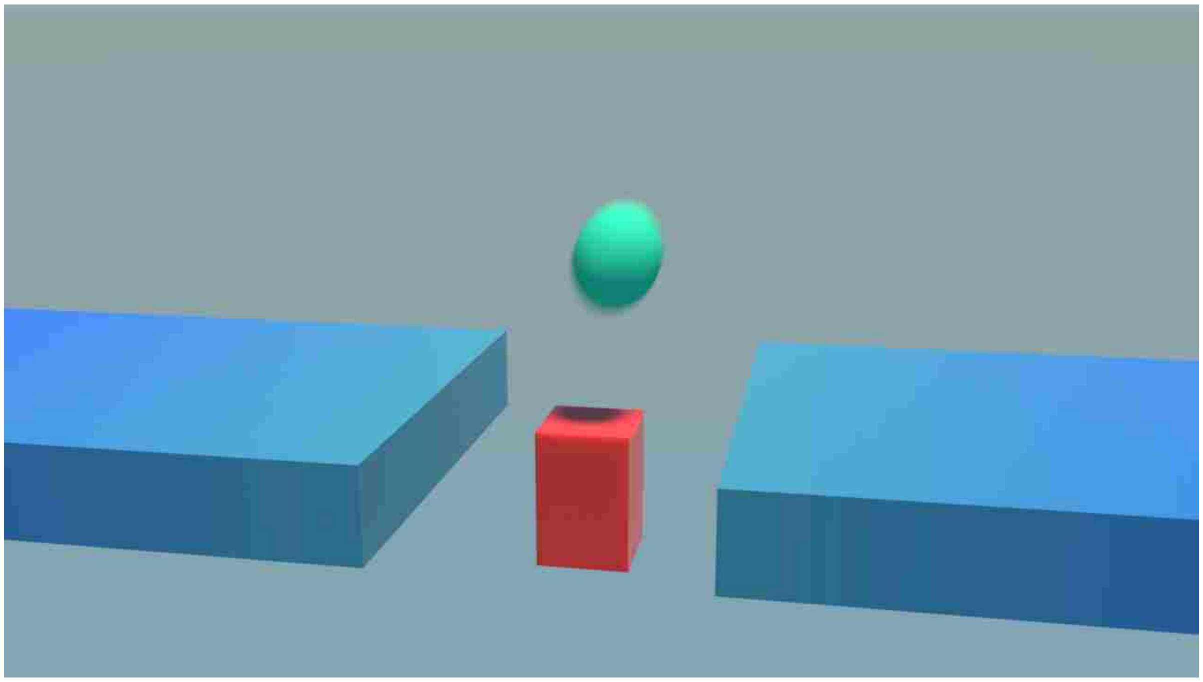


Figure 5.1: A selected frame from the Ball Bouncing on Hypercube animation

5.3.1 Ball Bouncing on Hypercube

Description: Two flat boxes are suspended in the air and separated by a wide gap. The cross-section of a hypercube floats between them. A 3D ball bounces from one box to the other, using the hypercube as a stepping stone. When the ball reaches the other box, it reverses its direction, intending to use the hypercube as a stepping stone again. However, as the ball approaches the hypercube, the hypercube moves Ana such that it has no intersection with the 3D hyperplane. With nothing to catch it, the ball falls down between the two boxes. See figure 5.1.

Link to animation: https://youtu.be/fRd379UpVpg

Workflow:

- In the File Explorer, create a project folder for this animation
- In Blender, create the ball, the two platforms it will bounce between, and a cube which serves as a placeholder for the cross-section of a hypercube.

- Animate the ball bouncing from one box to the hypercube to the other box, then back to where the hypercube was. Animate it falling down the gap.
- Select the animated ball and hypercube placeholder and export them as a single COLLADA file.
- In Fourveo, import the COLLADA file exported from Blender.
- Create a hypercube and transform it so that its cross-section perfectly overlaps the hypercube placeholder.
- Animate the hypercube reacting to the ball bouncing off it. Also animate the hypercube so that its cross-section disappears.
- Select the hypercube and export frames 1-72 (the duration of the animation) as OBJ files
- In the File Explorer, find the exported files and move them to the project folder.
- In Blender, load the 72 OBJ frames.
- Watch the animation and decide whether it needs any adjustment
- If it needs adjustment:
  - In Blender, optionally make adjustments to the bouncing ball.
  - Export the bouncing ball to COLLADA.
  - In Fourveo, replace the old COLLADA file with the new COLLADA file.
  - Optionally make adjustments to the hypercube.
  - Export the hypercube as OBJ frames.
  - In the File Explorer, find the exported files and move them to the project folder.
  - In Blender, import the new OBJ frames.
  - Watch the animation and decide whether it needs adjustments.
  - If further adjustment is required, repeat these steps.

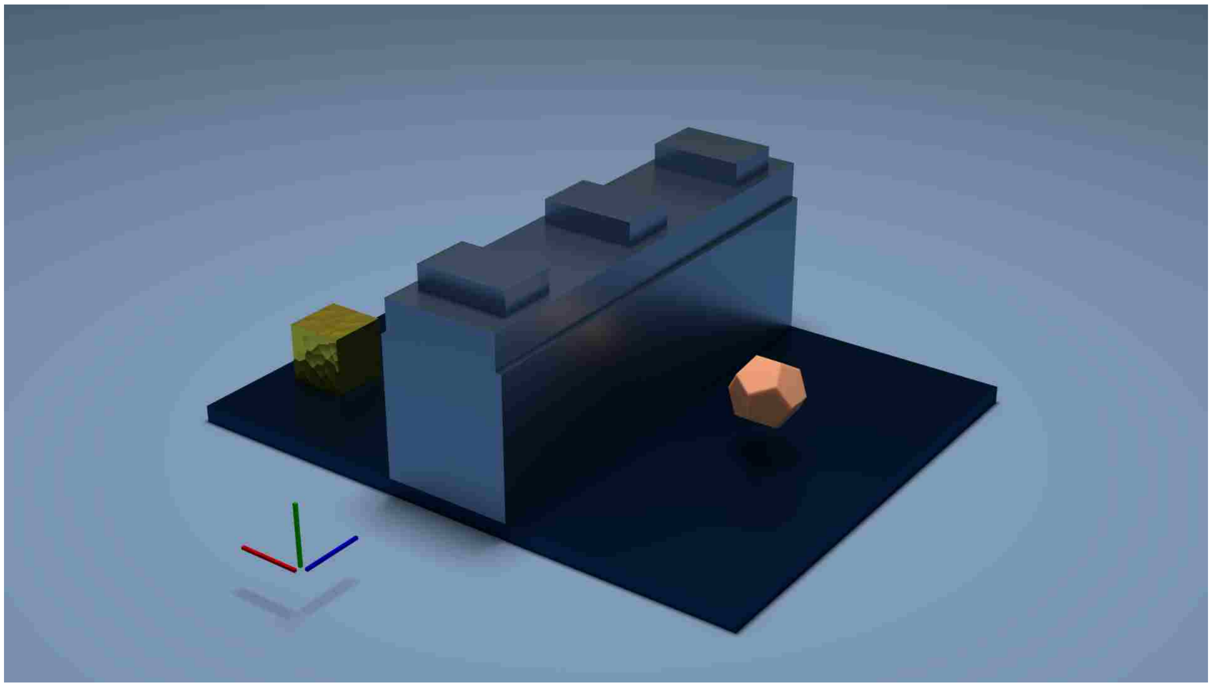


Figure 5.2: A selected frame from the Walking Through Walls animation

- In Blender, add lights, shaders, and camera positioning.
- Render the animation

5.3.2 Walking Through Walls in the Fourth Dimension

Description: A dodecahedron is separated from a large, gold treasure by a large wall. It has the ability to rotate in 4D, so it turns to face the 4th dimension and travels some distance along the W axis. It turns back and notices that the wall is much shorter at this location in 4D space, so it crosses the wall. It turns again to face the 4th dimension and walks backward some distance along the W axis. When it turns back, it finds itself on the other side of the wall and can claim the gold treasure. See figure 5.2.

Link to animation: https://youtu.be/ceacqWXYR5Y

Workflow:

- In the File Explorer, create a project folder for this animation.

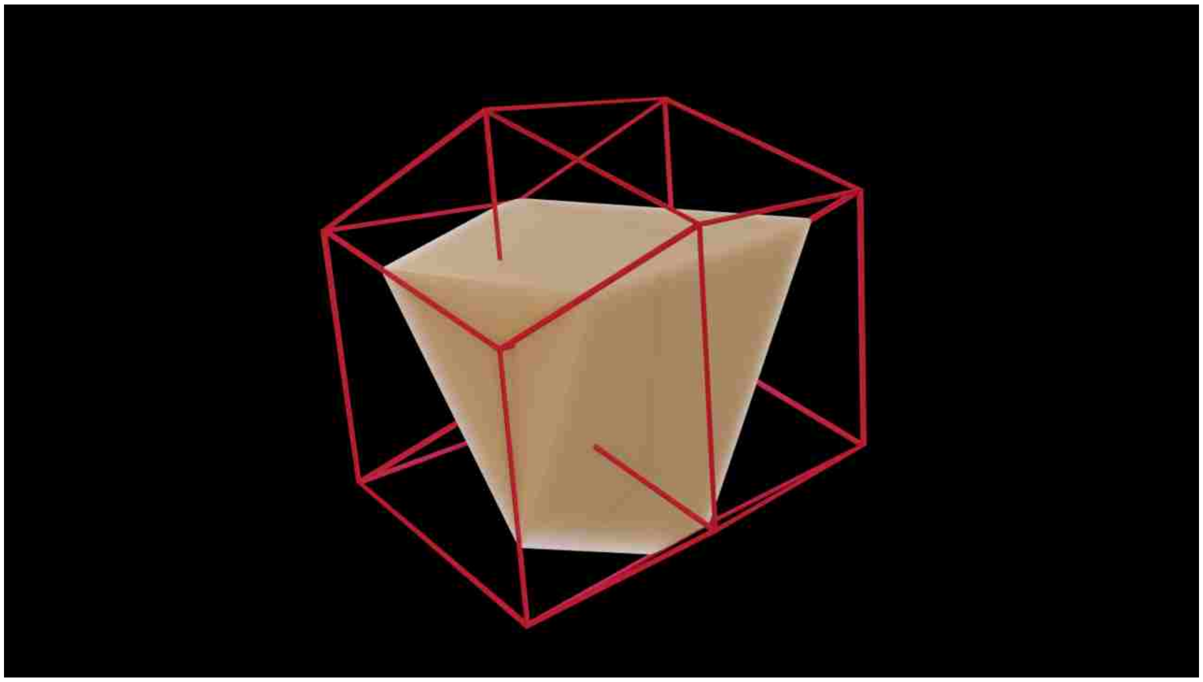


Figure 5.3: A selected frame from the Cross-section inside Projection animation

- In Fourveo, create the ground, the tall wall, the short wall, the dodecahedron, the gold treasure, and the axes.
- Animate all of the scene rotations and movements of the dodecahedron.
- For each distinct shape:
  - In Fourveo, select the shape.
  - Export frames 1-800 (the duration of the animation) as OBJ files.
  - In the File Explorer, find the exported files and move them to the project folder.
  - In Blender, import the OBJ sequence for this object
- In Blender, add lights to the scene and shading to each object.
- Also add a large ground plane.
- Render the animation.

5.3.3 Hypercube Cross-Section inside Parallel Projection

Description: A hypercube passes through the 3D hyperplane. Both its cross-section and parallel projection are shown. See figure 5.3.

Link to animation: https://youtu.be/9aFdtx2VgEA

Workflow:

- In the File Explorer, create a project folder for this animation.
- In Fourveo, create a hypercube and rotate it some amount in each of the six rotation planes.
- Create an empty object and make the hypercube a child of the empty object.
- Animate the empty object along the W axis such that the child hypercube passes completely through the 3D hyperplane.
- Select the hypercube and export frames 1-120 as OBJ files.
- Switch the visualization mode to Projection and set the camera to Parallel Projection.
- Select the hypercube and export frame 1 (since the projection does not move with this particular animation).
- In the File Explorer, find the exported files and move them to the project folder.
- In Blender, load the 120 OBJ files from the cross-section visualization.
- Load the OBJ file from the parallel projection visualization.
- Add lights to the scene and shading to the hypercubes.
- Render the animation.

5.3.4 Transforming Perspective Projected Hypercube

Description: A hypercube is visualized with a perspective projection as a wireframe mesh. The hypercube rotates 90 degrees, expands to twice its size, then undergoes two uniform shearing transformations. The visualization includes some 4D camera movement. Finally, all of the transformations revert to their original configurations. The animation is rendered with red-cyan anaglyph stereoscopic rendering. See figure 5.4.

Link to animation: https://youtu.be/1xNMGksou2w

Workflow:

- In the File Explorer, create a project folder for this animation.
- In Fourveo, create a hypercube and three empty objects: one for scaling, one for shearing X based on Y, and one for shearing Z based on W.
- Switch the visualization mode to Projection and set the camera to Perspective Projection.
- Enable Stereo Anaglyph rendering.
- Set the hypercube as a child of the scaling object, the scaling object as a child of the shearing XY object, and the shearing XY object as a child of the shearing ZW object.
- Animate the hypercube rotating, the scaling object scaling, and the two shearing objects shearing.
- Animate the 4D camera's position.
- Select the hypercube and export frames 1-300 as OBJ files.
- In the File Explorer, find the exported files and move them to the project folder.
- In Blender, load the 300 OBJ files from the cross-section visualization.
- Load the OBJ file from the perspective projection visualization.
- Add lights to the scene and shading to the hypercubes.
- Render the animation.

5.3.5 Features Used by Case Studies

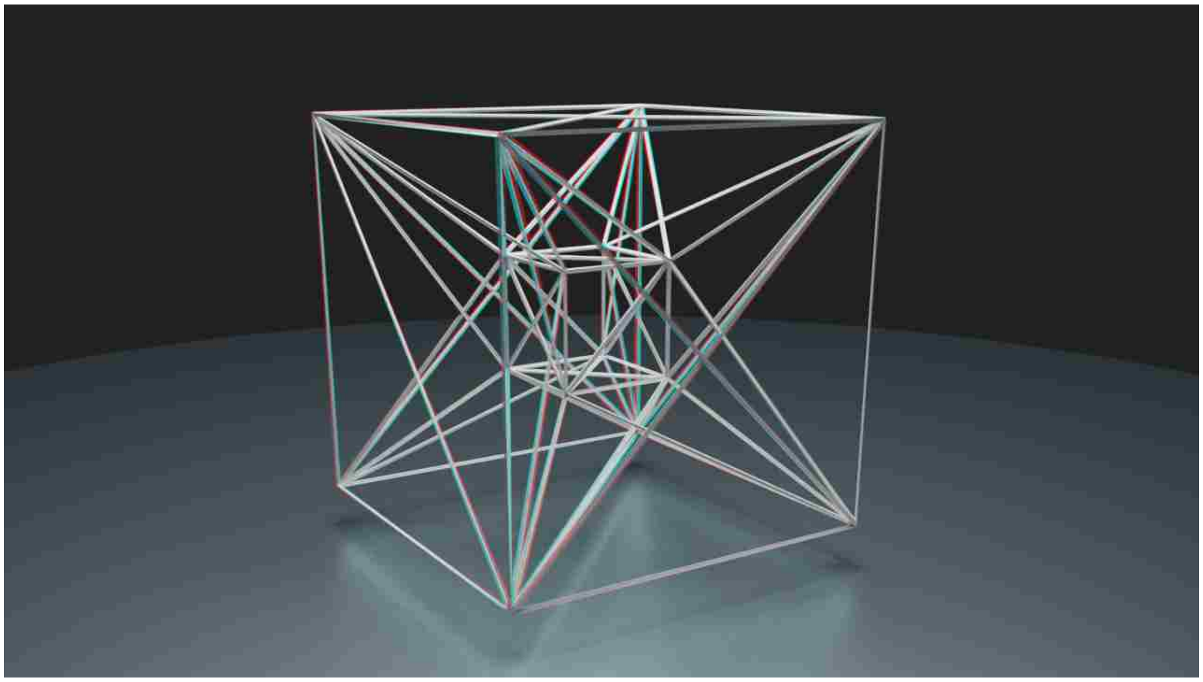


Figure 5.4: A selected frame from the Perspective Projected Hypercube animation

	Bouncing Ball	Walk through Walls	Cross-section inside Projection	Perspective Projected Hypercube
Translation	x	x	x	
Rotation	x	x		x
Scaling				x
Shearing				x
Camera Movement				x
Keyframe interpolation	x	x	x	x
Animation graph editor	x	x	x	x
3D cross-section	x	x	x	
4D to 3D perspective				x
4D to 3D parallel			x	
4D to 2D projection			x	x
Wireframe			x	x
Solid	x	x	x	
3D stereoscopic				x
Parenting		x		x
Multiple objects		x		x
Realtime playback	x	x	x	x
Non-linear editing	x	x	x	x

Table 5.1: Fourveo features used by each animation case study

Chapter 6

Validation

6.1 Animation System Testing

To establish the validity of Fourveo, we performed a set of tests that establish the correctness of the core features, showed that they work in concert, and demonstrated that Fourveo can be used in a 3D animation workflow.

6.1.1 Core Feature Tests

These features were tested via unit testing. Vector and matrix math classes and functions were tested. All affine transformation matrices are based on previous work by [3, 29, 30, 41] and were tested in Fourveo. The projection matrices, both parallel and perspective, are based on previous work by [29, 30, 41] and were tested in Fourveo. Nested transformations depend on a correct implementation of matrix multiplication, which was tested. Keyframe animation, both the classes for keyframes and the functions for keyframe interpolation, was tested.

6.1.2 Novel Feature Tests

Novel features were also tested. The novel slicing algorithm was tested by enumerating all tetrahedron slicing configurations and testing each of them. As part of this test, consistent winding order in cross-section geometry was verified in Fourveo, exported to Blender and verified there. Some parts of this feature were based on previous work by [18, 19, 29]. Finally, the 4D interactive transformation widget was tested with all supported transformations.

6.1.3 Animation Workflow Tests

In addition to unit testing of core and novel features, several short animations were produced. These are practical tests which, collectively, cover the full feature set of Fourveo (see table 5.1). The workflows and results are described in detail in section 5.3. Both the specific steps required to create each animation, and the animations themselves demonstrate that 4D animations created in Fourveo can be included in a 3D animation workflow.

6.2 Superiority of the 4D Animation System

Table 6.1 shows that Fourveo includes a feature that is not included by any other 4D animation system: the Animation Graph Editor. Additionally, there are features included in Fourveo that are possible in existing systems only via user-provided scripts: the Shear transformation, the Reflection transformation, Camera Movement, Object Hierarchy (nested transformations), and support for Multiple 4D Objects. These are possible in Mathematica and POV-Ray but their implementations require a deep understanding of four-dimensional geometry, computer graphics, and linear algebra. Finally, while scripting usually offers the most freedom to an animator, Fourveo still provides a high degree of freedom without the need for custom scripting. Hence, Fourveo represents the most complete 4D animation system.

Includes a greater number of desirable features: Fourveo has a set of features that allows true, general-purpose 4D animation. No existing system has been reported which offers this capability and is designed to be familiar to 3D animators. Referring to the final row of Table 6.1, when measured against this set of features, Fourveo achieves a score of 85.7, a score substantially higher than the score of its nearest existing system's score of 52.3. In contrast to the existing systems, Fourveo provides a substantially higher number of the desirable features for basic general-purpose 4D animation. It includes the relevant affine transformations, a camera which can be animated, keyframe interpolation that can be carefully adjusted, a scene hierarchy for nested transforms, and several visualization techniques. This feature set

	Stella4D	HyperSpace	Peek	Meshview	Matlab	Mathematica	POV-Ray	Fourveo
Translate	S		S	S	*	*	*	S
Rotate	S	S	S	S	*	*	*	S
Scale	S				*	*	*	S
Shear					*	*	*	S
Reflect					*	*	*	
Camera movement					*	*	*	S
Keyframe interpolation			S	S	*	*	*	S
Animation graph editor								S
Dope Sheet								
Scripted Animation					S	S	S	
3D cross-section	S	S	S		*	*		S
4D to 3D Perspective	S		S	S	*	*		S
4D to 3D Parallel	S	S	S	S	*	*		S
4D to 2D projection		S	S	S	*	*		S
Wireframe	S	S	S	S	*	*		S
Solid	S	S	S	S	*	*		S
3D Stereoscopic	S			S				S
Parenting					*	*	*	S
Multiple objects					*	*	*	S
Realtime Playback	S			S	*	*		S
Non-linear editing				S				S
Score	47.6	28.5	42.8	52.3	23.8	23.8	15.4	85.7

Table 6.1: Comparison of features of 4D animation software with those of Fourveo. S = supported, * = not supported, but possible via scripting. Scores are calculated as follows: ((F + 0.25s)/N) * 100, where F is the number of supported features, s is the number of features supported via scripting, and N is the total number of features.

allows an animator to import several 4D shapes and take them through the entire animation process without having to rely on other software.

Familiar to 3D animators: A primary goal of Fourveo is to make 4D animation accessible to 3D animators. Fourveo shares several core features of mainstream 3D animation software including affine transformations, keyframe interpolation, real time visualization, and scene hierarchy, among others. The 4D objects supported by Fourveo are hierarchical data structures (consisting of vertices, edges, faces, etc.), similar to the polygonal mesh representation of 3D objects in 3D animation software. Finally, the GUI offers the same basic functionality as that of 3D animation software (see figure 4.10).

High degree of control: Scripting provides the most freedom to an animator at the cost of introducing a more complex interface. Fourveo offers a high degree of control without requiring knowledge of scripting. Most entities in Fourveo have several properties that can be adjusted and/or animated. Affine transformations can be adjusted by the user in one of up to three different ways:

1. Interactively in a real time viewport: this method allows for multiple transformation values to be set simultaneously. It is useful for when speed, not precision, is a priority.
2. Interactively in the animation curve editor: this method allows for quick adjustment of values for a transformation, one at a time. It is the only way to fine-tune smooth interpolation.
3. Directly in the properties panel: this method is the slowest, but offers the greatest precision because the user can enter the values manually.

Object hierarchy (nested transforms): An object inherits the transformation of its parent object. It can also have its own transformation which is relative to the parent's transformation. Just as in 3D animation systems, this allows for the creation of complex motion that would otherwise be difficult to calculate and tedious to implement. Nested transforms allow a complex motion to be broken up into simple components. The presence of an object hierarchy also implies the ability to have more than one object present in the scene. Neither of these features has been implemented previously in 4D animation software.

Camera movement and animation: There are two types of cameras in Fourveo: 1) an interactive camera and 2) an animatable camera. The interactive camera's function is to allow a user to navigate quickly through the scene while creating an animated sequence. This camera is provided as a convenience to the animator; its orientation is disregarded in the final output of the animation. The animatable camera is one whose orientation may be animated and from whose viewpoint a scene may be rendered. This camera is considered part of the animation and will potentially require as much work to animate as any other animated object in the scene. Just as in conventional filmmaking, this camera allows the animator (the hyper-cinematographer, if you will) to guide a viewer's attention to certain parts of an animation and convey the mood through camera placement and movement.

Animation export: Fourveo allows the user to export a .OBJ sequence of the animated scene. This allows him to use the projected or sliced geometry in most conventional 3D animation software. This opens up numerous possibilities for further refinement of a 4D animated sequence that are not yet possible with this system. Some of these possibilities include lighting, shading/texturing, and incorporation of 4D animation into 3D animated sequences. Because of this feature, Fourveo can be included in an existing 3D animation workflow.

Chapter 7

Conclusion

7.1 Summary

Fourveo combines animation and computer graphics techniques that enable general-purpose 4D animation. This 4D animation capability is presented in an interface that resembles the interface of conventional 3D animation software. Fourveo introduces novel features that are not found in any existing systems, including Projected Markers, compatibility with conventional 3D animation software and 3D animation workflows, support for multiple 4D objects and nested transformations, and a 4D tetrahedral mesh file format. Fourveo has been designed and implemented as a general-purpose 4D animation system. Its features are natural extensions of conventional 3D animation system features with the intent of minimizing the learning curve for 3D animators.

7.2 Conclusions

Several useful conclusions were reached in the process of designing and implementing Fourveo. First, both the logical and graphical interfaces to conventional 3D animation software do extend to 4D. The aim of this specific extension is to lessen the learning curve associated with 4D space and geometry. Second, interactivity with 4D objects enhances intuition of 4D objects (see section 4.3). The 4D transformation widget facilitates interactivity and creates a tight feedback loop for the animator. Finally, trends in literature and popular culture show that interest in 4D topics is alive and well. As stated in section 1.3, several 4D-themed games and films have been produced in the recent past and have gained notoriety. Additionally, research in interactive 4D techniques continues with work published in 2013 [53], 2015 [34], and 2016 [48].

7.3 Future Work

Fourveo facilitates and invites the development of meaningful, specialized features which are beyond the intended scope of the work reported here. These features include 4D skeletal animation and a tighter integration with 3D animation software.

4D Skeletal Animation: 3D animated characters commonly are animated with the 3D skeletal animation technique. 4D animated characters could be animated with the extension of skeletal animation. 3D skeletal animation depends on quaternion rotation to rotate bones about arbitrary lines. Quaternion rotation works for 3D rotation, but needs to be extended to 4D rotation for 4D skeletal animation. This feature was not implemented because of the intentionally limited selection of simple 4D shapes provided by Fourveo and the absence of software for creating arbitrarily-complex 4D shapes. The simple shapes provided do not merit the implementation of a 4D skeletal animation system.

Tighter Integration with 3D Animation Software: Fourveo's OBJ sequence export feature allows it to be used with almost any 3D animation software, but introduces the time-consuming step of exporting then importing the sequence. Fourveo could be extended to integrate with 3D animation software in one of two ways: First, as a plugin for a specific 3D animation package. While this method would offer the tightest integration and the quickest workflow, the plugin would have to be re-implemented for each major 3D animation system. The core algorithms would be the same, but the source code likely would require customization. Second, Fourveo could be implemented as a standalone system, but with plugins to exchange geometry and animation data between itself and the intended 3D animation system. While this option offers more versatility than the first option, it would still require a dedicated plugin for each major 3D animation system. The paradigm we have implemented, where Fourveo is a standalone package and exports animation to sequences of OBJ files, has a more time-consuming workflow, but offers the greatest versatility because of the universality of the Wavefront OBJ file format. Versatility has been a priority in our work, but those who prefer an enhanced workflow may prefer to implement one of the other two paradigms described.

Several related activities are also recommended for future work. Since 4D animation is not yet widespread, there are no established conventions for 4D animation and storytelling. To establish 4D animation conventions, we might first decide which 3D animation conventions will extend readily to 4D, then experiment with new conventions specific to 4D. Modeling of arbitrary 4D geometric shapes is conspicuously absent from both Fourveo and any published work. 4D geometric modeling needs to allow a user to create complex monolithic shapes. To be successful, it needs to substantially increase the user's intuition about 4D and instill confidence that the actions they perform will produce a desired outcome. It must allow the user to discover the geometric possibilities of the added dimension.

References

[1] Winners & nominees. https://www.oscars.org/oscars/ceremonies/2015, 2015. [Online; accessed 15-June-2017].

[2] Edwin A Abbott. *Flatland: A romance of many dimensions*. OUP Oxford, 2006.

[3] William P Armstrong. Hyperdimensional graphics: the computer display of objects in hyperspace. Master's thesis, Brigham Young University, 1981.

[4] Fabrice Aubert and Dominique Bechmann. Animation by deformation of space-time objects. *Computer Graphics Forum*, 16(3):C57–C66, 1997. URL http://dx.doi.org/10.1111/1467-8659.00142.

[5] Autodesk. 3ds max help. docs.autodesk.com/3DSMAX/15/ENU/3ds-Max-Help/index.html, 2013. [Online; accessed 15-May-2015].

[6] Autodesk. Maya user's guide. download.autodesk.com/global/docs/maya2014/en_us/index.html, 2014. [Online; accessed 15-May-2015].

[7] Autodesk. Softimage user's guide. download.autodesk.com/global/docs/softimage2014/en_us/userguide/index.html, 2014. [Online; accessed 15-May-2015].

[8] Thomas F Banchoff. computer animation and the geometry of surfaces in 3-and 4-space. In *Proceedings of the International Congress of Mathematicians, Helsinki*, pages 1005–1013, 1978.

[9] David Banks. *Interacting with surfaces in four dimensions using computer graphics*. PhD thesis, University of North Carolina at Chapel Hill, 1993.

[10] Nelson Bond. The monster from nowhere. In *The Thirty-first of February*, pages 211–212. Gnome Press, 1949.

[11] Marc Ten Bosch. Designing a 4d world: The technology behind miegakure [hide&reveal]. https://www.youtube.com/watch?v=vZp0ETdD37E, 2016. [Online; accessed 8-March-2016].

[12] Paul Bourke. Hyperspace, user manual. http://paulbourke.net/geometry/hyperspace/, 1990. [Online; accessed 7-April-2015].

[13] Claude F Bragdon. *A primer of higher space (the fourth dimension)*. Cosimo, Inc., 2005.

[14] Sylvain Brandel, Dominique Bechmann, and Yves Bertrand. Stigma: a 4-dimensional modeller for animation. In *Proceedings of Computer Animation and Simulation*, pages 103–126. Springer, Vienna, 1999.

[15] Gary L Bringhurst. Hyperdimensional imagery: the raytracing of hyperobjects. Master's thesis, Brigham Young University, 1987.

[16] Edward G Britton, James S Lipscomb, and Michael E Pique. Making nested rotations convenient for the user. In *Proceedings of ACM SIGGRAPH Computer Graphics*, number 3, pages 222–227. ACM, 1978.

[17] Scott A Carey. A four-dimensional shading model. Master's thesis, Brigham Young University, 1987.

[18] SH Choi and KT Kwok. A tolerant slicing algorithm for layered manufacturing. *Rapid Prototyping Journal*, 8(3):161–179, 2002.

[19] Alan Chu, Chi-Wing Fu, Andrew J Hanson, and Pheng-Ann Heng. Gl4d: A gpu-based architecture for interactive 4d visualization. *IEEE transactions on visualization and computer graphics*, 15(6):1587–1594, 2009.

[20] Michael D'Zmura, Philippe Colantoni, and Gregory Seyranian. Virtual environments with four or more spatial dimensions. *Presence: Teleoperators and Virtual Environments*, 9(6):616–631, 2000.

[21] Thorsten Fleisch. Animating the fourth dimension. http://www.fleischfilm.com/html/texts.htm, 2005. [Online; accessed 2-April-2015].

[22] Blender Foundation. Blender manual contents. https://www.blender.org/manual/, 2015. [Online; accessed 3-March-2015].

[23] The Foundry. Modo online help. https://help.thefoundry.co.uk/modo/901/, 2015. [Online; accessed 10-May-2015].

[24] Andrew J Hanson. Rotations for n-dimensional graphics. *Graphics Gems V*, pages 55–64, 1995.

[25] Andrew J Hanson and Pheng-Ann Heng. Visualizing the fourth dimension using geometry and light. In *Proceedings of the 2nd conference on Visualization*, pages 321–328. IEEE Computer Society Press, 1991.

[26] Andrew J Hanson, Konstantine I Ishkov, and Jeff H Ma. Meshview: Visualizing the fourth dimension. *Overview of the MeshView 4D geometry viewer*, 1999.

[27] Charles H Hinton. *A New Era of Thought*. S. Sonnenschein & Company, 1888.

[28] Christoph M Hoffmann and Jianhua Zhou. Visualization of surfaces in four-dimensional space. Technical report, Purdue University, 1990.

[29] Steven R Hollasch. Four-space visualization of 4 d objects. Master's thesis, Arizona State University, 1991.

[30] Paul L Isaacson. Computer graphic presentation of hypothesized four-dimensional phenomena. Master's thesis, Brigham Young University, 1984.

[31] Michio Kaku. *Hyperspace: A scientific odyssey through parallel universes, time warps, and the tenth dimension*. OUP Oxford, 1995.

[32] Gordon Kindlmann. Polytope visualization: Peek. `http://www.cs.utah.edu/~gk/peek/`, 2000. [Online; accessed 2-June-2015].

[33] Hüseyin Koçak, Frederic Bisshopp, Thomas F Banchoff, and David H Laidlaw. Topology and mechanics with computer graphics: Linear hamiltonian systems in four dimensions. *Advances in Applied Mathematics*, 7(3):282–308, 1986.

[34] Nico Li, Daniel J Rea, James E Young, Ehud Sharlin, and Mario Costa Sousa. And he built a crooked camera: a mobile visualization tool to view four-dimensional geometric objects. In *SIGGRAPH Asia 2015 Mobile Graphics and Interactive Applications*, page 23. ACM, 2015.

[35] Mei-chi Liu. A four-dimensional shadow algorithm. Master's thesis, Brigham Young University, 1982.

[36] Shusen Liu, Bei Wang, Jayaraman J Thiagarajan, Peer-Timo Bremer, and Valerio Pascucci. Visual exploration of high-dimensional data: Subspace analysis through dynamic projections. Technical report, Scientific Computing and Imaging Institute University of Utah, 2014.

[37] Henry P Manning. *The Fourth Dimension, Simply Explained*. Dover Publications, 1960.

[38] Mathworks. Matlab documentation. http://www.mathworks.com/help/matlab/, 2015. [Online; accessed 19-April-2015].

[39] Maxon. Cinema 4d quickstart documentation. www.maxon.net/support/documentation.html, 2015. [Online; accessed 20-May-2015].

[40] Randall Munroe. Flatland. https://xkcd.com/721/, 2010. [Online; accessed 10-November-2014].

[41] A Michael Noll. A computer technique for displaying n-dimensional hyperobjects. *Communications of the ACM*, 10(8):469–473, 1967.

[42] A Michael Noll. Computer animation and the fourth dimension. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part II*, pages 1279–1283. ACM, 1968.

[43] Clifford A Pickover. *Surfing through hyperspace: Understanding higher universes in six easy lessons*. Oxford University Press on Demand, 1999.

[44] POV-Ray. Pov-ray for unix version 3.7 documentation. http://www.povray.org/documentation/3.7.0/, 2013. [Online; accessed 20-May-2015].

[45] Prabhat, David H Laidlaw, Thomas F Banchoff, and Cullen D Jackson. Comparative evaluation of desktop and cave environments for learning hypercube rotations. 2005.

[46] David Robinson. Animation: The first chapter 1833–1893'. *Sight & Sound, Autumn*, pages 251–254, 1990.

[47] Amit K Sanyal. Geometrical transformations in higher dimensional euclidean spaces. Master's thesis, Texas A&M University, 2001.

[48] Barret Schloerke, Hadley Wickham, Dianne Cook, and Heike Hofmann. Escape from boxland. *R JOURNAL*, 8(2):243–257, 2016.

[49] William Sleator. *The boy who reversed himself*. Puffin, 1998.

[50] Side Effects Software. Houdini 14.0. www.sidefx.com/docs/houdini14.0/, 2015. [Online; accessed 20-May-2015].

[51] Kenneth V Steiner. Hidden-volume elimination in four dimensions. Master's thesis, Brigham Young University, 1986.

[52] James N Wadley. Representation and display of hyperdimensional objects using sphyxel trees. Master's thesis, Brigham Young University, 1992.

[53] Weiming Wang, Xiaoqi Yan, Chi-Wing Fu, Andrew J Hanson, and Pheng-Ann Heng. Interactive exploration of 4d geometry with volumetric halos. In *The 21th Pacific Conference on Computer Graphics and Applications-Short Papers*, pages 1–6. Citeseer, 2013.

[54] Robert Webb. Stella4d manual. `http://www.software3d.com/StellaManual.php?prod=stella4D`, 2014. [Online; accessed 14-April-2015].

[55] Daniel Wilding. Four dimensional data navigation. Master's thesis, Brigham Young University, 2007.

[56] WildStar2002. Hypersphere. `https://www.youtube.com/watch?v=BqfwPQvb7KA`, 2008. [Online; accessed 2-March-2016].

[57] Wolfram. Wolfram language & system documentation center. `http://reference.wolfram.com/language/`, 2015. [Online; accessed 16-July-2015].

Appendix A

4D Transformation Matrices

A.1 Translation Matrix

The 4D translation matrix requires four parameters to construct: translation in T_x, T_y, T_z, and T_w. The matrix is constructed as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & T_x \\ 0 & 1 & 0 & 0 & T_y \\ 0 & 0 & 1 & 0 & T_z \\ 0 & 0 & 0 & 1 & T_w \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure A.1: The 4D Translation matrix

A.2 Scaling Matrix

The 4D scaling matrix requires four parameters to construct: scaling in S_x, S_y, S_z, and S_w. The matrix is constructed as follows:

$$\begin{bmatrix} S_x & 0 & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 & 0 \\ 0 & 0 & S_z & 0 & 0 \\ 0 & 0 & 0 & S_w & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure A.2: The 4D Scaling matrix

A.3 Rotation Matrices

There are six rotation matrices, one for each plane of rotation. Each requires one parameter, angle of rotation θ. The matrices are listed below. As mentioned in section 1.6.2, the sine and cosine terms are positioned based on the two principal axes that comprise a given plane of rotation.

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos(\theta) & 0 & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure A.3: Rotation matrices for the XY (top-left), XZ (top-middle), YZ (top-right), XW (bottom-left), YW (bottom-middle), and ZW (bottom-right) planes.

A.4 Shearing Matrix

The uniform shearing matrix requires three parameters to construct: the two shearing axes, and the shearing factor, λ. If the first four columns and rows of the matrix represent the four principal axes, then the shearing matrix is constructed by creating an identity matrix and replacing the entry at the column and row corresponding to the first and second shearing axes with the shearing factor. There are 16 possible matrices, but only one is listed here for the sake of brevity.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \lambda & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure A.4: A 4D Shearing matrix. The shearing factor, λ, is located in the third column and fourth row, which correspond to the Z and W axes, respectively, and represents shearing the W coordinate based on the value of the Z coordinate.

A.5 Perspective Projection Matrix

The 4D perspective projection matrix requires one parameter: f, or the distance between the viewpoint and the image plane. The matrix is constructed as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/f & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure A.5: A 4D Perspective Projection matrix

A.6 Parallel Projection Matrix

The 4D parallel projection matrix requires no parameters. It is constructed as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure A.6: A 4D Parallel Projection matrix

Appendix B

Fourveo Geometry File

Section 4.4.3 describes a novel 4D tetrahedral mesh file format and includes a code listing describing the rules of the format. Listed below is an actual file in the Fourveo Geometry format. This file contains a hypercube. The hypercube is divided into eight cubic cells, each of which is composed of five tetrahedra. Mapping tetrahedra to cells is optional, but it enables a post-processing step in the slicing algorithm which simplifies the geometry of the cross-section mesh.

```
# A sample Fourveo Geometry file
# This file contains a Hypercube centered at (0,0,0,0)
# with a side length of 2

# The 16 vertices in the hypercube
v -1 -1 -1 -1
v 1 -1 -1 -1
v 1 -1 1 -1
v -1 -1 1 -1

v -1 1 -1 -1
v 1 1 -1 -1
v 1 1 1 -1
v -1 1 1 -1

v -1 -1 -1 1
v 1 -1 -1 1
v 1 -1 1 1
v -1 -1 1 1

v -1 1 -1 1
v 1 1 -1 1
```

```
v 1 1 1 1
v −1 1 1 1

# No texture coordinates
# No tetrahedron normals

# The 40 tetrahedra in the hypercube
t 1 6 5 8
t 3 7 6 8
t 1 4 3 8
t 1 3 6 8
t 1 3 2 6

t 1 5 6 13
t 10 6 14 13
t 1 10 9 13
t 1 6 10 13
t 1 2 10 6

t 2 6 7 14
t 11 7 15 14
t 2 11 10 14
t 2 7 11 14
t 2 3 11 7

t 3 7 8 15
t 12 8 16 15
t 3 12 11 15
t 3 8 12 15
t 3 4 12 8

t 4 8 5 16
t 9 5 13 16
t 4 9 12 16
t 4 5 9 16
t 4 1 9 5
```

```
t 1 9 10 12
t 3 10 11 12
t 1 3 4 12
t 1 10 3 12
t 1 2 3 10

t 5 14 13 16
t 7 15 14 16
t 5 8 7 16
t 5 7 14 16
t 5 7 6 14

t 9 13 14 16
t 11 14 15 16
t 9 11 12 16
t 9 14 11 16
t 9 10 11 14


# A list mapping tetrahedra to their cubic cells (optional).
# Each entry corresponds to a tetrahedron in the mesh.
# The number corresponds to the index of a hyperplanar cell
# composed of multiple tetrahedra.
c 1
c 1
c 1
c 1
c 1
c 2
c 2
c 2
c 2
c 2
c 3
c 3
```

c	3
c	3
c	3
c	4
c	4
c	4
c	4
c	4
c	5
c	5
c	5
c	5
c	5
c	6
c	6
c	6
c	6
c	6
c	7
c	7
c	7
c	7
c	7
c	8
c	8
c	8
c	8
c	8