Developing a Transformation Pipeline Visualiser using XNA and Windows Forms

Submitted in fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

By Alastair Nottingham. Supervised by Prof. Peter Wentworth.

Grahamstown, South Africa
November 2008

Abstract

The transformation pipeline is an essential component of any 3D graphics system, responsible for the positioning and orienting of 3D models and vertices within a scene, and presenting that scene, through the view of a camera, to the screen. The transformation pipeline depends on matrix algebra in order perform its calculations, a topic poorly understood by a number of students with weak mathematical backgrounds. This document details the creation of a Transformation Pipeline Visualiser application, intended to provide open-ended graphical exploration of the XNA transformation pipeline, developed as an interactive teaching tool for use within an undergraduate game development course.

Contents

1 Introduction **4**

1.1 Background 4

1.2 Problem Statement 5

1.3 The Transformation Pipeline Visualiser 5

1.4 Document Overview 7

2 Analysis **8**

2.1 Overview 8

2.2 Teaching Considerations 8

2.2.1 Computer-Assisted Instruction 9

2.2.2 Learning Theory 10

2.3 Graphics 12

2.3.1 The Transformation Pipeline: An Overview 12

2.3.2 Matrix Operations 13

2.3.3 3D Transform Geometry 14

2.3.4 World, View and Projection Matrices 21

2.3.5 Color and Lighting 23

2.4 Skyboxes 27

2.5 Summary 27

3 Integrating XNA and Windows Forms **29**

3.1 Overview 29

3.2 Architecture Selection 29

3.3 Application Requirements 30

3.3.1 Performance 30

3.3.2 Simplicity 31

3.3.3 Content Sharing 31

3.4 Acceptable Methods 32

3.4.1 The Thorough Method: Building a Custom Game Object 32

3.4.2 The Simple Method: Presenting XNA Graphics to an Existing Control . 33

3.5 Performance Results 35

3.6 Selection 36

3.7 Summary 37

4 Design and Implementation **38**

4.1 Overview 38

4.2 Data Structures 38

4.2.1 TransformObject 39

4.2.2 Transformation Nodes 40

4.2.3 Transformation List 43

4.2.4 ModelRegistry 44

4.3 Global GUI Controls 45

4.3.1 Filtered TextBox 46

4.3.2 Matrix Control 46
4.4 Transformation Visualisation 47
4.4.1 Overview 47
4.4.2 Functionality 47
4.4.3 Associated GUI Controls 57
4.5 Matrix Operation Visualisation 59
4.5.1 Overview 59
4.5.2 Features 60
4.5.3 Associated GUI Controls 61
4.6 View And Projection Manipulation 64
4.6.1 Overview 64
4.6.2 Features 64
4.6.3 Associated GUI Controls 65
4.7 Controlling Lighting 66
4.7.1 Overview 66
4.7.2 Features 67
4.7.3 Associated GUI Controls 68
4.8 Synthesis 69
4.8.1 Overview 69
4.8.2 Features 69
4.8.3 Associated Controls 71
4.9 Summary 72

5 A Sample Lesson **74**

5.1 Overview 74

5.2 Practical 74

5.3 Solving the Practical 76

5.3.1 Section 1: Space Solution 76

5.3.2 Section 2: DNA Solution 77

5.4 Summary 77

6 Conclusion **79**

6.1 Overview 79

6.2 Available Functionality 79

6.3 Future Work 81

6.3.1 Improvements 81

6.3.2 Extensions 82

6.4 Summary 84

List of Figures

1.1 The Main Form 5

1.2 Stepping into a calculation 6

2.1 Matrix Addition 13

2.2 Matrix Subtraction 14

2.3 Matrix Multiplication 15

2.4 Translation Matrix 15

2.5 Scale Matrix 16

2.6 Rotation Matrices 17

2.7 Gimbal Lock 18

2.8 Orbits 19

2.9 Objects encircling the y axis 20

2.10 View Transformation 23

2.11 A Viewing Frustum[12] 24

2.12 Orthogonal and Perspective Projection 24

2.13 Emissive Light [6] 25

2.14 Ambient Light [3] 26

2.15 Diffuse Light [5] 26
2.16 Specular Light [8] 26
2.17 Skyboxes 28

4.1 Class Diagram providing an overview of the relations between objects 39
4.2 Scalar multiplication using a scaling matrix 41
4.3 A MatrixControl component 47
4.4 The Transform Manager 48
4.5 Object Controls 48
4.6 The Element Context Menu 51
4.7 The Element Builder 52
4.8 The Waypoint Context Menu 53
4.9 Rename Element Form 54
4.10 Transform Node Views 56
4.11 Transform List 57
4.12 The Matrix Calculation Viewer Form 60
4.13 Matrix Addition and Subtraction Layout 61
4.14 Matrix Multiplication Layout 62
4.15 Matrix Operation Viewer 62
4.16 DrawnMatrix Control 63
4.17 MatrixBlock and MatrixBlockList 64
4.18 Camera Controls 64
4.19 Field of View and Camera Information 65

4.20 The Effects of Custom Object Lighting 67

4.21 Directional Lighting Controls 68

4.22 Directional Lighting Effects 68

4.23 ColourSelector Control 68

4.24 Edit Skybox Form 70

4.25 A Small Skybox 71

5.1 Example Practical: Solution Images 78

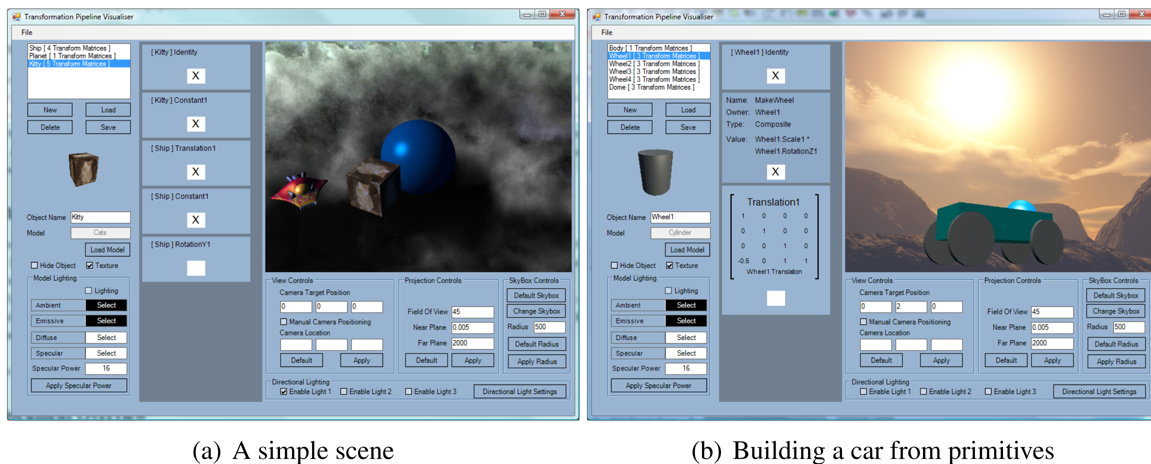
Chapter 1

Introduction

1.1 Background

Development of 3D interactive content, once reserved for experienced, specialised developers, is fast becoming an accessible and rewarding hobby for many individuals, due in part to the rapid advancement in technology, and the introduction of game development frameworks such as Microsoft XNA Game Studio. Due to the growing interest in interactive 3D development, particularly regarding 3D games, the Computer Science department of Rhodes University intends to introduce an elective Game Programming course as part of the 3rd year Computer Science curriculum. This course will instruct students in the basics of game development, and will serve as an introduction to basic 3D graphics programming.

As the course is intended as an introduction, it will focus primarily on aspects fundamental to all 3D development, rather than specific, complex effects. One such fundamental topic is that of the transformation pipeline, which is responsible for scaling, positioning and orienting objects relative to one another in a scene, positioning the camera, and projecting the camera's view to the screen, which it achieves utilising matrix algebra. Unfortunately, many third year students are unfamiliar or uncomfortable with matrix algebra, and thus find the transformation pipeline both confusing and complicated. Hence, it is desirable to develop a means to help such students properly visualise how the transformation pipeline works, as this will improve their comprehension of both transformations and matrix algebra in general, both fundamental to computer graphics.



(a) A simple scene (b) Building a car from primitives

Figure 1.1: The Main Form

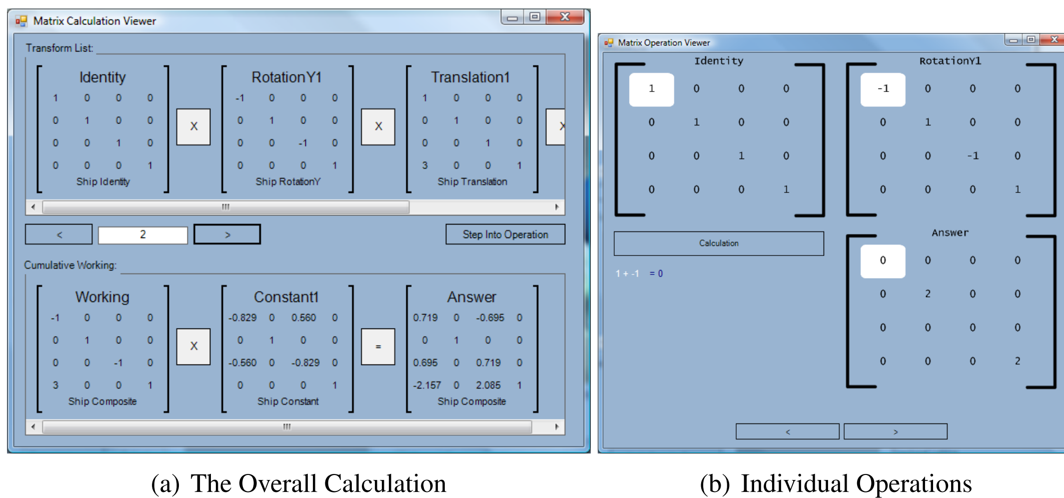
1.2 Problem Statement

The processes involved in developing interactive 3D content for the XNA framework, and graphical API's in general, are heavily dependent on matrix algebra and the transformation pipeline, which are responsible for positioning objects within a scene and projecting them to the screen. Despite being relatively simple in concept, the mathematics involved presents a barrier to learning[18] for many students. As these concepts are fundamental to a proper understanding of 3D development, particularly within the XNA framework, we desire an application to provide real time visualisation of the transformation pipeline in order to provide a mechanism for concept exploration and to help solidify understanding of basic techniques.

1.3 The Transformation Pipeline Visualiser

In order to address these concerns, a Transformation Pipeline Visualiser application has been developed, which leverages XNA and Windows Forms, to help students grasp these concepts. In this section, we supply a simple overview to provide context for the topics discussed in the following chapters.

The Transformation Pipeline Visualiser is an application which allows for graphical exploration of the transformation pipeline used to position and orientate objects in an XNA game. The application consists of a main form, housing transformation, lighting, camera and environment controls, which the user can use to build complex, animated scenes.



(a) The Overall Calculation (b) Individual Operations

Figure 1.2: Stepping into a calculation

The Transformation Pipeline Visualiser facilitates stepping into the calculation of an objects World matrix, allowing the user to view each step in the calculation of each matrix, and the calculation of the list overall.

The application supports both static and dynamic transformations, as well as a variety of matrix operations, and parenthesis, to allow the user to explore the creation of 3D scenes, comprising dynamic object models which demonstrate interdependent behaviors. The application also allows the user to manipulate both object and scene lighting, as well as the scene view and projection parameters. To improve the visual diversity and quality of scenes, the application allows the user to specify custom environments through use of a Skybox creation component. Environments, objects and entire scenes may be saved for later use, or for use on another machine.

Essentially, the Transformation Visualiser Application is an attempt to provide a simple means for students to familiarise themselves with the transformation pipeline within a 3D game. The application is designed as a flexible, exploratory medium, but may be used as a platform for more structured practical tasks. The application presents an open view of the state of each component, and provides the user with real time feedback regarding how changes affect the objects in a scene.

In this document, we discuss the development of this application.

1.4 Document Overview

The document is divided into sections. In the following chapter we consider theory relevant to the development of the application. This includes both theories of learning, used to develop the teaching concept for the system, and graphics theory, which introduces the concepts the application will be used to explore. Chapter 3 discusses the methods considered for displaying XNA content within a Forms environment, providing justification for the chosen method. Chapter 4 details the features of the system in detail, elaborating on the implementation of the functionality where relevant. Chapter 5 provides a sample practical which illustrates how the application may be used to teach graphics concepts. Chapter 6 concludes by summarising the application and its features, and then considers a number of possible improvements and extensions, which may be pursued in order to improve the application.

Chapter 2

Analysis

2.1 Overview

In this chapter we consider relevant theory, in the realms of both teaching and graphics, as a foundation for project development. Teaching theory considers the paradigms of both Operant Conditioning and Discover and Resolve Tutoring (DART) in order to provide a theoretical basis for the teaching methodology being utilised by the Transformation Pipeline Visualiser. In particular, this section aims to highlight the benefits of a DART inspired implementation which advocates personal discovery over conditioned understanding.

Graphics considers relevant graphical theory, in order to provide a thorough overview of what the application intends to achieve. Since the applications scope is limited to the transformation pipeline and scene lighting, discussion will be focussed around these topics.

2.2 Teaching Considerations

The following section considers aspects fundamental to the project, with the intention of providing a theoretical foundation for design decisions regarding both the medium and method of knowledge dissemination. Discussion first considers the field of Computer-Assisted Instruction, elaborating on both early history, and recent research relevant to this project. This is followed by an overview of relevant psychological theory (namely Behaviorism and Discover And Resolve Tutoring) in order to qualify the viability of the intended teaching methodology.

2.2.1 Computer-Assisted Instruction

Background

Computer-Assisted Instruction, or alternatively Computer-Based Training, is a field focussed on utilising current computing technologies to improve student understanding of particular ideas and paradigms whilst reducing inherent strain on educators, particularly in large teaching classes. The field of Computer-Assisted Instruction was arguably established in the late 1920's and early 1930's by Sydney Pressey, an educational psychology professor who developed and deployed a machine to mechanically provide practice multiple choice questions to students in his introductory psychology course [25]. The field has developed for nearly a century, embracing such technologies as the Internet, computer animation, graphical systems and embedded systems in order to improve the effectiveness of digital education[14, 36, 2, 25]. Currently, research persists on the educational potential of interactive computer systems, with eLearning, web based educational tools, and educational collaboration software receiving considerable attention[35].

Implications

This section discusses similar Computer-Assisted Instruction based projects, as well as their implications for this project. Many similar projects have been attempted, and have generally met with success[2, 33, 14, 23], with a few exceptions [31]. In this regard, failures tended to occur when the topic matter was less structured than those found in computer science courses, for instance in Psychology[31], where graphical interaction with computerised models is less applicable. Furthermore, failures tended to coincide with studies which focused on using the technology as a primary educator, rather than as a supplement for formal lectures [31, 14, 36, 2]. As our application is envisioned as a supplement to formal lectures, and illustrates a process which lends itself fundamentally to visual representation, chances of success are significantly increased.

With regard to applications which attempt to instruct computer science concepts, and which are reliant primarily on visual interactive components, we find that despite success, the graphical technology utilised was generally of a very simplistic nature. Notably, in A. Naiman's interactive modules for teaching computer graphics [26], the visual component consisted primarily of 2D line drawings, resulting in an abstract visualisation removed somewhat from an actual implementation. This simplification is largely a result of the limitations of technology at that time,

and the complexity inherent in developing a visually advanced graphical tutoring application from scratch. With the arrival of XNA and its respective libraries, graphical development has been greatly simplified [24], thus allowing rich 3D environments and their respective behaviors to be developed in relatively short time frames. As our intention is to teach games programming in the XNA environment, this provides the added benefit of removing all layers of abstraction, as results will be identical in any XNA game. As such, lessons may be far more engaging and informative than was previously possible.

2.2.2 Learning Theory

In this section, we shall consider theory regarding the human learning process. In this regard, we shall illustrate why, with respect to this project, a DART approach promises a more engaging and flexible learning environment than is possible using a behaviorist approach.

Behaviorism

Computer-Assisted Instruction has its roots in behaviorist theories, and has often been linked to B.F. Skinner's *Operant Conditioning* [29], an alternative to Pavlov*'s Classical Conditioning* concerning reflex response association[27]. While classical conditioning involves establishing a causal link between a neutral stimulus and a conditioned response, treating the learner as a passive observer within the environment, Operant Conditioning sees the learner as an active participant. In essence, it contends that the probability that the learner will engage in a behavior is increased by an *operant reinforcer*, or positive consequence, but decreased by a *punisher*, or negative consequence[27].

Thus, behavior can be conditioned by providing the right stimulus-response patterns[27, 30]. This can be achieved through *Behavioral Shaping*, which attempts to reward successive approximations of a desired behavior, thus improving behavior performance in stages [22, 27, 30]. As positive reinforcement continues, *generalisation* over similar stimuli should occur, producing *secondary reinforcers* which, unlike innately pleasurable *primary reinforcers*, are learned through association with other primary and secondary reinforcers [27, 22, 30].

While behaviorist theories provide the foundation for modern Computer-Assisted Instruction, we wish to elaborate on another more recent, qualitative approach which places greater emphasis on exploration than explanation.

DART: Discover And Resolve Tutoring

DART is a relatively new theoretical approach to improving success in student tutoring in scientific subjects [18]. As DART is intended specifically for one-to-one tutoring sessions between teacher and student, it is not readily applicable to computer based learning, but instead provides significant insight that may improve the user experience. Essentially, DART stipulates that student learning may be *blocked* by perceptual error or misunderstanding, which needs to be identified in order to remove the *block*, thus allowing them to grasp the concept properly. In order to maximise success, the student is encouraged to demonstrate their knowledge of an area, without lecture or demonstration from the tutor, so that their understanding of relevant concepts can be ascertained. Should a student get stuck, the tutor should provide simple insights to aid the student in completing the task, or alternatively, set a simpler version of the task to be attempted first. By taking advantage of watching the students attempt to solve the problem, the tutor can identify where the source of the *block* is, and help the student correct the problem[18].

With regard to the project at hand, while not generally applicable, DART provides several relevant insights. Firstly, the paradigm favors exploration over instruction, which in turn allows the student to solve the problem, rather than simply supplying the student with a solution to be learned. Secondly, DART recognises that misunderstandings occur as a result of conceptual discrepancies beyond the scope of the problems presented, and so focusing solely on the problem area without consideration for the root cause serves to only increase confusion. Thirdly, DART values simplicity and incremental approaches to teaching, whereby the student develops their understanding from less complicated and more constrained examples, and applies them in more complicated instances, reducing confusion and anxiety[18].

Teaching Methodology

Classical behaviorism advocates a constant question-answer-feedback loop, intended to allow students to assess and adjust their own knowledge based on the results of prior actions[27]. Whilst effective, it relies on a structured progression, which assumes student understanding at each step, and generally focuses on issues within the locus of the problem at hand without consideration for related areas which may affect accurate perception. In contrast, DART operates in far less structured circumstances, and utilises student exploration of the problem to a far greater degree[18]. Whilst DART's application is limited in the context of Computer-Assisted Instruction, due primarily to the severe limitations of machine intelligence, elements of exploration can be implemented within the application to provide user flexibility in addressing their problems.

In this regard, the application has been designed such that all material is accessible without prerequisite, and relevant additional information is available where necessary. This will entail deviating from the question-answer paradigm, since this assumes prior knowledge of the student's intention, which is necessarily absent in an open-ended system.

Since both paradigms advocate an incremental approach to learning[18, 27], we shall facilitate exploration of concepts in isolation and in cooperation, and allow the student to control the complexity to as great a degree as possible.

2.3 Graphics

This section aims to provide an overview of the mathematics utilised in the fundamental concepts demonstrated. The purpose of this section is to both elaborate on the relevant mathematical and graphical theory, and to provide an overview of the procedures the application is intended to demonstrate once deployed.

2.3.1 The Transformation Pipeline: An Overview

The Transformation Pipeline refers to the sequence of coordinate transformations applied to the vertices of a mesh in order to present them, in their correct positions, in a viewport[9, 32]. The transformation pipeline consists of four distinct stages[9], of which we are interested in demonstrating the first three, namely the World, View and Projection transformations. The World transformation converts model coordinates (where the model coordinates origin is typically the center of the model) to world coordinates (where the model coordinates origin is the center of the scene). Primarily, this allows for relative model positioning.

The View transformation converts object coordinates from world space into *camera space* [9], where objects are positioned relative to the camera. Projection transformations move coordinates from *camera space* into *clip space*, where meshes are clipped to a viewing volume, or frustum, and allows for perspective distortion[9]. We shall not be considering the transformation from clip space into pixel coordinates, as it is not typically relevant. In the following sections, we discuss the first three components of the transformation pipeline, and simple lighting techniques.

Let $A = [a_{ij}]$ and $B = [b_{ij}]$ be $m \times n$ matrices. Then the sum $A+B$ is an $m \times n$ matrix $C = [c_{ij}]$ where

$$c_{i,j} = a_{i,j} + b_{i,j}$$

Specifically,

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} + \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{bmatrix}$$

$$= \begin{bmatrix} a_{1,1}+b_{1,1} & a_{1,2}+b_{1,2} & a_{1,3}+b_{1,3} & a_{1,4}+b_{1,4} \\ a_{2,1}+b_{2,1} & a_{2,2}+b_{2,2} & a_{2,3}+b_{2,3} & a_{2,4}+b_{2,4} \\ a_{3,1}+b_{3,1} & a_{3,2}+b_{3,2} & a_{3,3}+b_{3,3} & a_{3,4}+b_{3,4} \\ a_{4,1}+b_{4,1} & a_{4,2}+b_{4,2} & a_{4,3}+b_{4,3} & a_{4,4}+b_{4,4} \end{bmatrix}$$

Figure 2.1: Matrix Addition

2.3.2 Matrix Operations

The transformation pipeline uses transformation matrices to scale, rotate, translate and project to the screen. Fundamental to this is the matrix multiplication operation, which allows such transformations to take place. In this section we shall discuss matrix multiplication, as well as matrix addition and subtraction, as it applies to the transformation pipeline. As matrix addition and subtraction are both very similar and conceptually simple, they shall be discussed first.

Matrix Addition

The sum of two matrices is equivalent to the sum of corresponding elements between both matrices2.1. While not typically used within the transformation pipeline, if appropriately utilised it can allow for sophisticated transformation effects.

Matrix Subtraction

Matrix subtraction is similar to matrix addition, in that the difference between two matrices is essentially the difference between corresponding elements between the two matrices2.2. Like matrix addition, matrix subtraction is rarely used in the transformation pipeline, but can be exploited to achieve sophisticated effects if properly understood.

Let $A = [a_{ij}]$ and $B = [b_{ij}]$ be $m \times n$ matrices. Then the difference $A - B$ is an $m \times n$ matrix $C = [c_{ij}]$ where

$$c_{i,j} = a_{i,j} - b_{i,j}$$

Specifically,

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} - \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{bmatrix}$$

$$= \begin{bmatrix} a_{1,1} - b_{1,1} & a_{1,2} - b_{1,2} & a_{1,3} - b_{1,3} & a_{1,4} - b_{1,4} \\ a_{2,1} - b_{2,1} & a_{2,2} - b_{2,2} & a_{2,3} - b_{2,3} & a_{2,4} - b_{2,4} \\ a_{3,1} - b_{3,1} & a_{3,2} - b_{3,2} & a_{3,3} - b_{3,3} & a_{3,4} - b_{3,4} \\ a_{4,1} - b_{4,1} & a_{4,2} - b_{4,2} & a_{4,3} - b_{4,3} & a_{4,4} - b_{4,4} \end{bmatrix}$$

Figure 2.2: Matrix Subtraction

Matrix Multiplication

Matrix multiplication is more complicated than either addition or subtraction, and is used frequently within transformation geometry. Matrix multiplication is non-commutative, and as such, correct ordering of matrices is essential. Matrix multiplication forms the foundation of almost all 3D transformations, and as such should be well understood2.3.

2.3.3 3D Transform Geometry

3D transformation geometry utilises matrix mathematics (a branch of linear algebra) in order to translate, scale, skew and rotate 3D objects within a three dimensional world[13, 37]. As a result, understanding how this process works is essential to understanding 3D development. As this project focuses on XNA development, and the XNA framework uses row major matrices which are multiplied from left to right[13], we shall be consider transformation order to be from left to right. To convert to right to left notation, or column major notation, as in OpenGL, one only needs to take the transpose of the transformation, and multiply the matrices in reverse order. This is merely a difference in notation, with row major matrices transforming Cartesian coordinates in the form of row vectors, and column major matrices transforming Cartesian coordinates in column vector format[38].

Let $A = [a_{ij}]$ be an $m \times n$ matrix, and let $B = [b_{kj}]$ be an $n \times s$ matrix. The matrix product AB is the $m \times s$ matrix $C = [c_{ij}]$, where $c_{i,j}$ is the dot product of the i^{th}row vector of A and the jth column vector of B.

Let $v = [v_1, v_2, \ldots, v_n]$ and $w = [w_1, w_2, \ldots, w_n]$ be n vectors. Then $v \cdot w = v_1 w_1 + v_2 w_2 + \ldots + v_n w_n = \sum_{j=1}^{n} v_j w_j$, where $v \cdot w$ is the dot product of the vectors v and w.

Specifically,

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{k=1}^{4} a_{1,k}b_{k,1} & \sum_{k=1}^{4} a_{1,k}b_{k,2} & \sum_{k=1}^{4} a_{1,k}b_{k,3} & \sum_{k=1}^{4} a_{1,k}b_{k,4} \\ \sum_{k=1}^{4} a_{2,k}b_{k,1} & \sum_{k=1}^{4} a_{2,k}b_{k,2} & \sum_{k=1}^{4} a_{2,k}b_{k,3} & \sum_{k=1}^{4} a_{2,k}b_{k,4} \\ \sum_{k=1}^{4} a_{3,k}b_{k,1} & \sum_{k=1}^{4} a_{3,k}b_{k,2} & \sum_{k=1}^{4} a_{3,k}b_{k,3} & \sum_{k=1}^{4} a_{3,k}b_{k,4} \\ \sum_{k=1}^{4} a_{4,k}b_{k,1} & \sum_{k=1}^{4} a_{4,k}b_{k,2} & \sum_{k=1}^{4} a_{4,k}b_{k,3} & \sum_{k=1}^{4} a_{4,k}b_{k,4} \end{bmatrix}$$

Figure 2.3: Matrix Multiplication

We shall be considering translation, rotation and scale matrices within our project, as skewing matrices are rarely used in game development[37]. A brief description of these matrices follows.

Translation Matrices

Translation matrices (Figure 2.4) position 3D objects by adjusting object placement by the x, y, and z values supplied, relative to the object's location. Translation matrices are commonly used for movement in 3D environments[13, 37].

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \Delta x & \Delta y & \Delta z & 1 \end{bmatrix}$$ where Δa is the relative change in position on axis a.

Figure 2.4: Translation Matrix

Scale Matrices

Scale matrices (Figure 2.5) adjust the size of a 3D object by percentage x, y and z values supplied. Scale matrices are used primarily to correctly size objects for a scene, or adjust object

size as a result of some event [13, 37].

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ where S_a is the scale applied to coordinates along axis a.

Figure 2.5: Scale Matrix

Rotation Matrices

Rotation matrices (Figure 2.6) adjust the x, y and z rotations of objects within a 3D scene. For simplicity, we shall only consider the Euler angles with regard to teaching, using separate matrices for each axis. To produce a single rotation matrix for all three planes, we simply use the multiplicative product of these three matrices. Rotation matrices are used to adjust the directional orientation of objects within a 3D scene[13, 37].

$$Rotation_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) & 0 \\ 0 & -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Rotation_y = \begin{bmatrix} cos(\alpha) & 0 & -sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\alpha) & 0 & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Rotation_z = \begin{bmatrix} cos(\alpha) & sin(\alpha) & 0 & 0 \\ -sin(\alpha) & cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

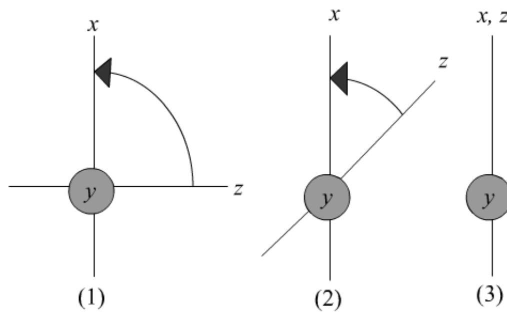
where α is the object angle adjustment in radians.
To produce a single rotation matrix, denote $cos(x) = x_c$ and $sin(x) = x_s$ for some angle x. Then

$$Rotation_{x,y,z} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha_c & \alpha_s & 0 \\ 0 & -\alpha_s & \alpha_c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \theta_c & 0 & -\theta_s & 0 \\ 0 & 1 & 0 & 0 \\ \theta_s & 0 & \theta_c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \phi_c & \phi_s & 0 & 0 \\ -\phi_s & \phi_c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \theta_c\phi_c & \theta_c\phi_s & -\theta_s & 0 \\ \alpha_s\theta_s\phi_c - \alpha_c\phi_s & \alpha_s\theta_s\phi_s + \alpha_c\phi_c & \alpha_s\theta_c & 0 \\ \alpha_c\theta_s\phi_c + \alpha_s\phi_s & \alpha_c\theta_s\phi_s - \alpha_s\phi_c & \alpha_c\theta_c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where α,$\theta$and ϕ are the object angle adjustment in radians for the x, $y$and z planes respectively.

Figure 2.6: Rotation Matrices



Shows how gimbal lock arises when the *y* axis is rotated by 90 degrees, causing the *x* and *z* axes to lock.

Figure 2.7: Gimbal Lock

Quaternions

While the Euler angles are sufficient for demonstration purposes, we use a more sophisticated rotation matrix for controlling the camera of the application. This is desirable as Euler angles are susceptible to *Gimbal Lock,* which occurs when the rotation on one axis overrides that of another axis thus locking them together and removing a degree of freedom[39]. The reason for this stems from the independent axis rotation used in Euler angles. For instance, assuming the Euler angles are applied in the order $x \rightarrow y \rightarrow z$, and the *y* axis is rotated by 90 degrees, the *z* axis will be rotated over the *x* axis, causing *Gimbal Lock* (see figure 2.7). This occurs because, unlike the *z* axis rotation, the *x* axis rotation has already been processed, and so is not affected by the *y* axis rotation.

Quaternions do not suffer from *Gimbal Lock* [39], and thus are used in camera positioning. Quaternions will not be demonstrated within the application, however, and as such will not be used for rotating scene elements within the transformation visualiser. This is to ensure consistency between the application and basic XNA programming techniques. As Quaternions do not feature significantly in the project, they will not be discussed further.

Orbits

Orbits are transformations which translate first, then rotate, in order to orbit an object around the starting point prior to the transformation [13]. Orbits are illustrated in figure 2.8. While orbits are rarely used, they are occasionally useful.

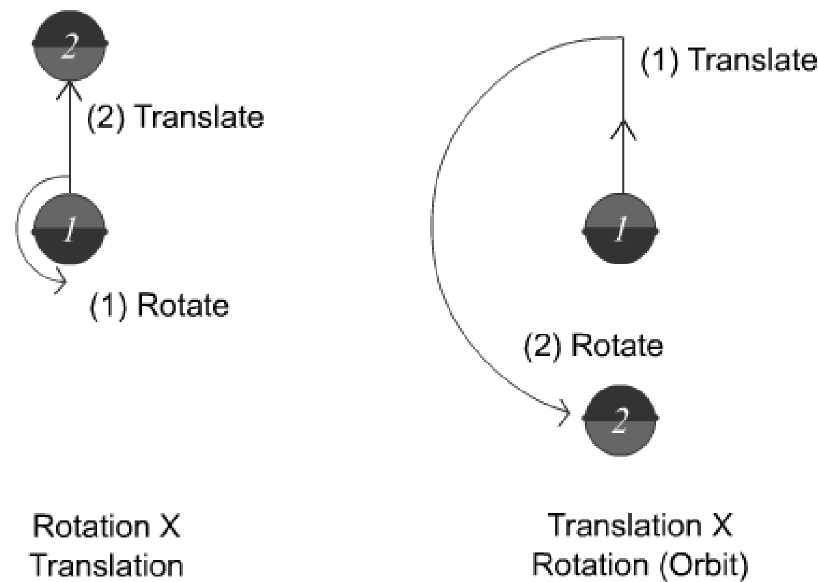


Figure 2.8: Orbits

Transform Application Sequence

Typically, linear translation matrices should be applied using the I.S.R.O.T. sequence [13], which assumes that all transformations start with the object at the origin, and are applied in the order: Scale, Rotations, Orbit, Translation. The sequence of application of transform matrices is important, due to matrix multiplication being non-commutative[13, 37]. The sequence composition and ordering is justified below:

1. While the Identity matrix is usually not necessary, it indicates that when no Scale, Rotation or Translation matrices are applied, the transform should just be the Identity matrix. At this point, the object is at the origin, untransformed.

2. Scaling is calculated first, in order to ensure that the object is correctly sized for the scene. If the object is translated before it is scaled, the translation distance will be scaled as well[13] which, in most situations, will be undesirable.

3. The re-sized object is then revolved until it is facing in the correct direction. As rotations occur about the origin[37, 13], rotations applied after translations result in orbits.

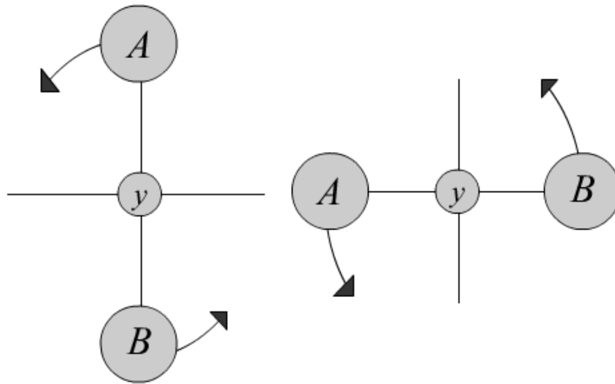


Figure 2.9: Objects encircling the y axis

4. If an orbit is necessary, this is applied next. Once applied, the object orbits a set distance from the origin, scaled and orientated correctly.

5. Finally, the object is translated to the correct position in the scene. If an orbit is being used, the object is translated such that the center of its orbit corresponds to the translation position. By applying translations last, we ensure that the translation is not scaled or rotated in any way.

The I.S.R.O.T sequence is used for the calculation of an object's World matrix, which is discussed later in the chapter.

Combining Transforms From Multiple Objects

In more complex instances, it is often useful to apply or use a transformation from another object in order to achieve complex effects. For instance, two objects multiplied by the same translation matrix will move in the same way. This may be utilised in order to achieve a moving platform effect, among other possibilities, whereby any object which touches the platform is transformed by the current position of the platform, thus creating the illusion of the platform carrying the object. Alternatively, one may apply the same variable rotation transformation to two separate objects at opposite sides of the axis of rotation, in order to make them encircle the axis (see figure 2.9). As can be seen, sharing transformations between objects allows for interesting effects that may be useful in particular situations.

Advanced Techniques

Finally, we consider the use of parenthesis, scalars, addition and subtraction in achieving some complex effects. These allow a finer degree of control over the transformation process, and can be exploited if understood. As an illustration, Listing 1 shows how addition, scalar multiplication and parenthesis can be utilised to translate an object to the center point between n objects. While such techniques are unnecessary in a typical case, such methods are occasionally useful, and understanding how to construct more complex transformations to solve a problem efficiently is an important aspect of games development[13].

2.3.4 World, View and Projection Matrices

The world, view and projection matrices are utilised within the first three stages of the XNA transformation pipeline [13], and are used to ensure that 3D objects are projected correctly on a 2D screen. World matrices converts model and vertex coordinates into world coordinates, view matrices set the camera's direction (thus defining what can be seen), and Projection matrices, or Perspective matrices, define the *frustum,* or cone shaped view of the camera[13].

World Matrix

The World matrix is calculated by applying the transformation geometry discussed in the previous section to a mesh or vertices, typically centered at the origin, in order to translate their local, or model coordinates into world coordinates[13, 9]. A separate World matrix is calculated for each object within the scene, necessary in order to allow individual placement of objects. As previously discussed, transformations applied to the World matrix should typically follow the I.S.R.O.T sequence, although this is not always the case, and depends on the requirements of the current scene.

View Matrix

The View matrix is used for converting from world coordinates into camera coordinates, where the camera is positioned at the origin, looking in the positive z direction[10, 13]. This essentially involves translating all the objects in the scene, such that their relative position to the camera is maintained once the camera has been positioned, which is the function of the View transformation matrix. An overview of this process is shown in 2.10[10].

Listing 1 Using parenthesis, scalar multiplication and addition to find the translation to the center point between n objects

Let A and B be translation matrices. The difference between A and B as a matrix, x**,** is easily found;

$A+x=B \Rightarrow x=B-A$

OR

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_1 & a_2 & a_3 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ x_1 & x_2 & x_3 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ b_1 & b_2 & b_3 & 1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ x_1 & x_2 & x_3 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ b_1-a_1 & b_2-a_2 & b_3-a_3 & 0 \end{bmatrix}$$

Thus,

$$\frac{x}{2} = \frac{B-A}{2} = \frac{1}{2} \times \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ x_1 & x_2 & x_3 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{b_1-a_1}{2} & \frac{b_2-a_2}{2} & \frac{b_3-a_3}{2} & 0 \end{bmatrix}$$

Hence, the transformation to the position between the points specified in A and B is:

$A+\frac{x}{2} = A+\frac{B-A}{2} = \frac{A+B}{2}$

OR

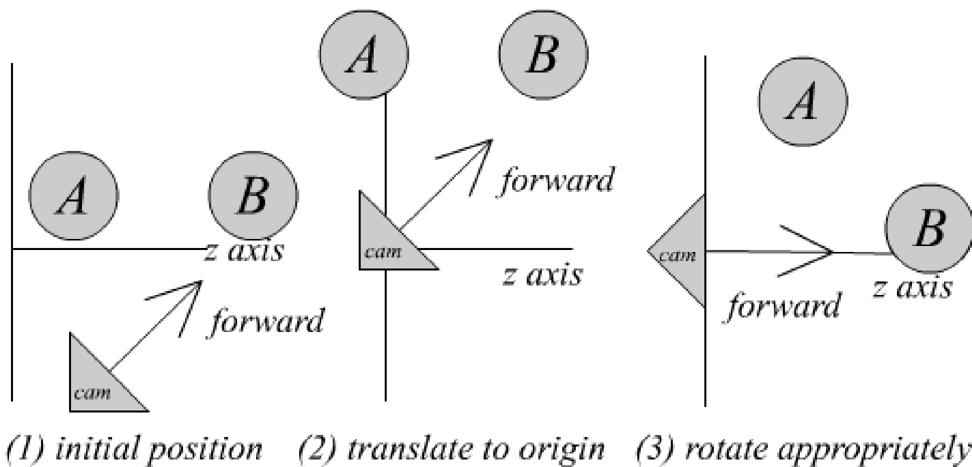
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_1 & a_2 & a_3 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{b_1-a_1}{2} & \frac{b_2-a_2}{2} & \frac{b_3-a_3}{2} & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{a_1+b_1}{2} & \frac{a_2+b_2}{2} & \frac{a_3+b_3}{2} & 1 \end{bmatrix}$$

$$= \frac{1}{2}\left(\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_1 & a_2 & a_3 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ b_1 & b_2 & b_3 & 1 \end{bmatrix} \right)$$

Similarly, the transformation to the center point of n objects is given by:

$$\frac{1}{n}\left(\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_{1,1} & x_{1,2} & x_{1,3} & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_{2,1} & x_{2,2} & x_{2,3} & 1 \end{bmatrix} + \ldots + \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_{n,1} & x_{n,2} & x_{n,3} & 1 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{\sum \left(x_{j,1}\right)_{j=1}^{n}}{n} & \frac{\sum \left(x_{j,2}\right)_{j=1}^{n}}{n} & \frac{\sum \left(x_{j,3}\right)_{j=1}^{n}}{n} & 1 \end{bmatrix}$$



View Matrix $V = T \cdot R_x \cdot R_y \cdot R_z$
where T is the translation to the origin, and R_α is the rotation about axis α.

Figure 2.10: View Transformation

Projection Matrix

The purpose of the projection matrix is to translate camera space into clip space and apply perspective distortion. Essentially, this is done using a *frustum[13, 7],* the shape of which affects how objects are projected from camera space to the screen (see figure 2.11). All objects outside the viewing frustum are clipped, as they are not seen by the camera and as such do not need to be rendered. We shall consider clip space and perspective distortion separately.

Clip space is a coordinate space which indicates which objects in the scene are visible by the camera, and hence need to be rendered. The frustum defines the boundaries of clip space, outside of which objects are not rendered. Perspective distortion refers to the distortion applied to the scene, and is typically separated into two types of projection. The simplest type is that of *Orthogonal* projection, in which the distance from the camera does not affect the size of an object on screen. In contrast, *Perspective* projection, reduces the size of objects as they move further away from the camera. This is achieved by scaling the x and y coordinates by a factor of the z axis, as the z axis represents the distance from the camera once the View transform has been applied (see figure 2.12).

2.3.5 Color and Lighting

Lighting methods in computer graphics allow for the appearance of an object to be manipulated, and is responsible for colour, surface reflectivity, gloss and shadows, among other effects.

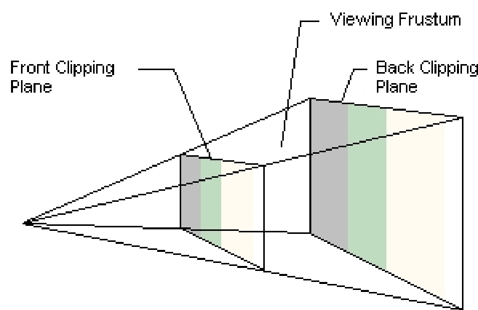


Figure 2.11: A Viewing Frustum[12]

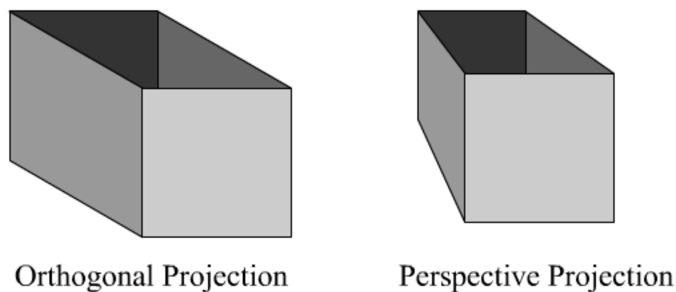


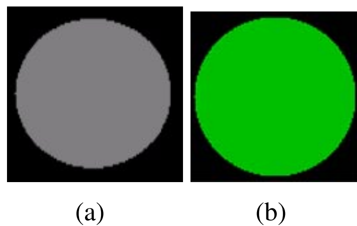
Figure 2.12: Orthogonal and Perspective Projection

Illumination may be local, where each object in the scene is illuminated outside of the context of the scene, or global, which allows for other objects in the scene to reflect off or otherwise affect the colour properties of the object being illuminated. We shall not be considering global illumination models in this visualiser, since we are only concerned with basic techniques. Furthermore, we shall limit our discussion to directional lighting, and ignore point lights and spot lights, as these are not available within XNA's BasicEffect shader.

This section considers directional lighting, useful in positioning light sources for all objects, and some basic local lighting methods, including emissive, ambient, diffuse and specular lighting, which can be used affect the the appearance of an object in several ways.

Directional Lights

Directional lights are lights which equally illuminate all objects within a scene, and is essentially light flowing in a particular direction. Unlike point and spot lights, directional lights have no specific source, and so the light intensity and direction is constant throughout the scene. Using directional lighting, surface illumination can be calculated using the object surface normals and the light direction vector, which are used to determine the degree of illumination on that surface.



(a) (b)

(a) Sphere with grey emissive light only
(b) Sphere with green emissive light only

Figure 2.13: Emissive Light [6]

Standard Scene Lighting

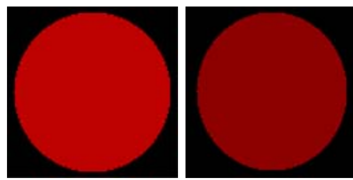
In order to improve the visual quality of the lit 3D models in a scene, the directional lights can be used to provide *key, fill* and *back* lighting[21]. The *key* light is the brightest of the three, and is used as the main illumination source for the scene. Key lights are typically positioned to correspond to the position of the main scene light source, such as the sun. The *fill* light is typically dimmer, and angled at 90 degrees to the key light, in order to soften shadows and improve definition in otherwise unilluminated regions. The *back* light is positioned behind the illuminated objects, in order to improve illumination around the edges. Of the three light sources, only the key light should be used to cast shadows.

Emissive Light

Emissive lighting has only one property, its colour, which it applies evenly to all vertices without shading. As such, using emissive lighting only will cause an object to look two-dimensional. Emissive lighting is typically used in combination with other lighting techniques to achieve more sophisticated effects[6].

Ambient Light

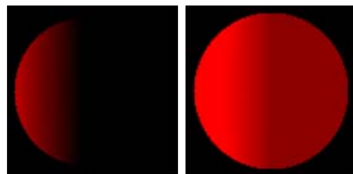
Ambient light refers to the background light within an environment, and is not affected by light intensity or direction[13]. Each material can have its own ambient light colour, which is used in conjunction with the global ambient light colour to calculate the ambient colour to apply to the surface[3].



(a) (b)

(a) Sphere with red ambient light only
(b) Sphere with red ambient light and grey emissive light

Figure 2.14: Ambient Light [3]



(a) (b)

(a) Sphere with red diffuse light only
(b) Sphere with red diffuse, red ambient and grey emissive light components

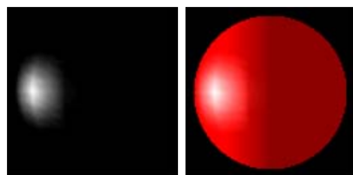
Figure 2.15: Diffuse Light [5]

Diffuse Light

Diffuse light specifies how a light source will illuminate a surface in its path. Diffuse light intensity is inversely proportional to the angle between the light direction vector and the surface normal[13, 5]. This ensures that the most illuminated surfaces are those directly facing the light source.

Specular Light

Specular lighting defines the shininess or gloss of a surface, and is dependent on both the viewer's and light's angle to the surface[13, 8]. Specular lighting utilises two properties, namely



(a) (b)

(a) Sphere with white specular light only
(b) Fully lit sphere

Figure 2.16: Specular Light [8]

the light colour, and the specular power, in order to light an object. The specular power refers to the focus of the specular light, with lower values providing a softer, more blurred effect providing a matte finish, while high values create more focused, intense light resembling a gloss finish[20, 13].

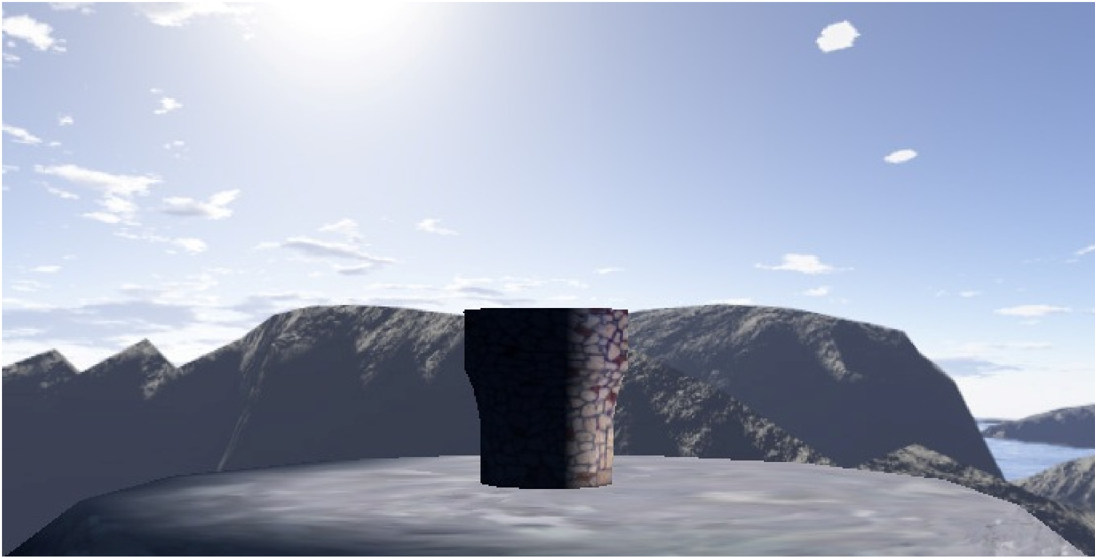
2.4 Skyboxes

Skyboxes provide an efficient means for creating a convincing environment for a scene. Skyboxes use six separate images, each viewing the horizon in a different direction, which if properly created, can be used to generate a seamless image of the horizon by surrounding a scene in a cube configuration[13]. As skyboxes are typically comprised of static prerendered images, they greatly reduce the processing necessary for creating a convincing environment within a 3D scene. Skyboxes will be included as a presentation component, but will not form part of the core functionality of the system. Thus, we shall not cover them in depth.

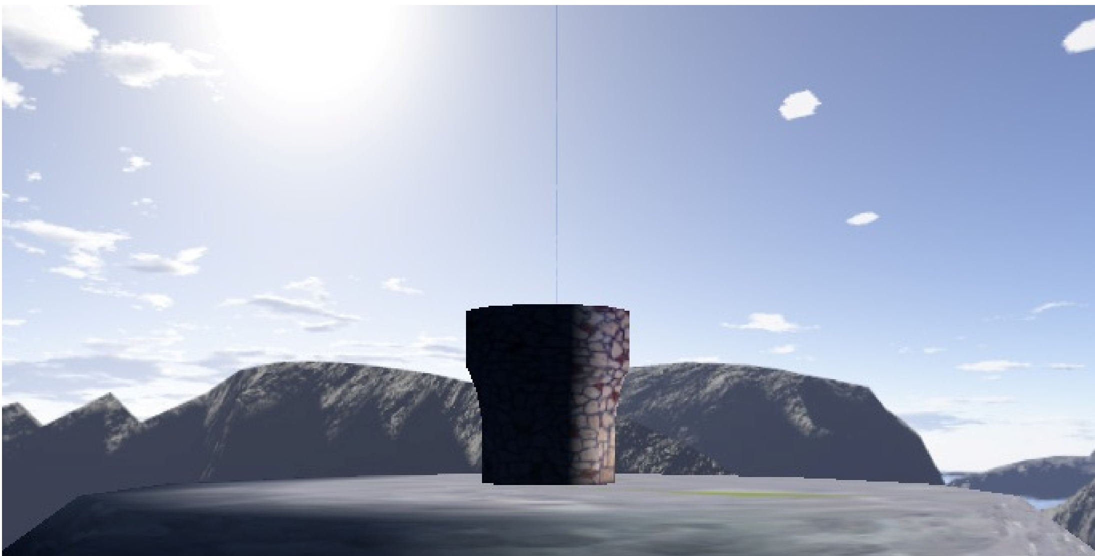
2.5 Summary

In this chapter, we have considered theory relevant to the project, with regard to both teaching methodology and the principles to be taught. The teaching considerations section provided a theoretical foundation for the project medium by elaborating on Computer-Assisted Instruction, and introduced both a classical behaviorist approach to teaching through Operant Conditioning and more recent approach through Discover And Resolve Tutoring (DART). The section concluded that while DART cannot truly be fully realised in software at this point, due to limitations in artificial intelligence, it was both possible and desirable to adapt the primary DART concepts of exploration and discovery into a feasible software based approach.

Regarding the graphics section, theory relevant to the concepts being taught was presented, both as an overview of terminology and outline of necessary functionality within the system. This section provided an summary of both the transformation pipeline and matrix algebra, and presented relevant graphical theory pertaining to transforming objects, presenting objects to screen, and lighting objects. This theory will be applied to the project, in order to provide a means for exploration of these concepts.



(a) A well configured skybox



(b) A Badly Configured Skybox Showing A Seam

See figure 4.25 for a more descriptive skybox image.

Figure 2.17: Skyboxes

Chapter 3

Integrating XNA and Windows Forms

3.1 Overview

This section considers two methods for displaying XNA 2.0 output within a C# 2.0 Windows Forms application, with particular focus on ease of integration and performance with regard to the transformation pipeline visualiser. We first provide some detail as to why such controls are necessary, elaborating on necessary functionality and performance, followed by an overview of both methods, with implementation specifics included where necessary. The section concludes with a discussion of results, which are used to justify the selection of the simpler of the two methods for use in this project.

3.2 Architecture Selection

Considering that the transformation pipeline visualiser application is intended to aid in teaching XNA programming, by developing the visualiser in XNA, consistency can be ensured between results of both the visualisation application and the framework the application intends to introduce. While developing the transformation visualiser entirely within the XNA framework was possible, there were numerous reasons which indicated that a marriage of XNA and .NET based controls would provide for a superior application foundation. An enumeration of these reasons follows.

Firstly, the Transformation Visualiser depends heavily on user input, which in turn requires that a number of input components be available. In this regard, .NET already contains a wide variety

of useful input controls that could easily be leveraged to provide for a number of input scenarios, whereas XNA, being focused on game development, would have necessitated development of a number of these controls from scratch. As such, a simple method which allows for the display of XNA content while provisioning input through .NET components is ideal, as it would play to the strengths of both frameworks. Furthermore, as the application would depend on a rich multi-windowed GUI environment comprising of controls which would not be inherently 3D, it seems unnecessary to process these controls in graphical hardware. Finally, given that the Transformation Visualiser would be designed specifically for a Windows environment and would not be deployed on XBox360 hardware, it would not be necessary for the application to be written entirely in XNA. Given this reasoning, a simple method for displaying XNA content in a Windows Forms application would greatly simplify project development.

3.3 Application Requirements

The methods considered for use in this application need to be compared in order to justify selection of an approach. As such, it is necessary to provide an overview of the primary concerns regarding presentation of XNA content within the application. These concerns include performance, ease of implementation and content sharing.

3.3.1 Performance

Performance refers primarily to graphical performance and speed, particularly in instances where multiple XNA based controls are present concurrently. It is important that the graphical presentation of XNA content be fluid, in order to cultivate a positive user experience. If the presentation quality or fluidity of the XNA content is significantly reduced, this may induce user frustration, and reduce the desire for the user to explore within the environment. Since the success of application is entirely dependent on the user's desire to utilise it as an exploration mechanism, it follows that maintaining acceptably high frame rates and graphic quality is essential.

Continuing along this thread, and considering that multiple components presenting entirely different content may be displayed concurrently within a form or multiple forms, it follows that the method need also be scalable. Scalability is a valid concern in this context, since each XNA game is essentially within an infinite loop[40]. While these loops are sophisticated enough to only iterate at specific intervals, typically ten milliseconds[40], scalability still remains a concern when many such loops are active simultaneously.

3.3.2 Simplicity

Simplicity refers to the minimisation of complexity inherent in displaying XNA content to a screen. While viable solutions need not be trivial, excessive complexity has implications regarding code maintenance and extension. With this in mind, it is desirable to minimise future architecture maintenance and extension problems, particularly in cases where such maintenance and extension is done by someone unversed in the applied presentation solution. Thus, conceptual simplicity is an important factor with regard to method selection.

3.3.3 Content Sharing

Content sharing refers to the ability for multiple XNA controls to share content between one another. Within a typical XNA game, content such as models or textures are converted at design time to XNB format, to allow for easy management by XNA. This content can then be loaded at run time by the game's content manager. The transformation pipeline visualiser, on the other hand, relies on run time compilation, by a custom ContentBuilder, of model files into XNB format, an operation which typically takes a number of seconds to complete. Once compiled, the XNB file can be loaded quickly by the content manager.

When utilising a single game thread to draw all the panels in the application, this presents no problem, as any model, once loaded, is available in all XNA components simultaneously. In instances where multiple game loops are used, it becomes necessary to reload a model when attempting to display it in a different game thread. This introduces two problems, concerning performance and resources respectively. With regard to performance, it is important that the first phase of the the content processing operation, namely the compilation of resources to XNB format, be avoided, since this operation is expensive and slow. This ensures that loading content within different game threads requires only that the XNB file be loaded into the Content Manager, a relatively fast operation. Thus, we can minimise loading times within the application, improving the user experience.

Regarding resources however, multiple game loops require each game to manage its own resources, and to manage the presentation of that resource to the graphics card. Thus, a model may be processed and sent to the graphics card several times by different panels, where as a solution with shared content management would only need to do this once. As graphics resources may be limited, this could adversely affect performance when large models and textures are used.

3.4 Acceptable Methods

This section considers two solutions to presenting XNA content to a Windows Forms control. We shall discuss each method individually, showing that both methods prove to be theoretically viable solutions within the context of the application requirements enumerated above.

3.4.1 The Thorough Method: Building a Custom Game Object

In this section, consideration is given to the more sophisticated method for displaying XNA content within a Windows Forms GUI. The method was developed and made available by Microsoft's XNA Creator's Club website[11], and is considered to be the most stable and flexible of the methods considered. We shall refer to this method as the *thorough method* for simplicity. The method entails creating a custom graphical control, which utilises components from the XNA framework to present output to the form. Primarily, this method involves creating an object similar to the typical Game object, which is capable of sharing a graphics device with other such controls, and can be used within a Windows Forms environment. As this method is made available freely by Microsoft, the source code shall not be presented, but can be found at the XNA Creator's Club Website[11].

While this method provides for a greater degree of flexibility with regard to functionality, as well as improved performance with regard to more complicated content, due primarily to a customisable game loop and reduced drawing overhead, it is comparatively more difficult to implement. In particular, as the method requires a reimplementation of the Game class, modified to operate on Windows Forms applications, many features available in the standard Game class need to be incorporated manually. Despite this, the thorough method is a viable solution, as its implementation is still relatively simple. Furthermore, once the custom control has been implemented, it may be inherited from as a foundation for any additional controls necessary, further reducing complexity.

Regarding scalability, while each control is itself a hybrid game object, these objects are not caught in an infinite loop. Instead, the draw method for the control is hooked to the application's idle process, ensuring that draw operations only occur when the application is idle, improving scalability. Finally, considering content sharing, the utilisation of separate game objects precludes the possibility of inherent content sharing between controls. However, model objects can be compiled to a single directory, utilised by all the game objects in the application, eliminating the need for recompilation of assets by different controls. Despite possible resource

exhaustion implications, the thorough method seems a viable solution within the context of this application.

3.4.2 The Simple Method: Presenting XNA Graphics to an Existing Control

A Monolithic Approach

The method described is based on an approach put forward by Pedro Güida in [17], which has become a popular alternative to the thorough method presented above[41]. This method is inherently monolithic, in that it requires all panels to be rendered within the same draw method of the main game class. Furthermore, each panel requires an accessor method within the form in order to retrieve the panel handle, which is necessary in order to present XNA content to the panels surface[17, 41]. While this method works well for programs using only a few panels displaying similar content, it becomes difficult to manage when many different panels need to display a wide variety of XNA graphics. This is primarily due to the requirement of incorporating multiple games into a single game file. Further difficulty arises when attempting to dynamically add or remove components, reducing this method's attractiveness for sophisticated applications.

In order to improve this method such that it becomes a viable solution for this project, it was modified to allow for a more modular approach, utilising the *DrawableGameComponent* class. As the implementation of this adjusted control is relatively trivial and is not readily available, the following section describes its development.

Modularising the Simple Method

As previously indicated, the implementation of this method is relatively simple. For simplicity, we shall refer to it as the *simple method*. Our intention is to produce a class that can be inherited from, and implemented as if it were XNA game component.

The method renders to the control passed as input, which should typically be a panel object of some form. The method uses the *GraphicsDevice*'s *Present* method[17] to render the game components output to the panel once the call to the base class is complete.

```
public class XNAContent : DrawableGameComponent
{
  protected readonly Control panel;

  //Constructor
  public XNAContent(Game game, Control panel):base(game)
  {
    this.panel = panel; //store reference to panel
    game.Components.Add(this); //add the component to the game
  }

  //on disposal, remove component from game
  ~XNAContent()
  {
    Game.Components.Remove(this);
  }

  //Draw method. Will be called when inheritor completes its draw method.
  public override void Draw(GameTime gameTime)
  {
    //remove the component from the game loop if the panel has been destroyed
    if (panel.IsDisposed) Game.Components.Remove(this);
    //otherwise present graphical output to a control
    else GraphicsDevice.Present(panel.Handle);
    base.Draw(gameTime);
  }
}
```

In order to use this control in a *Forms* environment, a few preparations are required. Firstly, the application's main form needs to be added to the main game class, passed the game object as a parameter, and shown. The games window must also be hidden, and finally, the game must exit when the main form exits[17].

```
  public Game1()
  {
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    //hide the game window
    ((Form)Form.FromHandle(Window.Handle)).Shown += form_Shown;

    //create a the form and pass it the game
    MainForm form = new Form1();
    form.Initialise(this);    //pass the game to the form
```

```
        form.HandleDestroyed += form_HandleDestroyed;
        form.Show(); //show the main form
    }

    //hide the game window if it is shown
    private void form_Shown(object sender, EventArgs e)
    {
        ((Form)sender).Hide();
    }

    //if the main form is closed, exit the game
    private void form_HandleDestroyed(object sender, EventArgs e)
    {
        this.Exit();
    }
```

By overriding the XNAContent class, we produce a GUI object in a similar fashion to that of a typical DrawableGameComponent object. To render to a specific panel, an object need simply be created, referencing the appropriate control. It is worth noting that the aspect ratio of the rendered content is dependent on the GraphicsDevice settings, and not on the dimensions of the control. Thus presented output will be stretched in order to fit into the control being rendered to. In order to correct this, one can simply adjust the viewport attributes from within the derived class's *Draw* method.

```
PresentationParameters tmp = GraphicsDevice.PresentationParameters.Clone();
tmp.BackBufferHeight = 200;
tmp.BackBufferWidth = 500;
GraphicsDevice.Reset(tmp);
```

As can be seen, this method is relatively trivial to implement, and can be used to make many different panels with relatively little extra code. Furthermore, as all components share the same base game, loaded content can easily be shared between controls. Finally, as all components are drawn from within the same game loop, the method inherently provides round-robin scheduling, reducing scheduling overhead and improving performance. Thus, the solution proves viable for use within the transformation visualiser application.

3.5 Performance Results

In this section we discuss the performance results seen when implementing either method. In order to compare both methodologies presented, six separate applications were created. The

Number Of Panels	1 Panel	4 Panels	12 Panels
Thorough Method, First Panel	60 fps	59 fps	58 fps
Thorough Method, Other Panels	N/A	54 fps	54 fps
Modular Simple Method, All Panels	59 fps	59 fps	59 fps

Table 3.1: Average frame rates witnessed with varying numbers of panels over 3 trials

first three were implemented using the thorough method, and included a single panel, a four panel application and a twelve panel application, displaying simple XNA content. The second three were identical to the first, except that they were implemented using the modular simple method. All six applications displayed the number of frames per second rendered after a thirty second interval. Results indicated that performance differences between methods in the single panel applications were roughly equivalent. Performance differences experienced with a larger number of panels was seemingly negligible, and such differences are likely due to the simpler method using the same game loop to render all panels, ensuring that all panels are rendered at the same rate. The thorough method, on the other hand, utilises a separate loop for each game panel, introducing competition for processing resources by each game thread.

3.6 Selection

While both methods prove viable options for presentation of XNA content within a Windows Forms GUI, the transformation visualiser has been developed using the modular simple method. This is due to several reasons. Firstly, within the testing applications, the modular simple method performed marginally better, and provided a more consistent set of frame rate averages. While this performance difference is not particularly significant, it is a performance difference none-the-less.

Secondly, the modular simple method proved to be less complicated to implement than the thorough method, retaining all functionality inherent in a typical game, while requiring significantly less source code. It is thus conceptually simpler to manage and extend. Finally, content sharing is built into the modular simple method as all controls share the same ContentManager. While content sharing can be implemented in a number of ways within the thorough method, this requires extra code, and thus more complexity. Given that the simple method performs better than the thorough method with regard to our applications criteria, the simple method seems to be the most appropriate choice.

3.7 Summary

This chapter has focused on methods for displaying XNA content within a Windows Forms environment. The chapter first considered the type of architecture to implement, arguing that the integration of XNA into a Windows Form GUI provided for a better application foundation than application developed entirely in XNA. Consideration was then given to the specific requirements of the application in question, as all methods have their strengths and weaknesses. Two methods , the thorough method and the modular simple method, were then presented, described, and shown to meet the minimum application requirements discussed. Upon consideration of results, despite negligible frame rate differences, the less sophisticated method was selected as it met the application requirements to larger extent than the thorough method.

Chapter 4

Design and Implementation

4.1 Overview

This section provides a specification for the system, detailing the implemented features, and their relation to the theory discussed in the previous chapter. For clarity, this chapter will be divided into sections. In the first section, the core data structures are introduced, as they form the foundation for the entire system. We then consider two global GUI controls, used throughout the application and thus not applicable to any individual section. The next four sections deal with specific elements of the transformation visualiser which facilitate exploration of the core topics. These include the visualisation of low level matrix algebra and high level graphical results, controlling the view and projection, and lighting controls. Finally, these elements are brought together and considered collectively, with attention given to the interaction between separate system components.

4.2 Data Structures

In this section, we shall provide an overview of the fundamental data structures used by the transformation visualiser to manipulate, present and store scene objects. Each object contained in the scene is stored as a TransformObject, which is responsible for storing name, model, colour and transformation information.

Transformation information is stored in a linked list based structure comprised of a number of *ListNodes*. These *ListNodes* contain a *ITransformNode* variant and an operation to be applied

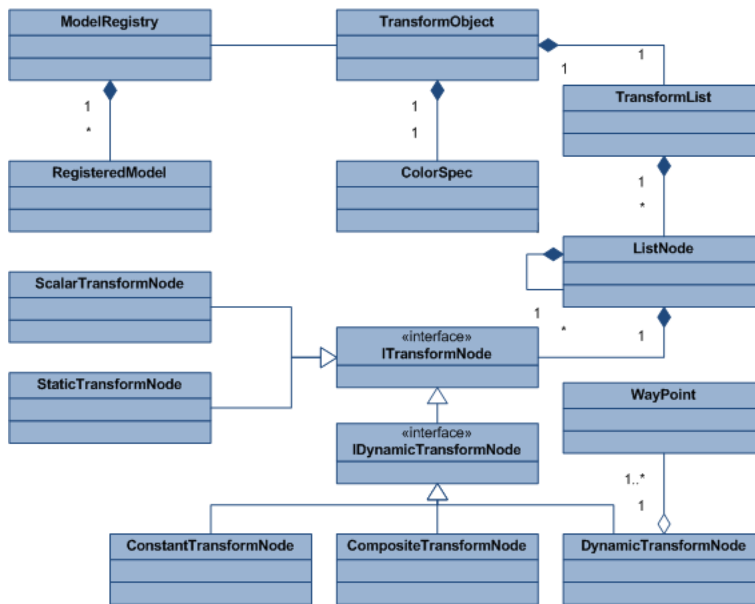


Figure 4.1: Class Diagram providing an overview of the relations between objects

to the next node in the list. There are five *ITransformNode* variants, each providing a different type of transformation effect. *ScalarTransformNodes* and *StaticTransformNodes* remain constant, and provide for scalar transformation and simple static transformation respectively. *DynamicTransformNodes* and *ConstantTransformNodes* provide dynamic behavior, the former providing a simple waypoint system, the latter providing a means of incrementing or decrementing a transform at a specific rate. *CompositeTransformNodes* facilitate the combining of two transforms into a single transformation node. As *Model* objects are not serializable, a *ModelRegistry* object is used to stored references to all the models loaded in memory, and provides them on request. This allows for *TransformObjects* to share models, reducing loading times.

These components are elaborated upon below.

4.2.1 TransformObject

The *TransformObject* class contains all the information necessary to position, colour and render an object within a scene. The position of the *TransformObject* is determined by the solution to the *TransformList*, a linked list comprising a number of transformations through which the object must go in order to be positioned correctly. The model to be drawn is saved as a string, and references a model in the registry stored under that identity. When an object needs to be rendered, the stored model name is passed to the *ModelRegistry*, which returns the requested model. The *TransformList* and its components, as well as the *ModelRegistry*, are covered in more detail in the following sections.

The *TransformObject* contains the *ColorSpecification* class, which stores custom lighting information for the object. Specifically, it stores ambient, diffuse, emissive and specular colour values, as well as a value for the specular power of the object. When custom lighting is enabled, these values are passed to the *BasicEffect* shader of each mesh within the object, thus applying the custom lighting effects. When custom lighting is disabled, default colour values are retrieved from the *ModelRegistry*.

Next we consider the objects which facilitate the object transformations within a scene.

4.2.2 Transformation Nodes

There are a variety of transformation nodes available, which collectively provide a means for creating sophisticated, animated transformation sequences. All transform nodes implement the *ITransformNode* interface, which specifies the necessary functionality each transform node needs to implement in order to be used within the *TransformList*. The system supports both static transforms, which remain constant over time, and dynamic transformations, which change over time. It is important to distinguish between dynamic transforms, which refers to any transform which changes over time, and the *DynamicTransformNode* class, which provides dynamic behavior through the use of a waypoint system. Essentially, the *DynamicTransformNode* is one of three available types of dynamic transformation node.

Transformation Node Interfaces

In order to allow for different node types within the *TransformList*, two interfaces are used – namely the *ITransformNode* and *IDynamicTransformNode* interfaces. The *ITransformNode* interface ensures that all transform nodes contain the minimum properties and methods necessary for the nodes to be used within the *TransformList.* The properties required include the name and owner of the object (used solely for identification purposes), the type of transformation the node represents, and the node's current transformation matrix. The *ITransformNode* interface also requires implementation of a *Clone* method (intended to return an identical Transform Node), which is used when transformations need to be copied by value to other TransformObjects.

The *IDynamicTransformNode* interface inherits from the *ITransformNode* interface, specifying several additional properties and methods which must be implemented by all dynamic nodes. These include methods to start and stop the dynamics of the node, both a property and method which return a boolean value indication whether the node has been started or not, and a method

Let M be some transformation matrix, and let α be some scalar value. Then:

$$\alpha M = \alpha(IM) = (\alpha I)M$$

So in order to scale M by a value α, we need to multiply it by the matrix αI.

Figure 4.2: Scalar multiplication using a scaling matrix

for updating the node's current game time, which facilitates animation effects when the node has been started.

ScalarTransformNode

The *ScalarTransformNode* class implements the *ITransformNode* interface, and is used to provide scalar multiplication of matrices. The *ScalarTransformNode* transformation matrix is calculated by multiplying the stored scalar value by the matrix identity, which is treated as a regular transformation matrix. This transformation matrix is used to provide scalar multiplication functionality within the list of transformations, and provides an extra measure of flexibility4.2.

StaticTransformNode

The *StaticTransformNode* class implements the *ITransformNode* interface, and is used to provide static (or constant) transformations, which do not change over time. The *StaticTransformNode* class allows for relevant types of transformation (see section 2.3.3), facilitating the creation of *Translation*, *Scale* and *Rotation* transformation nodes. The *StaticTransformNode* class also allows for the creation of *Identity* nodes, which contain the matrix identity as the transformation matrix.

DynamicTransformNode

The *DynamicTransformNode* class implements the *IDynamicTransformNode* interface, and provides support for dynamic transformations that operate using *WayPoints*. Each *WayPoint* is comprised of a both transformation matrix, which can include *Transformation*, *Rotation* and *Scale* components and are multiplied together in *I.S.R.O.T* order (see section 2.3.3), and a duration component, which specifies the time until the *WayPoint* position.

DynamicTransformNodes use the current *GameTime* to determine which two *WayPoints* the current transform should be in between, and what percentage of the current waypoint transition

has already been completed. The *DynamicTransformNode* uses the transformation matrices of the two *WayPoints*, as well as the percentage representing the completeness of the current transition, in order to determine the expected transformation matrix for the specified game time. This is done using the *Lerp* function of XNA's *Matrix* class, which linearly interpolates the two matrices using the specified percentage interpolation value. The *WayPoints* within the *DynamicTransformNode* are applied circularly, so that the final *WayPoint* within the *WayPoint* list is connected to the first *WayPoint* in the list. This ensures that the *DynamicTransformNode* can move indefinitely.

ConstantTransformNode

The *ConstantTransformNode* class implements the *IDynamicTransformNode* interface, and provides support for dynamic transformations that operate by constantly incrementing at a specific rate over time. For instance, a constant transform node can specify that an object should rotate 30 degrees every second, or move 15 units down the x axis every 6.5 seconds. This introduces the problem of when to stop incrementing a transformation, which is tackled using boundaries.

Boundaries indicate at which point a constant transformation should invert, to prevent it growing, shrinking or translating in a particular direction infinitely. Regarding translation, should the position of the object on a particular axis exceed the boundary value, the respective constant translation component corresponding to that axis is reversed, so that the object is reflected off the boundary wall. Similarly, should the scale value for a particular axis exceed the maximum scale value or drop below the minimum scale value, the scaling direction is inverted by reversing the application rate components. Since rotations are inherently circular, no boundary specifications are necessary.

CompositeTransformNodes

The *ConstantTransformNode* class implements the *IDynamicTransformNode* interface, and provides a means to both combine adjacent transformation nodes, and apply parenthesis rules. *ConstantTransformNodes* contain two parent nodes, which reference the original two nodes used for the creation of the node, and an operation to apply between them.

The current transformation is calculated on request by applying the contained operation between the two parent nodes. This is essentially the same as multiplying the two parent nodes within

parenthesis, thus providing a greater degree of flexibility within the application. When a *ConstantTransformNode* is updated, each parent is updated individually, provided that parent implements *IDynamicTransformNode.* Since either of the parent nodes could be a *ConstantTransformNode,* many transformation nodes can be contained within one *ConstantTransformNode,* providing a simple mechanism for producing a single node which represents a complex transformation procedure.

4.2.3 Transformation List

The *TransformList* class is used to manage the list of transformations applied to a *TransformObject.* The *TransformList* is a linked list structure, which utilises *ListNodes* to string transformations together. These classes are discussed in the following sections. This is followed by an overview of *cloning,* which explains how nodes can be cloned by reference or value and sent to a different *TransformList* object.

ListNode

The *ListNode* class provides the linked list functionality required by the *TransformList,* and is essentially comprised of an *ITransformNode* variant, a reference to the next *ListNode* in the list, and a matrix operation to apply between them. The operations available are contained within the *MatrixOperator* enumeration, which can be used to specify *Add*, *Subtract* and *Multiply* operations. Since exactly one *ListNode* will be at the end of every list, a fourth operation type – *None* – is also included. The matrix operation cannot be set as *None,* but the *ListNode* always returns this value when it is the last node in a list.

ListNodes are responsible for both creating *CompositeTransformNodes,* and creating clones of the list structure. These aspects will be discussed in a later section.

TransformList

The *TransformList* is responsible for the geometric transformation of objects within a scene. The object is comprised of a single *ListNode*, which acts as the head of the linked list, and provides a number of list management procedures used by a multitude of other objects within the system. These operations include the addition of new nodes, removal of existing nodes, and moving nodes to different positions within the list.

The *TransformList* is responsible for calculating the *World* matrix for a particular object in a scene (see section 2.3.4), which is computed by iterating through the linked list, applying each transformation in the correct order, using the specified operator.

Cloning

Cloning refers to the creation of new *ListNodes* which either contain the same transforms as the original, as is the case when copying by reference, and new transforms which are identical to the original, used when copying by value. We shall refer to two nodes as being *related* if either node is the clone of the other, or both nodes are clones of the same parent node, and *referentially related* if the related nodes share the same *ITransformNode.*

Cloning by reference is useful when a number of *ListNodes* need to share a transformation node, as is often the case when the movement of two or more objects are related, as any changes to the transform of a *ListNode* will be visible in all referentially related *ListNodes*. Cloning by reference is partially achieved by the internal *SharedStatus* class, which is shared between all referentially related *ListNodes,* and contains the *ITransformNode*, identification information, and the nodes input vector, used for storing form based input.

Cloning by value is useful when two objects require similar transformations, but their movement is not specifically related, and is achieved by creating a new *ITransformNode* with identical parameters, and adding this to a new *ListNode.* In this case, changes to any *ListNodes* cloned by value will not result in changes to other related nodes.

Cloning functionality is primarily provided by the *ListNode* class, and does not specifically involve the *TransformList* class.

4.2.4 ModelRegistry

The *ModelRegistry* class is used for storing and retrieving *TransformObject* models, and provides a mechanism for sharing models between *TransformObject.* When a model is loaded from disk, and it is not contained within the *ModelRegistry*, it is compiled to XNB format so that it can be used by the XNA framework, and loaded into the *ModelRegistry* as a *RegisteredModel* object.

The *RegisteredModel* object contains the model, its bone transformations, and the default colours for each effect in the model. The name of the model is also stored, as it is used to identify and

compare *RegisteredModels*, and is derived from the model file name. The default colours are stored so that they can be recovered should the effect colours be changed. This occurs when custom lighting is applied to the object, and since multiple objects may be using the same model with different lighting options, the default colours are essential for returning the model back to its original state after each object has been drawn.

The *ModelRegistry* stores all the models loaded into the application within a *List* of *Registered-Models,* and ensures uniqueness of models by disallowing two models with the same name to be contained within the list. Furthermore, models are only added to the registry if a model with the same name is not already in the list. Should a matching name be found, the specified file will not be loaded, and the associated *TransformObject* will assume that the model in the model registry is the correct model. Despite introducing a relatively minor limitation, which prevents two different models of the same name from being loaded into the same scene, this design was adopted for a number of reasons:

1. The limitation can be avoided simply by providing unique names for all model files.
2. By default, content management in XNA also utilises file names as unique identifiers.
3. While manual specification of unique model names was possible, this introduced unnecessary complexity.
4. This method all but prevents the loading of duplicate content, improving resource utilisation.

During draw operations, models are retrieved from the *ModelRegistry* using the model name stored in the respective *TransformObject.* Should custom lighting be disabled, default colour is also retrieved and applied to the effects in the model.

4.3 Global GUI Controls

In this section, brief consideration is given to two important GUI controls which are utilised for a variety of tasks. We first consider a custom *TextBox* control for collecting textual user input, followed by a controls which presents matrices to a form. We shall not discuss the *XNAContent* control in this section, as it provides no functionality and has already been discussed (see section 3.4.2).

4.3.1 Filtered TextBox

The *FilteredTextBox* is the primary input component within the Transformation Pipeline Visualiser application. It is derived from the *TextBox* class, extending the basic functionality to provide for the selection of different input types, which represent the various types of input expected by different controls. Input types include *Standard*, *Numeric*, *Alphabetical, Alphanumeric*, *Decimal* and *NegativeDecimal*. Each type specifies a different set of rules of key press filtering, thus eliminating the possibility of invalid characters corrupting user input. Selecting the Standard input type causes the *FilteredTextBox* to act like the typical *TextBox* provided by the .NET framework. All other input types are custom extensions.

Numeric input specifies that the *TextBox* will only accept numeric characters, and will ignore all others. Alphabetical input accepts only uppercase and lowercase alphabetical characters, while Alphanumeric input accepts only uppercase and lowercase alphabetical characters and numeric characters.

Decimal input expects numeric characters and the decimal point character. If a decimal is already contained in the *Text* field, then the input is treated as numeric and no further decimal points will be allowed until such time as the existing decimal point is removed. Furthermore, if the decimal point is the first character in the *Text* field, a '0' is inserted in front of it automatically. Finally, should the last character in the *Text* field be a decimal point when the *FilteredTextBox* loses focus, it will automatically be removed.

NegativeDecimal input is similar to Decimal input, except that it supports negative values. If the '-' character is detected, it will be added to the front of the *Text* field if the value was initially positive, and remove it from the first index of the *Text* field if the value was originally negative.

The *FilteredTextBox* ensures that all input is well formed, providing very strong input validation. The modifications also improve ease of data entry, as values are automatically filled in within certain contexts. Thus the *FilteredTextBox* provides a number of benefits, improving both usability and stability.

4.3.2 Matrix Control

The *MatrixControl* is a custom Windows Forms control for displaying the *Matrix* value of the *ITransformNode* contained within a *ListNode.* The MatrixControl shows the matrix values to three decimal places for legibility, as well as the name of the node, its owner and its type. The *MatrixControl* is drawn using the *OnPaint* method, and does not collaborate with XNA.

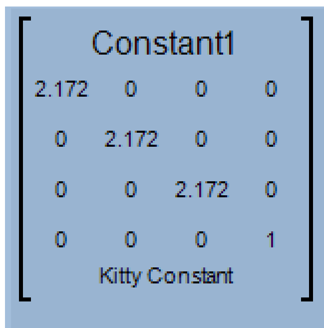


Figure 4.3: A MatrixControl component

4.4 Transformation Visualisation

4.4.1 Overview

In this section we consider elements which enable the core functionality of the system, namely manipulation of the transform lists of objects within a scene. This includes the creation and management of *TransformObjects*, and their associated transformations, both detailing the necessary functionality and providing an overview of its implementation.

While rendering objects to screen forms an important part of this process, it is integral to many other system components and will thus be discussed in a later section (see section 4.8).

4.4.2 Functionality

Before we can discuss the implementation of specific elements within the system, it is important to understand exactly what the implementation actually achieves, as the operation of these elements was largely determined by the functionality to be incorporated. In this section we detail exactly what functions the transformation visualisation component facilitates, including an overview of the Forms used to capture input, and the functions available to manipulate the orientation of objects within a scene.

Creating, Deleting and Accessing Objects in a Scene

In order to allow for the manipulation of *TransformObjects*, these objects first need to be created. In order to create a *TransformObject,* two pieces of information need to be collected. First, we

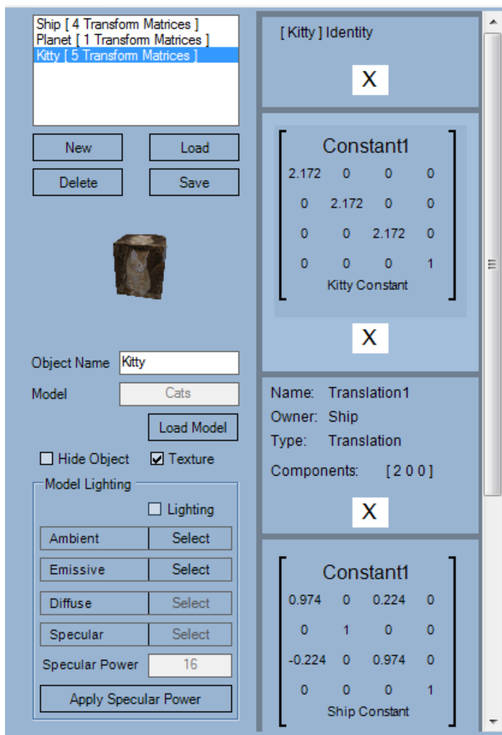
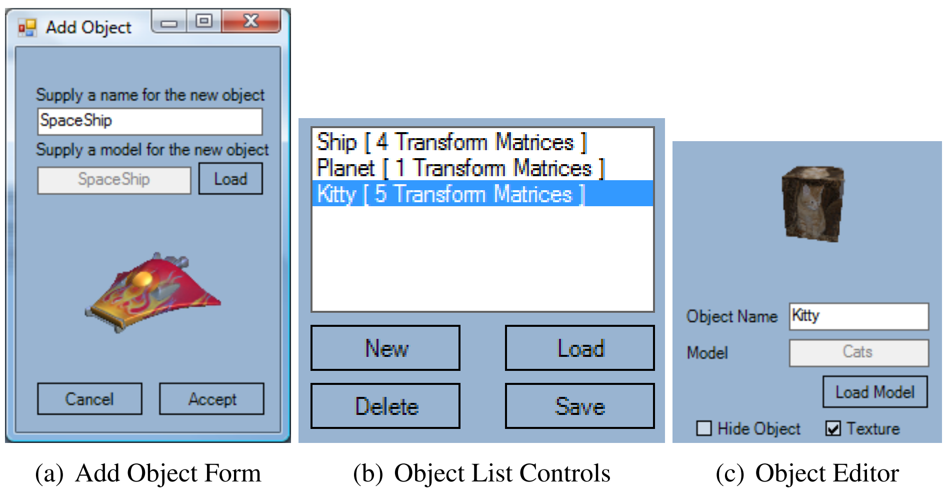


Figure 4.4: The Transform Manager



(a) Add Object Form (b) Object List Controls (c) Object Editor

Figure 4.5: Object Controls

require the name associated with the object. This is essentially an identifier for the object, and it must be unique. Second, the model associated with the object must be selected, and loaded into the *ModelRegistry* if necessary.

Creation of objects is achieved using the *Add Object* Form. This form ensures that both the name and model assigned are valid before allowing the object to be created. The name text field is a *FilteredTextBox* which allows only alphanumeric input, and the model can only be selected using the load button. The model loading process comprises three stages. First, the *ModelRegistry* is checked to ascertain whether a model by the same name has already been registered. If it is not registered, the loading process attempts to load a compiled version of the model, as it may have been compiled by a previous session. If this fails, then the model is compiled using the *ContentBuilder*. This three step process ensures minimal loading times without introducing unnecessary operation complexity. This basic procedure is used in all model loading operations.

The selected model is displayed within a *ModelViewer* control, to help verify that the correct model has been loaded. Once an object has been added to the scene, it becomes available for selection within the Object List. From here, it may be deleted or saved, while further objects may be created or loaded, using the available Object List controls.

Loading and Saving Objects

Objects can be saved and reloaded for use in other scenes. Since the *TransformObject* and all of its constituent components are serializable, these objects are simply serialized into a binary file, and saved to disk. Loading objects is slightly more complicated, as loaded objects may reference models which are not contained within the *ModelRegistry,* thus requiring that those models be recovered. The recovery operation is performed by the object in question upon loading, and follows a similar three step procedure discussed in the previous section, except that if a compiled version of the model cannot be found, the application prompts the user to select the appropriate model to load. This allows for objects to be loaded with different models, should the original models not be available. As object names are required to be unique, the loading procedure also forces a rename of a loaded object if it shares the same name as another object in the list.

While most information is retained by this procedure, all referential relationships to *ITransformNodes* contained in other objects are lost. Loaded objects do not retain these relationships for two primary reasons:

1. Retaining a relationship to an unloaded object is difficult, and makes little sense.

2. Referential relationships between objects can be maintained by saving the entire workspace (see section 4.8), eliminating the necessity for maintaining relations to objects that will likely not be included in the scene.

Editing Objects

Once an object has been selected from the Object List, its name and model can be changed from the Object Editor, which is located beneath the Object List. The Object Editor prevents the renaming of objects to a non-unique value, while the model loading component acts in an almost identical fashion to its Add Object Form counterpart.

The Object Editor also allows for models to be hidden, which prevents them from being drawn. This functionality is achieved through a *CheckBox* component, and is intended to help simplify complex scenes by allowing the user to remove objects from a scene temporarily, in order to focus on other objects without obstruction. Similarly, the Object Editor allows the user to turn model texture support on and off. This allows the user to see untextured objects, improving understanding of how textures are applied to objects.

The Object Editor also provides control over custom lighting. While custom lighting is relevant to the Transformation Visualiser components, it will be discussed at a later point (see section 4.6).

Adding and Editing Transformations

In order to transform objects within a scene, transformation matrices need to be applied to those objects. Initially, all objects have only one Identity transformation applied to them, and will maintain at least one node in their object's *TransformList* at all times. This is necessary, as manipulation of the *TransformList* is achieved through a context menu accessed by right-clicking a node in the list, and thus without any nodes in the list, it would be impossible to interact with the system. This method was used because the options available at any time are sensitive to the node being operated on. For instance, static nodes cannot have their dynamics started, and nodes at the top of the list cannot be moved further up. By positioning the mouse cursor over any available options will display a *ToolTip* explaining what the particular option does, while *ToolTips* for deactivated options explain why the option is not available. We shall consider adding and editing transformation nodes in this section, and consider other operations in the next section.

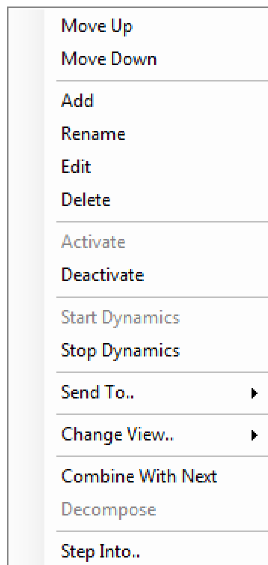
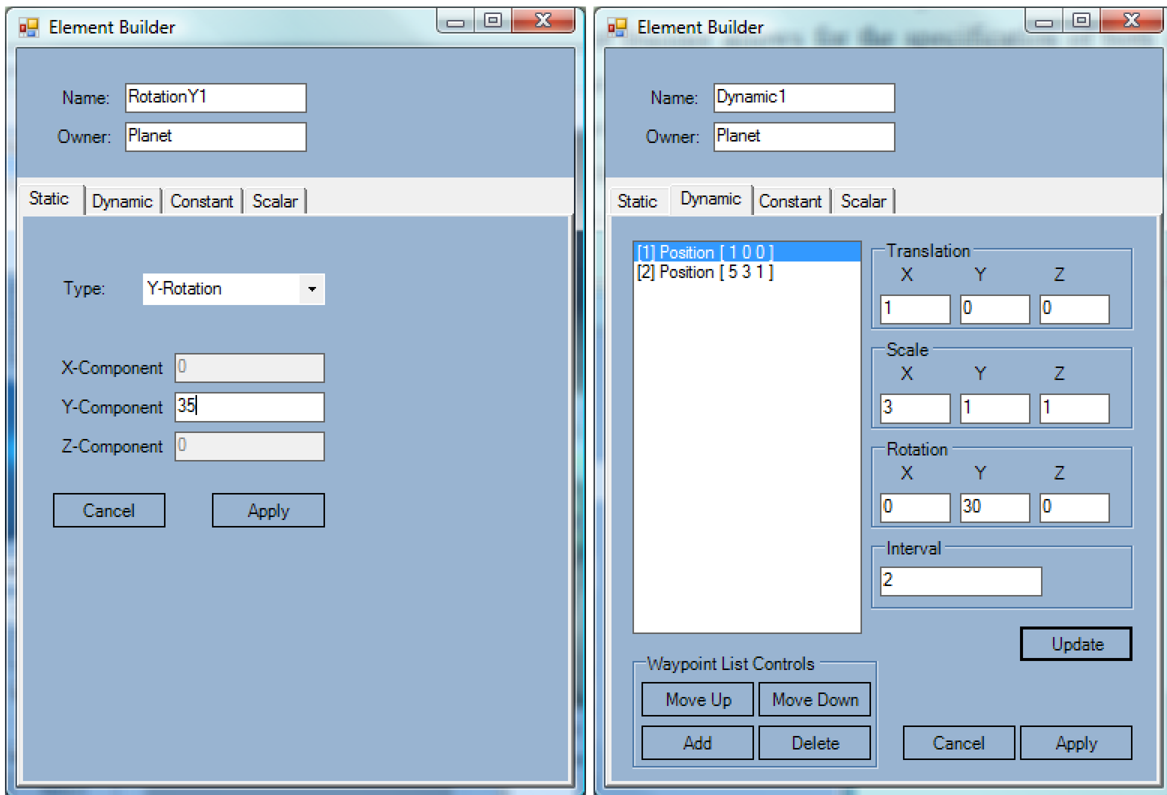


Figure 4.6: The Element Context Menu

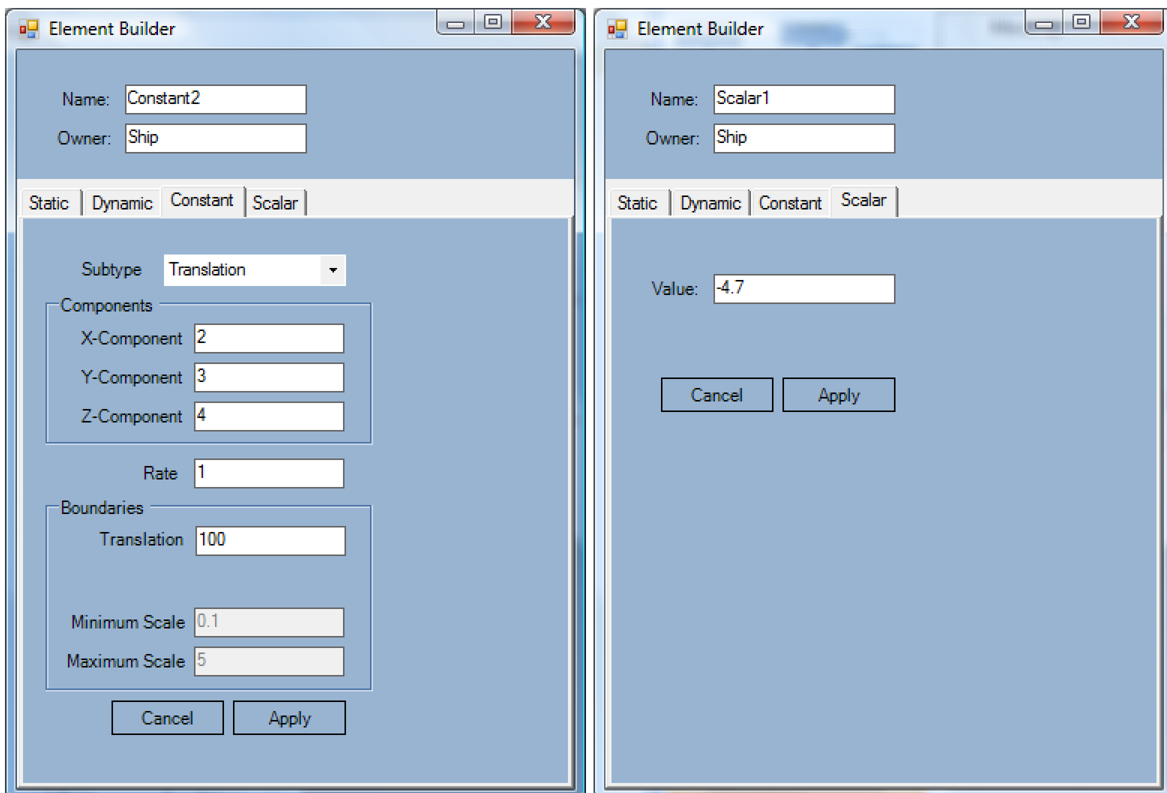
The process of adding new nodes is initiated by selecting the 'Add' option in the Element Context Menu. This opens the *Element Builder* Form, which facilitates the creation of *StaticTransformNodes*, *DynamicTransformNodes*, *ConstantTransformNodes* and *ScalarTransformNodes*. The Element Builder allows for the specification of both a name and owner for the transformation node, but will provide default values for these so as not to necessitate unessential input. As long as the transform name has not been changed, changing the type of transformation will result in the default name changing to reflect this type. The owner field allows for a simple way to keep track of the origins of transforms. As it does not provide or affect transformation functionality, it may be changed without consequence should the need arise. In the majority of cases however, it should be left as is.

The form contains a number of tabs, each intended for creating a different type of *ITransformNode* and adding it to the transformation list. The first tab facilitates the creation of *StaticTransformNodes,* and allows for translation, scaling and rotation transformations, as well as static nodes containing the matrix identity.

The Dynamic Tab allows for the creation, deletion, editing and ordering of *WayPoint* objects of a *DynamicTransformNode.* Initially, the Waypoint List contains a single identity transform. Adding, deleting and changing the order of WayPoints is facilitated by the Waypoint List Controls, located directly below the list. These options can also be reached from the Waypoint Context Menu, accessible by right clicking a waypoint in the Waypoint List. The Waypoint Context Menu also allows for the copying and pasting of WayPoints within the Waypoint List, which is intended to ease creation of similar WayPoints within a list.



(a) The Static Tab (b) Dynamic Tab



(c) Constant Tab (d) Scalar Tab

Figure 4.7: The Element Builder

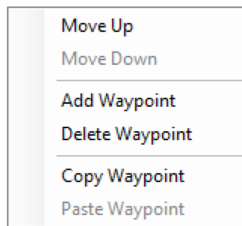


Figure 4.8: The Waypoint Context Menu

Once a Waypoint has been selected from the Waypoint List, its component transformations can be edited, and updated once complete. If a single Waypoint utilises multiple transformation components of differing types, then the resultant matrices of these transformations are multiplied together in I.S.R.O.T order.

The Constant Tab allows for *ConstantTransformNodes* to be created and, when relevant, allows adjustment of the translation and scale boundaries. Boundary editing is only enabled if the selected type of transformation correlates to the boundary. For instance, setting the translation boundary is only allowed if the node is of a *Translation* type.

The Scalar Tab is used to create *ScalarTransformNodes,* and provides only a single input field, the scalar value, to generate the appropriate node. All input within the Element Builder Form is collected using *FilteredTextBoxes,* which are set as *Alphanumeric* for all textual input fields, *NegativeDecimal* for all component vector input fields, and *Decimal* for all rate input fields. This helps ensure valid input, and improves the stability of the system. By clicking apply within a specific tab, the appropriate *ITransformNode* is created and added to the appropriate Transformation List.

Editing of objects is almost identical, except no automatic name generation is used, and changes to the Waypoint List and its WayPoints are visible within the rendered output. All changes can still, however, be undone by clicking the cancel button within any tab. In the following section, we shall be considering the transformation list as a whole, and the operations available for transformation manipulation.

Manipulating the Transformation List

In order to properly position and orientate an object in a scene, and to visualise the effect of certain operations such as changing the order of transformations, a number of manipulation operations are supported, and are accessible through the Element Context Menu. The most vital of these, namely the *Move Up* and *Move Down* operations, allow for the reordering of

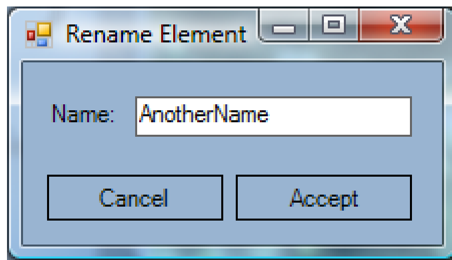


Figure 4.9: Rename Element Form

nodes within the Transformation List. This provides both a means for visualising the effects of changing the order of transformations, and for developing and fine tuning more complex transformations.

While we have already discussed adding and editing transformations in the previous section, it is worth considering the delete and rename operations. Deleting a node removes it from the list, but does not necessarily destroy its transformation. Thus, delete can be applied to any node, while leaving all referentially related nodes intact. The rename operation opens the Rename Element form, which provides a simplified mechanism for renaming transform elements quickly.

Each node is capable of being in either an active or inactive state. Upon creation, all nodes are initially in an active state, and operate as expected. Deactivating a node essentially ensures that the node is ignored when calculating the solution to the transformation list. This provides a simple mechanism for visualising the effect a node has on the calculation of the *World* matrix, and for testing the effect of removing a node without actually having to remove it. As keeping track of active and inactive nodes could easily become unmanageable without assistance, the Transformation List colours all inactive transformation nodes *SlateGrey* to provide an easily recognisable indication of which nodes are inactive.

Nodes which implement the *IDynamicTransformNode* interface can have their dynamics started and stopped at any time. On creation, all nodes are active, as would be expected. Editing, however, does not restart stopped nodes. This facility is intended primarily for *DynamicTransformNodes,* allowing for their initial positions to be set without interference, but can be used by *ConstantTransformNodes* to achieve a similar, if slightly less useful, effect.

Transformations can be sent to other transformation lists, or they can be sent to their own transform list, either of which can be done by value or by reference. Sending transformations to a different list by reference results in a referentially related transform node being appended to the end of that list. When sent by value, the appended transform node will have the same value,

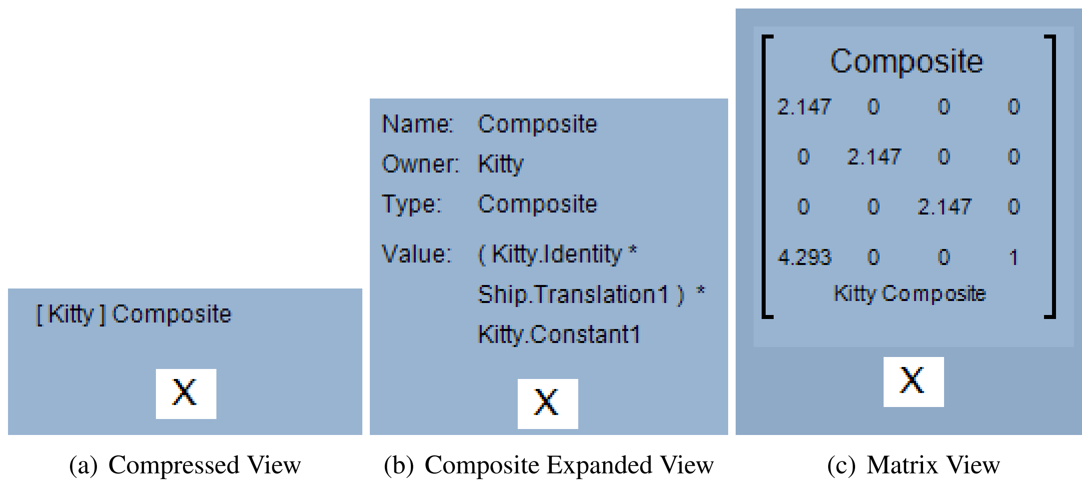
but will not be linked to the original transform. As the specific differences between referential and value sending have already been discussed (see section 2.3.3), we shall not discuss them further. Sending nodes to the same list provides a method for squaring transformations, should this be necessary.

Nodes can be viewed at three different levels of abstraction, namely *Compressed*, *Expanded* and *Matrix* views. The Compressed view shows only the owner and name of the transform node, and is the highest level of abstraction. For more detail, the Expanded view shows specific, type sensitive information relating to the *ITransformNode.* In this view, most relevant information is displayed, with the exception of *DynamicTransformNodes,* which display only the positions for each *Waypoint.* This view provides a textual overview of the transformation, providing enough detail to discern what the node does. The Matrix view shows the actual transformation of the matrix, utilising the *MatrixControl* as a presentation mechanism. When the transformations are dynamic, the MatrixControl display is updated frequently to help illustrate how the dynamics affect the values in the matrix. This is particularly beneficial in collaboration with the rate components of *IDynamicTransformNode,* which allow a transformation components to remain unchanged, but be applied over a far longer length of time. This enables the user to reduce the rate of change of the transformation, thus providing a better view of how matrices change over time. The view of a node can be changed, either by using the Element Context Menu, or by double clicking the node. When double clicking the node, views are cycled iteratively.

The Element Context Menu provides the only means for creating *CompositeTransformNodes,* which is achieved by selecting 'Combine With Next'. This operation creates a new *CompositeTransformNode,* using the the current and next nodes as parents, and replacing them in the TransformList. Composite nodes may be combined with other composite nodes, or decomposed to replace them by their parent nodes. Combining nodes does not affect any nodes which are referentially related to those nodes, as the *ITransformNodes* themselves persist. This ensures flexibility, and allows for *CompositeTransformNodes* to be used as a safe mechanism for building complex transforms, and reducing the visual complexity of long transformation lists.

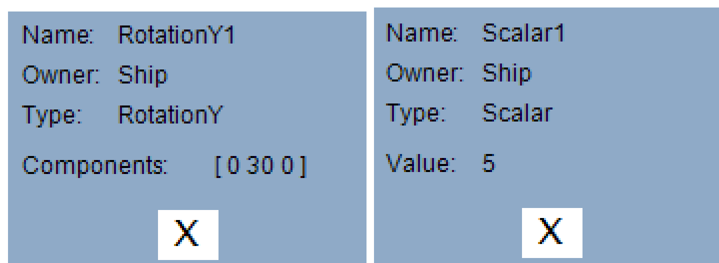
The final operation available in the Element Context Menu – 'Step Into' – allows the user to step into the calculation of the Transformation List solution. This functionality is discussed in section 4.5, and so will not be considered further at this point.

All Transformation Nodes contain a *MatrixOperation* button, which allows the user to specify which matrix operation to apply between the current element and the next element in the list. Operations can be cycled by clicking the button, or specified by right-clicking the button to access its context menu.

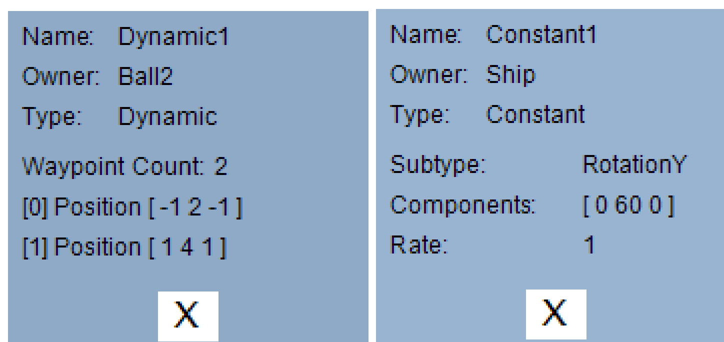


(a) Compressed View (b) Composite Expanded View (c) Matrix View

Other Expanded Views:



(d) Static Expanded View (e) Scalar Expanded



(f) Dynamic Expanded View (g) Constant Expanded View

Figure 4.10: Transform Node Views

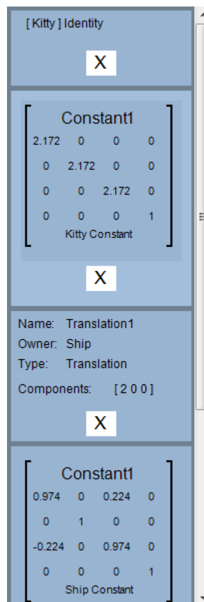


Figure 4.11: Transform List

Together, this functionality ensures that relatively sophisticated World matrices can be created using the Transformation List, allowing the user significant depth of exploration.

4.4.3 Associated GUI Controls

The Transformation Visualisation component is achieved using a number of custom components. In this section, we elaborate on these controls, detailing interesting functionality.

Model Viewer

The *ModelViewerControl* is used to display the model associated with an object outside of the context of a scene, and is used in both the Add Object Form and the Object Manager. The *ModelViewerControl* supports run time model switching, accepting both default and custom object colours to ensure correct representation. Upon loading the model, the *ModelViewerControl* calculates the center of the model, and rotates the y axis about this point, resulting in the model slowly rotating. The *ModelViewerControl* inherits from *XNAContent* in order to present XNA output to a control.

TransformElementList

The *TransformElementList* is responsible for managing the Transformation List, and acts as a container for *TransformElement* controls. The *TransformElementList* is responsible for both element positioning, and servicing certain element events resulting from Element Context Menu operations, such as element reordering, creation and deletion. Any changes made to the *TransformElementList* are instantly reflected in the *TransformList* of the relevant object, as well as within the scene viewport.

TransformElements

The *TransformElement* class is responsible for the bulk of the applications Transformation List manipulation capabilities, typically accessible through the Element Context Menu. The Element Context menu is dynamically generated when the user right-clicks an element, and checks the state of the contained *ListNode* and the *TransformList* of which is apart, to determine which options should be disabled. This is facilitated using a number of delegates and events, which allows elements to communicate with the *TransformList,* both to manipulate it, and to poll it for information.

Object Controls

The Object Controls allow the user to rename objects, change their models, and set their visibility and texture options. These controls also include custom lighting options, but as these controls are more relevant to the lighting facilities of the application, they shall be discussed within that context (see section 4.7). As previously stated, the application does not allow two objects with the same name to exist within the transform list. While the application prevents a object with a duplicate name from being added to the object list using the Add Object form, this may still occur as the result or either loading an object with the same name, or renaming the object from the Object Controls. To prevent this from happening, the *FilteredTextBox* used for renaming will retain focus until the name specified is unique. Attempts to do other operations will result in a *MessageBox* informing the user that they must first resolve the name conflict before they can continue. In order to resolve conflicts when an object is loaded, the *FilteredTextBox* is given focus directly after loading. Thus, if a name conflict exists, the user is forced to rectify this before he can continue.

In order to hide objects, the *Hide* checkbox simply sets the *Active* property in the *ListNode,* indicating that the object should not be drawn. When drawing objects, inactive objects are filtered out. Disabling textures is done in a similar fashion, as the *Texture* checkbox value is applied to the *BasicEffect* objects contained in the mesh during the draw operation.

While Model Loading is important, it has already been discussed in a previous section (see section 4.4.2), and thus will not be discussed here.

Element Builder Form

The *ElementBuilderForm* facilitates the creation of elements within the *TransformElementList,* and thus *ITransformNodes* within the *TransformList.* Elements are built within a specific tab, and upon clicking Accept, relevant input is combined to produce a new *ITransformNode,* which is either added to the list within a new *ListNode*, or used to replace a transform within an existing *ListNode*. The *ElementBuilderForm* uses an *ElementNameManager* object to generate appropriate default names. The *ElementNameManager* maintains an array of instance counts for each *ITransformNode* type*,* and in the case of *StaticTransformNodes*, the count of each transformation type. If the name field has not been manually changed, then the name field will changed to reflect the selected node type, whenever the node type is changed, removing the need for users to manually specify unique names, without introducing too much ambiguity.

4.5 Matrix Operation Visualisation

4.5.1 Overview

The transformation pipeline is heavily dependent on matrix algebra, which is responsible for combining any number of separate transformation matrices into one. While in the large majority of cases only matrix multiplication is used, the operations of matrix addition and subtraction can occasionally be utilised to achieve transformation effects otherwise difficult or impossible to create. This section discusses the matrix visualisation component of the transformation visualiser, which allows the user to step into the calculation of the world matrix for a scene object.

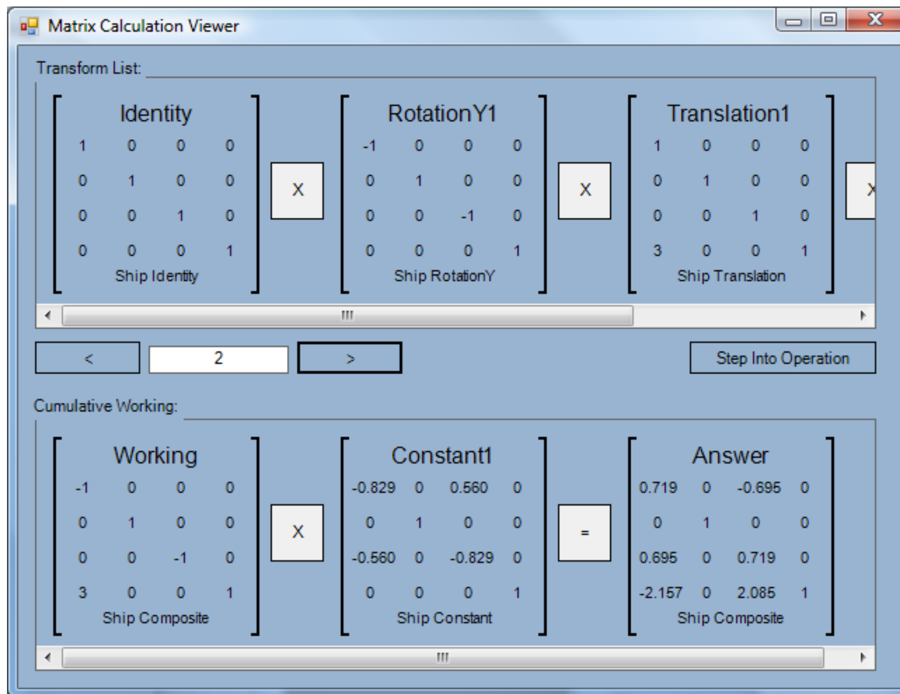


Figure 4.12: The Matrix Calculation Viewer Form

4.5.2 Features

Matrix Operation Visualisation within the Transformation Pipeline Visualiser application is comprised of two *Forms,* which present transformation pipeline calculation information at different levels of abstraction. The first of these forms, the *Matrix Calculation Viewer* Form, is accessed from the Element Context Menu by selecting 'Step Into', and shows both the current transformation pipeline, but also the working calculation up to the node which initiated the operation. The backward and forward buttons facilitate movement to any point in the calculation to view the current working up to that point. All transformations within the list are represented as matrices, using the *MatrixControl* as a presentation mechanism, and are updated in real-time to show how the dynamics of a system affect both the individual matrices, and the calculation as a whole. When nodes are inactive, they are not included in this calculation, and are skipped when moving between steps in the calculation. Finally, changing the width of the form will automatically re-size the widths of each *MatrixBlockList*, allowing the user to expand the number of visible transformations displayed on screen.

The second form allows for the user to step into an algebraic operation between two matrices, which utilises simple animation to illustrate how the operation is applied to each node to achieve a result. The *OperationViewerForm* is accessed using the 'Step Into Operation' button, and shows the calculation of the current step in the Matrix Calculation Viewer Form. Given

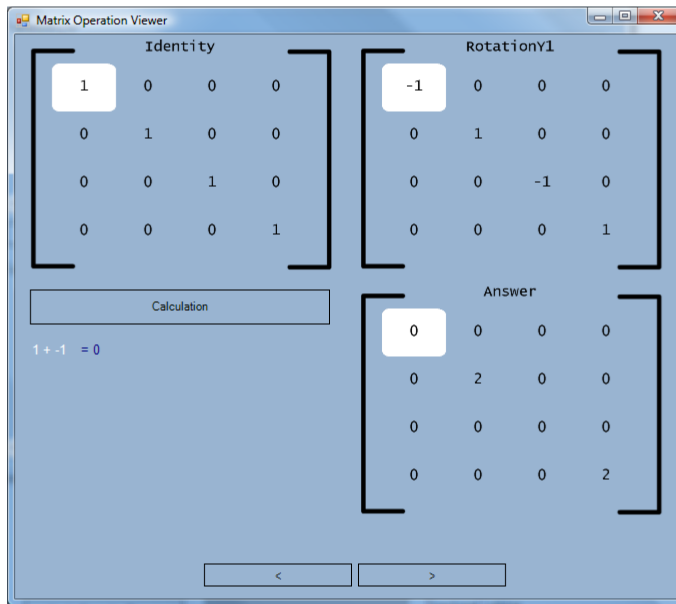


Figure 4.13: Matrix Addition and Subtraction Layout

the dramatic differences between Matrix Addition and Subtraction and Matrix Multiplication, different Form layouts are utilised to better illustrate the sequence of calculations involved in producing an answer matrix. The matrix values are rendered in real-time, in order to aid in visualising dynamic transformations.

The *OperationViewerForm* includes an Information Panel, which provides three different textual representations of the matrix operation calculation sequence. These include the *Calculation,* where the current calculation is represented numerically, with the current working highlighted; *Working*, which is similar to Calculation, except that working is accumulated; and *Operation,* which uses string based representations of matrix values to show the calculation symbolically. These views can be cycled by clicking their title. This is intended to help illustrate exactly what operations occur, in sequence, when calculating the answer to a matrix operation.

Together, these forms provide a means for 'drilling down' into the calculation of the transformation pipeline calculation sequence, and can be utilised to improve familiarity with transformation sequences and matrix operations.

4.5.3 Associated GUI Controls

There are several controls dedicated specifically to the visualisation of the calculation procedure. These controls are discussed in the following section, drawing attention to interesting functionality.

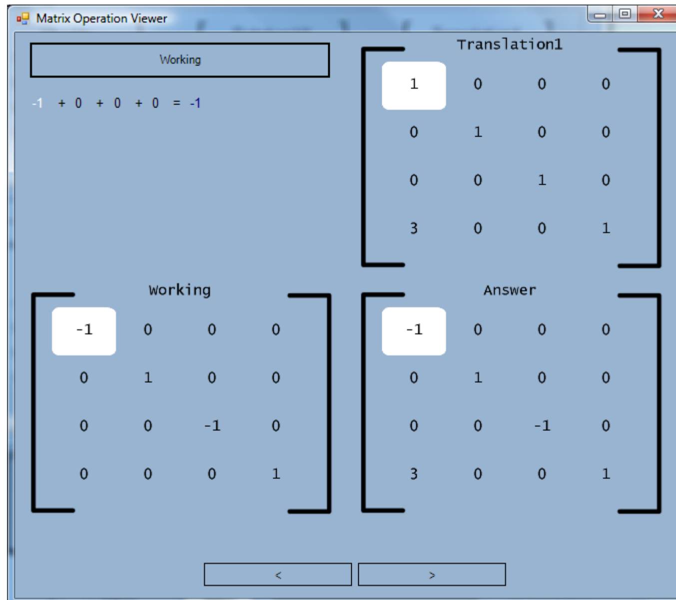
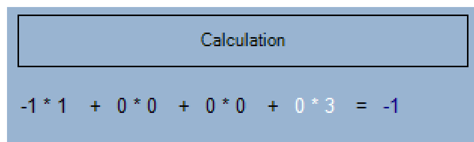
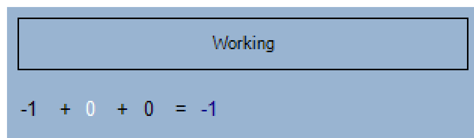


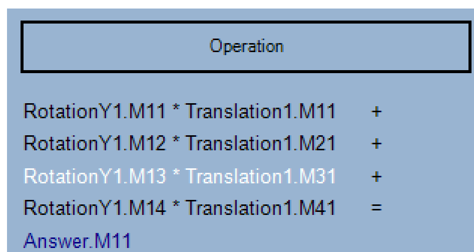
Figure 4.14: Matrix Multiplication Layout



(a) Calculation View



(b) Working View



(c) Operation View

Figure 4.15: Matrix Operation Viewer

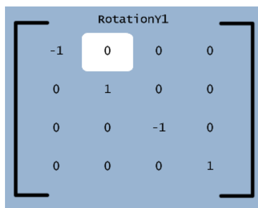


Figure 4.16: DrawnMatrix Control

DrawnMatrix

The *DrawnMatrix* control is an XNA Game Component which extends *XNAContent* and presents its output to an *XNAPanel*. Implementation within XNA was desirable, so as to allow for efficient and smooth animation of the white tile used to represent the current position in the matrix. Furthermore, as the *DrawnMatrix* polls the relevant matrices on each Draw iteration, the most current values for dynamic transformations are inherently displayed. Tile positioning is handled by the *TileManager* class, which uses the current *GameTime* and position interpolation to show the transition from one cell to another. Only one transition can be underway at any given time, and each transition takes 200ms to complete. This was done for simplicity. The movement of the Tile is dependent of the state of the *TileManager,* which, in turn, is dependent on the type of operation being applied. The *DrawnMatrix* control depends on a *SpriteBatch* to display both text and graphics.

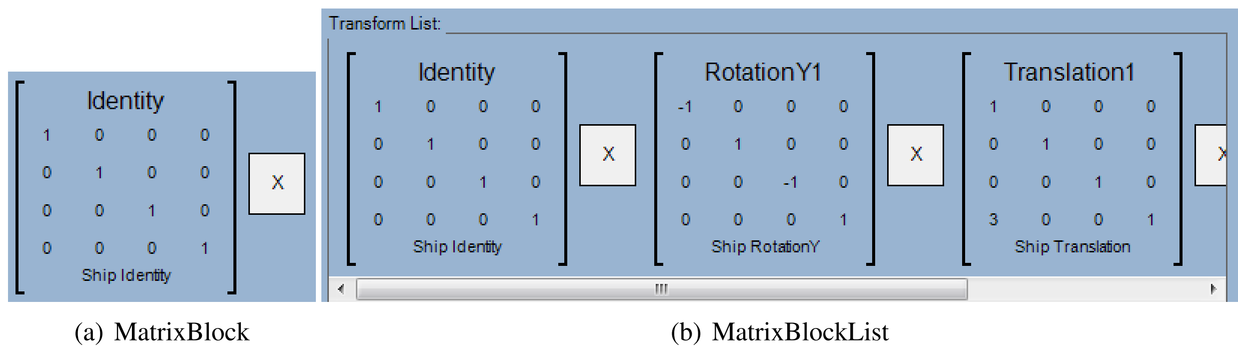
OperationViewer

The OperationViewer is responsible for positioning the DrawnMatrix controls and InformationPanel, based upon the operation to be visualised, and coordinates the concurrent updating of all contained components to the correct phase of the calculation.

MatrixBlock and MatrixBlockList

The *MatrixBlock* control is responsible for displaying the transformation matrix and operator of its contained *ListNode,* and is used for representing *ITransformNodes* within the *MatrixBlockList.* The Matrix is presented to the control using a *MatrixControl.*

The *MatrixBlockList* is a more sophisticated control, which can be used to display various stages of the transformation calculation, as well as display the solution to a transformation list using the *Answer* class. The *Answer* class is essentially contains a MatrixBlock control which



(a) MatrixBlock (b) MatrixBlockList

Figure 4.17: MatrixBlock and MatrixBlockList

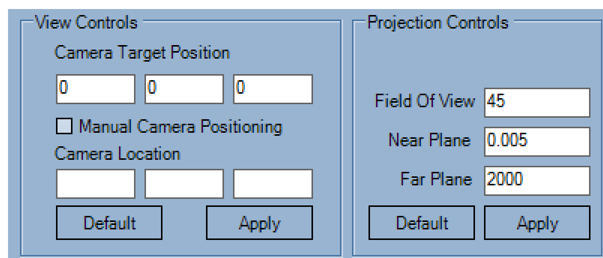


Figure 4.18: Camera Controls

contains a *CompositeTransformNode* comprised of all the *ITransformNodes* in the list, applied in the correct order. This *CompositeTransformNode* is then updated by the game loop, ensuring the values remain current.

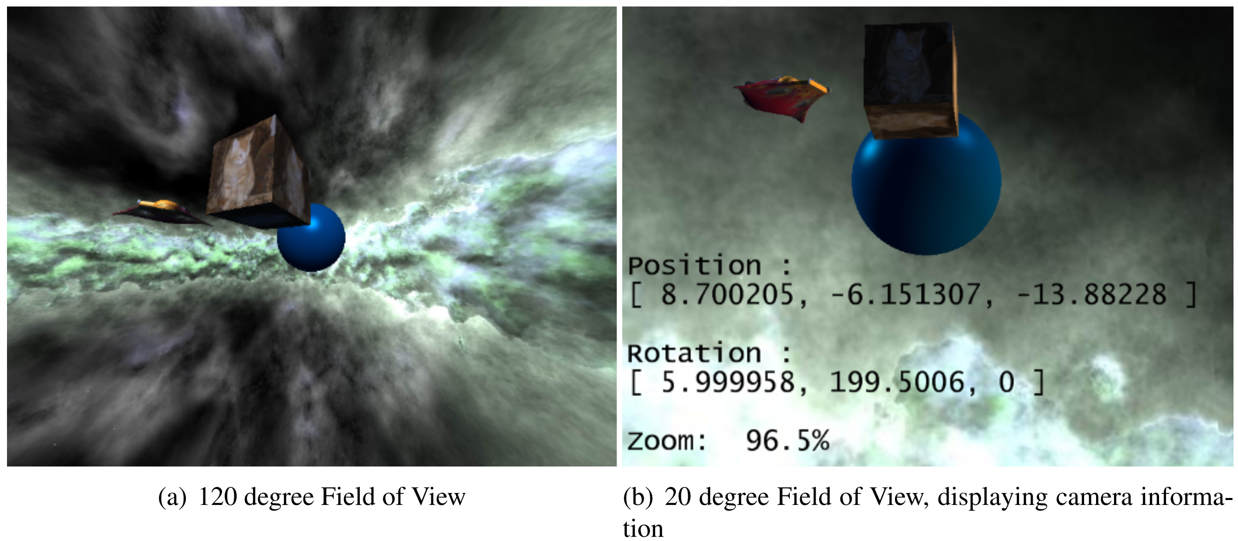
4.6 View And Projection Manipulation

4.6.1 Overview

View and Projection manipulation is provisioned by the Camera Controls, found below the main viewport. They provide an interface for specifying view and projection parameters, which are used to create the appropriate View and Projection matrices within the *Camera* object, used when rendering to the main viewport.

4.6.2 Features

View and Projection matrix generation is relatively simple, and as such, requires relatively little functionality. View Controls include the ability to specify the camera target position, as well as



(a) 120 degree Field of View

(b) 20 degree Field of View, displaying camera information

Figure 4.19: Field of View and Camera Information

to enable and apply manual camera positioning. When manual camera positioning is disabled, the camera can be rotated around the origin and zoomed in and out using a combination of directional keys, +, - and the ctrl key after clicking on the viewport. x and y axis rotations are applied using the left and right, and up and down directional keys respectively. z axis rotation is accessed by holding down the *ctrl* key, and pressing the left and right directional keys. *Zooming* is achieved using the + and - keys, while fast zooming can be accessed by holding down the *ctrl* key when zooming.

Projection controls allow for the specification of the camera Field of View (see figure 4.19), as well as the near and far draw planes of the projection frustum. The camera Field of View can vary between 1 and 179 degrees, allowing for both wide and narrow angle lens specifications. The camera object ensures that the Field of View falls within this range, and that the near plane is always closer to the camera than the far plane. View and Projection changes can be applied using their respective Apply buttons, or be reset to default values using the respective Default buttons.

4.6.3 Associated GUI Controls

GamePanel

The *GamePanel* is a more sophisticated *XNAPanel*, which is capable of capturing keyboard input, which can be used to control the camera. The *GamePanel* inherits from the *TextBox* control, but does not display the captured input. Instead, it sets relevant states in a *Keyboard* object,

which is passed to the camera on each game loop iteration. The *GamePanel* has its Multiline property enabled, so as to allow vertical resizing. As XNA content is presented over the *TextBox* graphics, and the *TextBox* control is designed to collect keyboard input, the *GamePanel* operates well in both capturing input and presenting output.

Camera

The *Camera* class used in the Transformation Pipeline Visualisation application is responsible for positioning the camera within a scene. The camera is directly interfaced by the *CameraControl,* allowing the manipulation of all available variables through this Form control. The *Camera* class also uses the *Keyboard* object generated by the *GamePanel* to rotate and and scale the camera about the origin, thus providing camera position and zoom manipulation capabilities through keyboard input.

CameraControl

The *CameraControl* is a custom Forms control which allows editing of View, Projection, Skybox, and Directional Lighting options. The Skybox and Directional Lighting elements are discussed in the following two sections.

4.7 Controlling Lighting

4.7.1 Overview

The Transformation Pipeline Visualiser Application provides a number of controls for customising both object and scene lighting. Object Lighting controls allow for the enabling and disabling of custom lighting for the object, as well as access to a number of light colour properties. Scene lighting is located in the Directional Lighting section of the *CameraControl,* and allows selection of both the ambient and specular properties, as well as direction, of three directional lights. These lights are enabled from the Camera Control. Relevant theory is discussed in section 2.3.5.

(a) Custom Lighting Enabled

(b) Custom Lighting Disabled

Figure 4.20: The Effects of Custom Object Lighting

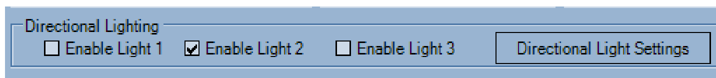
4.7.2 Features

Object Lighting

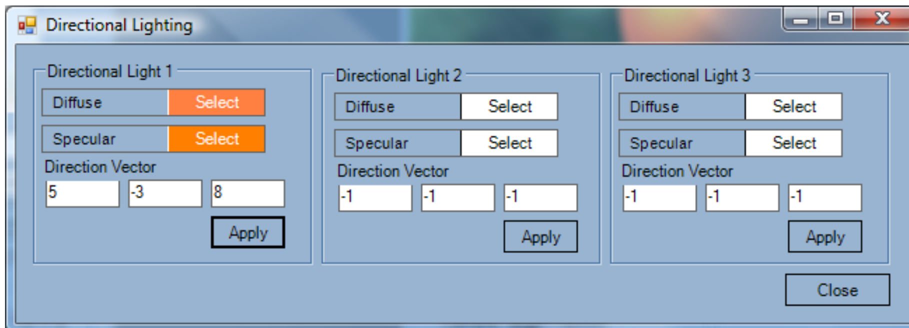
Object Lighting features allow the user to specify the *Ambient*, *Emissive*, *Diffuse* and *Specular* light colours for the selected object model, and adjust the applied Specular Power value. Colours can be adjusted by clicking the appropriate Select button, when Custom Lighting is enabled. The currently selected colour is displayed as the background to the Select button, making the currently loaded colour easy to discern. The Specular Power field allows the user to specify a specular power value, accepting *Decimal* input. High Specular Power values make a material appear smooth and reflective, while low values correspond to a dull, matte texture.

Direction Lighting

Lighting Direction features allow the user to manually configure the three independent directional lights made available by the XNA *BasicEffect* class. Individual directional lights can be enabled and disabled from the Camera Control panel below the main viewport.. Light parameters are configured from the Directional Light Settings form, also accessed from the Camera Controls panel. Parameters include Diffuse and Specular light colours, as well as the direction vector, for each light.

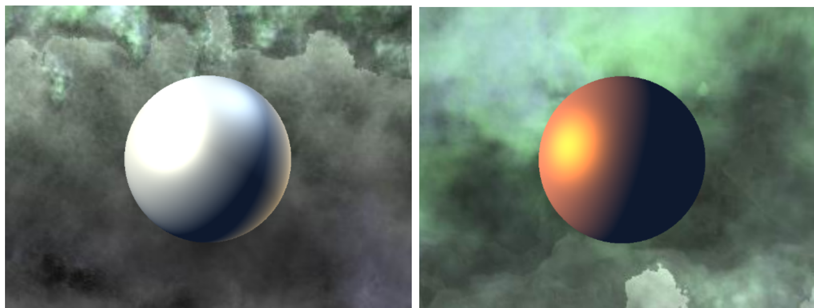


(a) Directional Lighting Controls



(b) Directional Light Settings Form

Figure 4.21: Directional Lighting Controls



(a) Defaultl Lighting

(b) Single Orange Directional Light Enabled

Figure 4.22: Directional Lighting Effects

Figure 4.22 shows the same object, rendered with and without directional lighting enabled.

4.7.3 Associated GUI Controls

Colour Selector

The *ColourSelector* control is used for specifying light colours for objects and directional lights. The control contains a *Select Button* which opens a ColorDialog, and sets the result to its back-

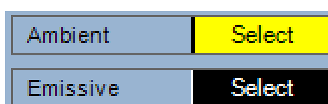


Figure 4.23: ColourSelector Control

ground colour, thus acting as an indicator of the currently applied colour. This colour can then be converted into a format usable by XNA, and stored within the relevant *TransformObject*.

Directional Light Settings Form

The *DirectionalLightForm* allows for the changing of directional light properties for three independent lights. The *DirectionalLightForm* allows properties of each light to be specified independently of others, and thus has three separate *Apply* buttons, each responsible for updating the properties of its respective directional light. The Directional Light Settings form is shown in figure 4.21.

4.8 Synthesis

4.8.1 Overview

This section considers the display of the transformed objects to the screen, loading and saving workspaces, as well as generation of a simple Skybox based environment for the scene.

4.8.2 Features

Skybox

The scene environment is created using a simple Skybox control, which uses six images to create a seamless cubic enclosure. In order to provide greater flexibility with regard to the visualisation of a scene, the application supports custom Skybox creation, and allows for these controls to be saved for later use in different scenes. These features are facilitated by the *Edit Skybox* form, which utilises managed input components to collect necessary information, ensuring that the Skybox is generated with well formed input. The form requires that a valid Skybox name be specified before any textures can be loaded, and further ensures that all textures are loaded before allowing Skybox generation. Once generated, texture panels can be rotated using the appropriate *R* button, located next to the texture name.

Valid skyboxes may be saved by clicking the Save button, while existing skyboxes may be loaded using the Load button. As *Texture* objects are not serializable, the texture file names are

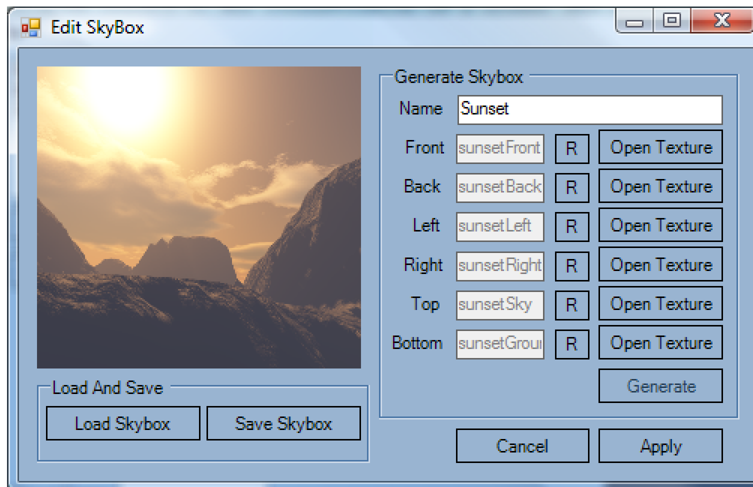


Figure 4.24: Edit Skybox Form

used to reload the original texture files. As this process can take a long time when textures have not been previously compiled, progress is indicated textually within the Skybox Viewer control, providing feedback to the user as to the expected completion time of the Skybox generation operation. On clicking *Apply*, the custom Skybox is placed in a buffer, where it is later collected by the *Update* procedure and used to replace the existing scene Skybox.

If necessary, the default Skybox can be loaded by clicking Default Skybox in the Camera Control. The Skybox diameter can also be manually adjusted using the *Radius* field. The radius of the Skybox is the distance from the center of the scene to its closest edge, and is thus half the diameter. By using a small radius, it is possible to see how the Skybox works. Figure 4.25 shows a Skybox with a radius of five, demonstrating how this facility can be used to better understand the illusion created by the Skybox.

Viewport Rendering

The Skybox and object models are rendered within the *Draw* method of the *BaseGame* class as a scene, which is then presented to the main viewport in the *MainForm* class. The viewport rendering procedure follows the standard procedure for rendering objects, collecting *BasicEffect* parameter values by polling objects such as the camera and model registry.

Loading and Saving

In order to allow dependencies between objects to be saved, the Transformation Pipeline Visualiser facilitates the saving and loading of *Workspaces*. A Workspace comprises all objects and

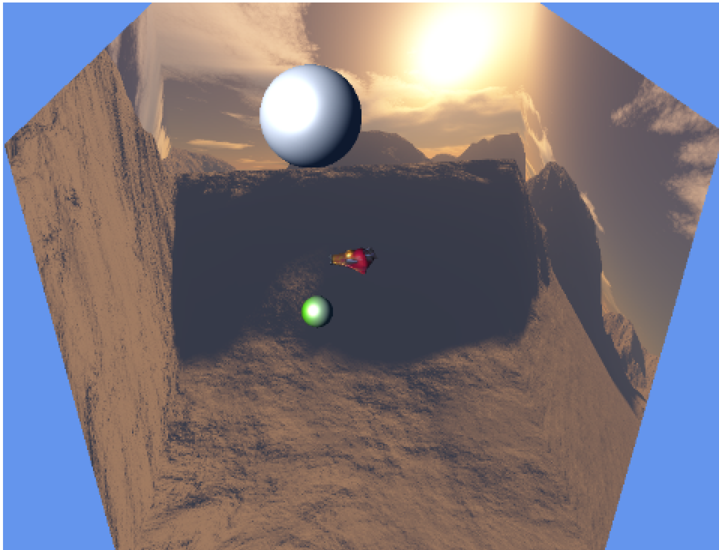


Figure 4.25: A Small Skybox

their transformations, the current *Camera*, *ModelRegistry* and *Skybox*. As the *ModelRegistry* and *Skybox* classes contain references to non-serializable objects, these objects are recovered using *Model* and *Skybox* loading procedures respectively, discussed earlier in this chapter. Loading these objects could take a significant amount of time, particularly in dense scenes, where none of the objects have been compiled before. Thus, during the loading procedure, the current operation being completed is displayed within the main viewport, to provide a modicum of feedback to the user. It is worth noting that manual calls to the Draw operation are necessary in order for the textual feedback to be presented to the viewport during the loading operation, as interrupts by the game loop are not serviced during content compilation.

The ability to load and save workspaces is significant, as it allows for scenes to be created for students to operate in. Furthermore, the ability for students to hand in scenes, either for advice or marking, exists, and may be exploited if desired.

4.8.3 Associated Controls

Skybox

The Skybox class is responsible for storing and presenting an environment to the scene. The Skybox uses four vertices to create a surface, which is used to render all six sides of the Skybox cube. First, the appropriate texture is applied to the surface. The surface is then rotated so that the image has the correct orientation based on the user supplied rotations on the Edit Skybox

form. It is then translated and rotated appropriately so that it is in the correct position for the current face of the Skybox. this procedure is repeated for each of the cube faces. The Skybox does not present to a control, as it is typically only part of a scene, and is always the first object drawn to avoid obstruction of scene objects.

SkyboxSave

The SkyboxSave class is used for saving and loading skyboxes. The SkyboxSave stores the name of the Skybox, the location of its textures, and the rotations to be applied to those textures. On loading, the SkyboxSave is used to retrieve the appropriate textures, and use them recreate the saved Skybox.

SkyBoxViewer

The *SkyBoxViewer* inherits from *XNAContent,* and presents the currently contained *Skybox* to a *GamePanel.* The *GamePanel* allows for the viewers camera to be positioned in real time using keyboard input, thus allowing the user to evaluate the Skybox before applying it to the scene. This is achieved in a similar fashion to that of camera input collection within the main game loop.

4.9 Summary

In this section we have considered the functionality and structure of the Transformation Pipeline Visualiser application. Primary focus was given to elaborating on functionality and features, while supplementary implementation information was provided where considered necessary. Consideration was first given the underlying data structures supporting the system, in order to provide insight into the application architecture, and how the system manages transformations and objects. This was followed by a short overview of two controls utilised within multiple sections of the project. The core application functionality was then presented, separated into sections which represent specific goals of the system. These included Transformation Visualisation, Matrix Operation Visualisation, View and Projection Manipulation and Controlling Lighting. Finally, aspects either unrelated or broadly related to other system components were discussed in Synthesis.

As such, the chapter represents an overview of the implementation of the application, and provides insight into its functionality and flexibility. As has been shown, the program provides a relatively broad array of functionality, allowing for the creation of sophisticated and visually compelling scenes, containing a collection of widely varied objects which demonstrate complex, dynamic and interdependent behaviors. The application also affords a great deal of flexibility, both to the learner's approach to the system, and to the educator's. While the application was primarily designed to facilitate exploratory freedom on the part of the learner, the system can be utilised in a far more structured way. For instance, a practical may be set to either create or modify a scene, in order to meet certain specifications. These may be simple or exhaustive, interesting or repetitive, highly specific or vague. Thus, the application can be used for highly structured, semi-structured and unstructured exploration, providing substantial educator flexibility with regard to their application of the program.

Chapter 5

A Sample Lesson

5.1 Overview

In this section we provide a sample practical, intended to demonstrate how the application may be used. The first section of the sample presents a structured problem to be solved incrementally, while the second section presents a somewhat less structured problem, allowing for more exploration on the part of the user. This practical is meant for illustration purposes only, and only utilises a small subset of the applications features.

5.2 Practical

Practical Outcomes:

- Become familiar with skyboxes, and how they are used to create environments.
- Understand how translation, rotation and scale transformations may be used to position objects in a 3D world.
- Create dependencies between objects.
- Learn how lighting can be used to improve a scene.

Introduction:

In this practical, you will incrementally create an animated 3D space scene, and a rotating double helix, using the Transformation Visualiser Application. Models and Skybox textures have been provided to help you achieve this.

Section 1: Space

For Submission:

The workspace file for your completed scene.

Part 1: The Milky Way

The first task is to create the environment in which the scene takes place. You may use the supplied skybox textures to do this. Skyboxes may be created within the Transformation Pipeline Visualiser application by clicking the 'Create Skybox' button in the Skybox Options panel under the main viewport.

Part 2: The Earth

(a) Your next task is to load the model of the earth provided, and position it correctly at the center of the scene. The earth should be tilted by 23 degrees on its axis.

(b) Add a spaceship, using the model provided, to the space scene. Re-size the planet so it is at least five times larger, and position the spaceship high above the equator, with the correct orientation.

Part 3: The Rotation

(a) Add a constant transformation to the earth, to that it rotates on its tilted axis. Assume that 1 second in the program is equivalent to 1 hour in reality.

(b) Without creating any new transformations, rotate the spaceship so that it remains in geosynchronous orbit with the planet, even if the planets rotation changes. The 'Send To' option within the transform element context menu may be useful.

Part 4: The Sun

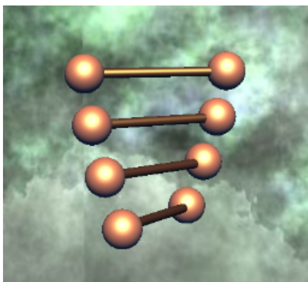
Complete the scene by adding sunlight. Use a directional light to achieve this. Assume that the sun is located infinitely far down the x axis, and has no y or z component.

Section 2: DNA

For Submission:

The workspace file for your completed scene.

The Problem:



Use the Icosphere and Cylinder models to create a rotating double-helix like the one shown above. When implementing this solution, take the following into consideration:

1. All the bars must rotate at the same speed.
2. The speed of rotation must be changeable from a single node.
3. The difference in rotation between successive bars must be variable, and changeable from a single node.

5.3 Solving the Practical

5.3.1 Section 1: Space Solution

The solution to section 1 is relatively trivial, requiring primarily that the user follow the instructions, and correctly order the transformations of the two objects in the scene. In solving this

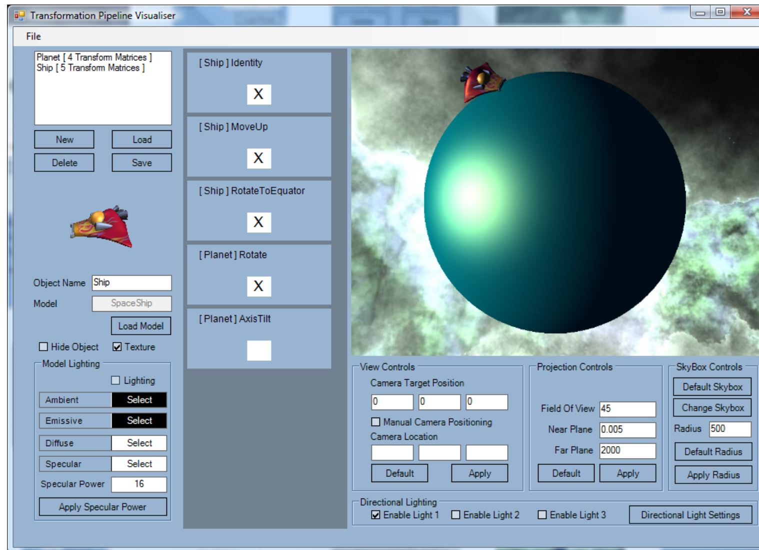
problem, the user will be exposed to the basics of positioning and orientating objects within a scene, introduced to how transformation sharing can be utilised, and encouraged to explore lighting effects. Should the user become confused, the Matrix Operation Visualiser may be used to see how the order of transformations affects the calculation of the World matrix.

5.3.2 Section 2: DNA Solution

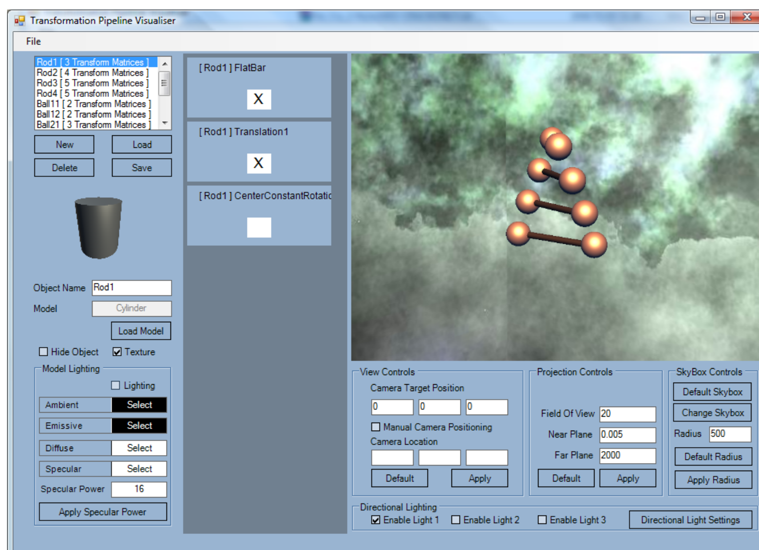
While not a difficult problem, the solution to the DNA section requires a greater degree of independent thinking and problem solving than that of the space section. This problem requires the user to build a structure from several model components, and coordinate them such that they act as a controllable object. The user is also required to apply transformation sharing intelligently, as the solution requires interdependence of objects.

5.4 Summary

In this section, we have presented an example practical demonstrating how the application may be used within a teaching environment. While these examples were quite simplistic, they illustrate two of a number of teaching approaches supported by the program.



(a) Space Solution



(b) DNA Solution

Figure 5.1: Example Practical: Solution Images

Chapter 6

Conclusion

6.1 Overview

In conclusion, we have successfully implemented a relatively full featured, graphical and interactive exploration tool for visualising the XNA transformation pipeline. The application leverages the XNA framework for the transformation visualisation component, while utilising the .NET framework for user input collection, thus allowing for real-time interactive scene representation, managed and interacted with by diverse and feature rich .NET GUI controls. By combining the visual elements of a game, and the structured input of Windows Forms, the application manages to afford the user a substantial degree of exploration initiative, while preventing such initiative from placing the application in an unrecoverable state. Thus, the Transformation Pipeline Visualiser provides a semi-structured and stable environment, which is flexible enough to facilitate a structured, semi-structured or unstructured learning environment, applied per educator preference.

We begin this section with a simple, abstract overview of the available functionality within the system. We then elaborate on interesting and desirable functionality currently not implemented within the application, either due to time or scope constraints, or implementation complexity.

6.2 Available Functionality

For simplicity, the available functionality of the application will be enumerated as a bullet point list. This is intended as an overview of primary functionality only, and thus should not be considered exclusively complete.

- The user can create a wide variety of static and dynamic transformation types, which together provide substantial flexibility in defining sophisticated behaviors for objects within a scene.
- The user can utilise matrix multiplication, addition and subtraction, as well as scalar multiplication, within transformation calculations. This allows the user to explore more sophisticated transformation sequences, while improving their understanding of matrix algebra in general.
- The user can reorder elements, parenthesize calculations and share transformation components between objects. With concrete, real time feedback, the user can experiment with different configurations to better understand transformation sequencing and its importance.
- A wide array of visualisation capabilities, and three levels of pipeline abstraction, provide a complete view of the transformation process for the user – from the graphical results it achieves, to the calculation of individual components of a single matrix in an objects transformation list.
- Strong input management ensures reduced frustration and improved stability, as the user is prevented from undertaking illegal operations and supplying malformed or incompatible input.
- The user can customise environments and load custom models into the application at run time, facilitating an essentially limitless number of visually compelling and diverse scenes, while intelligent content management ensures minimal loading times.
- The user can customise the scene camera in order to visualise view and projection effects on the rendering process.
- The user has control over both object lighting parameters, and three directional lights, which allows for familiarisation with lighting and insight into how such lighting may be exploited to dramatically improve a 3D scene.
- The user can save and load individual objects, environments, and entire scenes for later use.

6.3 Future Work

In this section we consider elements missing from the current implementation which would improve the usefulness of the application. Two types of extension are considered, namely those which fall within the scope of the current application features, either supplementing or improving existing elements, with specific focus on improved usability, and those which fall outside the scope of the current application, adding entirely new features to facilitate broader exploration.

6.3.1 Improvements

Undo and Redo

Currently, the application does not support undo and redo operations, due primarily to the complexity of the application state space, and scope limitations on the project. While undo and redo functionality is highly desirable, the architecture of the application, which is heavily dependent on the maintenance of implicit, unmanaged relationships, makes this a relatively difficult undertaking. In the interests of producing a more feature rich application, focus was placed on the significantly simpler task of ensuring valid input, slightly mitigating the problem. Despite this, implementing Undo and Redo operations remains feasible, and could be achieved, for example, by using an event logger which is capable of undoing each logged event.

Alternatively, a database could be utilised to store checkpoints containing the full application state at a particular point in time. Each time the user undertakes an action, an event is generated representing this action. The current program state is then determined by loading the last saved checkpoint, and applying all the commands collected since the checkpoint was made. After a certain number of operations, a new checkpoint is created, and the procedure begins again. As this method does not require inverses for all operations, it would be considerably easier to implement.

Graphical Feedback for Waypoint Positioning

DynamicTransformNode generation within the Element Builder form is likely the least accessible operation within the Transformation Pipeline Visualiser application, as little graphical feedback is given as to how the supplied node configuration will affect the object it is being applied to. A possible resolution to this problem involves creating a new *XNAContent* based

control, which shows the path the object will take drawn as lines using custom vertices, and an object moving along that path, being translated, rotated and scaled appropriately. This would be both highly desirable, and relatively trivial, to include.

Drag and Drop Support

Drag and Drop support was originally included within all the list based controls in the application, to make position management more fluid and efficient. This support was later removed due to slight inconsistencies in operation under certain conditions. This was done so as to ensure that the primary mode of input was consistent and dependable, and never unpredictable. This was due to the lack of undo and redo support, which necessitated the minimisation of irreversible changes, particularly when these changes were the result of an operation which did not always operate consistently. As a consistent method of manipulation has been fully implemented, the addition of drag and drop support is highly desirable.

Extended Camera Options

Camera targeting is currently quite limited within the application, only allowing for manual specification of static target coordinates. Similarly, camera positioning can only be done manually, somewhat limiting the way a scene can be viewed. In order to improve upon this, a number of possible extensions could be incorporated. For instance, support for tying the camera position or target to a moving object could be incorporated, allowing for the camera to follow an object, or rotate to keep it in view. This may be extended further, by providing the camera with its own transformation list, thus allowing the camera to be treated as a regular scene object. This same principle may be extended to allow for camera target position transformation, greatly improving the flexibility of scene visualisation.

6.3.2 Extensions

Object Grouping

An additional facility to combine a number of meshes, with transformations applied, into a single object is a desirable feature which would improve both usability and scene scalability. Primarily, this would allow for reuse of scene components comprised of multiple loaded meshes,

currently unsupported by the application. This could be achieved by extending the *TransformObject* class to allow it to contain a sub-scene. This sub-scene is then managed as a single, closed entity, but drawn as a collection of objects.

Model Animation Support

Support for animated models would significantly extend the teaching scope of the application. Possible animation related features include:

- A model bone transformation visualisation component which provisions stepping through the bone transformation tree.
- An animation event system, which initiates specific animations at defined points in a transformation.

Point Light Source Support

Currently, the application only supports directional light sources, as these alone are provisioned by the *BasicEffect* class. Extending the lighting features to include point light would allow for more interesting scenes, as well as provide a more diverse understanding of lighting. For instance, user creation of point light sources within a scene may be added, allowing for both colour and light intensity to be specified.

Particle Visualiser

Particle effects are used often within games, and allow for convincing environmental effects such as smoke, fire, rain and snow. A particle system would necessitate a method of allowing the user to specify not only how long a particle should exist, but also how it should look and behave throughout its life.

Global Lighting

The Transformation Pipeline Visualiser application provides only local lighting capabilities, which lights each object individually and in isolation within a scene. Global lighting lights

objects within the context of a scene, allowing for such effects as reflection and drop shadows. Global lighting facilities would improve not only the visual quality of the application, but increase its scope.

Extended Behaviors

Currently, an object's behavior is determined by only the transformation nodes which form its transformation list. While this is satisfactory for our current purposes, adding decisional behavior and event triggers to objects would provision exploration of more diverse and interesting environments. For instance, the application may extended to allow for nodes to act differently based on the state of other nodes, or to change course when a collision is detected.

6.4 Summary

In this chapter we have reviewed the primary functionality of the system, demonstrating the flexibility of the application. We then presented a list of improvements and extensions that would increase both the usability and scope of the application. Essentially, we have shown that the application has met the goals specified, and demonstrates significant potential for extension.

Bibliography

[1] Albahari, J. "Threading in C# Part 3: Using Threads", 2007. URL: http://www.albahari.com/threading/part3.aspx, Last Accessed: September 21st 2008.

[2] Allenstein, B., Yost, A., Wagner, P., and Morrison, J. 2008. *"A query simulation system to illustrate database query execution." In Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA, March 12 - 15, 2008). SIGCSE '08. ACM, New York, NY, 493-497. DOI= http://doi.acm.org/10.1145/1352135.1352301, Last Accessed: 25th of May, 2008

[3] Anonymous, *Ambient Lighting (Direct3D 9).* Microsoft Developer Network (MSDN), Microsoft Corporation, 2008. URL: http://msdn.microsoft.com/en-us/library/bb172256(VS.85).aspx, Last Accessed: 21st October 2008.

[4] Anonymous, *Content Pipeline Architecture.* Microsoft Developer Network (MSDN), Microsoft Corporation, 2008. URL: http://msdn.microsoft.com/en-us/library/bb447745.aspx, Last Accessed: 22nd October 2008.

[5] Anonymous, *Diffuse Lighting (Direct3D 9).* Microsoft Developer Network (MSDN), Microsoft Corporation, 2008. URL: http://msdn.microsoft.com/en-us/library/bb219656(VS.85).aspx, Last Accessed: 21st October 2008.

[6] Anonymous, *Emissive Lighting (Direct3D 9).* Microsoft Developer Network (MSDN), Microsoft Corporation, 2008. URL: http://msdn.microsoft.com/en-us/library/bb173352(VS.85).aspx, Last Accessed: 21st October 2008.

[7] Anonymous, *Projection Transform (Direct3D 9).* Microsoft Developer Network (MSDN), Microsoft Corporation, 2008. URL: http://msdn.microsoft.com/en-us/library/bb147302(VS.85).aspx, Last Accessed 20th of October, 2008.

[8] Anonymous, *Specular Lighting (Direct3D 9).* Microsoft Developer Network (MSDN), Microsoft Corporation, 2008. URL: http://msdn.microsoft.com/en-us/library/bb147399(VS.85).aspx, Last Accessed: 21st October 2008.

[9] Anonymous, *The Direct3D Transformation Pipeline.* Microsoft Developer Network (MSDN), Microsoft Corporation, 2008. URL: http://msdn.microsoft.com/en-us/library/bb206260(VS.85).aspx, Last Accessed 17th of October, 2008.

[10] Anonymous, *View Transform (Direct3D 9).* Microsoft Developer Network (MSDN), Microsoft Corporation, 2008. URL: http://msdn.microsoft.com/en-us/library/bb206342(VS.85).aspx, Last Accessed 20th of October, 2008.

[11] Anonymous, "WinForms Series 1: Graphics Device Sample", XNA Creators Club Online, Microsoft, 2008. URL: http://creators.xna.com/en-us/sample/winforms_series1 , Last Accessed 19th September 2008.

[12] Anonymous, *XNA Game Studio 2.0: Viewports and Frustums.* Microsoft Developer Network (MSDN), Microsoft Corporation, 2008. URL: http://msdn.microsoft.com/en-us/library/bb975157.aspx, Last Accessed 20th of October, 2008.

[13] Cawood, S., McGee, P., *Microsoft XNA Game Studio Creator's Guide.* McGraw-Hill Companies, United States of America, 2007.

[14] Coe, P. S., Williams, L. M., and Ibbett, R. N. 1996. *"An interactive environment for the teaching of computer architecture." In Proceedings of the 1st Conference on integrating Technology into Computer Science Education* (Barcelona, Spain, June 02 - 06, 1996). ITiCSE '96. ACM, New York, NY, 33-35. DOI= http://doi.acm.org/10.1145/237466.237518, Last Accessed: 25th of May, 2008

[15] Conger, D., *Physics Modeling for Game Programmers.* Thompson Course Technology, Boston, United States of America, 2004.

[16] Eberly, D., *Game Physics.* Morgan Kaufmann Publishers, San Fransisco, United States of America, 2004.

[17] Güida, P., *XNA & Beyond: The Path to VS 2008.* The Code Project, 2007. URL = http://www.codeproject.com/KB/game/XNA_And_Beyond.aspx, Last Accessed: 25th of May, 2008

[18] Gurr, H.S., *The AHA In Teaching & Tutoring*. Department of Physical Sciences, University of South Carolina at Aiken, Aiken, SC 29801, 2005. URL = http://www.usca.edu/math/~mathdept/hsg/TeachingTutoring+Pix.html, Last Accessed: 22nd of May, 2008

[19] Hanisch, F. and Straßer, W. 2006. *"Making of an interactive teaching gem." In ACM SIGGRAPH 2006 Educators Program* (Boston, Massachusetts, July 30 - August 03, 2006). SIGGRAPH '06. ACM, New York, NY, 53. DOI= http://doi.acm.org/10.1145/1179295.1179349, Last Accessed: 25th of May, 2008

[20] Hargreaves, S., *Specularity*, Shaun Hargreaves Blog, MSDN Blogs, 2007. URL: http://blogs.msdn.com/shawnhar/archive/2007/04/12/specularity.aspx, Last Accessed: 21st October 2008.

[21] Hargreaves, S., *The Standard Lighting Rig*, Shaun Hargreaves Blog, MSDN Blogs, 2007. URL: http://blogs.msdn.com/shawnhar/archive/2007/04/09/the-standard-lighting-rig.aspx, Last Accessed: 21st October 2008.

[22] Kearsley, G., *Operant Conditioning (B.F. Skinner).* The Theory Into Practice Database, 2008. URL: http://tip.psychology.org/skinner.html, Last Accessed: 25th of May, 2008

[23] Kremenska, A. 2007. *"Measuring student attitudes to computer assisted language learning." In Proceedings of the 2007 international Conference on Computer Systems and Technologies* (Bulgaria, June 14 - 15, 2007). B. Rachev, A. Smrikarov, and D. Dimov, Eds. CompSysTech '07, vol. 285. ACM, New York, NY, 1-6. DOI= http://doi.acm.org/10.1145/1330598.1330677, Last Accessed: 25th of May, 2008

[24] Landry, N., *Microsoft XNA: Ready for Prime Time?* CoDe Magazine, 2007. URL: http://www.code-magazine.com/article.aspx?quickid=0709041&page=1, Last Accessed: 25th of May, 2008

[25] McNeil, S., *A brief history of instructional design.* The Instructional Technology Program, Department of Curriculum and Instruction, College of Education, University of Houston, 2008. URL: http://www.coe.uh.edu/courses/cuin6373/idhistory/index.html, Last Accessed: 25th of May, 2008

[26] Naiman, A. C. 1996. *"Interactive teaching modules for computer graphics." SIGGRAPH Comput. Graph.* 30, 3 (Aug. 1996), 33-35. DOI= http://doi.acm.org/10.1145/232301.232330, Last Accessed: 25th of May, 2008

[27] Nicholas, L., *Introduction to Psychology.* UCT Press, Landsdowne, South Africa, 2003.

[28] Rodger, S. H. 1995. *"An interactive lecture approach to teaching computer science." SIGCSE Bull.* 27, 1 (Mar. 1995), 278-282. DOI= http://doi.acm.org/10.1145/199691.199820, Last Accessed: 25th of May, 2008

[29] Schneider, D.K., *Computer-based training.* EduTech Wiki, 2006. URL: http://edutechwiki.unige.ch/en/Computer-based_training, Last Accessed: 25th of May, 2008

[30] Schneider, D.K., *Programmed Instruction.* EduTech Wiki, 2007. URL: http://edutechwiki.unige.ch/en/Programmed_instruction, Last Accessed: 25th of May, 2008

[31] Snyder, K. 2002. *"The use of interactive learning modules for the teaching of undergraduate curriculum." J. Comput. Small Coll.* 17, 3 (Feb. 2002), 203-208., URL: http://portal.acm.org/citation.cfm?id=772669&coll=ACM&dl=ACM&CFID=12487090&CFTOKEN Last Accessed: 25th of May, 2008

[32] Stahler, W., Clingman, D., Kahrizi, K., *Beginning Math and Physics for Game Programmers.* New Riders, 2004.

[33] Stern, L. and Sterling, L. 1996. *"Teaching AI algorithms using animations reinforced by interactive exercises." In Proceedings of the 2nd Australasian Conference on Computer Science Education* (The Univ. of Melbourne, Australia). ACSE '97, vol. 2. ACM, New York, NY, 78-83. DOI= http://doi.acm.org/10.1145/299359.299372, Last Accessed: 25th of May, 2008

[34] Subramanian, K. R. and Cassen, T. 2008. *"A cross-domain visual learning engine for interactive generation of instructional materials." SIGCSE Bull.* 40, 1 (Feb. 2008), 488-492. DOI= http://doi.acm.org/10.1145/1352322.1352300, Last Accessed: 25th of May, 2008

[35] Syrjakow, M., Berdux, J., and Szczerbicka, H. 2000. *"Interactive Web-based animations for teaching and learning." In Proceedings of the 32nd Conference on Winter Simulation* (Orlando, Florida, December 10 - 13, 2000). Winter Simulation Conference. Society for Computer Simulation International, San Diego, CA, 1651-1659. URL: http://portal.acm.org/citation.cfm?id=510620&coll=ACM&dl=ACM&CFID=12487090&CFTOKEN Last Accessed: 25th of May, 2008

[36] Tran, Q. 2006. *Interactive computer algebra software for teaching and helping students to study foundations of computer science.* J. Comput. Small Coll. 22, 1 (Oct. 2006), 131-143. URL: http://portal.acm.org/citation.cfm?id=1181831&coll=ACM&dl=ACM&CFID=12487090&CFTOKEN Last Accessed: 25th of May, 2008

[37] Tremblay, C., *Mathematics for Game Developers.* Thompson Course Technology, Boston, United States of America, 2004.

[38] Van Verth, J.M., Bishop, L.M., *Essential Mathematics for Games and Interactive Applications.* Morgan Kaufmann, 2008.

[39] Void, S., *Quaternion Powers.* GameDev.Net, 2003. URL: http://www.gamedev.net/reference/articles/article1095.asp, Last Accessed: October 27th, 2008.

[40] Witters, K., “XNA Game Loop Basics”, 27 July 2007. http://www.nuclex.org/articles/xna-game-loop-basics

[41] ”XNA Framework GameEngine Development Part 15: Adding WinForm Support”, blog, 29th February 2008. http://roecode.wordpress.com/2008/02/29/xna-framework-gameengine-development-part-15-adding-winform-support-step-1-to-world-builder/