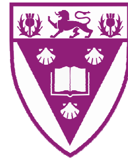
3D Logic Games and Tutorials for BingBee



RHODES UNIVERSITY
Where leaders learn

Principal Investigator: Bwini Mudimba
Supervisor: Prof Peter Wentworth

Submitted in partial fulfilment of the requirements for Bachelor of Science (Honours) degree at Rhodes University

3 November 2008

ABSTRACT

Games provide a forum in which learning arises as a result of tasks stimulated by the content of the games, knowledge is developed through the content of the game, and skills are developed as a result of playing the game [15]. There are various types of games, that is, board games, card games, puzzles, and other types of non computerized and computerized games that children play. Besides gaining logic skills like critical thinking, problem solving and so on, children also learn how to use a computer by playing computer games. It is asserted that the use of such games can stimulate the enjoyment, motivation and engagement of users, aiding recall and information retrieval, and can also encourage the development of various social and cognitive skills [20]. 3D games have been shown to be very popular with game players but there is only one on BingBee. Introducing 3D Noughts and Crosses (implemented using XNA Game Studio) will help introduce more 3D gaming on BingBee with the prospect of adding more challenging 3D games with time to further increase the skills that children acquire when they play the games.

ACKNOWLEDGEMENTS

Firstly, I would like to thank the rock of my salvation, Jesus Christ for seeing this work unto its completion. He was my strength especially through the personal trials I encountered in the second semester. Thank you Lord.

I also acknowledge the patience and dedication of my supervisor, Prof Wentworth without whom this project would not have been successful. Thank you for your guidance and exceptional programming skills. Mr John Ebden, your help and guidance is also acknowledged.

I also acknowledge the Telkom Centre of excellence (CoE) at Rhodes who secured funding for me to study at Rhodes this year. Thank you.

A big thank you also goes to my family and friends who were there for me throughout this year. Special mention goes to Peter Nkomo, Sinini Ncube and Flora Tasse. Thank you for your love and support.

I also acknowledge the financial and technical support of this project of Telkom SA, Comverse SA, OpenVoice, Stortech, Tellabs, Amatole, Mars Technologies, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence at Rhodes University.

Table of Contents

Chapter 1: Introduction……………………………………………………………….8

1.1 Background…………………………………………………………...8

1.2 Motivation for research……………………………………………......10

1.3 Project aims…………………………………………………………...11

1.4 Thesis structure………………………………………………………11

Chapter 2: Related work………………………………………………………………12

2.1 Microsoft XNA framework…………………………………………..12

2.1.1 Development…………………………………………………………..12

2.1.2 Microsoft XNA Game Studio…………………………………………13

2.1.3 The XNA Framework Content Pipeline………………………………14

2.1.3.1 Content Importer……………………………………………………14

2.1.3.2 Content DOM…………………………………………...............15

2.1.3.3 Content Processor……………………………………………………15

2.1.3.4 Content Compiler……………………………………………………16

2.1.3.5 Content Manager……………………………………………………16

2.2 Noughts and Crosses…………………………………………………16

2.3 Summary………………………………………………………………18

Chapter 3: Design and Implementation………………………………………………20

3.1 How an XNA game works………………………………………………20

3.2 3D graphics in XNA…………………………………………………..22

3.2.1 Matrices……………………………………………………………...27

3.2.2 Transformations…………………………………………………………..28

3.2.2.1 Translation…………………………………………………………...28

3.2.2.2 Rotation…………………………………………………………..29

3.2.2.3 Scaling……………………………………………………………….29

3.3 3D Noughts and Crosses……………………………………………...30

3.3.1 3*3*3 matrix cube…………………………………………………..30

3.3.2 Detecting that a cubelet has been clicked……………………………...31

3.3.3 Importing 3D models…………………………………………………..33

3.3.4 Adding sound……………………………………………………….....34

3.3.5 Animation…………………………………………………………...35

3.4 Game Classes……………………………………………………….36

3.5 Summary……………………………………………………………37

Chapter 4: Results…………………………………………………………………….38

4.1 3D Noughts and Crosses…………………………………………………38

4.1.1 Version 1…………………………………………………………….39

4.1.2 Version 2…………………………………………………………….39

4.1.3 Version 3…………………………………………………………….39

4.2 First User Interface………………………………………………………39

4.3 Second User Interface……………………………………………………41

4.4 Summary…………………………………………………………....43

Chapter 5: Conclusion…………………………………………………………….. 45

5.1 Challenges faced in the project ……………………………………….45

5.2 Possible project extensions ……………………………………………46

5.3 Project achievements …………………………………………………….47

References…………………………………………………………………………….48

Appendix……………………………………………………………………………51

Table of Figures

Figure 2.1: Importers that are used in XNA Game Studio……………………….........15

Figure 2.2: Shows how the XNA framework content pipeline works……………………16

Figure 2.3: Standard 2D Noughts and Crosses…………………………………………...17

Figure 2.4: 3D Noughts and Crosses by using three 2D layers…………………….........17

Figure 2.5: 3D Noughts and Crosses by using a 3*3*3 matrix of cubelets………........18

Figure 3.1: A flow diagram showing the XNA game loop……………………………….22

Figure 3.2: Shows a wire frame 3D model……………………………………….........23

Figure 3.3: Shows a textured 3D model………………………………………………..24

Figure 3.4: Shows the right hand orientation ……………………………………………..25

Figure 3.5: Shows triangle strips and triangle lists ………………………………........25

Figure 3.6: Shows the normal vector on a triangle ……………………...……………..26

Figure 3.7: Showing the bounding sphere of a 3D model ………………………………..31

Figure 3.8: Shows the model with the cubelet under the mouse highlighted …………….32

Figure 3.9: Shows the model after a cubelet is clicked, the computer has made another move, and the mouse has now moved over another cubelet……………..........33

Figure 3.10: The "Ducky" model…………………………………………………………34

Figure 3.11: The "Sonic" model…………………………………………………………..34

Figure 3.12: The MVC design pattern…………………………………………………….37

Figure 4.1: BingBee touchpad…………………………………………………………….41

Figure 4.2: The second user interface for Noughts and Crosses………………………….42

Figure 4.3: Shows the 3*3*3 matrix cube after some cubelets have been played………...43

Table of Figures

Table 1.1: The elements that make computer games engaging ………………….............9

CHAPTER 1: INTRODUCTION

1.1 Background

To get a good understanding of what computer games are, it is important to first define a few terms. According to Prensky [23], play is something one chooses to do as a source of pleasure, which is intensely and utterly absorbing and promotes the formation of social groupings, play also increases our involvement, which also helps us learn.

A game on the other hand is a set of activities involving one or more players. It has goals, constraints, payoffs and consequences. A game is rule-guided and artificial in some respects. Finally, a game involves some aspect of competition, even if that competition is with oneself [6].

Prensky [23] states that computer games can be characterised by six key structural elements which, when combined together, strongly engage the player. These elements are:

- rules
- conflict or competition or challenge or opposition
- goals and objectives
- interaction
- outcomes and feedback
- representation or story.

There are various types of computer games and these include action games, adventure games, fighting games, platformers (where game characters run and jump along and onto platforms), knowledge games, simulation or modelling or role-playing games such as management and strategy games, drill-and-practice games, logical games and maths games [11]. Game play can be competitive, cooperative or individualistic [3].

Video games on the other hand differ from computer games in that they include an interactive virtual playing environment and the player has to struggle against certain opposition.

BingBee has about 40 activities ranging from logic games and puzzles, drill practise for Xhosa/English vocabulary, videos, music, school plays and presentations from local artists.

BingBee [26] (website has details on what BingBee is, its purpose, the activities that it contains and the people running BingBee) is an information kiosk designed to improve literacy and numeracy skills in children through entertainment. BingBee currently targets

children between the ages of seven to fourteen years. There are various hardware systems that can be used to deliver games and these include:

- mobile phones
- handheld devices for example Nintendo Wii
- game consoles for example Microsoft Xbox, Sony PlayStation
- personal computers

BingBee makes use of personal computers as a delivery system for the various activities and computer games.

The major reason why computer games are used for learning is that they are engaging, that is, they hold one's attention. Some of the characteristics of a game that make it engaging are, networking (multiplayer games), varying levels of difficulty, speed and graphic representation. Prensky [23] also identified twelve elements (Table 1) that make a game more engaging.

Characteristics of the computer Game	**How characteristics contribute to players' engagement**
Fun	Enjoyment and pleasure
Play	Intense and passionate involvement
Rules	Structure
Goals	Motivation
Interaction	Doing (that is the activity)
Outcomes and feedback	Learning
Adaptive	Flow
Winning	Ego gratification
Conflict/competition/challenge and opposition	Adrenaline
Problem solving	Sparks creativity
Interaction	Social groups
Representation and a story	Emotion

Table 1.1: The elements that make computer games engaging

Gee [11] argues that good computer games are not just entertainment but incorporate as many as 36 important learning principles. Taking as long as 100 hours to win, some are very difficult and they encourage the player to try different ways of learning and thinking, which can be experienced as both frustrating and life-enhancing.

1.2 Motivation for research

One of the arguments for making use of games in education is that they grab and hold the attention of the player. It is therefore paramount for game developers to design and implement games that will attract and hold the players' attention.
Solving problems, accomplishing tasks and making decisions lie at the heart of any game. Games provide a type of learning environment that can help to foster general learning applicable in a wide range of problem-solving situations. Frequent gaming is also said to help users to adjust to a computer-oriented society [11].

Graphics, colour and audio effects are characteristics that players like in a game, so quality and design is important [6]. According to research carried out at the University of Natal by Amory et al [1], students prefer three dimensional (3D) adventure and strategy games to other types of games. 3D games are more engaging because they seem more life-like, that is, like a real world object compared to 2D games which have a flat structure. Another factor that makes 3D games more worthwhile is play control, whereby the player is able to freely move around the world. Currently BingBee has one 3D logic game called DropBlocks which was implemented using Microsoft DirectX 3D technologies. The popularity of DropBlocks on BingBee also reinforces the extent to which players find 3D games more appealing. This project therefore seeks to delve into the graphics component of games by increasing the 3D content on BingBee to pave way for increasingly rich 3D games that are more fun, and perhaps more challenging.

In games, learners ask for novelty, surprise and humour and instructions which are clear and concise. The nature of performance feedback and remediation is also very important [13]. Players also want the game to be fast and challenging; a time element should be incorporated in graphic form [22]. They want the game to become more difficult as they improve. Educational software is typically disliked by students "because the fun factor is missing"

[14]. To develop educational software which maintains the fun factor, it is important to take into consideration the above elements that the players like in a game.
This project was also motivated by the emergence of Microsoft XNA Game Studio as an allegedly “better and easier” game development platform, thus the games in this project were developed using the XNA development platform.

1.3 Project aims

The aim of the project is to increase 3D content on BingBee by adding a 3D game and possibly paving the way for the future addition of richer 3D games that simulate the environment. A new 3D logic game, namely 3D Noughts and Crosses was implemented using Microsoft XNA Game Studio. The intended approach was to take advantage of the popularity of Noughts and Crosses and implement its 3D version to familiarise the players with manipulating an object in 3D.

Microsoft XNA Game Studio is a fairly new game development platform which has not been used before for developing games hosted on BingBee. This project therefore also seeks to evaluate the usefulness or ease of using XNA in developing educational software.

1.4 Thesis structure

Chapter 2 outlines the development of XNA Game Studio and its emergence. The features of XNA Game Studio that put it at an advantage over other game development platforms are also outlined here. This chapter ends by discussing the origin and educational value of the game developed in this project. Chapter 3 delves into the design and implementation of the game using XNA Game Studio. Chapter 4 outlines the results from the game development and the conclusion and possible project extensions for the future are discussed in chapter 5.

CHAPTER 2: RELATED WORK

2.1 MICROSOFT XNA FRAMEWORK

2.1.1 DEVELOPMENT

Over the years Microsoft has continually improved the tools used for game programming starting with the release of the Windows 95 operating system. This was a great improvement as it cut down on the tedious low-level coding that was done in DOS. However Windows still had shortfalls because there was no way to get down to hardware level to do things that required a lot of speed and Windows had a protected memory model that kept developers from directly accessing the low-level interrupts of the hardware. To solve this, Microsoft created a technology called DirectX. DirectX enabled developers to write games with one source that would work on all PCs regardless of their hardware [5].

Starting with DirectX 1.0, Microsoft released a series of upgrades to DirectX 10. As part of the upgrades Microsoft introduced Direct3D which allows developers to create 3D objects inside 3D worlds. When there was no graphics hardware, games were slow, but they were very flexible. Later, as hardware rendering became prominent, the games were faster, but they were not very flexible in that all of the games really started to look the same. Now with shaders, the speed of the hardware is combined with the flexibility for each game to render and light its 3D content differently. DirectX 10 was released at the same time as Microsoft Windows Vista. In fact, DirectX 10 only works on Vista and requires Shader Model 4.0 hardware. DirectX 9 is the foundation for Managed DirectX, an API that exposed the core DirectX functionality to .NET Framework developers and it requires Shader Model 3.0. The XNA Framework uses DirectX 9 in the background and developers used Managed DirectX as a launching pad although XNA was not built on top of Managed DirectX. The XNA Framework is a set of managed libraries based on the Microsoft .NET Framework 2.0 that are designed for game development. A series of XNA integrated development environments (IDEs) have been released starting with Microsoft XNA Game Studio Express and the latest being Microsoft XNA Game Studio 2.0. A beta version of XNA Game Studio 3.0 was

released in September 2008 and it is mainly targeting the Zune platform. Zune (a competitor for Apple's iPod) is a brand of digital media players and services sold by Microsoft.

2.1.2 MICROSOFT XNA GAME STUDIO

Initially, XNA was available in two versions, that is, XNA Game Studio Express which was free for amateurs, and XNA Game Studio Professional for professional game developers. These two platforms were merged into XNA Game Studio 2.0 to make only one tool available to everyone, that is, students, hobbyists and professionals. In this project we used Microsoft XNA Game Studio 2.0 which is an integrated development environment (IDE) to develop our XNA games. Microsoft XNA Game Studio is a set of tools that plugs into supported versions of Microsoft Visual Studio tools that allow students and hobbyists to build games for both Microsoft Windows and Xbox 360. XNA Game Studio 2.0 in particular can be used with any Visual Studio 2005 products. The XNA framework provides support for both 2D and 3D games and it has the following useful functionalities:

- Easy access to the input devices (keyboard, game pad or controller, mouse) [24]. All three controller types are provided in the framework for Windows games hosted on a PC, while only the Gamepad is available for Xbox games. The developer does not have to worry about acquiring or releasing a device, all they do is call `GetState` on the appropriate controller type. This is useful for this project because the keyboard and mouse are used as an interface between the player and the game, for example using the arrow keys on the keyboard to rotate the 3*3*3 matrix cube in the game so that all faces can be viewed.
- Easy access to the graphics hardware.
- Easy control of audio using Microsoft Cross-Platform Audio Creation Tool (XACT). This was valuable in adding sounds to the game to make it more engaging and interesting to the target group (children between the ages of 7-14 years).
- Provides the ability for us to store information like high scores and even saved games.
- Networking capability which provides multiplayer functionality.
- Math API which provides types that are often used in game programming, for example, Vector2, Vector3, Vector4, Matrix, Plane and Ray. The Ray is particularly useful for determining if a 3D object has been selected which is an important aspect

of the first user interface in the game. The Math API also provides bounding volume types like the BoundingSphere, BoundingBox and BoundingFrustrum. This reduces the workload on the game developer because they do not have to create the bounding volume from scratch and it is also useful in selecting a 3D model [24].

Another useful feature of XNA Game Studio is the XNA framework content pipeline which handles all the content that the game uses.

2.1.3 THE XNA FRAMEWORK CONTENT PIPELINE

Content is a large part of a game and getting content into a game is not a very easy task. There are so many difficulties that one faces, for example, getting the right tools to export or import, process and display content in the game. The XNA framework content pipeline makes this work a lot easier. It is simpler to use; it's highly extensible and customizable to what you're doing; and it gives you choices when putting your game together, both in the tools you use to create content and in the engine you use [13].

With the XNA Framework Content Pipeline, the content is managed inside Visual Studio and it is added just as one would add code files. The content pipeline is one of the components of XNA Game Studio that make development easier. The XNA Framework content pipeline is made up of different components. The bulk of the work – importing the fonts, textures, 3D models and sound files is all done at compile time, and is compiled down to an internal format that can be loaded at run time by the Content Manager. The main content pipeline components are:

2.1.3.1 Content Importer

An importer is used, at compile time, when you begin to add data to your game. It is responsible for taking data and normalizing it. For example, it manages which way the content is facing and which way is up in the content creation tool. The importer then takes files that have been saved from your content creation tools and imports them into Visual C#. The importers found in XNA Game Studio include Fbx (.fbx), effect (.fx), fontdescription (.spritefont), texture (.bmp, .jpg, .tga), X (.x), XACT (.xap) and Xml [13]. Figure 2.1 shows some of the importers that exist in XNA Game Studio.

3D File Formats	2D File Formats	Material File Formats	Audio File Formats
.FBX .X	.DDS .BMP .JPG .PNG .TGA	.FX	.XAP (XACT)

Figure 2.1: Importers that are used in XNA Game Studio

2.1.3.2 Content document object model (DOM)

The job of the importer is to build an object of type DOM. The data in the content DOM is represented in a consistent internal format, that is, a series of vertices or textures will look the same regardless of the type of file they came from [13].

2.1.3.3 Content Processor

A processor takes data from the content DOM and "compiles" it to the objects which will be used at run time. This object can be as simple as a model, or as complex as multiple processors for your game. Some examples of available processors include `Model` (for simple objects with textures), `FontTexture` (for sprites), `FontDescription` (converts a spritefont file into font) and `Effect` (for handling the Model's materials). The `Model` processor makes it possible to use readymade objects though the flexibility to create your own objects from points and lines is still available. In our 3D Noughts and Crosses game described in this document, there are three kinds of 3D objects in use: two external models called *Ducky* and *Sonic* represent the player and the computer's pieces, and an array of cubelets represents the places where the players can play (Figure 2.5). The `Model` processor was used in 3D Noughts and Crosses at compile time to prepare the Ducky and Sonic objects for the game. The cubelets on the other hand were created at run time from points and vertices.

2.1.3.4 Content Compiler

This component manages content building. When you click Build, all the output from the known content will be built, saved to disk, and ready to be used at run time. The compiled files have an extension .xnb. Incremental building is used, that is, when a change is made in the game, only the items using the changed object will be rebuilt [13].

2.1.3.5 Content Manager

This is the run-time component that loads all the assets into the game (often from the .xnb files that were created at compile time), and ensures that all associated assets are also loaded. (Some .xnb files can reference external textures, for example.). Below is code that shows how a content manager works:

```
ContentManager myManager = new ContentManager (GameServices);
model = myManager. Load<Model> ("ship");
```

The above components of the XNA framework content pipeline work together as illustrated in Figure 2.2 below.

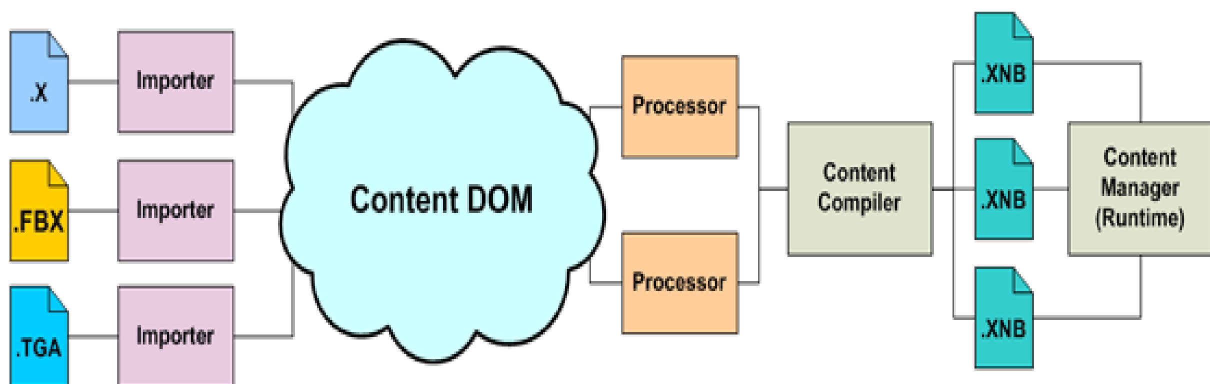


Figure 2.2: Shows how the XNA framework content pipeline works

2.2 NOUGHTS AND CROSSES

Noughts and Crosses, also called Tic-tac-toe and many other names, is a game for two players, who take turns marking the spaces in a 3×3 grid [25]. The player who succeeds in placing three respective marks in a horizontal, vertical or diagonal row wins the game. Currently BingBee has a popular game of Noughts and Crosses played on 2D squares. In BingBee this game is often the first one used by newcomers, and so has a role of

familiarizing them with the system and its user interaction. This project will implement an extended 3D version of the game in a 3*3*3 matrix cube. The player will be able to navigate to the various facets of the cube by using arrow keys or other inputs to rotate and manipulate the cube. Our objective is to leverage the user's familiarity with the 2D game, so that the extended 3D version becomes an easy way to get them familiar with manipulating an object in the 3D world. To win in 3D Noughts and Crosses the player has to complete a row either horizontally, vertically, diagonally or through the cube. Many developers, including Jeremiah McLeod [16], have modified the standard Noughts and Crosses [Figure 2.3] and implemented it on 3 layers of 2D squares [Figure 2.4]. The difference with our approach is that we are implementing Noughts and Crosses on a solid 3*3*3 matrix cube which is a 3D model [Figure 2.5] and using 3D models in place of the conventional X and O.

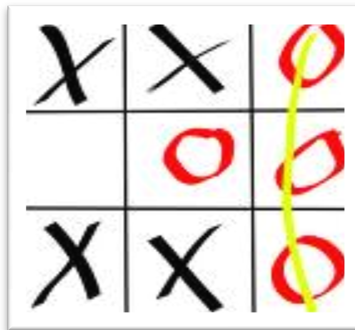


Figure 2.3: Standard 2D Noughts and Crosses

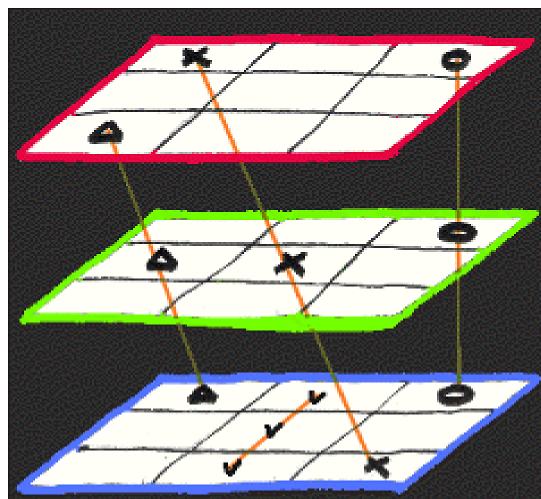


Figure 2.4: 3D Noughts and Crosses by using three 2D layers, as often done by others

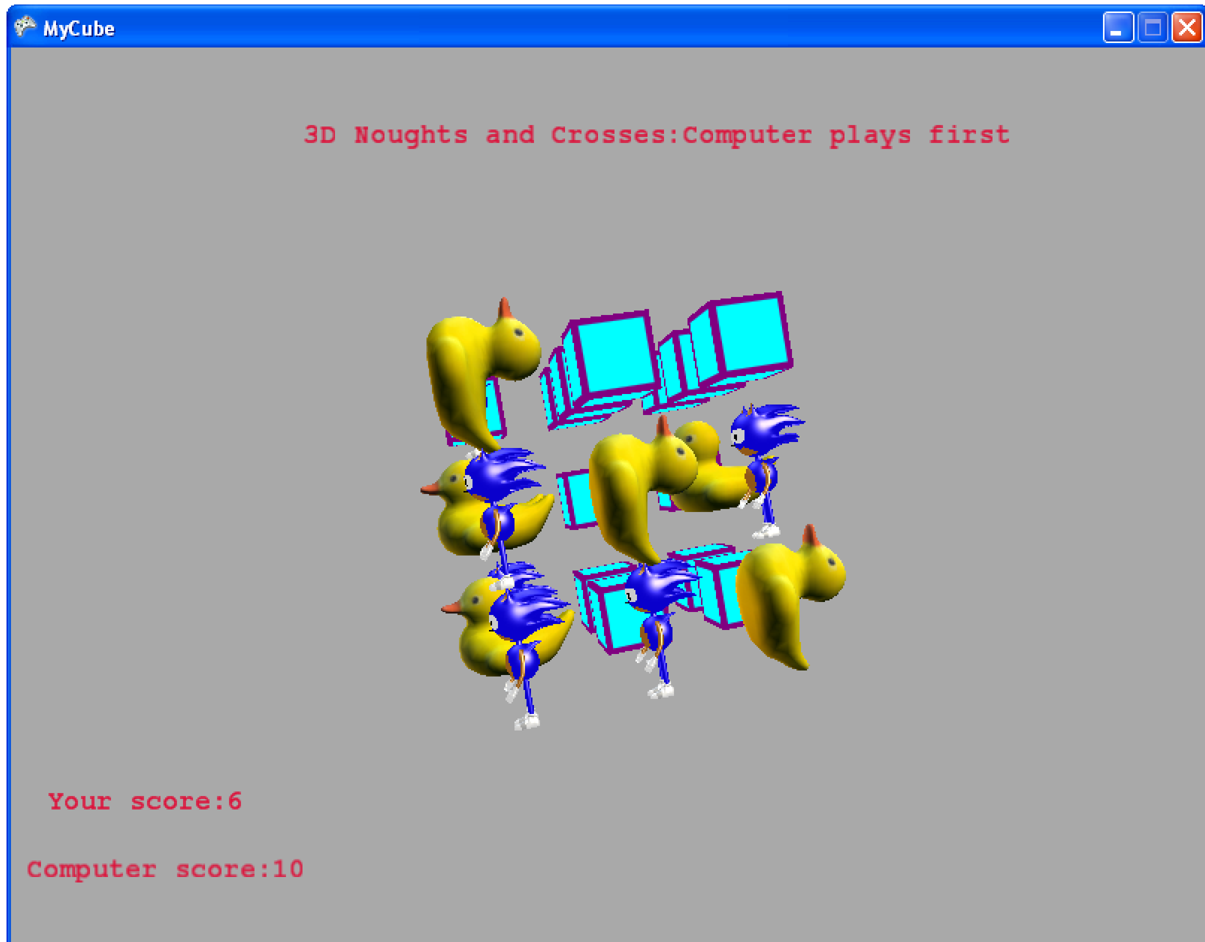


Figure 2.5: Our 3D Noughts and Crosses by using a 3*3*3 matrix of cubelets

2.3 Summary

This chapter outlines some aspects of the XNA framework, and the features that it has that make XNA Game Studio an easier game development tool. The aim of designing XNA Game Studio was to reduce the difficulties and inefficiencies that game developers had to deal with when developing games. XNA Game Studio is therefore a game development platform that makes game programming easier because it provides useful classes that reduce the amount of work that the developer has to do. One of these classes is the XNA Framework Content Pipeline which is responsible for handling all the content in the game project. The content pipeline comes with built in importers, processors and a compiler which handle the content therefore giving the game developer an opportunity to develop the actual game without having to spend too much time fussing about how the content is handled behind the scenes.

The other useful features that come with XNA which were particularly useful in developing 3D Noughts and Crosses include:

- the Math API
- the Graphics API,
- the input API
- the sound API.

The XNA features, therefore cut down on the workload carried by the game programmer, thus making the game development process easier. Another useful feature of XNA Game Studio is that it comes with a lot of tutorials that take a beginner through step by step procedures on how to create a game using XNA.

CHAPTER 3: DESIGN AND IMPLEMENTATION

3.1 How an XNA game works

The XNA Framework's logic flow [4] works like this:

1. The Main application calls the Game Constructor.
2. The Game Constructor will create any game components and call their constructors.
3. The XNA Framework calls the game's `Initialize` method.
4. The XNA Framework calls each of the game component's `Initialize` methods.
5. The XNA Framework calls each of the `Drawable` game component's `LoadGraphicsContent` methods.
6. The XNA Framework calls the game's `LoadGraphicsContent` method.
7. The XNA Framework calls the game's `Update` method.
8. The XNA Framework calls each of the game component's `Update` methods.
9. The XNA Framework calls the game's `Draw` method.
10. The XNA Framework calls each of the `Drawable` game component's `Draw` methods.
11. Steps 7 through 10 are repeated many times each second, as fast as the computer allows.
12. If the device is lost (user moved the window to another monitor, screen resolution is changed, window is minimized, etc.), then a call to `UnloadGraphicsContent` is made. (This is automatically handled by the XNA Framework)
13. If the device is reset then we start back at step 6 again.
14. The gamer exits the game.
15. The XNA Framework calls the game's `Dispose` method
16. The game's `Dispose` method calls the base object's `Dispose` method
17. The XNA Framework calls each of the game component's `Dispose` methods.
18. The XNA Framework calls the game's `UnloadGraphicsContent` method.
19. The game's `Dispose` method gets focus back and the game exits. [4]

The game loop [9] is the heartbeat of every game and it is a series of steps that happen while the game is running regardless of what the player is doing. The game loop alternates between the `Update` and `Draw` methods until the player ends the game.

Because computers differ in speed, the XNA framework has an inter-frame timer which is passed to the Update call (step 7 above). The programmer can use the elapsed time between

frames to scale movement or action in the game. With care, the programmer can ensure that in a game like a car-racing game, the speed of the car appears the same on a fast machine or a slow machine: the faster machine simply renders more frames per second, and the action looks smoother.

The algorithm below shows a simple outline or skeleton of an XNA game with the game loop.

```
Initialize ()
LoadGraphicsContent ()
while (stillPlaying)
{
     Update ()
     Draw ()
}
UnloadGraphicsContent ()
```

The `Initialise` method starts the game and this is followed by loading the graphics resources that are needed in the game. While the game is still running, the `Update` and `Draw` methods are called simultaneously. The `Update` method as the name suggests updates the game, that is, it is called when the game determines that there is game logic to be processed. Examples of game logic include, getting player input, rotating objects on the screen, checking for collisions and making decisions (is there any winner or is the game over?) based on the previous three examples. The `Draw` method handles the drawing or rendering for the game program. When the game stops running, the graphics resources are unloaded before the game is closed. Figure 3.1 shows a clear illustration of the game loop.

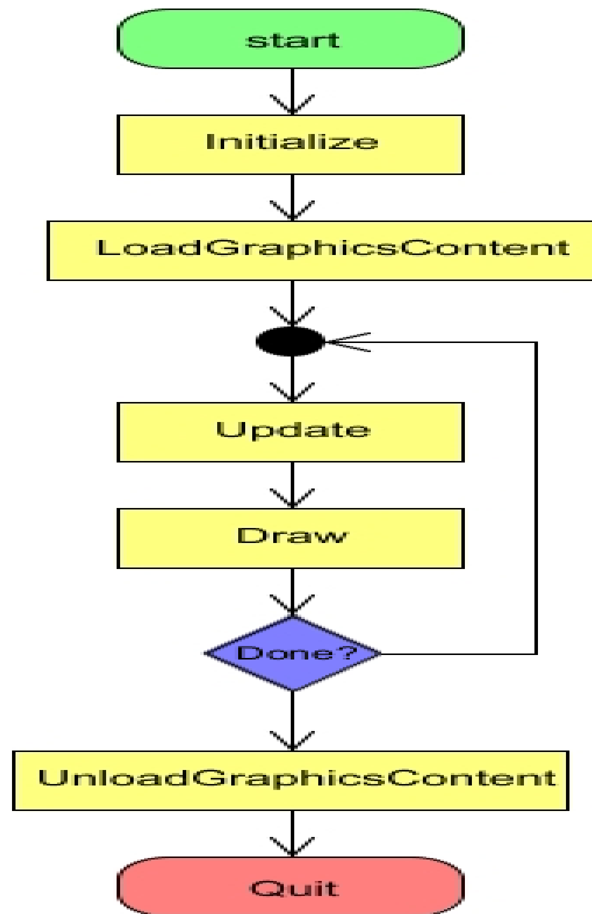


Figure 3.1: Flow diagram showing the XNA game loop

The unloading of the graphics content is an important step that is also handled automatically by the XNA framework. Generally, the games make use of advanced graphics hardware and processors. The graphics hardware has its own memory and resource constraints. Loading models and textures into the game often involves sending the points or the textures from the CPU to the Graphics hardware device, where they are available during hardware-accelerated rendering. It is important to free these resources on the graphics hardware when the game ends.

3.2 3D graphics in XNA

3D graphics can be a great challenge especially to a beginner in 3D graphics programming, but once one understands the geometry it becomes easier. The first challenge is drawing 3D primitive shapes like points, lines and triangles. This challenge can however be overcome by simply importing a complete textured model from a tool like Blender, for example. This is

made possible by the Model processor in the XNA framework content pipeline. However, there are times when a game programmer needs to construct the model from primitives. This approach is also supported in XNA Game Studio. Everything in a 3D game is represented by 3D points. XNA stores these points or vertices in vectors. XNA provides three different vector structures - `Vector2`, `Vector3`, and `Vector4`. `Vector2` only has an x and y component and this 2D vector is used in 2D games and when using a texture. `Vector3` which adds in the z component is used to represent the primitive points in a 3D game. In this project we will use `Vector3` to store coordinates for our points. These points make up triangles which are a basis for all 3D models (Figure 3.2) [5].

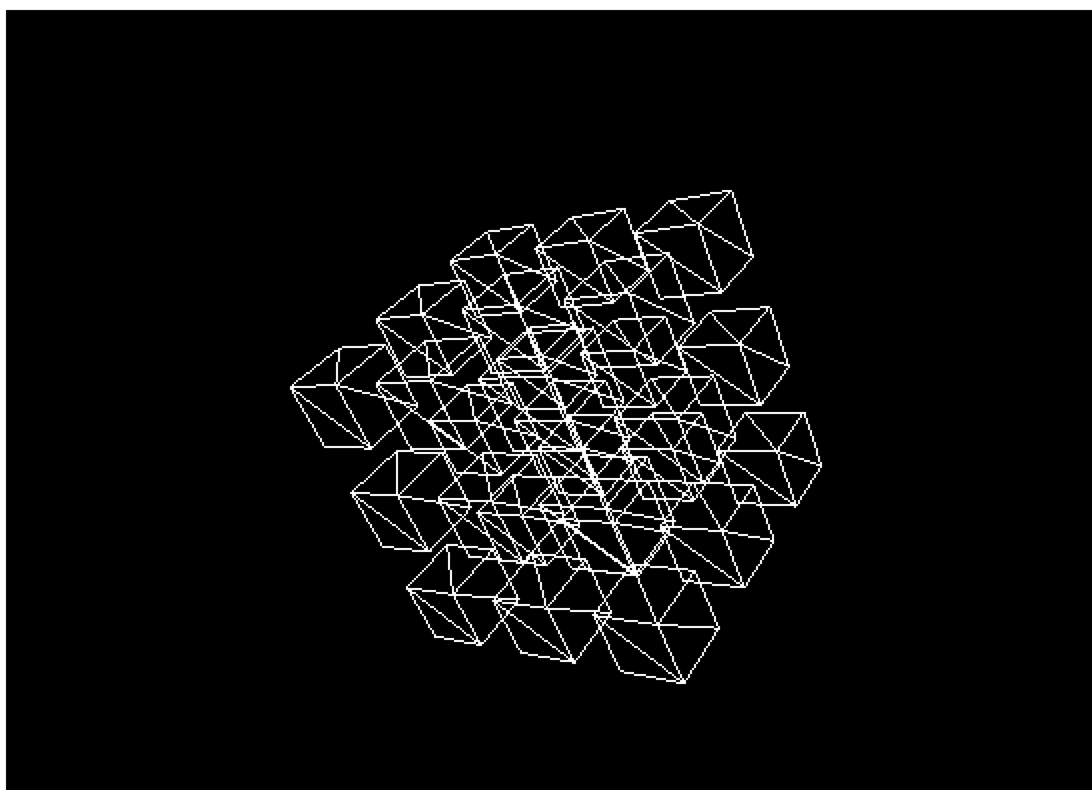


Figure 3.2: Shows a wire frame 3D model

To make the 3D model look solid, we have to fill the triangles in the wire frame model. This can be done with solid or blended colours, or we can add pictures to the wire frame model, that is, texture the model (Figure 3.3). Textures are images applied to the surfaces of primitive objects. Without textures, especially in a 3D game, the models appear less interesting and engaging. Texturing 3D models make them look more like real world objects. `Vector2` coordinates are used for texturing because a texture is a two-dimensional object.

The points on a three-dimensional triangle are mapped onto corresponding positions on the texture. To learn these fundamentals, a tutorial is available on the XNA Game Studio documentation [19] and it's entitled "How to: Draw points, lines and other 3D primitives".

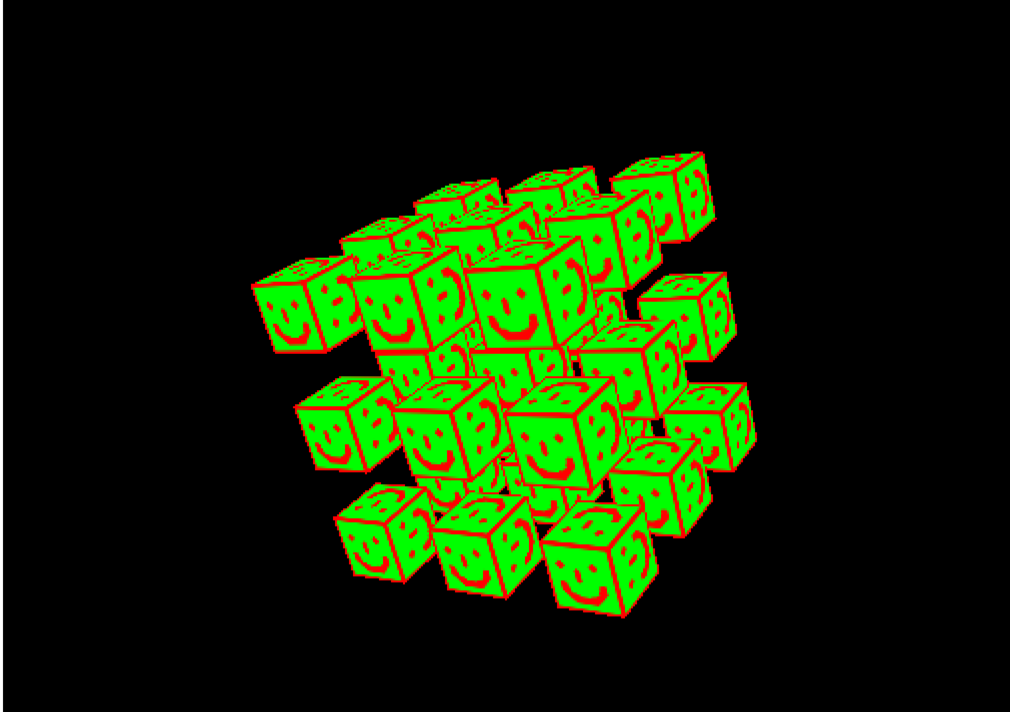


Figure 3.3: Shows a textured 3D model

XNA uses a right-handed coordinate system [4] where the x axis is aligned with the thumb of the right hand, the y axis is aligned with the index finger of the right hand and the z axis is aligned with the middle finger of the right hand (Figure 3.4). This means that the x axis goes from left to right (left being negative and right being positive), the y axis goes down and up (down being negative and up being positive), and z goes forward (positive values of z and coming towards the viewer) and backward (negative values of z and moving away from the viewer).

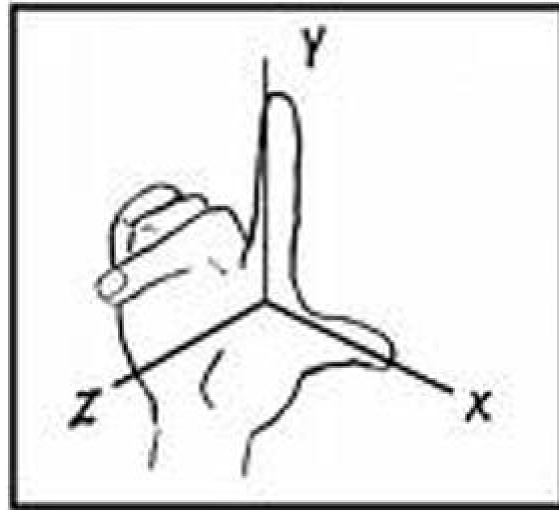


Figure 3.4: Shows the right hand orientation

Triangles and lines can be drawn in strips or in lists (Figure 3.5). Lists are required for drawing separate points, lines, or triangles. Strips, on the other hand, are more efficient where the lines or triangles are combined to create one complex shape like a 3D model. Strips are also more efficient than lists for saving memory and, as a result, enable faster drawing [5]. When you're drawing a triangle strip, adding one more vertex to the strip generates one more triangle. A strip practically cuts the memory requirements for vertex data in half when compared to a list:

Total triangle list vertices = N triangles * 3 vertices

Total triangle strip vertices = N triangles + 2 vertices

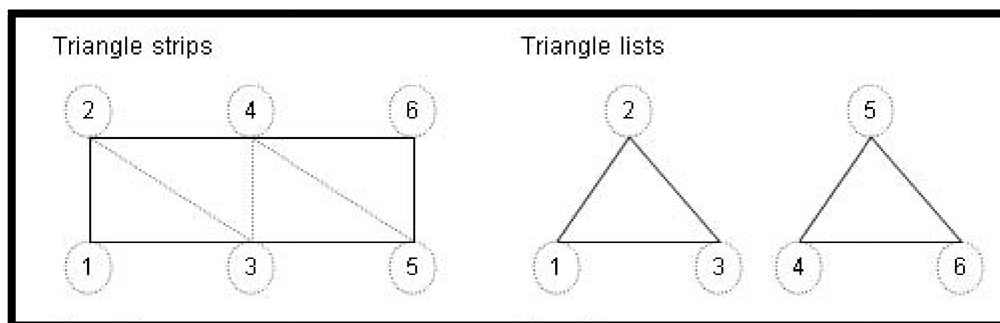


Figure 3.5: shows 4 triangles in a strip and two triangles in a list

When triangle lists are used, a single corner on a cubelet features on as many as six different triangles, the point on the texture can be individually assigned for each of those cases. Using the more compact strip notation means that the textures need to be organized so that the same

texture point is also shared when the point on the triangles is shared, and this makes it harder to create the mapping between the wireframe points and the texture.

In this project therefore, triangle lists were used to draw the 3D cube because it is much easier to texture, we have a small number of triangles, and we don't have the same need for speed as high-action games might have.

When defining a wireframe cubelet, one has a choice to create vertices that are associated with specific colours (`VertexPositionColor`), or with specific positions in a texture (`VertexPositionTexture`). (For this purpose, a texture is created from a simple 2D image, and its points become addressable by values between 0 and 1 on each axis.) Colour-mapped vertices cause the rendering hardware to fill the triangles with (optionally blended) solid colours, while texture-mapped vertices cause the rendering hardware to fill the resultant triangles by choosing pixel colours from the texture.

With texture-mapped vertices, which are what we used for the cubelets, there are two points of particular interest:

- different texture sources can be attached at the time of rendering. This allows us to render the same cubelet object, but make it look different (highlighted or numbered) by changing the texture we apply at rendering time.
- the renderer has to know which surface of the triangle to apply the texture to. Each face in a mesh has a perpendicular unit normal vector (Figure 3.6). The vector's direction is determined by the order in which the vertices are defined and by whether the coordinate system is right- or left-handed. The face normal points away from the front side of the face and a front face is one in which vertices are defined in clockwise order [17]. The texture should be applied on the front face and when one gets this order wrong, the texture will not be visible and the 3D model will appear to have holes in it

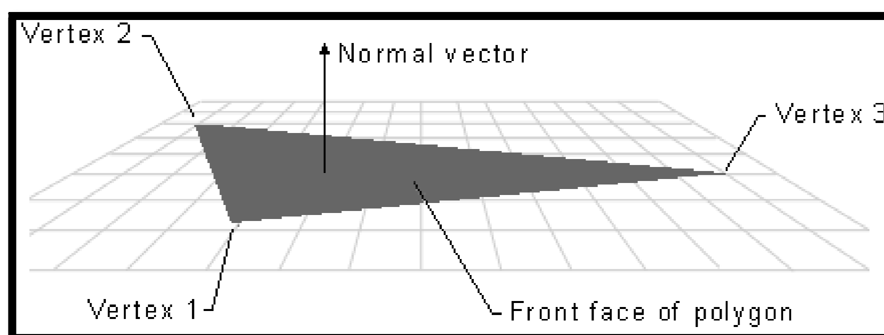


Figure 3.6: Shows the normal vector on a triangle

3.2.1 Matrices

In this project, another useful structure which is provided by XNA is the matrix. In XNA a matrix is a 4x 4 table of data. In rendering any 3D model, three important matrices come into play, namely, the world matrix, view matrix and the projection matrix [18]. In 3D graphics, these three matrices need to be set before 3D geometry can be rendered. These matrices are used to transform the 3D geometry from its initial definition in model space into the final image drawn on the screen in 2D screen space.

The geometry goes through the following steps:

- First you define your 3D object in an editor or by hard coding the values. We chose to use (0, 0, 0) as the origin of the cubelet object, and made the cubelet size (1, 1, 1), with our vertex positions defined relative to the centre. These positions are defined in **model space** [8].
- The next step is to position, rotate and scale our 3D model into our game world. In order to do this we must provide a matrix that XNA Game Studio can use to convert the vertex positions from model space into this new space, known as **world space.** This matrix is called the **world matrix.** The world matrix therefore transforms 3D data from model space into world space and this should be set before rendering every entity in your world [8]. Each object in the world will have its own world matrix that positions and possibly scales the object relative to other objects in the world. If an object is moving in the world, or rotating, its world matrix will be continually changing to reflect this. Changing the world matrix for each 3D model was particularly useful in 3D Noughts and Crosses because only one Ducky or Sonic model was imported into the game and only one cubelet was constructed from individual points. From these three object definitions, the world matrix was manipulated to draw the twenty seven instances of the cubelets in different positions, varying the textures as drawing took place.
- The world will be viewed from a certain position. This position is the location of the eye or camera. So we provide a matrix that allows XNA Game Studio to convert from World Space into **view space** (sometimes known as camera space). This matrix is called the **view matrix.** The view matrix therefore transforms from world space into view space. There is only one view matrix which applies to all the objects in the scene, and this only needs to be modified each time your camera changes position [8].

In 3D Noughts and Crosses there was no need to ever move the camera, so the view space matrix does not change while the game is being played.

- Finally we need to tell XNA Game Studio how to project this 3D View Space onto our flat 2D screen. So we provide a matrix that allows XNA Game Studio to convert from view space into **screen space.** This matrix is known as the **projection matrix**. The key parameters for constructing this matrix define the aspect ratio (width and height) of the rendering surface, and the field-of-view (allowing for effects like zoom in and zoom out). The projection matrix therefore transforms from view space into screen space and this is normally set just once during initialization [8]. In our game we have no need to resize the drawing window dynamically, and no zoom capability, so the matrix remains constant.

3.2.2 Transformations

Transformation data is held in the world matrix and there are three common transformations, namely translation, scaling, and rotating. Transformations, in accordance with the name, transform our 3D objects.

3.2.2.1 Translation

Translation simply moves an object. Translation moves all the points in the 3D Model in the direction of the offset vector you specify with the OffsetX, OffsetY, and OffsetZ properties. For example, an offset vector of (0.0f, 1.6f, 1.0f) would, during rendering, cause the model vertex (2.0f, 2.0f, 2.0f) to instead be rendered at (2.0f, 3.6f, 3.0f). The cube's vertex remains, always, at (2.0f ,2.0f ,2.0f) in model space, but now that model space has changed its relationship to world space so that (2.0f ,2.0f ,2.0f) in model space becomes (2.0f ,3.6f ,3.0f) in world space [18]. This is the transformation that is used in this project to render instances of the initial cubelet at other positions to draw the twenty seven cubelets positioned in a 3*3*3 matrix cube. There is therefore iteration during rendering as the world matrix is manipulated to translate the rendering of the initial cubelet from one position to a different neighbouring position until we have rendered twenty seven small cubelets that make up a bigger 3*3*3 matrix cube [Figure 2.5].

3.2.2.2 Rotation

A transformation that is applied to an object will appear to turn the object on one or more axes when the object is rendered. Rotation transforms the world matrix before rendering the model. If the model was not created about the origin, or has been translated previously, the rotation appears to "pivot" the model about the origin instead of rotating in place. To rotate the model "in place," specify the model's actual centre as the centre of rotation [8]. This transformation is used in this project to enable the player to spin the cube on the x, y and z axis by using input methods like the arrow keys on the keyboard. The player can therefore rotate the cube and view all six faces and click on any of the cubelets as they choose. This improves visualisation of all the game elements on the cubelets.

Notice that cubelets that are displaced from the centre position (that is, those on the corner) do have their translation applied before the rotation, so in fact they do pivot about rather than rotate in place.

We also used another form of rotation in our game: when a position on the game cube becomes part of a winning line, we apply a special "pitch rotation" to the rendering of those cubelets to highlight the winning line by making the pieces do somersaults. This winning-line pitch rotation has to be applied to the cubelet before the translation and rotation of the big cube.

3.2.2.3 Scaling

Scaling an object will make the object larger or smaller. Scaling changes the model's scale by a specified scale vector. One can specify a uniform scale, which scales the model by the same value in the X, Y, and Z axes, to change the model's size proportionally. For example, setting the transformation's ScaleX, ScaleY, and ScaleZ properties to 0.5 halves the size of the model; setting the same properties to 2 doubles its scale in all three axes [8]. In our case, scaling was important because we imported two external models, Sonic and Ducky, with their own model space and sizes. We used our own cubelet size as the "standard" for the world, and then scaled the imported models to fit our world appropriately.

3.3 3D Noughts and Crosses

3.3.1 3*3*3 matrix cube

As discussed earlier, there are two ways of getting 3D models into XNA Game Studio, that is, importing an existing model using the .x or .fbx importer or by drawing the model from 3D points. The second approach was used for the 3*3*3 matrix cube. The `VertexPositionTexture` structure was used which contains `Vector3` coordinates for the position of the cube and another set of `Vector2` coordinates for the texture. The code snippet below shows how to draw the front face of the cube using the `VertexPositionTexture` structure.

```
// Front face
trianglelist[0] = new VertexPositionTexture(new Vector3(0.0f, 0.0f, 0.0f),
new Vector2(0.0f, 0.0f));
trianglelist[1] = new VertexPositionTexture(new Vector3(0.0f, 1.0f, 0.0f),
new Vector2(0.0f, 1.0f));
trianglelist[2] = new VertexPositionTexture(new Vector3(1.0f, 0.0f, 0.0f),
new Vector2(1.0f, 0.0f));
trianglelist[3] = new VertexPositionTexture(new Vector3(1.0f, 0.0f, 0.0f),
new Vector2(1.0f, 0.0f));
trianglelist[4] = new VertexPositionTexture(new Vector3(0.0f, 1.0f, 0.0f),
new Vector2(0.0f, 1.0f));
trianglelist[5] = new VertexPositionTexture(new Vector3(1.0f, 1.0f, 0.0f),
new Vector2(1.0f, 1.0f));
```

As shown above, each face is made up of six points which are joined together by vertices to make triangles which are the basis of 3D models. Each vertex has a 3D position, and a 2D position that determines its corresponding point on a texture. To make up a cube, six faces were required. One important aspect that was confusing at the beginning of this project was to ensure consistency in the *winding order* of the triangles, that is, drawing the vertices in a clockwise manner so that the normal is pointing towards the viewer to ensure that the texture is placed correctly on the front face. The next step was to actually draw or render the cube in the `Draw` method. In order to draw the cube in 27 different places to form the 3*3*3 matrix cube, a translation matrix is used which specifies different positions for each of the cubelets. In our design, we created an object instance for each of the cubelets. In the game, this keeps the state of the cubelet (whether it has been played, by whom, as well as precomputed offsets from the origin, and some other data).The translation matrix is shown below, and it uses the translations from the specific object that describes the state of this cubelet.

```
Matrix tr = Matrix.CreateTranslation(new Vector3(posns[i].X, posns[i].Y,
posns[i].Z));
```

3.3.2 Detecting that a cubelet has been clicked

After the 3*3*3 matrix cube was drawn the next step was to work on the 3D visual effects of the game which are key components of the user interface. This is handled by the `Update` method. The crux of the visual impact of this game is in actually being able to play with the cubes like real life objects which is what makes 3D games engaging. One exciting aspect of this game is the ability to rotate the cube in the yaw, pitch and roll directions. The game also has a mouse-over effect whereby the cubelet that is under the mouse is rendered with a different texture. This improves the interaction that the player has with the game. The gameplay is in actually clicking on a cubelet in order to put your piece into it and own it. This can be quite tricky considering you only want to select one object at a time but this is made quite easy by using the XNA `Unproject` method, the `Ray` structure and the `BoundingSphere` which are part of the built in Math API.

Picking a 3D object from a scene using a mouse click can be complicated because 3D objects are made up of many triangles joined together and the 2D mouse coordinates have to be converted to 3D coordinates that are usable with the 3D objects. To simplify this task, a `BoundingSphere` was defined for each cubelet in the 3*3*3 matrix cube. A bounding sphere (Figure 3.5) is a hypothetical sphere that completely encompasses an object. It is defined by a 3D coordinate representing the centre of the sphere, and a scalar radius that defines the maximum distance from the centre of the sphere to any point in the object.

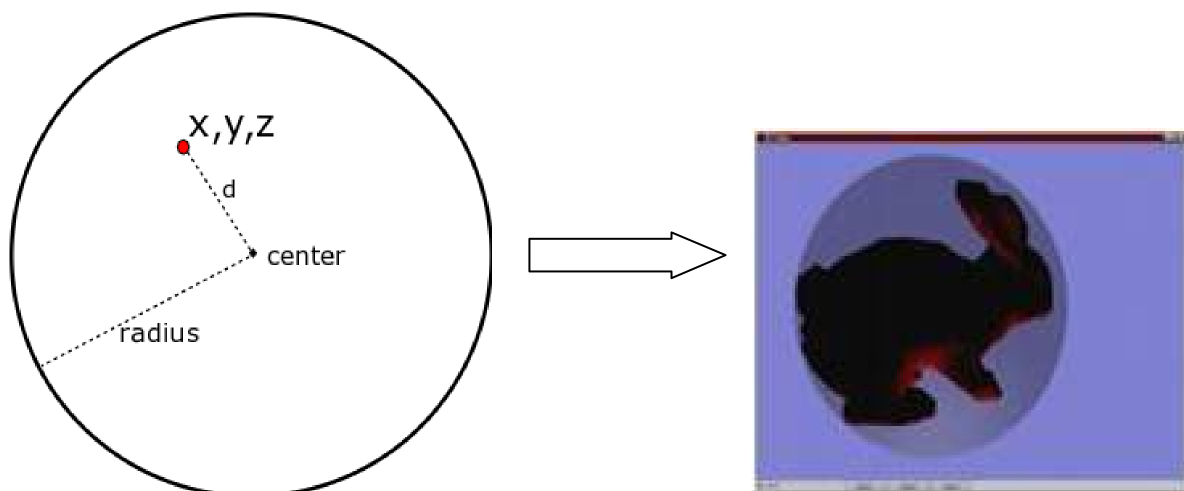


Figure 3.7: Showing the bounding sphere of a 3D model

When the mouse is clicked, there is a method that gets the position of the mouse. The `Unproject` method (which projects a vector from screen space to object space , that is, it

uses the projection matrix to do the inverse of what the projection matrix does) is used to convert the mouse coordinates to a 3D representation. This unprojection, going from a 2D point to 3D, cannot uniquely identify a point in the 3D space. Rather, it produces a line object that runs away from the eye into the scene. In this same method, a `nearsource` and `farsource` which are both `Vector3` are defined using the X and Y coordinates of the mouse position and a Z value. The `nearsource` and `farsource`, together with the unprojected line, make up the `Ray` object with the nearsource being the starting point and the farsource being the furthest point. The next step is to find all objects that intersect with the `Ray`, loop through the objects to find one nearest to the viewer and that is the one which is selected (Figure 3.6 and Figure 3.7).

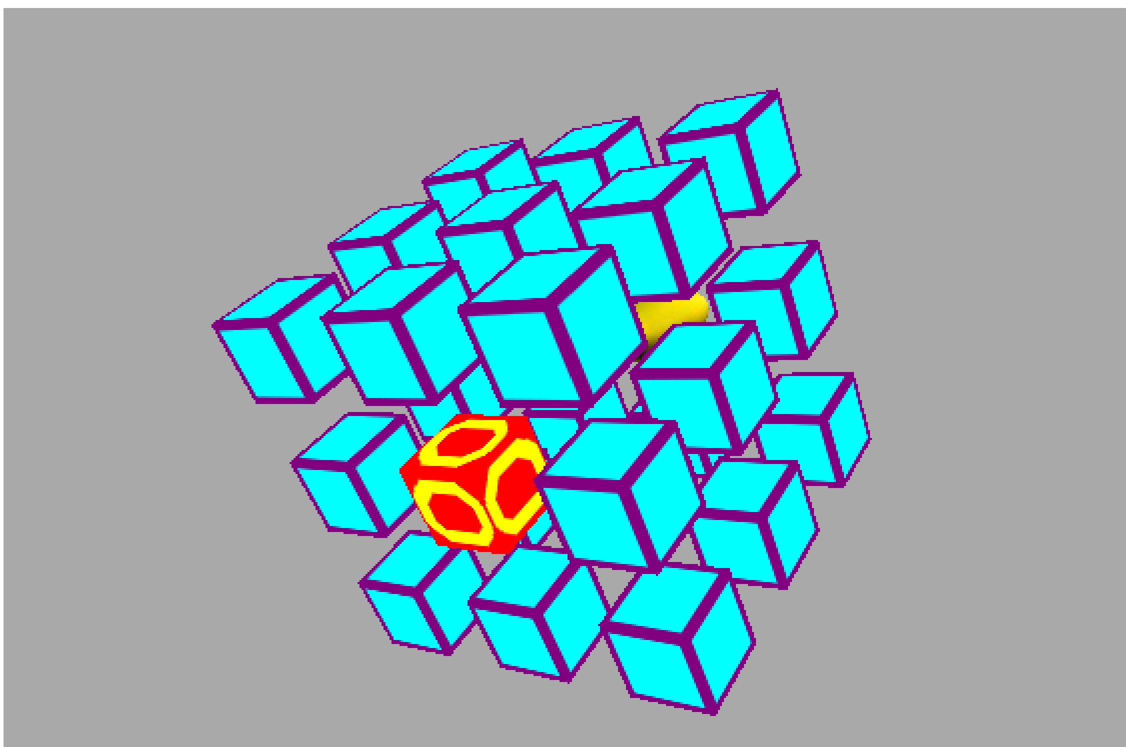


Figure 3.8: Shows the model with the cubelet under the mouse highlighted

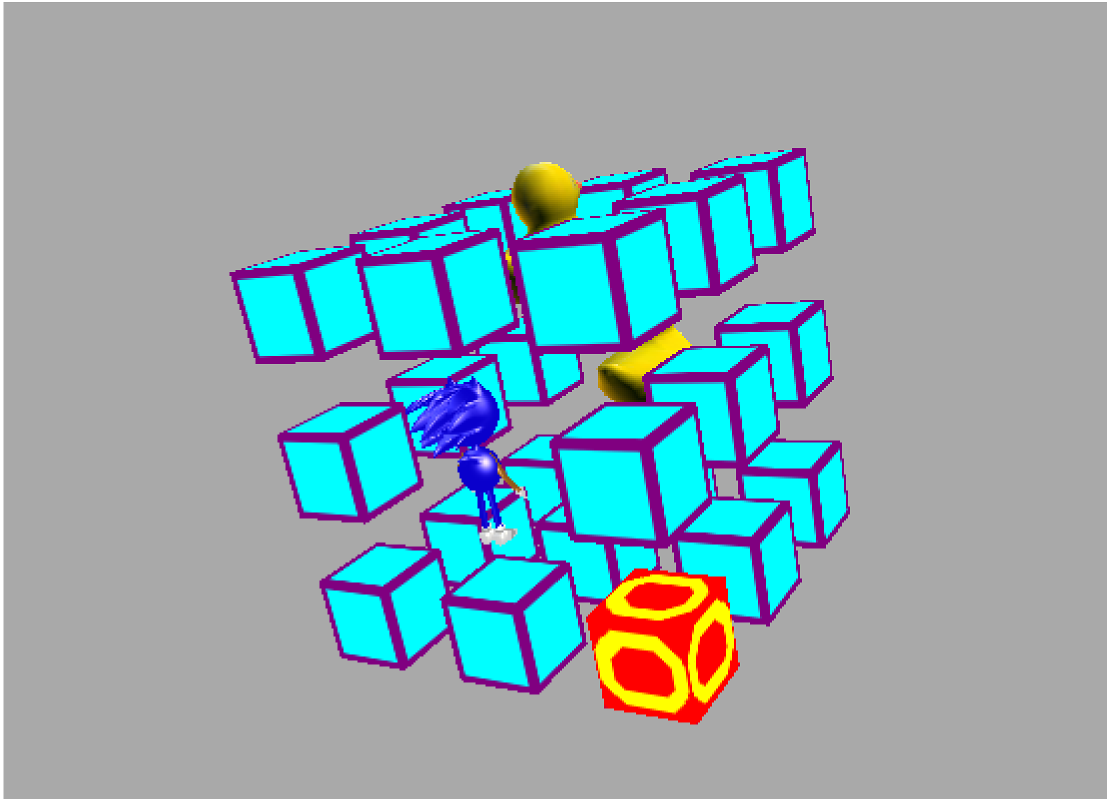


Figure 3.9: Shows the model after a cubelet is clicked, the computer has made another move, and the mouse has now moved over another cubelet.

3.3.3 Importing 3D models

In our game we used a combination of techniques. Two pre-defined models, called "Ducky" (Figure 3.10) and "Sonic" (Figure 3.11), were downloaded and imported into the game through the content importer during initialization. These objects will be rendered when the computer or the user has played one of the available twenty seven cubelet positions. For the positions that have not been played, we constructed our own cubelet wireframe model, and rendered it with one of a number of different textures.

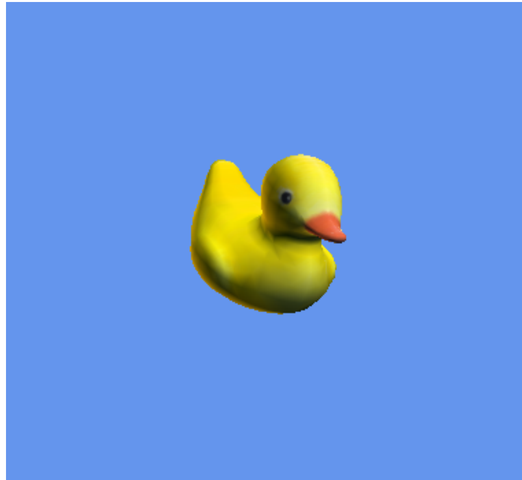


Figure 3.10: The "Ducky" model

Figure 3.11: The "Sonic" model

As shown in Figure 3.9, when a player clicks on a cubelet, a duck model is drawn in place of the cubelet. Importing the model into an XNA game is made possible by the .x or .fbx importers and the Model processor. The models are loaded in the `LoadGraphicsContent` method as shown in the code snippet below:

```
ducky = Content.Load<Model>("Models\\ducky_highres");

sonic = Content.Load<Model>("Models\\sonic");
```

3.3.4 Adding sound

Music sets the atmosphere for our games and sound effects add to the realism of our games. To add sound, we used the Microsoft Cross-Platform Audio Creation Tool (XACT) which

comes with XNA Game Studio. XACT takes completed wav files and puts them in a format that XNA can read and use [4]. Although XACT only supports wav files, one can add multiple sound files to the wave bank.

To implement XNA audio, you must, at least, use these five main objects:

- XACT audio project file
- Audio engine
- Global settings
- Wave banks
- Sound banks

The XACT audio project file is the file created from XACT and it has a .xap extension. This file stores the wave file references, sound cue instances, and their playback settings. The `AudioEngine` object instantiates and manipulates core sound objects for playing game audio. Global settings are the definitions for the audio controls created by the sound designer. This file is used to initialize the sound engine. A wave bank is a collection of wave files loaded and packaged in an .xwb file. A sound bank is a collection of instructions and cues for the wave files to regulate how the sounds are played in your program. Cues are used to trigger audio playback from the sound bank [4]. The code snippet below shows how XACT audio is used in this game:

```
engine = new AudioEngine("Content\\Audio\\soundXOX.xgs");
soundBank = new SoundBank(engine, "Content\\Audio\\Sound Bank.xsb");
waveBank = new WaveBank(engine, "Content\\Audio\\Wave Bank.xwb");
// Play the sound.
soundBank.PlayCue("bugsbunny1");
soundBank.PlayCue("beep");
soundBank.PlayCue("cheering");
```

3.3.5 Animation

Animation is a visual technique that provides the illusion of motion by displaying a collection of images in rapid sequence [2]. Each image contains a small change, for example a leg moves slightly, or the wheel of a car turns. When the images are viewed rapidly, your eye fills in the details and the illusion of movement is complete. When used appropriately in your application’s user interface, animation can enhance the user experience while providing a

more dynamic look and feel. Animated models make a game more exciting and realistic which are some of the characteristics that engage the player. In this game, we did not implement any animation of the models themselves (that is, Sonic does not wave his hands, nor does Ducky change shape). However, the models in the winning line are spun for about 5 seconds before the next game starts.

3.4 Game classes

The system is designed around three main classes, namely, `Cubelet`, `XOXModel` and `Game`. The architecture (loosely) adheres to the Model, View and Controller (MVC) design pattern because MVC (Figure 3.8) separates the concern of storing, displaying, and updating data into three components that can be tested individually [21]. The MVC pattern separates the modelling of the domain, the presentation, and the actions based on user input into three separate classes. The `XOXModel` class is the MVC model of the game and it is responsible for managing the behaviour of the cubelets, managing the game states and also responds to instructions from the controller. The `Cubelet` class is also part of the MVC model: it keeps the state of each of the cubelets during play, and also keeps some relative information about the position of the cubelet relative to the origin. The MVC viewer uses this data extensively when the game is rendered. The `Game` class is the one supplied by the XNA framework, (and extended by ourselves). It is the controller of the game and it is responsible for handling events or logic in the game. The `Game` class `Update` method interprets user inputs from the keyboard or mouse and gives instructions to the model or view to change accordingly. The `Draw` method in the Game class corresponds to the view of MVC. The controller also keeps track of Yaw, Pitch and Roll values in order to allow the viewer to rotate the 3*3*3 matrix cube.

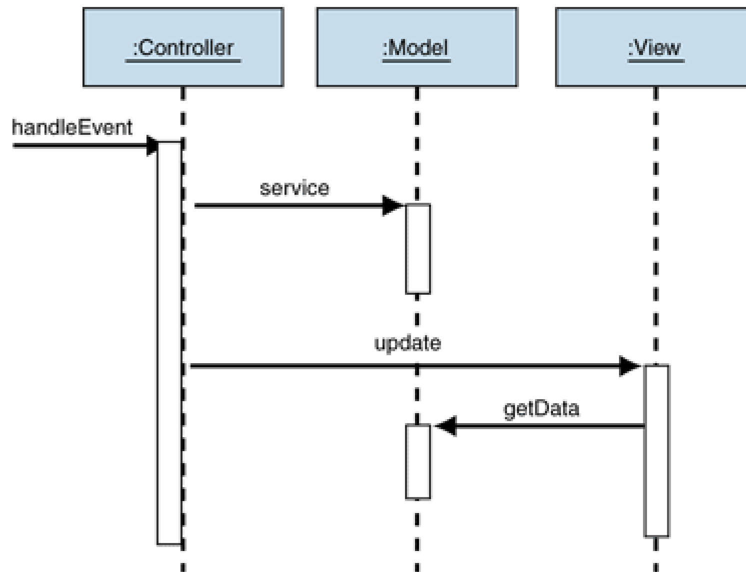


Figure 3.12: The MVC design pattern

There are some minor anomalies in our design where the MVC architecture is not strictly followed. For example, our activity keeps scores and plays multiple games in a row. It was more convenient to simply handle this directly in the Game class, although in strict MVC we should have represented the scores and multiple-game/who-starts-next logic as part of the MVC model.

3.5 Summary

In order to implement 3D Noughts and Crosses it was important first to understand the basic geometry of 3D graphics in order to be able to model the cube. This was made quite easy by the introductory tutorial that is found in the XNA documentation [18]. This chapter therefore showed the features of the XNA framework that were discussed in chapter two coming into use. The Math and Input API's in particular made implementing the game a lot easier. The Math API was fundamental in actually modelling and rendering the 3*3*3 matrix cube using the `Vector` and `Matrix` structures. Designing the user interface which involved using the `BoundingSphere` and `Ray` types was made easier by the Math API. The Input API which defines the Keyboard and Mouse controller types was useful in actually manipulating the cube in order to view all faces. The ease with which sound is played using XACT also made it possible to add various sounds to the game to be played on separate events in order to aid in emphasising the distinction between events.

CHAPTER 4: RESULTS

4.1 3D Noughts and Crosses

In all the versions of the game, the human plays against the computer. Instead of the conventional X and O that is used in the 2D game, we used two 3D models as pieces for each player, that is, Ducky and Sonic. The models were imported into the XNA game using the .x importer. In this game there are 49 possible winning lines which go either vertically, horizontally, diagonally or through the 3*3*3 matrix cube. Whenever a new game starts, the first player is alternated between the human and the computer. Sound has been incorporated into the game such that beeps accompany every move that is made. The beeps have a different tone to distinguish between the two players.

There are four possible states of ownership that a cubelet can be in; none, player 1, player 2 and void. At the beginning of the game, all the cubelets have the `Owner.None` state. When a cubelet is selected or played, it is assigned either player 1 or player 2 depending on whether it is the human or computer that played. This logic of owning a cubelet is done in response to inputs detected during the `Update` method of the XNA game loop.

The Draw method interrogates the XOXModel object and takes note of the cubelet ownerships to render the game content as follows:

- If the state is `Owner.None`, a textured cubelet will be drawn
- If the state is `Owner.Player1`, Ducky will be drawn in place of the cubelet
- If the state is `Owner.Player2`, Sonic will be drawn in place of the cubelet
- If the state is `Owner.Void`, the cubelet is not rendered at all (this is used in one of the game variations where we prefer not to render the centre cubelet)

The state of the game is also tracked in the Update method. At the beginning of every game the state of the game is `GameOutcome.None`. If any of the winning lines is owned by one player, then there is a winner and the game state changes to `GameOutcome.P1Wins` or `GameOutcome.P2Wins` depending on the player who has won.

When there is a winner, the winning cubelets are identified and marked as part of the winning line. During rendering, an extra rotation matrix is applied to all the objects in the winning line such that they spin for a while. A cheering sound also accompanies every winning move

which makes the game interesting and likely to be great fun for the target audience. After every win, and an appropriate timeout, a new game is loaded. 3D Noughts and Crosses also has a scoring scheme whereby a score is awarded for every move made and even higher points are awarded to the winner in each round. The scoring is continuous for the number of rounds that the player plays before quitting the game. This system therefore determines the ultimate winner if more than one game is played. This game tested two different user interfaces with one particularly designed to suit BingBee which uses a touchpad (Figure 4.1) instead of the conventional mouse and keyboard. A number of other variations were also implemented:

4.1.1 Version One

This first version is the easiest one in which the intelligence of the computer is oriented towards or prioritises blocking the human player. When the human player takes possession of the two cubelets in a winning line, the computer moves to block the third cubelet and prevent the human from winning.

4.1.2 Version Two

The second version of the game is such that the aim of the computer is to win and not to block the human player. The computer therefore aims to own three winning cubelets at any given opportunity.

4.1.3 Version Three

In this version, the middle cubelet in the 3*3*3 matrix cubelet is deactivated to try and make the game easier for the target audience. Therefore, instead of 27 playable cubelets, we now have 26. The game code still leaves the option for activating the middle cubelet. Deactivating the middle cubelet leaves only 36 possible winning lines instead of the initial 49 and this is less complex for the player.

4.2 First User Interface

This interface during development only took into consideration the keyboard and mouse as the medium of user input. With this interface, the player uses the arrow keys to rotate the 3*3*3 matrix cube so that all sides can be viewed and the appropriate cubelet can be selected.

Using the method for determining which cubelet has been selected, moving the mouse pointer over the cubelets highlights the selected one by giving it a different texture. When the mouse is clicked on the highlighted cubelet, the cubelet is assigned an owner and the appropriate model is drawn in that position. This interface worked well using the keyboard and mouse and it was implemented on BingBee. After a few trials on BingBee, this interface proved to be quite tedious. BingBee has a pad which can mimic mouse movement (but it is not as accurate or responsive as a real mouse), and it needs a separate key-press instead of a convenient click of a button that is right under the user's finger. BingBee uses a touchpad (Figure 4.1) as a means of input. Using the first user interface, the player selected a cubelet as follows:

- the player made use of the arrow keys to rotate the 3*3*3 matrix cube
- the mouse pad (the area written BingBee in Figure 4.1 below) was used to highlight a potential cubelet and
- the enter key was pressed to play in that cubelet.

As a result, that user interface had to be changed to make it a bit easier and faster. The BingBee combination of moving the cursor and then hitting an enter key proved somewhat difficult, so it gave rise to the second user interface.

Figure 4.1: BingBee touchpad

4.3 Second User Interface

The new user interface was such that each cubelet on the face that is facing forward was assigned a number from 1 to 9, labelled as such using textures (Figure 4.2) and the number keys were dynamically assigned to each cubelet on the front face.

The first step was to determine, during `Update` after new rotations could have been applied to the cube, which face on the 3*3*3 matrix cube is closest to the viewer. Every matrix object in XNA has six easily assessable unit vectors pointing up, left, right, down, forward and back. We looked at these 6 unit vectors in the rotation matrix, and found the one which had the largest Z value, that is, closest to the viewer. From this we know which the forward-facing

face of our cube is. This is shown in the code snippet below, where **tf** is the transformation matrix controlling the pitch, yaw and roll of the cube:

```
faces[0] = tf.Backward.Z;     //front
faces[1] = tf.Left.Z;         //left
faces[2] = tf.Forward.Z;      //back:forward vector is away from the viewer
faces[3] = tf.Right.Z;        //right
faces[4] = tf.Up.Z;           //up
faces[5] = tf.Down.Z;         //down
```

Once we know which face is forward-facing, we can associate numbers with the cubelets on that face. Therefore to play a cubelet; the player had to just press the corresponding number on the BingBee number pad. In the case where the centre cubelet is also active in the game, it can be selected by using the number 0 on the keypad.

Figure 4.2: The second user interface for Noughts and Crosses

Pressing the number 1 will draw the sonic model in the cubelet currently labelled 1. When the 3*3*3 matrix cube is rotated, the face that is facing forward will be relabelled (Figure 4.3) and the upside down sonic models show the winning line. (Notice that only the cubelets that have not been get textured with this labelling.)

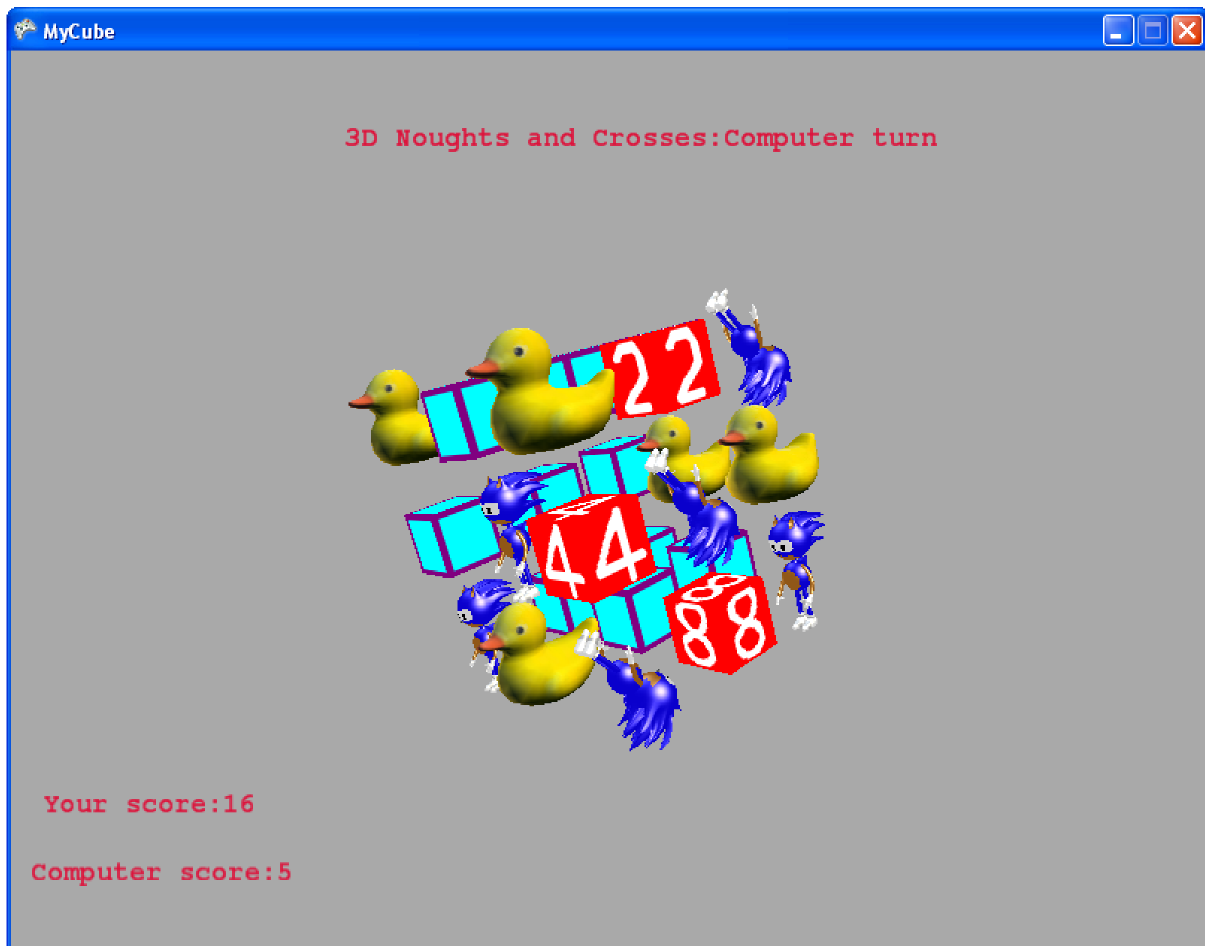


Figure 4.3: Shows the 3*3*3 matrix cube after some cubelets have been played

4.4 Summary

Game variations were incorporated into the game to accommodate the target audience. The objective in 3D Noughts and Crosses is to incorporate varying levels of difficulty which is an aspect that players enjoy in a game. The easiest version is whereby the computer's priority is to defend. The human player therefore only concentrates on evading the defence and winning. In this version therefore the computer only won by chance if the line it was defending coincided with its winning line. The second version which is a bit more challenging requires the human player to defend and to make winning moves at the same time because the computer prioritizes winning.

A third variation was included whereby the middle cube was deactivated and this actually produced a more balanced game whereby the computer either defended or won interchangeably depending on the ownership of the cubelets in any winning line.

CHAPTER 5: CONCLUSION

Computer and video games have become an increasingly prevalent form of entertainment. While the primary purpose of games is entertainment, the underlying design employs a variety of strategies and techniques which require players to analyse, synthesize, and to use critical thinking skills. Ironically, these are also many of the same types of critical thinking skills educators and instructional designers attempt to foster when creating educational materials and media [7]. “3D Noughts and Crosses” is a strategy game that requires critical thinking and evaluation of the player’s position and that of the opponent before making a move. As a result, the player exercises and improves their intellectual skills by playing the game. This therefore confirms the potential of BingBee as a means of not only entertaining the children but educating them at the same time. The key therefore to successful edutainment is to maintain the fun factor in the games so that the players will keep on coming back for more. Game players have also been shown to love challenge in a game, it is therefore important to maintain an increasing level of difficulty as the player progresses.

XNA Game Studio has also proved to make 3D game development reasonably accessible to non-expert game developers, and to programmers who prefer the ease of a modern garbage-collected language like C# in a fully-featured development environment, that is, Visual Studio. This is because of the content pipeline that handles the content of the game. The math of the game, for example, detecting that a 3D object has been clicked is also made easier because of the built in `KeyboardState` and `MouseState` classes. XACT also makes the addition of audio files into the game easier which is important because sound is a crucial part of the game if it is to remain engaging.

For one who is using XNA Game Studio for the first time, the tutorials that are available in the documentation make learning relatively easy.

5.1 Challenges faced in the project

The first problem is that XNA Game Studio requires a graphics card that supports DirectX 9.0c and Shader Model 1.1. Shader Model 2.0 is recommended and required for some Starter Kits provided by Microsoft. Some of the machines at the deployed BingBee did not have the required graphics capability so the hardware had to be upgraded in order to run the game. This therefore delayed the deployment of the game.

The game was developed on a standard computer using the keyboard and mouse as input devices. BingBee on the other hand uses a touch pad as an input device. When the game was deployed on a BingBee machine, the user interface was tedious, that is, using the arrow keys to rotate the 3*3*3 matrix cube, using the mouse pad for selecting a cubelet and using the enter key to play a cubelet. To accommodate the touchpad the user interface had to be changed as described in 4.1.3. Other concurrent projects [Building an intuitive interface for BingBee by Ray Musvibe] are attempting to provide more sophisticated gesture recognition on BingBee's input pad, and this might change the ease of use.

The game loop in XNA Game Studio calls the `Update` and `Draw` methods while the game is running. In certain instances, for example, checking if the mouse has been clicked, this was not desirable because the game loop kept on reading in one key press several times per second. It is also in the `CheckMouseClick` method, that the current players were alternated. As a result of this game loop, the current player would not alternate consistently, for example, the human would play 3 times before the computer gets its turn. As a result, this complicated the game logic and the code had to be modified as follows in order to read a key press only once:

```
bool gotThisClick;          // a flag that is true while the mouse is down
                            // so that we react to the down transition only
bool CheckMouseClick()      // return true if a move is valid
        {
            MouseState mouseState = Mouse.GetState();
            if (mouseState.LeftButton != ButtonState.Pressed)
            {
                gotThisClick = false;
            }
            else
            {
                if (!gotThisClick)
                {
                    gotThisClick = true;

                    //method logic
                }
            }
        }
```

5.2 Possible project extensions

The purpose of this project was to introduce more 3D content on BingBee to pave way for more challenging 3D games. One possible project extension would be to delve more into 3D serious gaming which simulates an actual environment, for example, a game that involves 3D buildings, cars, vegetation and people to make the games more engaging. Another beneficial

aspect would be to develop networked games because XNA Game Studio supports multi player games.

5.3 Project achievements

Our initial question of whether XNA is a suitable programming platform for game development, specifically for use in BingBee, has been positively answered. Our sample game has shown the feasibility of this approach, and the additional work (outside the scope of this project) that was undertaken to integrate XNA games into BingBee has established a new capability that will be a useful future building block for new BingBee activities. The objective of adding more 3D content on BingBee has also been fulfilled by the deployment of this game (3D Noughts and Crosses).

References

[1] Amory Alan, Naicker Kevin, Vincent Jacky, Adams Claudia, *The use of computer games as an educational tool: identification of appropriate game types and game elements* British Journal of Educational Technology 30 (4), p311–322, 1999: [Retrieved on 28/02/20]
www.nu.ac.za/biology/staff/amory/bjet30.rtf

[2] Apple Inc., *Animation overview* © 2008 Apple Inc.
http://developer.apple.com/documentation/GraphicsImaging/Conceptual/Animation_Overview/Animation_Overview.pdf

[3]Becta, *Designing software for schools*, ©Becta, 2001
http://partners.becta.org.uk/page_documents/research/summary.pdf

[4] Carter Chad, *Microsoft ® XNA™ Unleashed: Graphics and Game programming for Xbox 360 and Windows* SAMS, USA, c2008
http://knowfree.net/2007/08/11/microsoft-xna-unleashed-graphics-and gameprogramming- for-xbox-360-and-windows.kf

[5] Cawood Stephen and McGee Pat, Microsoft ® *XNA ™ Game Studio Creator's Guide: An Introduction to XNA Game Programming,* The McGraw Hill Companies, USA, c2007
http://knowfree.net/2008/01/02/microsoft%C2%AE-xna-game-studio-creators-guidepaperback.kf

[6] Dempsey, J. V., Lucassen, B. A., Haynes, L. L. & Casey, M. S. (1996, April). *Instructional applications of computer games*, Paper presented at the annual meeting of the American Education Research Association, New York.
http://eric.ed.gov/ERICDocs/data/ericdocs2sql/content_storage_01/0000019b/80/14/7c/a7.pdf

[7] Dickey Michelle D, *"Ninja looting" for Instructional Design: The Design challenges of creating a Game-based learning environment*, 2006

[8] Ditchburn Keith (A lecturer on the Games Programming Courses at the University of Teesside) *Spaces and Matrix in Direct3D*, © 2004-2008
http://www.toymaker.info/Games/html/matrices.html

[9] Eveland Evan, *"XNA development-Basic concepts: The game loop"*, blog, March 12, 2008
http://evelands.net/evan/xna02.php

[10] Falco Marsha Jean - The Creative Genius behind Set, ©Copyright Set Enterprises Inc. 2004
http://www.setgame.com/set/index.html

[11] Gee James Paul, *What video games have to teach us about Learning and Literacy*, ACM Computers in Entertainment, Vol. 1, No. 1, October 2003, BOOK01

[12]Greenfield et al, *Cognitive socialisation by computer games in two cultures: Inductive discovery or mastery of an iconic code?* Journal of applied developmental psychology 15, pg 59-85, 1994

[13] Klucher Michael (Program manager – XNA Game Studio) –*The XNA Framework Content Pipeline*, XNA team blog, August 29, 2006
http://blogs.msdn.com/xna/archive/2006/08/29/730168.aspx

[14]Leddo, J. (1996). An intelligent tutoring game to teach scientific reasoning, Journal *of Instruction Delivery Systems, 10*(4), 22-25.

[15] McFarlane Angela (Professor), Sparrowhawk Anne, Heald Ysanne: *Report on the educational use of games: an exploration by TEEM of the contribution which games can make to the education process*, 2002 [Retrieved on 28/02/2008]
http://www.teem.org.uk/publications/teem_gamesined_full.pdf

[16] McLeod Jeremiah, 3D - Noughts and Crosses (Tic-Tac-Toe)
http://www.murderousmaths.co.uk/games/ttt/ttt.htm

[17] Microsoft Corporation: *3D coordinate systems and geometry*, © 2008
http://msdn.microsoft.com/en-gb/library/bb324489(VS.85).aspx

[18] Microsoft Corporation: *3D Transformations overview*, © 2008
http://msdn.microsoft.com/en-us/library/ms753347.aspx

[19] Microsoft Corporation: XNA Game Studio 2.0 documentation, © 2008
http://msdn.microsoft.com/en-us/library/bb200104.aspx

[20] Mitchell Alice and Savill-Smith Carol – *The use of computer and video games for learning* © Learning and Skills Development agency, 2004
http://www.lsda.org.uk/files/PDF/1529.pdf

[21] Microsoft patterns and practices, *Model-View-Controller*, ©2008 Microsoft Corporation
http://msdn.microsoft.com/en-us/library/ms978748.aspx

[22] Moursund David, *Some Personal Thoughts about Research on Using Games in Education*, NECC 2006

[23] Prensky Marc, *Digital game-based Learning*, ACM Computers in Entertainment, Vol. 1, No. 1, October 2003, Book 02.

[24] Walker Mitch (25/08/2006) Program manager | XNA Framework: *What is the XNA Framework*, XNA team blog, August 25, 2006 [Retrieved on 01/03/2008]
http://blogs.msdn.com/xna/archive/2006/08/25/724607.aspx

[25] Weisstein Eric W, “Tic-Tac-Toe." From *MathWorld*--A Wolfram Web Resource.
http://mathworld.wolfram.com/Tic-Tac-Toe.html

[26] http://www.bingbee.com/

[27] http://creators.xna.com/

Appendix

C# code for 3D Noughts and Crosses

Cubelet class (Model)

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace MyCube
{
    public enum Owner { None, Player1, Player2,Void };
    class Cubelet
    {
        public Owner theOwner;
        public bool inWinningLine = false;
        private int onFrontFace = -1;

        public int OnFrontFace
        {
            get { return onFrontFace; }
            set { onFrontFace = value; }
        }

        public Cubelet( )
        {
            theOwner = Owner.None;
        }
    }
}
```

XOXModel class (Model)

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework;

namespace MyCube
{
     public enum GameOutcome { None, P1Wins, P2Wins, Full };

     public class XoXGameModel
     {
         Cubelet[] cubes;

         Random rand = new Random();

         internal Owner TheOwner(int selectedIndex)
         {
             return cubes[selectedIndex].theOwner;
         }
```

```
        internal bool IsInWinningLine(int i)
        {
            return cubes[i].inWinningLine;
        }

        internal void SetCubeOwner(int selectedIndex, Owner owner)
        {
            cubes[selectedIndex].theOwner = owner;
        }

        internal void SetCubeOnFrontFace(int i, int val)
        {
            cubes[i].OnFrontFace = val;
        }

        internal int GetCubeOnFrontFace(int i)
        {
            return cubes[i].OnFrontFace;
        }

        public XoXGameModel()
        {

            cubes = new Cubelet[27];
            for (int i = 0; i < 27; i++)
            {
                cubes[i] = new Cubelet();
                cubes[i].theOwner = Owner.None;
                cubes[i].inWinningLine = false;

            }
            cubes[13].theOwner = Owner.Void;
        }

        bool full = false;

        int[,] Winners = {{0, 3, 6}, {1, 4, 7}, {2, 5, 8}, {0, 1, 2}, {3,
4, 5}, {6, 7, 8}, {0, 4, 8}, {6, 4, 2} , {9, 12, 15}, {11, 14, 17}, {9, 10,
11}, {15, 16, 17},  {18, 21, 24}, {19, 22, 25}, {20, 23, 26} , {18, 19,
20}, {21, 22, 23}, {24, 25, 26}, {18, 22, 26}, {24, 22, 20}, {0, 12, 24},
{2, 14, 26}, {0, 10, 20}, {6, 16, 26}, {0, 9, 18}, {1, 10, 19} , {2, 11,
20}, {3, 12, 21}, {5, 14, 23}, {6, 15, 24}, {7, 16, 25}, {8, 17, 26}, {24,
16, 8}, {18, 10, 2}, {20, 14, 8}, {18, 12, 6}};

        public int CalculateCompMove()
        {
            int pos = GoforBlock();
            if (pos >= 0) return pos;
            pos = GoforWin();
            if (pos >= 0) return pos;
            pos = GoforCenter();
            if (pos >= 0) return pos;

            return GoforRandom();
        }

        private int GoforCenter()
        {

            if (cubes[4].theOwner == Owner.None)
```

```
            {
                cubes[4].theOwner = Owner.Player1;
                return 4;
            }

            if (cubes[22].theOwner == Owner.None)
            {
                cubes[22].theOwner = Owner.Player1;
                return 22;

            }
            return -1;
        }

        private int GoforBlock()
        {
            for (int w = 0; w < 36; w++) //49
            {

                if (cubes[Winners[w, 0]].theOwner == Owner.Player2 &&
cubes[Winners[w, 1]].theOwner == Owner.Player2 && cubes[Winners[w,
2]].theOwner == Owner.None)
                {
                    cubes[Winners[w, 2]].theOwner = Owner.Player1;
                    return Winners[w, 2];
                }

                if (cubes[Winners[w, 0]].theOwner == Owner.Player2 &&
cubes[Winners[w, 1]].theOwner == Owner.None && cubes[Winners[w,
2]].theOwner == Owner.Player2)
                {
                    cubes[Winners[w, 1]].theOwner = Owner.Player1;
                    return Winners[w, 1];
                }

                if (cubes[Winners[w, 0]].theOwner == Owner.None &&
cubes[Winners[w, 1]].theOwner == Owner.Player2 && cubes[Winners[w,
2]].theOwner == Owner.Player2)
                {
                    cubes[Winners[w, 0]].theOwner = Owner.Player1;
                    return w;
                }
            }
            return -1;
       }

       private int GoforWin()
       {
           for (int w = 0; w < 36; w++)  //49
           {
               if (cubes[Winners[w, 0]].theOwner == Owner.Player1 &&
cubes[Winners[w, 1]].theOwner == Owner.Player1 && cubes[Winners[w,
2]].theOwner == Owner.None)
               {
                cubes[Winners[w, 2]].theOwner = Owner.Player1;
                   return Winners[w, 2];
               }

               if (cubes[Winners[w, 0]].theOwner == Owner.Player1 &&
cubes[Winners[w, 1]].theOwner == Owner.None && cubes[Winners[w,
2]].theOwner == Owner.Player1)
```

```
               {
                cubes[Winners[w, 1]].theOwner = Owner.Player1;
                   return Winners[w, 1];
               }

               if (cubes[Winners[w, 0]].theOwner == Owner.None &&
cubes[Winners[w, 1]].theOwner == Owner.Player1 && cubes[Winners[w,
2]].theOwner == Owner.Player1)
               {
                 cubes[Winners[w, 0]].theOwner = Owner.Player1
                   return Winners[w, 0];
               }
           }
           return -1;
         }

         private int GoforRandom()
         {
             int comp = rand.Next(26);
             if (cubes[comp].theOwner == Owner.None)
             {
                 cubes[comp].theOwner = Owner.Player1;
                 return comp;
             }
             return -1;
         }

         private void CheckFull()
         {
             for (int p = 0; p < 27; p++)
             {
                 if (cubes[p].theOwner == Owner.None)
                     full = false;
                 else
                     full = true;
             }
         }

         public GameOutcome CheckOutcome()
         {
           for (int w = 0; w < 36; w++)  //49
           {
            if (cubes[Winners[w, 0]].theOwner == Owner.Player2 &&
cubes[Winners[w, 1]].theOwner == Owner.Player2
&& cubes[Winners[w, 2]].theOwner == Owner.Player2)
              {
                    cubes[Winners[w, 0]].inWinningLine = true;
                    cubes[Winners[w, 1]].inWinningLine = true;
                    cubes[Winners[w, 2]].inWinningLine = true;
                    return GameOutcome.P2Wins;
             }
            else if (cubes[Winners[w, 0]].theOwner == Owner.Player1 &&
cubes[Winners[w, 1]].theOwner ==Owner.Player1 && cubes[Winners[w,
2]].theOwner == Owner.Player1)
                {
                    cubes[Winners[w, 0]].inWinningLine = true;
                    cubes[Winners[w, 1]].inWinningLine = true;
                    cubes[Winners[w, 2]].inWinningLine = true;
                    return GameOutcome.P1Wins;
                 }
             }
```

```
            for (int i = 0; i < 27; i++)
            {
                if (cubes[i].theOwner == Owner.None)
                {
                    return GameOutcome.None;
                }
            }

            return GameOutcome.Full;

        }
    }
}
```

Game class (Controller)

The code below shows the Update method that handles game logic.

```
float mdlRotation = 0f;
bool gotKey = false;

protected override void Update(GameTime gameTime)
   {
       mdlRotation += 0.04f;
       KeyboardState newState = Keyboard.GetState();
       // Allows the game to exit
       if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Escape))
           this.Exit();

       if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Up))
       {
           pitch -= (float)gameTime.ElapsedGameTime.TotalMilliseconds
           *MathHelper.ToRadians(0.1f);
       }
       if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Down))
       {
           pitch += (float)gameTime.ElapsedGameTime.TotalMilliseconds
           *MathHelper.ToRadians(0.1f);
       }
       if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Left))
       {
           yaw -= (float)gameTime.ElapsedGameTime.TotalMilliseconds *
           MathHelper.ToRadians(0.1f);
       }
       if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Right))
       {
           yaw += (float)gameTime.ElapsedGameTime.TotalMilliseconds *
           MathHelper.ToRadians(0.1f);
       }
       if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.OemComma))
       {
           roll -= (float)gameTime.ElapsedGameTime.TotalMilliseconds *
           MathHelper.ToRadians(0.1f);
       }
       if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.OemPeriod))
       {
           roll += (float)gameTime.ElapsedGameTime.TotalMilliseconds *
           MathHelper.ToRadians(0.1f);
       }
```

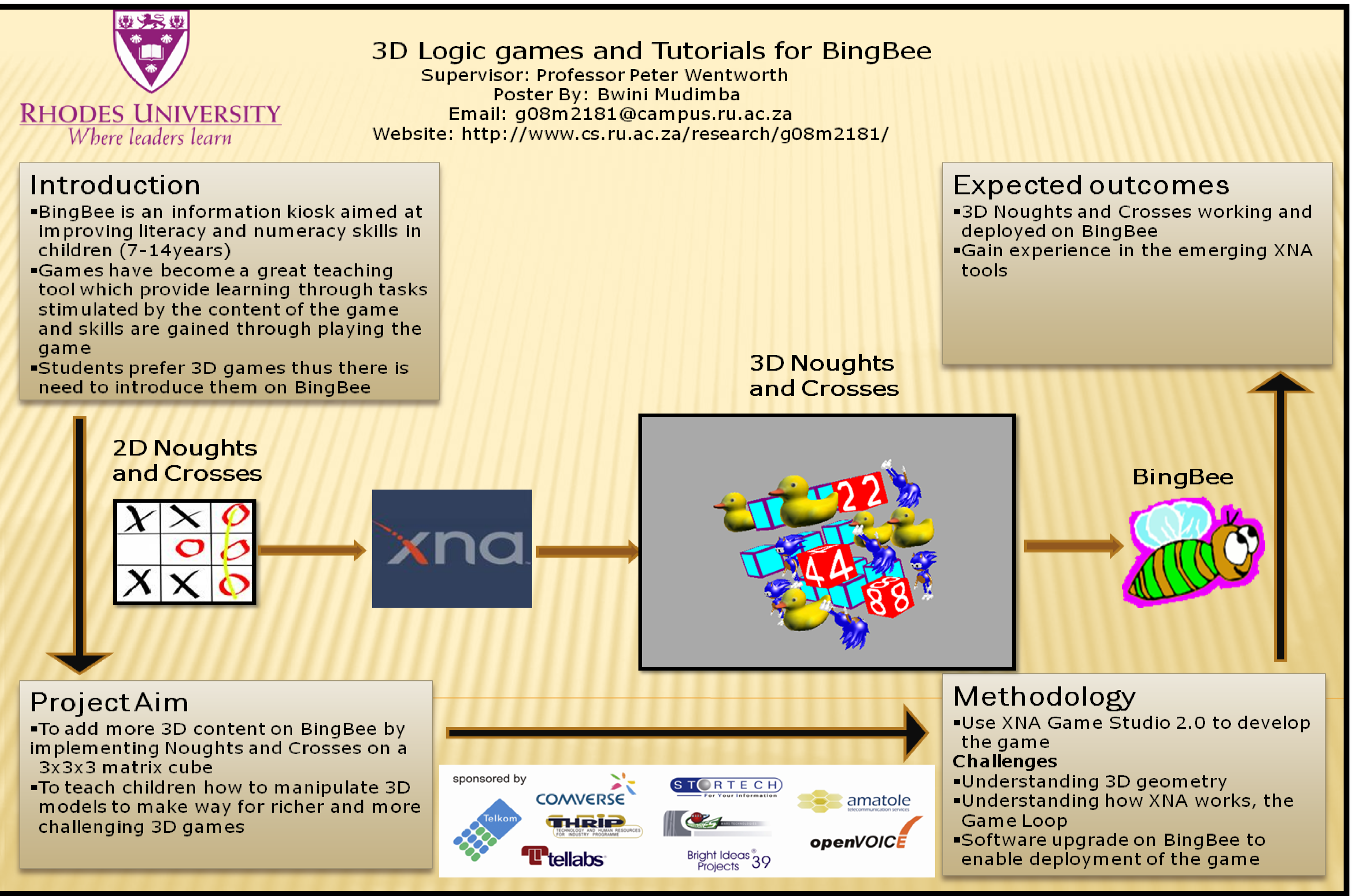
```
labelForwardFacingCubelets();

 if (DateTime.Now.CompareTo(dontPlayUntil) > 0)
 {
     if (newGamePending)
     {
         StartAnotherGame();
     }
     else
     {
         if (playerToMoveNext == Human)
         {
           Keys[] theKeys =
              Keyboard.GetState(PlayerIndex.One).GetPressedKeys();
           if (theKeys.Length == 0)
           {
          gotKey = false;
          }
          else
          {
            if (theKeys.Length == 1)
             {
              bool played = false;
              if (!gotKey)
              {
               switch (theKeys[0])
               {
                case Keys.NumPad1: played = selectCubelet(1);
                break;
                case Keys.NumPad2: played = selectCubelet(2);
                break;
                case Keys.NumPad3: played = selectCubelet(3);
                break;
                case Keys.NumPad4: played = selectCubelet(4);
                break;
                case Keys.NumPad5: played = selectCubelet(5);
                break;
                case Keys.NumPad6: played = selectCubelet(6);
                break;
                case Keys.NumPad7: played = selectCubelet(7);
                break;
                case Keys.NumPad8: played = selectCubelet(8);
                break;
                case Keys.NumPad9: played = selectCubelet(9);
                break;
                default: break;
              }

              if (played)
                 {
                    soundBank.PlayCue("blip1");
                    dontPlayUntil =
                    DateTime.Now.AddSeconds(0.8)
                    playerToMoveNext = Computer;
                  }
                    gotKey = true;
                 }
               }
            }
         }
```

```
                else
                {
                     theModel.CalculateCompMove();
                     compscore++;
                     soundBank.PlayCue("beep");
                     playerToMoveNext = Human;
                 }
                 switch (theModel.CheckOutcome())
                 {
                    case GameOutcome.None:
                    break;
                    case GameOutcome.P1Wins:
                    soundBank.PlayCue("cheering");
                    dontPlayUntil = DateTime.Now.AddSeconds(5);
                    winner = true;
                    this.BoardState = GameState.ComputerWins;
                    compscore += 10;
                    newGamePending = true;
                    break;
                    case GameOutcome.P2Wins:
                    soundBank.PlayCue("applause");
                    dontPlayUntil = DateTime.Now.AddSeconds(5);
                    winner = true;
                    this.BoardState = GameState.HumanWins;
                    score += 10;
                    newGamePending = true;
                    break;
                    case GameOutcome.Full:
                    dontPlayUntil = DateTime.Now.AddSeconds(5);
                    newGamePending = true;
                    break;
                 }
            }
        }
        engine.Update();
        base.Update(gameTime);
    }
```

The code below is the `Draw` method of the game which forms the View component of the design.

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Clear(Color.DarkGray);

  // The effect is a compiled effect created and compiled elsewhere
  // in the application.
 for (int i = 0; i < posns.Length; i++)
  {
   Matrix centercubelet = Matrix.CreateTranslation(-0.5f, -0.5f,
                     0.5f);
   Matrix centerbigblock = Matrix.Identity;
   Matrix tr = Matrix.CreateTranslation(new Vector3(posns[i].X,
                    posns[i].Y, posns[i].Z));
   Matrix mm =  tr * centerbigblock *
                  Matrix.CreateFromYawPitchRoll(yaw, pitch, roll);
   switch (theModel.TheOwner(i))
    {
    case Owner.None:
```

```
graphics.GraphicsDevice.VertexDeclaration = vertexDeclaration;
       basicEffect.Begin();

         switch (theModel.GetCubeOnFrontFace(i))
         {
             case 1:
               basicEffect.Texture = one;
               break;
             case 2:
               basicEffect.Texture = two;
               break;
             case 3:
               basicEffect.Texture = three;
               break;
             case 4:
               basicEffect.Texture = four;
               break;
             case 5:
               basicEffect.Texture = five;
               break;
             case 6:
               basicEffect.Texture = six;
               break;
             case 7:
               basicEffect.Texture = seven;
               break;
             case 8:
               basicEffect.Texture = eight;
               break;
             case 9:
               basicEffect.Texture = nine;
               break;

             default:
               basicEffect.Texture = unclicked;
               break;
          }

             basicEffect.TextureEnabled = true;
             basicEffect.World = centercubelet * mm;

             foreach (EffectPass pass in
                          basicEffect.CurrentTechnique.Passes)
             {
                pass.Begin();
                DrawTriangleList();
                pass.End();
             }

             basicEffect.TextureEnabled = true;
             basicEffect.End();
             break;
             case Owner.Player1:
                  DrawModel(ducky, mm, theModel.IsInWinningLine(i));
                  break;
             case Owner.Player2:
                  DrawModel(sonic, mm, theModel.IsInWinningLine(i));
                  break;
           }
      }
```



3D Logic Games and Tutorials for BingBee