NTNU
Norwegian University of
Science and Technology

Evaluation and Extension of an XNA Game Library used in Software Architecture Projects

Trond Blomholm Kvamme
Jan-Erik Strøm

Master of Science in Computer Science
Submission date: June 2008
Supervisor: Alf Inge Wang, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This master thesis evaluates the introduction of XNA game development in software architecture projects, as well as evaluating and extending an XNA game library used in the projects. This evaluation will investigate how well XNA in general is suited for software architecture projects, if the game library was useful and if it was used, and improvements for future projects. The project will also reflect on the mathematical and programming prerequisites for creating 3D games with XNA.

Assignment given: 15. January 2008
Supervisor: Alf Inge Wang, IDI

Abstract

For most young people growing up today, video games have been a part of their life on the same level as music, films, and other entertainment. They regard video games as a fun, exciting, and absorbing source of entertainment and stimulation. Transferring these properties into an educational context can prove to be very valuable and motivational. In this master thesis, the introduction of video game development with the XNA game development platform in software architecture projects at the Norwegian University of Science and Technology (NTNU) is evaluated. This includes an evaluation of a 2D XNA game library used in the projects. In addition, we present an assessment of the effort and time spent required to grasp the necessary 3D concepts and techniques involved in producing 3D games with XNA. We also describe our improvements and extensions of the game library to support and include 3D features, based on the evaluation and assessment.

The students of the course had the choice between the traditional project (a robot simulation) and the new XNA project. We find that the students who chose the XNA project were more motivated, struggled less, and thus required less assistance. On the other hand, the XNA students admitted to over focusing on the gameplay of their game, at the expense of the software architecture. This should even out when more learning material specific to the XNA project becomes available.

40% of the XNA students used the game library in their project. Overall, they were satisfied with the usefulness and usability of it, but did not think it helped them focus less on technical matters and more on the architecture.

Preface

This report is a Master's Thesis conducted during the 2008 spring semester at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

The work was conducted by two computer science students under the supervision of associate professor Alf Inge Wang, and is a continuation of a student depth project called "Exploration of the XNA Game Programming Framework and its Suitability for Teaching Software Architecture"[1] written in the autumn semester of 2007.

We would like to thank our supervisor Alf Inge Wang for his guidance and feedback throughout the project period.

Table of Contents

Part I
Introduction

1. Introduction

The use of video games for educational purposes is a relatively new approach. Many students today have grown up playing video games, and have a clear understanding of what they have to offer. Video games have properties such as fun, absorbing, and engaging. Players often find themselves in deep concentration and tend to forget about the world around them. Often video games represent an escape from the real life into a virtual world where the rules and content keep the players glued to the television or computer screen. Because of these properties, video games have a great motivational factor. Leveraging this factor is key to successful use of video games in education.

Many students struggle with motivation when trying to learn in an educational context. Material seems dry and boring, and the teaching methods are not very engaging or exciting. Incorporating video games in the teaching may increase student's motivation and help them achieve better results. The Norwegian University of Science and Technology (NTNU) in Trondheim has established a research program in video games, and some courses are already using games as an integrated part of the curriculum[2].

This master thesis deals with the integration of student game projects in a software architecture course at NTNU. In these projects, the students are tasked with creating games using the XNA game development platform from Microsoft. They apply topics learned in the course through the design and implementation of a software architecture for their game.

In the autumn semester of 2007, we conducted a depth study where we explored XNA and its suitability for teaching students software architecture. As part of this, we also developed a 2D XNA game library called XQUEST (XNA QUick and Easy Starter Template), which contains reusable code for accomplishing common game development tasks. In this master thesis, we build upon the results gained from this depth study, and conduct evaluations of both the introduction of XNA game projects in the software architecture course, and XQUEST. We also improve and extend XQUEST, based on the evaluation and the desire to support 3D features.

1.1 Project Definition

This master thesis evaluates the introduction of XNA game development in software architecture projects, as well as evaluating and extending an XNA game library used in the projects. The evaluation will investigate how well XNA in general is suited for software architecture projects, if the game library was useful and if it was used, and improvements for future projects. The project will also reflect on the mathematical and programming prerequisites for creating 3D games with XNA.

1.2 Project Context

Part of our master thesis involves conducting an evaluation of the student projects in the course TDT4275 Software Architecture at NTNU. We present the results of this evaluation in Part III, but we explain the background and organization of the student projects here.

The goal of the student projects in the course is to give the students practical experiences of designing and implementing a software architecture. In previous semesters the student projects involved developing a robot simulation program in Java, in which the students were tasked with programming an autonomous robot navigating an area, picking up balls, and moving them to a destination. The outcome of the project was a software architecture and an implementation of the architecture for the robot simulation program. However, this semester the students were given the choice between the robot project and a new XNA game development project.

The students worked in groups of three to five people. Each group was assigned a quality attribute to focus on, which would drive their architecture. The robot project groups were assigned either the *modifiability*, *testability*, or *safety* quality attribute. The XNA project groups were assigned either *modifiability* or *testability*.

The project went through several phases:

- *Requirements and architectural description*

  In this phase the groups authored two documents; a requirements document eliciting requirements, and an architecture document describing elements such as architectural drivers, structures, views, and quality tactics.

- *Architectural Trade-Off Analysis Method*

  In this phase, the groups evaluated each other's architecture using ATAM[3] techniques. As far as it was possible, the robot groups evaluated the XNA groups and vice versa.

- *Implementation*

  In the implementation phase, the groups implemented their architecture. They also created an implementation document describing the implementation details.

- *Post-mortem*

  After the hand-in of the projects, the groups performed a post-mortem analysis[4] of their work.

- *Presentation*

  Selected groups were given a chance to present their work.

During the run of the course, the students used the it's learning e-learning platform[5]. This platform is a web-based service where students keep track of the courses they are enrolled in. With it's learning, the students can get information about schedules, lectures, exercises, and exams, and they can deliver exercises and download course material. In the software architecture course, the students delivered the outcome of each phase using it's learning.

Both the authors of this thesis were involved in the course a student assistants. As such, we gained insight and experiences that proved useful in our master thesis.

1.3 Report Outline

The thesis is divided into parts. Each part contains several numbered chapters that again may contain several sections. Part I introduces the report, explains the background, goals, and research agenda. Part II contains the results of our prestudy, where we present the underlying concepts and technologies behind our work. In this part we also summarize the findings in our depth study. Part III presents a survey of the student projects in the software architecture course. Part IV presents our work on XQUEST, the XNA game library we developed as part of our depth project, and which we improved and extended for this master thesis. Finally, Part V concludes the thesis and outlines further work.

2. Research Agenda

In this chapter we present our research agenda. This includes our research questions; what we want to answer in this project, and research methods; how we go about answering them. The research questions drive our project work, and the outcome and success of the project will depend on the degree at which we successfully answer the questions using the chosen methods.

2.1 Research Questions

In our depth study, our main focus was on investigating the suitability of the XNA framework for use in student projects in the software architecture course. We also gauged the difficulty level of learning C# programming, the XNA API, and the tools, given the student's backgrounds and experience levels. Lastly, we investigated how the results of the project could be introduced to the students. Based on our depth study, this master thesis' main research agenda revolves around activities that are two-fold:

- *Evaluation*: We evaluate the use of XNA in the student projects and compare with the robot simulation project. We also evaluate the use of XQUEST.

- *Extension*: We go beyond the 2D-only focus from our depth study and take a more in-depth look at the 3D features of XNA, as well as improving and extending XQUEST with more useful software elements, both 2D and 3D.

With this in mind, we will be answering the following research questions:

RQ1: What are the pros and cons of using XNA in the student projects as opposed to the traditional robot simulation?

a) Does game programming steal too much focus with respect to the learning goals of the software architecture course?

b) Are the students who chose the XNA game project more motivated than the students who chose the robot simulation project?

c) Will the students who chose the XNA game project need more assistance than the students who chose the robot simulation project?

d) Will the choice of COTS influence the different architectural sides of the project?

RQ2: How useful is our XNA game library for the students?

a) Will the students actually use our game library or find other similar libraries or software packages better suited?

b) What parts of our game library are most useful for the students?

c) What are the features of the game library that students found missing or hard to use?

RQ3: How can our game library be extended and improved?

a) What are the necessary changes and additions for introducing 3D concepts in the game library?

RQ4: What is the difficulty level/what kind of effort is required for learning the 3D concepts and production of 3D content needed to be able to create 3D games with XNA?

a) What are the prerequisites in terms of mathematical and programming knowledge for creating 3D games with XNA?

b) Is it reasonable to expect that 3D content such as model meshes, skinning, bone animation etc should be produces by the students themselves?

The first two questions reflect our evaluation focus. The third question deals with XQUEST and how it can be extended and improved. We dig a bit deeper into the XNA framework and look at the more advanced concepts. The fourth question involves looking at 3D material and implementing 3D features into XQUEST.

2.2 Research Methods

To answer our research questions, we need to use research methods that are suitable. Our research questions cover areas that are very different in nature, and it is obvious that we need to use several different research methods in order to answer them.

2.2.1 Experimental Models

For doing experimentation in software engineering, Basili[6] mentions three approaches:

- *The engineering method.* This method observes existing software and software processes in order to identify problems and propose solutions. The solutions may be in the form of modification to the existing software. Vital parts of this method are careful analysis and measurement. The process is repeated until no more improvements can be identified.

- *The empirical method.* The empirical method uses statistical analysis to validate a certain hypothesis about the software or software processes. It starts with the proposition of the introduction of a new model, and analyzes and evaluates the effect it has on the system.

- *The mathematical method.* By using mathematical and formal methods to develop a theory, the mathematical method provides a deductive approach to the problem at hand. The results can be considered a framework for developing models to be used in further work. The results can also be compared with empirical observations.

Our first two research questions, which focus on evaluation of the usage of XNA to support the learning of software architecture, are answered by an empirical approach. Through observation, communication, and feedback gathering, we analyze and measure the effect the introduction of the XNA game project has on the course. The outcome of this process is a comparative evaluation of the robot projects versus the XNA game projects, some general conclusions about how the course projects were perceived (from both sides), and an evaluation of XQUEST based on usability and usefulness. In Section 2.2.3, we will present in more detail our strategy for carrying out these evaluations.

For the third research question, an engineering approach is used. We will improve and extend our game library with more advanced concepts. For this, we identify components that are useful to be included in such a library, and then analyze the impact these components has on the game development process. For identifying the most useful components, we also use an empirical approach, either directly through communication and feedback gathering from the students of the course, or indirectly through the technical assistance we provide, thus identifying what they are struggling with and how we can best help them.

For our fourth research question, none of the experimental models described by Basili makes a clear candidate. This question is highly theoretical in nature, and so other methods need to be considered.

2.2.2 Technology Validation Models

Zelkowitz and Wallace[7] states that for experimentation in software engineering to be effective, new approaches that are more specific to the characteristics of software are needed. Thus, they have created a taxonomy consisting of twelve experimental approaches grouped into three main categories: *observational* methods, which collect data from a project as it is developed, *historical* methods, which apply collected data from previous project to new ones, and *controlled* methods, which uses multiple instances of an operation to provide statistical validity. The twelve validation methods are summarized in Table 2.1.

Table 2.1: Twelve methods for validating technology.

Observational Methods	**Historical Methods**	**Controlled Methods**
Project monitoring	Literature search	Replicated
Case study	Legacy data	Synthetic
Assertion	Lessons learned	Dynamic analysis
Field study	Static analysis	Simulation

Both observational and historical methods are of interest for us. The *field study* method involves external examination of multiple projects to find out if a certain new tool or process has an effect on the projects. This method fits well for our evaluation focus (RQ1 and RQ2). We examine how the introduction of the XNA game project alongside the robot project will affect the learning processes, as well as comparing the two versions of the project. Furthermore, for our extension focus (RQ3), combinations of observational and historical methods are used. *Project monitoring* is employed to monitor and gather data passively from the XNA game projects. This can lead to identification of new components that students either suggest or can benefit from. In addition, historical methods such as *literature search* and *lessons learned* are used. We dig deeper into XNA to gain an understanding of the more advanced concepts in order to support them in our game library. Books, web articles, and tutorials are our primary sources for the literature study. We also consider the results from our depth study. Improvements to existing components of the game library are made; particularly the more abstract features are updated to support 3D concepts. For the last research question (RQ4), historical methods are used. We study the literature pertaining to 3D graphics, and the mathematics behind the curtains.

2.2.3 Empirical Strategies

There are three major strategies for conducting an empirical investigation[8]:

- *Survey*. A survey is used when the object under evaluation (tool, technique) has been in use for a while. The data is typically gathered using questionnaires or interviews, often using only a small representative part of the population under investigation. The results from the survey are analyzed in order to reach conclusions.

- *Case study*. A case study is an observational study where projects, activities or assignments are monitored. Data is collected throughout the run of the study. The study often focuses on certain attributes or relationships between attributes. Statistical analysis can then be applied to the gathered data to reach conclusions.

- *Experiment*. Experiments are done in a certain environment and provide a high level of control. The subjects of the experiment are exposed to the same treatment randomly, and by measuring the manipulation of one or more variables a statistical analysis can be performed.

Our research pertaining to the empirical study is both qualitative and quantitative in nature. It is qualitative in that we are following the student projects as student assistants, and through assistance we can gain an understanding of how the students interpret their tasks and how they perceive the COTS (robot or XNA) they are using. However, it is mostly quantitative. An essential part of our evaluation is to compare the XNA projects against the robot projects. Data is collected after delivery to form a basis for comparison and statistical analysis. However, we lack the control needed to do rigid experiments and thorough analyses of statistical data. Instead, the research is mostly explorative and focuses on the gathering of both qualitative and quantitative data. Therefore, an experiment, which focuses solely on gathering quantitative data and requiring an elevated level of control, is not a valid strategy for us. The approach we use is a combination of survey and case study. We follow the students throughout the course as student assistants, and help them with their exercises and projects. We also have the opportunity to assess their work, and from that we gather data that is valuable for us in a qualitative way. However, the main approach is a survey study where we collect quantitative data using questionnaires that the students will respond to after the completion of their project.

Part II
Prestudy

In this part we present topics that lay the foundation for the rest of report. It starts with a summary of the relevant theoretical findings in our depth project in Chapter 3, and continues in Chapter 4 with a presentation of the results that were achieved and the conclusions that were inferred in the same project. In Chapter 5, we present an overview of the 3D concepts used in 3D game development, as well as how these concepts are represented in XNA.

3. Theory from the Depth Study

In this chapter we present a summary of the theoretical findings that are relevant for this master thesis from the prestudy part of our depth project. For a more detailed text, we refer to the depth study itself[1].

3.1 XNA

XNA is game development platform developed by Microsoft, which includes a programming framework and a set of tools to offer a complete game development package[9]. Based on the .NET platform, XNA offers game development for the PC, the Xbox 360, and more recently, the Zune[10] media player, using the C# programming language. XNA mainly targets students, hobbyists, and independent game developers. XNA is free to use for anyone, but to deploy games on the Xbox 360, a subscription to the XNA Creators Club[11] is required. XNA was motivated by an earlier attempt at bringing the DirectX C++ multimedia API[12] over to the .NET platform, called Managed DirectX[13], which was essentially a 1:1 mapping of the DirectX API onto .NET. XNA took the idea one step further and provides a complete game development solution, not just the programming API.

First released in version 1.0 in December 2006, the current version of XNA is 2.0 (released December 2007), with XNA 3.0 scheduled to be released in late 2008.

3.1.1 Architecture

Although inspired by DirectX, and unlike Managed DirectX, XNA is a completely fresh solution, adhering to the more modern .NET standards, making it a novelty in the game development field.

Figure 3.1 shows a deployment overview of XNA, depicting the big picture component-wise. *XNA Game Studio* is the development environment; it integrates itself into any Visual Studio 2005 edition, providing a set of starter kits, project templates, and features editor-support for extended high-level components such as the Content Pipeline and the Application Model. The *XNA Framework* is the programming API. Modular, object-oriented designed; this API enables developers to program their games using a modern, managed programming language (C#), and with the comfort and functionality of the class libraries of the .NET Framework on the PC, or the .NET Compact Framework on the Xbox 360 and Zune.

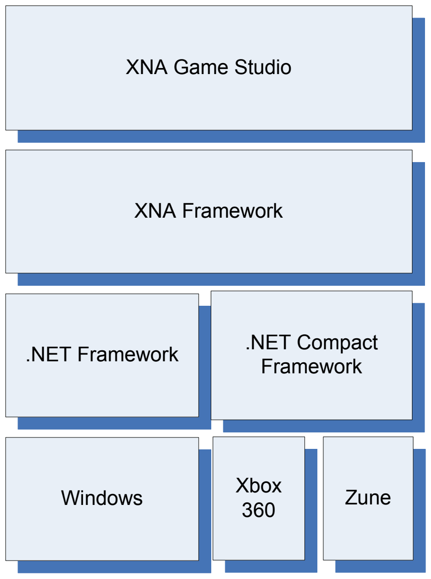


Figure 3.1: Deployment view of XNA.

Further dissection of the XNA Framework reveals the layers shown in Figure 3.2.

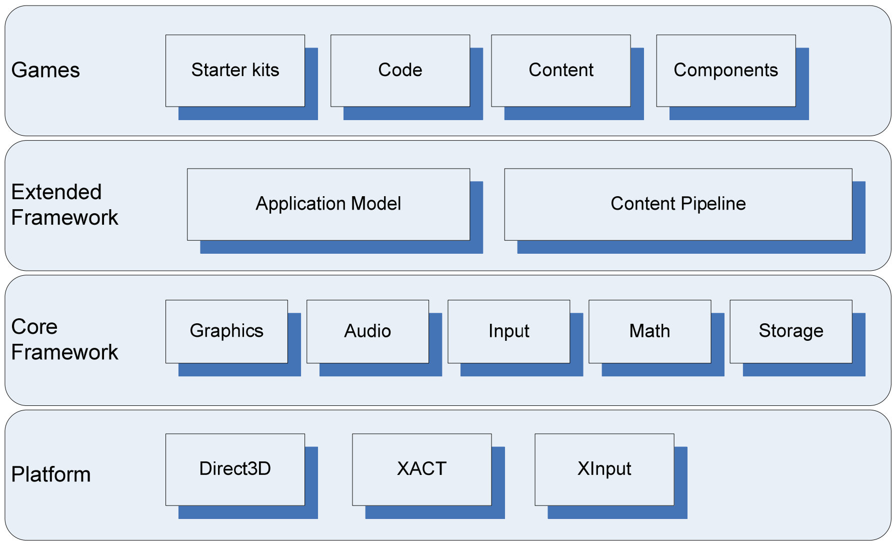


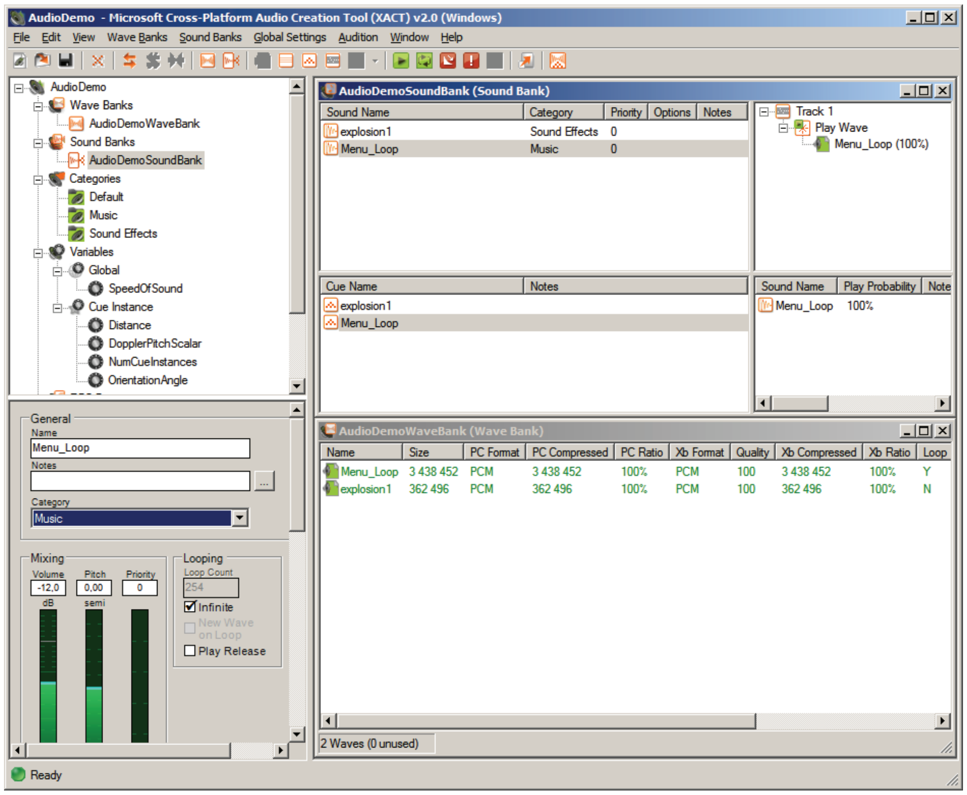
Figure 3.2: Layers of the XNA Framework.

At the bottom layer, we find the platform components that provide the graphics (Direct3D) functionality, audio (XACT) functionality, and input (XInput) functionality. On top of that, the *Core Framework* includes programming interface for common game development tasks in the categories *Graphics, Audio, Input, Math,* and *Storage.* The *Extended Framework* layer contains two major components that are at the heart of the XNA game development experience; the *Application Model* and the *Content Pipeline*. The upper layer represents community resources, provided by both Microsoft and the development community.

Following we will look at some of the aforementioned components (that have relevancy for this master thesis) in more detail.

3.1.2 XACT

XACT (Microsoft Cross-Platform Audio Creation Tool)[14] is a tool for managing audio. It was originally released with the DirectX SDK, but has been adopted for use in XNA as well. It sports a convenient graphical user interface that lets users manage audio resources by grouping them up into categories and setting properties like volume, pitch, looping, and so on. The name is a little misleading; the tool does not let users create audio files, but is a way of organizing audio created in other audio packages. Figure 3.3 shows a screen shot of the XACT tool.



Sound Name	Category	Priority	Options	Notes
explosion1	Sound Effects	0		
Menu_Loop	Music	0		

Cue Name	Notes
explosion1	
Menu_Loop	

Sound Name	Play Probability	Note
Menu_Loop	100%	

General
Name: Menu_Loop
Notes:
Category: Music

Mixing
Volume: -12,0 dB
Pitch: 0,00 semi
Priority: 0

Looping
Loop Count: 254
[✓] Infinite
[ ] New Wave on Loop
[ ] Play Release

Name	Size	PC Format	PC Compressed	PC Ratio	Xb Format	Quality	Xb Compressed	Xb Ratio	Loop
Menu_Loop	3 438 452	PCM	3 438 452	100%	PCM	100	3 438 452	100%	Y
explosion1	362 496	PCM	362 496	100%	PCM	100	362 496	100%	N

2 Waves (0 unused)

Figure 3.3: Screen shot of the XACT tool.

The XACT tool creates an XACT audio project, which can be imported into XNA Game Studio through the content pipeline. The audio resources defined in the XACT project are then accessible through the XNA Framework audio API.

3.1.3 The Application Model

The Application Model is an application-level framework that "provides functionality for accomplishing common game development tasks"[15, 16]. The heart of the Application Model is the *Game* class, which provides a base class for game simulation processing. When a new game project is created in the XNA Game Studio development environment, a skeleton code is provided, which contains a class derived from the XNA *Game* class. There are four virtual methods in the *Game* class that make up the game simulation loop, also known as just the game loop (see Figure 3.4). The game loop is managed by the XNA Framework, which calls these methods at appropriate times. The four methods are:

- *Initialize*. Is called before the game loop starts to initialize resources.
- *LoadContent*. Responsible for loading game content such as textures, audio, 3D models, sprite fonts, etc, and is called after the *Initialize* method but before the game loop starts.
- *Update*. Updates the game logic and state of objects in the game.
- *Draw*. Draws graphics. A pass through the *Draw* method in the game loop is called a *frame*.

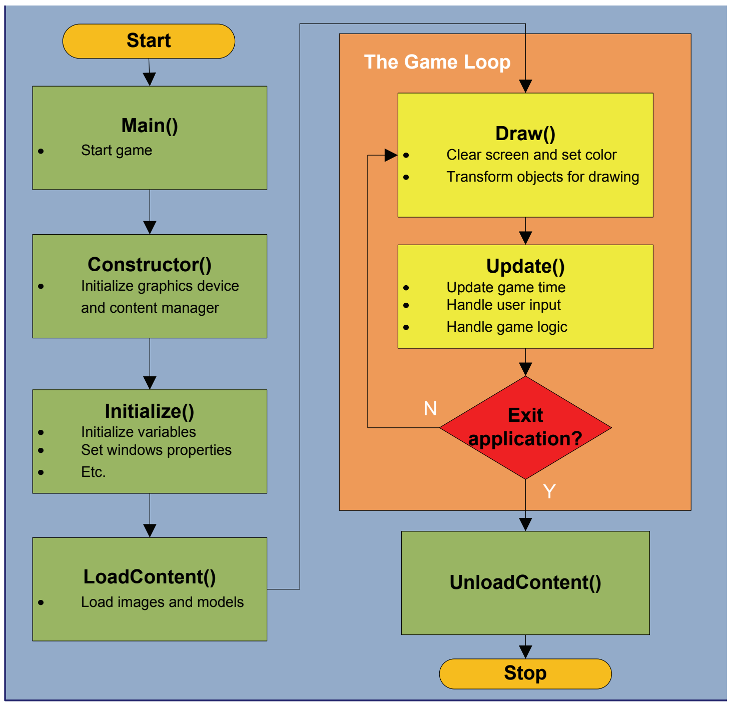


Figure 3.4: The game loop.

Even though the game loop really only involves the *Update* and *Draw* method, the *Initialize* and *LoadContent* methods are considered parts of it because they provide the necessary setup before the game loop can begin.

An important feature of the game loop is the fixed time step vs. variable time step modes. In fixed time step mode, the *Update* method is called in fixed intervals. By default, the game loop runs in fixed step mode with the *Update* method being called every 1/60th of second. The *Draw* method is only called after a successful call to *Update*, which means the game loop will idle after a *Draw* call when it is not time to call *Update* yet. On the other hand, in variable time step mode, the *Update* and *Draw* methods are called continuously. Fixed time step and variable time step modes can be toggled through the *Game.IsFixedTimeStep* boolean property.

The Application Framework is extensible through modular *game components* that can be plugged in through the *Game* class. A game component is created by deriving a class from *GameComponent* or *DrawableGameComponent*. The component then overrides the *Initialize*, *LoadContent*, *Update*, and *Draw* methods. A game component is plugged into the *Game* class by adding it to the *Game.Components* collection. The *Game* class will then manage the component by calling its respective methods in *Game.Initialize*, *Game.LoadContent*, *Game.Update*, and *Game.Draw*.

Another feature of the Application Framework is game services. Game services provide a way to maintain loose coupling between objects that need to communicate with each other. Game services follow a publish/subscribe pattern, where objects can register themselves in the *Game.Services* collection so that other objects can retrieve them. Game components make good candidates for game services, since they provide distinct coherent functionality that is well suited for reuse.

3.1.4 Sprites

A sprite in computer graphics usually refers to a two-dimensional image or animation. In XNA, sprites are represented as *Texture2D* objects[17]. To draw sprites, XNA uses something called a sprite batch, exposed through the *SpriteBatch* class. The *SpriteBatch.Draw* method[18] and its overloads take a *Texture2D* object along with several other parameters that describe the position and appearance of the sprite. All calls to *SpriteBatch.Draw* must occur only after *SpriteBatch.Begin* has been called, and before *SpriteBatch.End* is called. Behind the curtains, *SpriteBatch* sets up render states and a 2D orthogonal coordinate system (see Figure 3.5) that is necessary for drawing 2D images. What this means is that users do not have to worry about setting up a 2D environment themselves in order to draw sprites.

Sprites drawn with a *SpriteBatch* can be scaled, rotated, tinted, and flipped. Animation can be simulated by drawing sequences of images in a timely fashion.

A sprite font is a special kind of sprite that draws text instead of an image. It is simply an XML file describing the properties of the font such as scale, character ranges, bold, italic, etc (see Figure 3.6). It can be created in XNA Game Studio from a regular Windows system font such as Arial or Courier. It is then imported into the game through the content pipeline, and drawn with *SpriteBatch.Draw*.

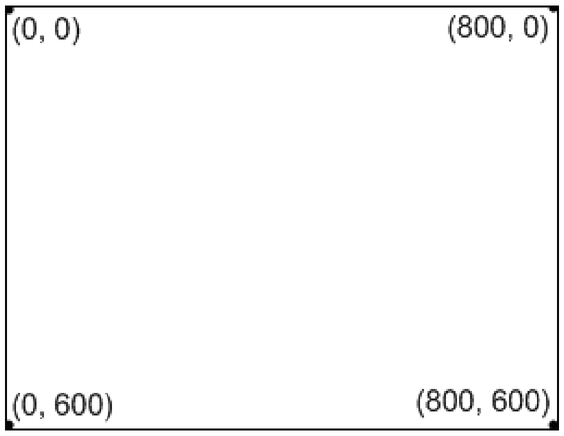
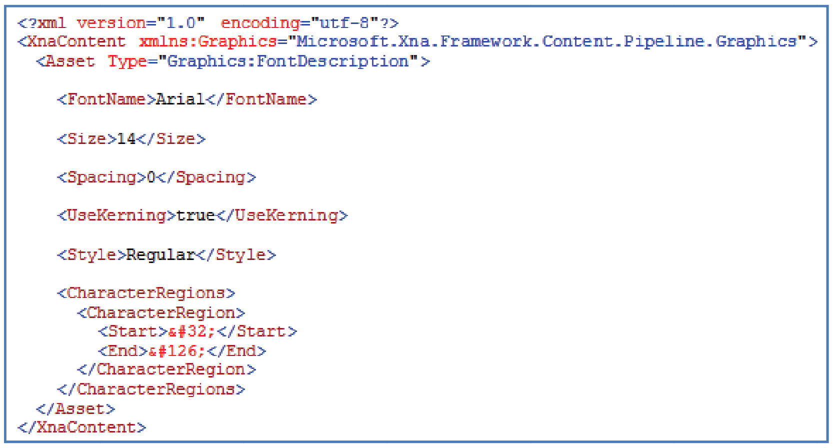


Figure 3.5: Coordinate system of *SpriteBatch*.



```
<?xml version="1.0"  encoding="utf-8"?>
<XnaContent xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
  <Asset Type="Graphics:FontDescription">

    <FontName>Arial</FontName>

    <Size>14</Size>

    <Spacing>0</Spacing>

    <UseKerning>true</UseKerning>

    <Style>Regular</Style>

    <CharacterRegions>
      <CharacterRegion>
        <Start>&#32;</Start>
        <End>&#126;</End>
      </CharacterRegion>
    </CharacterRegions>
  </Asset>
</XnaContent>
```

Figure 3.6: A sprite font's source XML. The sprite font is using the Arial system font with size 14 and regular style.

4. Results from the Depth Study

Here we present our own contribution in the depth study; the XQUEST game library, as well as the results and conclusions that were made as part of our exploration of XNA game development framework and its usefulness for teaching software architecture.

4.1 The XQUEST 1.0 Game Library

In our depth study, we developed a small 2D XNA game library consisting of several game components and helper classes that could be used to simplify certain game programming tasks, such as sprite drawing and input handling[1]. We did not give the library an official name in the depth study. Later however, we felt that since we were to release the library to the students of the software architecture course, we had to name it something. After some consideration, the choice fell on XQUEST, short for *XNA QUick and Easy Starter Template*. We feel that this name describes the library fairly accurate. It is meant to be a quick and easy way to get started using XNA, and we imagined most people would use it as a template on which they based their code.

XQUEST 1.0 was developed using version 1.0 of the XNA framework. XNA 2.0 was released right before Christmas 2007, which means that the modifications and extensions we made to XQUEST as part of this master thesis uses this version of XNA. The new and extended version of XQUEST will therefore be referred to as XQUEST 2.0.

Time was a big issue when we were considering what kind of components for XQUEST we should make. The students only had about one to two months at their disposal for implementation, and so our components would contribute the most if they reduced implementation time in any way. Through our research we found that our game library had to focus on the modifiability quality attribute to speed up the implementation of game specific functionality.

Through our work with a prototype platform game we identified several framework components that are present in most 2D games. By providing the students with this common functionality, we could help them achieve faster implementation.

XQUEST 1.0 consisted of the following three modules as shown in Table 4.1.

Table 4.1: The three modules of XQUEST 1.0 and the C# classes contained in each of them.

Sprite Animation Framework (SAF)	**Game Object Management Framework (GOMF)**	**Helper classes**
Sprite.cs AnimatedSprite.cs Animation.cs	IGameObject.cs BasicGameObject.cs ICollidable.cs GameObjectManager.cs	TextureStore.cs AudioManager.cs InputManager.cs TextOut.cs

4.1.1 The Sprite Animation Framework

The goal of our sprite animation framework was to conceptualize and build on XNA's sprite drawing and in addition introduce sprite animation. This makes it easier for users to use sprites in a more modular and object-oriented way, compared to the standard way it is done in the XNA framework.

4.1.2 The Game Object Framework

A game object is any object in the world that has some kind of life and/or behaviour. Game objects can interact with each other in many ways to create a dynamic game world. We conceptualized the game object into a framework for creating and managing the game objects in a 2D game.

The framework keeps track of all game objects that are present in the game world at any time and is able to handle collisions and other interactions between them. We designed it to be as abstract and generic as possible to allow flexibility and reusability.

4.1.3 The Helper Classes

These helper classes are not part of any overall framework, but are stand-alone separate classes that provide convenient interfaces to commonly used functionality. We created four of these classes that help a user handle textures, audio, input, and drawing text on the screen.

4.2 Results

In our depth study we investigated the results of applying the game library to create actual games. Two small games were created, Pong and Breakout, and we identified the game library's impact on development time as well as ease of integration.

Table 4.2 shows which of the classes in the game library were used in the games, their role in the games (how they were used), and how we would have done it without using any of these classes.

Table 4.2: Classes used in the Pong and Breakout games.

Classes used in the Pong and Breakout games	**Role in game**	**XNA Alternative**
Sprite	Paddles, ball, and bricks modeled as sprites.	Use the SpriteBatch.Draw method with lots of parameters.
BasicGameObject	Paddles, ball, and bricks represented as game objects.	For Pong, create Paddle and Ball classes from scratch with position, velocity, speed, state, etc. Keep track of instances of Paddle and Ball, and implement collision detection directly in the Pong class. For Breakout, in addition manage collection of Bricks.
ICollidable	Implemented in the Ball class to handle collisions with the paddles.	
GameObjectManager	Automatic management of the paddles and ball. Calls update and draw for all game objects, and checks for collisions.	
AudioManager	Plays sound effects when the ball hits the paddles or bricks and when a player scores a point.	Manage the three classes AudioEngine, WaveBank, and Soundbank.
InputManager	Checks for input from keyboard or a game pad.	Query the state of the keyboard and the game pads manually.

Since both Pong and Breakout are two very simple games we did not experience a huge impact using our game library, because the XNA framework already provides a high level API with which such simple games can easily be created. However we did see that our game library worked well, and were we to extend the games in some way it could be more easily achieved. Also, the game components helped keeping the code clean and manageable.

A third game was also created during our depth study project, a larger platform game. It served as a test bed for what components should go into the game library. Through its development we found that if we had had the components of the game library from start, we would have been able to create the final version of the platform game in a lot less time. It uses the entire range of classes in the game library. The game object framework was especially helpful as well as the sprite animation framework. The texture store and input manager were also good and much used assets.

4.3 Conclusions from the Depth Study

The overall goal of the depth study project was to investigate the XNA framework and its usefulness for teaching students software architecture. We found that creating games takes much time, even with such a high level API as XNA, and we therefore decided to try to reduce the implementation time by creating a game library that provides functionality that makes certain aspects of game programming easier.

The results of implementing Pong and Breakout show that the most useful part of the library is the game object management framework. However, these games were too small and simple for an effective evaluation. For instance, there was no animation going on, so some parts of the game library remained unused. The platform game however used the entire library and benefited much from it in the end.

Another issue we looked into was the difficulty level of learning C# and XNA. We concluded that learning the pre-requisites was a fairly easy task for fourth grade students in computer science, and that there exist plenty of help resources on the web aimed at teaching students and hobbyists exactly this.

The great increase in complexity when we go from 2D to 3D made us limit the game library to 2D. This limitation gave us the time to create a better framework and we reduced the risk of having students who set upon making a 3D game and running out of time. Learning 3D game programming would also require much more time from the students, and quite possibly be at the expense of learning software architecture.

Not all of the quality attributes presented in the software architecture course were relevant for game development projects. Performance was not ideal because of the difficulty of measuring it for such small projects and because it steals much focus away from other quality attributes and software architecture itself. Security is neither a good quality attribute to focus on, because it is not easily applied and of relevance to the types of games resulting from these projects. We concluded that modifiability, testability, availability, and usability are the most relevant quality attributes to focus a game development project on in the software architecture course.

5. XNA and 3D Concepts

In this chapter we will present an overview of the more advanced subjects of XNA, focusing especially on 3D concepts, and investigate how easy it is and what the requirements learning such concepts within a limited timeframe are. In our depth project, we chose to focus solely on 2D, except brief mentions of 3D concepts in the general XNA chapters. We did this because we believed that 2D-only was a good way to limit the amount of required knowledge and experience that the students need to acquire to successfully complete the XNA game project. While we still believe that limiting themselves to 2D is a sensible approach, we still want to conduct an investigation into 3D concepts in order to understand better what this will require in regards of background knowledge, time spent gaining new knowledge, and difficulty level of making a 3D game while still focusing on the architectural side.

5.1 Basic 3D Mathematics

Here we present some fundamental mathematical topics needed to understand the basic 3D concepts of the next part. The students of the software architecture course should all be familiar with the mathematics involved as it does not go much further than the fundamentals of linear algebra and basic trigonometry. The source for much of the information over the next sections is the excellent DirectX 9.0 introductory book by Frank Luna[12].

5.1.1 Vectors

A vector is pretty much a quantity that possesses both magnitude and direction. This makes them ideal for representing forces, displacements, velocities, and even positions in video games.

The geometrical representation of a vector is a line segment and by applying geometric operations we can use vectors to solve problems involving vector-valued quantities (e.g. how far and in what direction would this character move, if affected by two different forces?). However, our computer can only crunch numbers and therefore the vectors need to be specified numerically. By introducing 3D coordinate systems and translating the vectors so that the tail coincide with the origin, we can represent a vector by the coordinates of its head.

Now we can perform basic vector operations on them, like addition, subtraction and multiplication by a real number. In addition we can calculate the dot product and the cross product of two vectors. Using the dot product we can find the angle between the two vectors. Using the cross product we can find another vector that is mutually orthogonal to the other two. This is all easy and something most if not all students would know from the courses in mathematics.

As mentioned above a vector is represented by numbers in relation to one specific coordinate system, but in video games we are often working with several coordinate systems depending on the situation. We might have one world coordinate system and several local to the objects in the world, or even local to each of the several meshes a 3D object consists of. Since a vector only is a quantity with magnitude and direction, the vector itself is equal in all coordinate system, but its representation is not. We can think of it as the boiling point of water. Some people say water boils at 212 degrees Fahrenheit while other people say 100 degrees Celsius, but the temperature really is the *same.* There are methods for converting this difference in representation both for temperatures and vectors, but for vectors it is a bit harder to get our head around.

What the students have not learned is how to apply their basic knowledge of vectors to solve more complex problems. How would we determine if a bullet hits its target, when it is starting from a particular position and travelling in a particular direction? A solution here would be to model the bullet with a ray[1] and the target with a bounding sphere, then check the two against each other for collision. But which coordinate system should we use? In this case it would be smart to convert the ray of the bullet from world coordinates to the local coordinate system of the target and go from there.

5.1.2 Matrices

A matrix is a rectangular table (2D) of *elements* (or *entries* as they are often referred to). Matrices are used a lot in game development when working on the visual/graphical part of the game. Because of the properties of a matrix it can be used to very compactly describe geometric translations such as scaling, rotation and translation. In addition they let us change coordinates of points and vectors easily from one coordinate system to another.

Just as vectors we can perform basic operations like adding or subtracting two matrices, or multiplying a matrix by another matrix, a vector or a real number. There are some constraints though: only matrices of equal size can be added together or subtracted and when multiplying we can only multiply a matrix whose number of columns is equal to the number of rows of the other matrix.

There is one special matrix called the identity matrix. This matrix can be looked at as the number 1 for matrices. Multiplying any given matrix by the identity matrix does not change the matrix. The

[1] A ray is represented by a starting point and a vector specifying the direction.

identity matrix is square and all its elements are zeros except the main diagonal where the elements are ones.

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

For some matrices, and only square matrices, there exists an inverse matrix such that if we multiply them together we get the identity matrix. If we have a matrix M, its inverse would be denoted M^{-1}. Because of this property it can be shown (see equation below) that if we multiply a vector A with matrix M and get a vector B, then if we multiply B with M^{-1} we get A again. This is what we call a multiplicative inverse operation and it lets us more easily change from one coordinate system to another and back again, or to reverse geometric translations.

$$P_A M = P_B$$
$$P_A M M^{-1} = P_B M^{-1}$$
$$P_A I = P_B M^{-1}$$
$$P_A = P_B M^{-1}$$

5.1.3 Transformations

We have three fundamental *affine transformations*[2], scaling, rotation and translation. They are all given by 4 x 4 matrices in game programming, while vectors and points are given by 1 x 4 matrices. In vectors the fourth element is a zero because we do not want translation applied to vectors, but for points we do and so the fourth element is a one.

$$[x \quad y \quad z \quad 1] \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ b_x & b_y & b_z & 1 \end{bmatrix} = [x' \quad y' \quad z' \quad 1]$$

When scaling an object we multiply all of its points with the matrix that represents scaling. If we want to undo it we multiply the new point with the inverse of the scaling matrix to get the starting point again. This goes for rotation and translation as well.

Suppose we want to perform both rotation and translation on an object consisting of 50 000 points, we might think we would have to do a total of 50 000 x 2 multiplications. Matrix multiplication however is *associative*[3], meaning the end product is the same no matter what order we perform the multiplications in. This has performance implications depending on how we go about doing the

[2] An affine transformation is a linear transformation plus a translation vector.
[3] Associativity means that the order of operations does not matter as long as the sequence of the operands is not changed.

multiplications. If we have the old and new points P_{old} and P_{new}, the translation matrix T and the rotation matrix R and we do:

$$(P_{old} \times R) \times T = P_{new}$$

We would be doing 100 000 multiplications, but if we instead multiply the transformation matrices together and then multiply each of the points with the product we would only have to do 50 001 multiplications. Like this:

$$P_{old} \times (R \times T) = P_{new}$$

In other words we can combine several transformation matrices into one net transformation matrix, which represents all of the individual transformations combined.

When working with transformations as a video game developer it is very important to keep in mind that matrix multiplication is not *commutative*[4], meaning matrix A multiplied with B is not the same as matrix B multiplied with A. This is easy to imagine if we think of rotation and translation on an object. If the object is centered at start and we apply rotation we would have it rotate around itself (around its axis), and we could then translate it away from center. If we apply translation first however, the rotation would still be around center meaning the object would revolve around the center instead of rotating around itself. Transformations are done with respect to the origin of the coordinate system and so the order in which we apply the transformations matters a lot. Table 5.1 shows transformation matrices for translation, scaling, and rotation, and their inverses.

Table 5.1: Transformation matrices.

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ b_x & b_y & b_z & 1 \end{bmatrix} \qquad T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -b_x & -b_y & -b_z & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad S^{-1} = \begin{bmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad {R_x}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[4] Commutativity is the ability to change the order of something without changing the end result.

$$R_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad {R_y}^{-1} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad {R_z}^{-1} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.2 Basic 3D Concepts in Computer Graphics

We have studied some of the more basic 3D concepts in order to better understand what exactly it is XNA makes so much easier[12, 19, 20]. The following math and illustrations are all based on Direct3D, which uses the left-handed Cartesian coordinate system[5].

5.2.1 Triangle Meshes and Modeling

A 3D model in computer graphics is just a list of triangle meshes which in turn are lists of lots and lots of triangles. These triangles again are all given by three points, each of them called a *vertex*. So if we are to describe an object in a way that the computer understands we need to specify triangles using three vertices, and then define the object by listing all these vertices.

Triangles can be used to model any thinkable shape, but for it to look realistic an immense amount of triangles are required. However, the more triangles, the more processing power is required, and so the models need to be balanced with regards to the end user's computer system.

With so many triangles we need an easy way of managing them. This is mainly done through third party 3D modeling applications like 3D Studio Max[6] and Maya[7]. Listing all of them manually is too cumbersome except for the simplest models like cubes and cylinders, or models that are easily described mathematically. Often vertices are used to hold additional information other then the shape of the model as well. Custom vertices can be made to hold information about color, normal and texture coordinates as well. A normal determines which direction a surface vertex is facing and is used a lot when calculating lighting. Texture coordinates are used when applying a texture to the model.

With so many triangles all connected in a model, there are of course many vertices overlapping. That is they have the same coordinates. More vertices mean increased memory requirements and

[5] The x-coordinate increases horizontally to the right, the y-coordinate increases vertically upwards and the z-coordinate increases "into" the screen.
[6] http://www.autodesk.com/3dsmax
[7] http://www.autodesk.com/maya

processing power performed by the graphics hardware. To minimize this, an indexing technique similar to that which search engines use, where we create two lists, can be used; a vertex list that holds all the unique vertices, and an index list that contains index values into the vertex list that defines how the vertices are put together to form triangles.

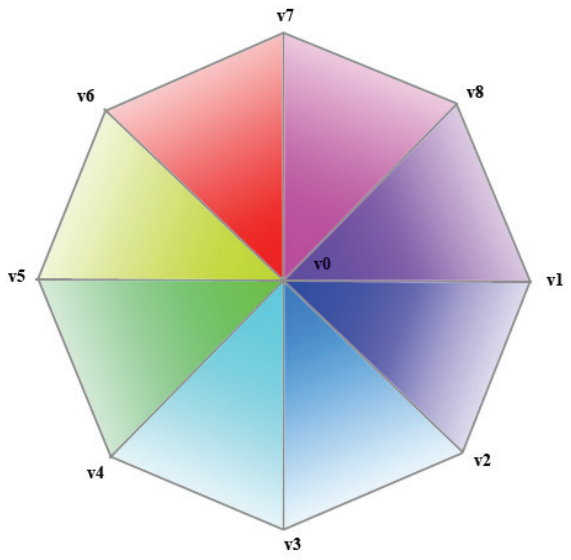


Figure 5.1: A rough circle built by several triangles.

Unique vertices
v0
v1
v2
v3
v4
v5
v6
v7
v8

Index list
0, 1, 2
0, 2, 3
0, 3, 4
0, 4, 5
0, 5, 6
0, 6, 7
0, 7, 8
0, 8, 1

In Figure 5.1, notice how the middle vertex, *v0*, is used by every triangle and the rest of the vertices are used twice. Using the indexing technique we can specify triangles by index numbers and only store nine unique vertices. This is opposed to specifying triangles by vertices and having to store, in this case, 24 vertices. Of course now we moved the duplication over to the index list, but as indices are integers they take up a lot less space than vertex structures.

Calling the above 3D modeling tools easy though is not exactly correct. It is true they do make it a lot easier to create 3D models, but for a student new to them they can be very hard to use and time-consuming to learn.

5.2.2 The Rendering Pipeline

The rendering pipeline is a sequence of necessary steps to display a 2D image on a monitor based on what a virtual camera sees of objects in a 3D scene.

Local and global space

A 3D scene may have a lot of objects that vary in size and detail. Some may even intersect each other, and so it would be real hard to build them properly if we had to build them all relative to the same coordinate system. Instead we can build the object in local space where it is alone and it is the center of the coordinate system. Later when it is finished, we can place it among the rest of the objects in the global scene called world space. This is easily compared to how we would build up a miniature army of tin soldiers. We pick up one uncolored tin soldier, bring it to our workbench and paint it, and then position it on the battlefield where it belongs. The benefits of doing so is that we do not need to pay respects to size, orientation and position relative to the other objects in the world.

Changing from local space to world space is the first step in the rendering pipeline and is called the world transform. This is done by multiplying each vertex with the *world transform matrix*. In order to construct the world transform matrix there are a few things we need to know: the origin, $\vec{p}$, and axis vectors, $\vec{r}, \vec{u},$ and $\vec{f}$ of the local coordinate system relative to the global coordinate system.

$$W = \begin{bmatrix} r_x & r_y & r_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ p_x & p_y & p_z & 1 \end{bmatrix}$$

Complicated as it may look it is really just a net transformation matrix that we talked about in Section 5.1.3. We could also use the identity matrix for the world transformation matrix. That would position the object in the center of the world with no rotation and scaling applied to it.

View space

The virtual camera may be thought of as another object in the 3D scene, and similarly it got its own set of coordinates and orientation. The camera's local space is called the view space, but instead of transforming the camera's coordinates to world space we want to take every object in world space and transform them to view space. Unlike the rest of the objects, the camera is not an object to be displayed. It is an object that helps define which of the other objects that are visible in the final 2D image. The operations needed for this process are easier and more efficient when the camera is the center of the scene, instead of at an arbitrary position and orientation in the world.

The world transform matrix W takes the camera from view space to world space, but we want the opposite and from our discussion on matrix algebra we know that W^{-1} does exactly this. Looking at W we see that this is just a rotation and a translation and if we call the matrix that transform from world space to view space, V, then we get the following:

$$V = W^{-1} = (RT)^{-1} = T^{-1}R^{-1} = T^{-1}R^{T}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_x & p_y & p_z & 1 \end{bmatrix} \begin{bmatrix} r_x & u_x & f_x & 0 \\ r_y & u_y & f_y & 0 \\ r_z & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} r_x & u_x & f_x & 0 \\ r_y & u_y & f_y & 0 \\ r_z & u_z & f_z & 0 \\ -\vec{p}\cdot\vec{r} & -\vec{p}\cdot\vec{u} & -\vec{p}\cdot\vec{f} & 1 \end{bmatrix}$$

At a first glance it does not look pretty, but in reality it is really easy to construct this view transformation matrix. What we need to know is the position of the camera relative to the world space, the point that the camera is looking at in world space and a vector, called the *up* vector, that gives the direction that is considered up in the 3D world (this is usually (0,1,0)).

Given this we can find the unit vectors of the camera relative to the world space and from there fill in our view transformation matrix. We find the z-axis vector ($\vec{f}$ in our view transformation matrix) by normalizing the vector given by the *look at* point and the camera's position. We get the x-axis ($\vec{r}$ in our view transformation matrix) by normalizing the cross product between $\vec{f}$ and the world up vector, $\vec{w}$. The y-axis ($\vec{u}$ in our view transformation matrix) is found by taking the cross product between $\vec{f}$ and $\vec{r}$.

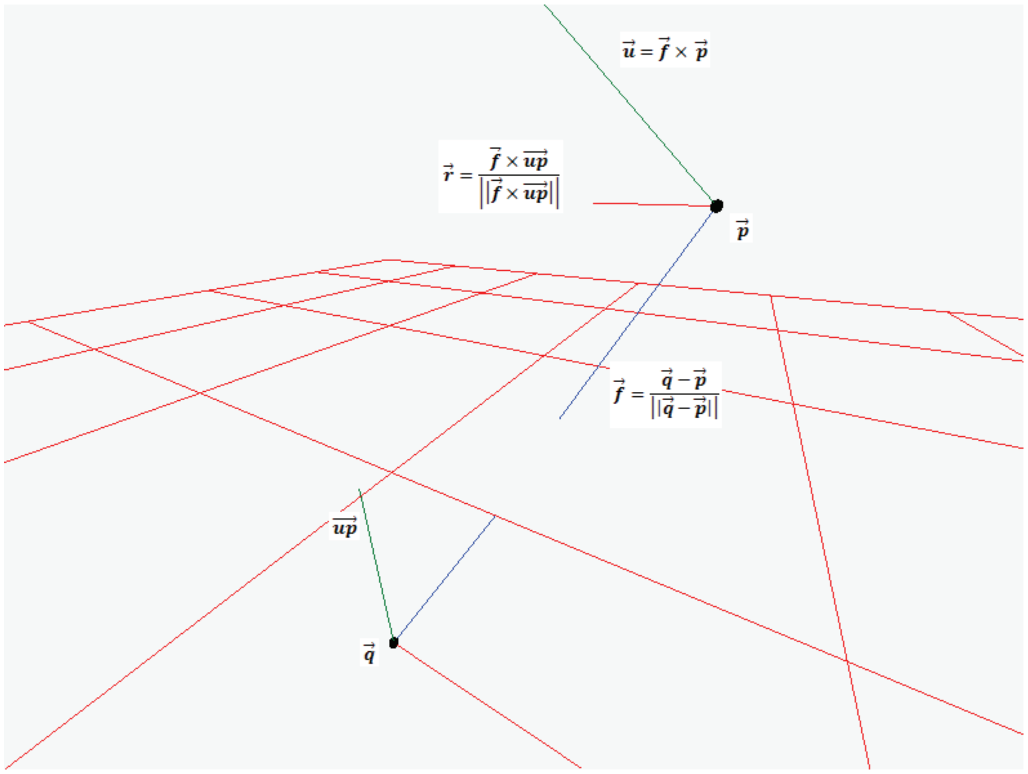

$\vec{u} = \vec{f} \times \vec{p}$
$\vec{r} = \frac{\vec{f} \times \vec{up}}{\|\vec{f} \times \vec{up}\|}$
$\vec{p}$
$\vec{f} = \frac{\vec{q} - \vec{p}}{\|\vec{q} - \vec{p}\|}$
$\vec{up}$
$\vec{q}$

Figure 5.2: Illustration showing how to find the different components of the view transformation matrix.

In Figure 5.2, the camera is located at $\vec{p}$ and looking down along the z-axis (the blue line) towards a position $\vec{q}$ in the world.

Projection space

In projection space the objects have had their coordinates transformed to 2D coordinates, where x and y are in range [-1, 1] and the z coordinates are in the range [0,1]. The x- and y-coordinates represent the 2D projected vertices, while the z-coordinate is used for depth buffering.

Needless to say the objects of a scene might be positioned anywhere around the camera (e.g. behind it), and these objects will not be displayed on screen. To determine which objects the camera does see, first thing we need to do is define something called a *frustum* that defines a viewing volume in the 3D world. The frustum is a contracted cone with its top in the center of the camera and it expands down the camera's z-axis (see Figure 5.3). We define a near-plane and a far-plane and the volume between these two planes is the volume that the displayed objects need to be within. The planes are given by a distance from the camera's center, the vertical field of view angle and the aspect ratio. Everything else we need to know can be derived using trigonometry.

The line from a point (x, y) in the frustum to the center of the camera intersects the near-plane in a point (x′, y′). The intersection point is found using the properties of similar triangles in trigonometry. After normalization for x and y to the range [-1, 1] and z to the range [0, 1], these are the coordinates passed along to the next step in the rendering pipeline. However, we are not going into more details about the normalization process as it falls a bit outside the scope of our report, but refer to [12] for the interested reader.

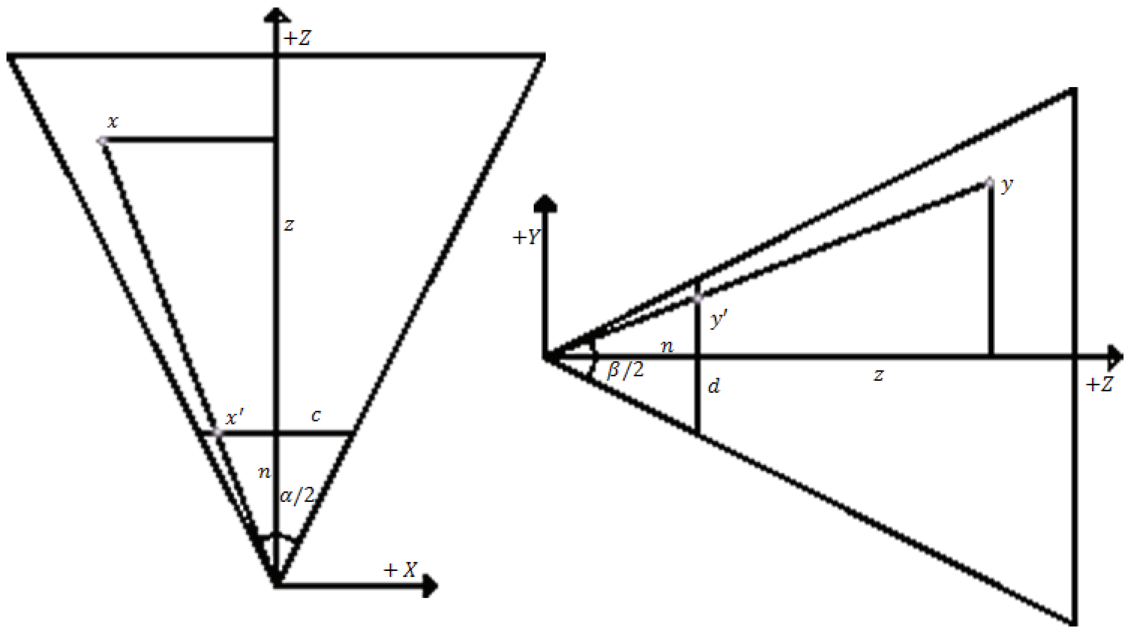


Figure 5.3: The frustum, showing how a vertex gets projected onto the near-plane.

Similarly to the previous steps there exists a *perspective projection matrix* that does the projection and normalization for us, to increase speed and efficiency. Actually this is a two step process were we first multiply the x, y and z coordinates of the vertex with the perspective projection matrix to get coordinates in something called *projection space*, or *homogenous clip space*. In this space some operations like back face culling and clipping can be done in a simplified manner. Next step is called the *perspective divide*, or the *homogenous divide*, where we divide the resulting vector from the first step by *z*. The output from the last step is called normalized device coordinates, and completes the projection process.

$$[x, \quad y, \quad z, \quad 1]\begin{bmatrix} \frac{1}{R\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{-fn}{f-n} & 0 \end{bmatrix}$$

$$= \left[\frac{x}{R\tan\left(\frac{\alpha}{2}\right)}, \quad \frac{y}{\tan\left(\frac{\alpha}{2}\right)}, \quad \frac{zf}{f-n} - \frac{fn}{f-n}, \quad z\right] = [z\tilde{x}, \quad z\tilde{y}, \quad z\tilde{z}, \quad z]$$

Above is the first step from view space to projection space. α refers to the vertical field of view angle, n and f are the distance to the near and far plane respectively in view space dimensions and R is the aspect ratio. Following comes the perspective divide. Notice however that the original z coordinate from view space is now called w which goes by the name of the *homogenous coordinate*.

$$\frac{1}{w}[w\tilde{x}, \quad w\tilde{y}, \quad w\tilde{z}, \quad w] = \left[\frac{x}{zR\tan\left(\frac{\alpha}{2}\right)}, \quad \frac{y}{z\tan\left(\frac{\alpha}{2}\right)}, \quad \frac{f}{f-n} - \frac{fn}{zf-n}, \quad 1\right]$$

Backface culling

Backface culling is an operation performed in perspective space before the perspective divide. Its function is to remove geometry data from the scene that would not be displayed anyway because of being obscured by other geometry. This is done by knowing which parts of an object are facing towards the camera, and which are facing away. The parts that are facing away can be discarded since they are obscured by the front facing parts anyway.

More specifically, triangles with vertices specified in a clockwise winding order in view space are front facing, while the rest are back facing and can be discarded.

It is not unusual that about half of the triangles in a scene are back facing, and by discarding them from further processing we can greatly increase performance. Notice however that this technique does not work if we have transparent objects in the scene, or if the camera is allowed to be positioned inside objects.

Clipping

Clipping is another operation that takes place in the perspective space before the perspective divide. Its function is to remove geometry data that falls outside the frustum. A triangle can be inside the frustum, outside the frustum or in between. Basically we discard the triangles outside, keep the ones that are inside and we clip the triangles that are partially inside and outside.

This is done by checking the coordinates against the boundaries of the frustum which after the normalization are x and y in range [-1, 1] and z in the range [0,1]. However, since clipping is done before the perspective divide we need to multiply the limits with the homogenous coordinate w first. And so x and y needs to be in range [-w, w] and z in the range [0, w].

Viewport transform

To explain the viewport we need to know a little about something called the back buffer and the front buffer. The front buffer holds the current frame that is being displayed, while the next frame that is going to be displayed is sent to the back buffer. This process is called *presenting* and it solves a problem with the monitor's refresh rate versus the application's frame rate. When the monitor is ready to display a new frame, the back buffer gets promoted to front buffer and the front buffer acts as the back buffer instead.

If we think of the back buffer as every pixel on the screen, then a viewport is an area defined by a rectangle on this back buffer. Our application can even use several viewports to render subsections of the back buffer.

From the perspective divide we were left with normalized device coordinates for the objects that we wanted displayed. Next thing we need to do is transform these coordinates into our viewport on the back buffer, which is just the matter of calculating the screen coordinates for x and y when we know where the viewport start and we know its height and width.

Notice that the back buffer only stores the x and y components, but that there may be several vertices that have the same x- and y-value but different z-value. How then do we determine which vertex to display? This is done using another buffer called the depth-buffer, which contains information about the depth value of every pixel in the back buffer. If two vertices want to be in the same pixel on screen, then the one that is closest, that is the one with the least depth-value, is chosen.

Rasterization

Finally we have a list of 2D triangles but we are missing information about every pixel inside and at the boundaries of these triangles. The last part of the rendering pipeline is called rasterization, which is the process of taking an image described in vectors graphics and converting it into pixels. Basically it is how we fill our triangles.

Texturing, pixel shaders, depth buffering and alpha blending occur in the rasterization stage but we are not going to go into details about all this. The most important thing to know here is how we determine the color of a pixel within one of these triangles.

A triangle is defined by 3 vertices and so that is 3 pixels accounted for, but what about the pixels in the middle of the triangle. What color do they get? Interpolation is the answer to that question, and with it we can calculate the color values for every remaining pixel. Texture coordinates and vertex normals can also be interpolated as well as depth values.

To explain quickly what interpolation is, we imagine a stick with one end black and the other end white. The color of the stick in the middle would, with interpolation, be shades of grey depending on how far from black, or close to white, we are (see Figure 5.4).

Figure 5.4: Interpolation between black and white

After the rasterization stage our back buffer is filled with colored pixels, or in other words, our back buffer is now an image ready to be displayed.

5.2.3 Shaders

A shader can be seen as a program run by the GPU to perform rendering effects such as lightning. Assembly was the main programming language used for writing these shaders in the start, but as the hardware capabilities increased, the need for higher level shader languages emerged. Today, there are plenty of them such as GLSL (OpenGL Shading Language), NVidia’s Cg language and HLSL (High-Level Shader Language) developed by Microsoft and used by XNA and DirectX [20]. HLSL is the one we have studied.

When we write a shader in HLSL, we use a C-like syntax to produce code that is compiled by Direct3D to the specified assembly version, and then executed by the GPU for each vertex or pixel that a mesh occupies on the screen. There are three types of shader programs; vertex shaders, geometry shaders, and pixel shaders. Figure 5.5 shows the Direct3D 10 Graphics Pipeline, which illustrates how the different shader types are invoked in the process of bringing pixels to the screen.

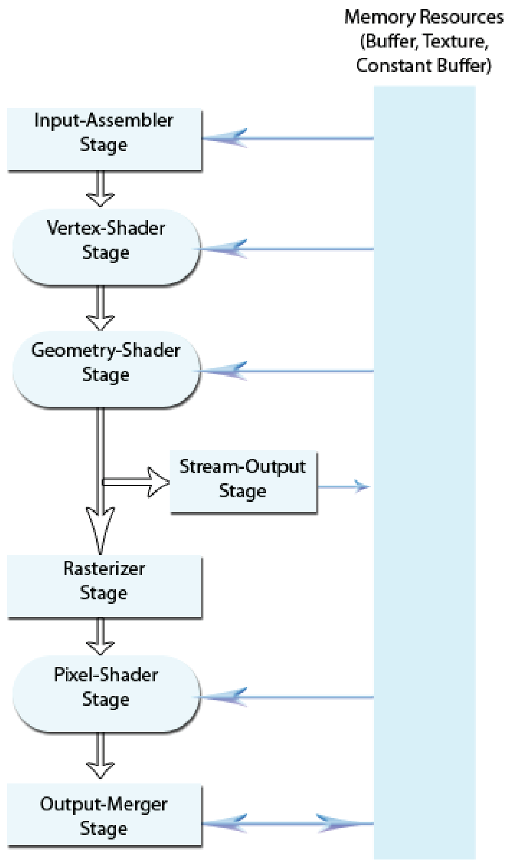


Figure 5.5: The Direct3D 10 Graphics Pipeline.

Vertex Shaders

Vertex shaders are executed one time for each vertex that is going to be displayed on the screen. The main purpose of this shader is to transform the vertex from its local object-space to screen-space, which are the first few stages in the rendering pipeline (see Section 5.2.2). This is a three step process where we first transform the vertex's position from object-space to world-space, then we take it from world-space to view-space, and finally from view-space to screen-space. Three vector-matrix multiplications for each vertex seems a lot, but as mentioned above in Section 5.1.3, we can reduce the number of multiplications by having one matrix called the World-View-Projection matrix that holds the net transformations needed to go directly from object-space to screen-space. Returning a transformed position is the absolute minimum a vertex shader needs to do.

Vertex shaders give us complete control over the vertices before they are displayed on the screen, and so we can do a lot of stuff with them to create different effects. For instance facial expressions on a game character or water can be animated using vertex shaders. These things are often more easily created using shaders, and it frees up processing power by executing on the GPU instead of the CPU. In addition GPUs are generally faster than CPUs when it comes to mathematical operations.

Geometry shaders

Geometry shaders are a relative new addition to the graphics pipeline, and were first introduced to HLSL with DirectX 10. Their function is to add or remove vertices from a mesh, and so they can be used to generate geometry or add detail to existing meshes. Unlike vertex shaders, which operate on single vertices, the geometry shader operates on all the vertices of a full primitive (a point, a line, or a triangle). Areas of use include particle systems, fur generation, and shadow volume generation [19].

Pixel shaders

Pixel shaders calculate the color of individual pixels usually to create some effect on an image like shadows, lighting, explosions, realism and bump mapping. It operates on a per-pixel basis so depending on screen resolution the pixel shader gets executed millions of times each frame. Frame rates of between 30 and 60 frames per second are considered minimally acceptable by some people.

The inputs to the pixel shader are bound to the outputs from the vertex shader or the geometry shader if it is used. The color of a pixel is usually interpolated using the three surrounding vertices and this is the color we will see on the screen. The pixel shader can leave the color data coming from the preceding stage as is, or it can perform different operations to alter the color, but at the very minimum it needs to output some color data.

5.3 3D In XNA

The whole idea behind XNA is to make game development easier, which is why Microsoft has abstracted away much of the graphics concepts discussed in Section 5.2. In most cases when working with graphics in XNA, we do not need to know *how* to do things like matrix multiplication and setting up a projection matrix, but knowing the *when* and *why* is still needed and very important.

XNA builds on DirectX 9 but unlike DirectX it uses the right-handed Cartesian coordinate system[8].

5.3.1 Vectors and Matrices

XNA provides structures for vectors and matrices that include methods that perform most if not all of the basic operations. If we want a rotation matrix we only need to write one line where we specify the axis of rotation and degrees to rotate. If we then want to rotate a point using this rotation matrix, we create a vector holding the position and transform it using the rotation matrix:

```
Matrix rotation = Matrix.CreateRotationY(MathHelper.ToRadians(180f));

Vector3 position = new Vector3(1f, 0f, 0f);

Vector3 newPosition = Vector3.Transform(position, rotation);
```

The code above rotates a point by 180 degrees around the Y axis. This is XNA's way of doing the following:

$$\begin{bmatrix}1 & 0 & 0 & 1\end{bmatrix} \times \begin{bmatrix}\cos\pi/2 & 0 & -\sin\pi/2 & 0\\ 0 & 1 & 0 & 0\\ \sin\pi/2 & 0 & \cos\pi/2 & 0\\ 0 & 0 & 0 & 1\end{bmatrix} = \begin{bmatrix}1\times\cos\pi/2 & 0 & 0 & 1\end{bmatrix} = \begin{bmatrix}-1 & 0 & 0 & 1\end{bmatrix}$$

Similarly there are classes and structures to handle bounding boxes, planes, points, quaternions and rays as well as a math helper.

5.3.2 Effects

Effects are text files written in HLSL that combine vertex shaders, pixel shaders, geometry shaders and graphic device states. Below is a real basic vertex and pixel shader that only renders white geometry:

[8] The x-coordinate increases horizontally to the right, the y-coordinate increases vertically upwards and the z-coordinate increases "out" of the screen.

```
float4x4 mWorldViewProj;  // World * View * Projection transformation

float4 Vertex_Shader_Transform(
    in float4 vPosition : POSITION ) : POSITION
{
    float4 TransformedPosition;

    // Transform the vertex into projection space.
    TransformedPosition = mul( vPosition, mWorldViewProj );

    return TransformedPosition;
}

float4 Pixel_Shader() : COLOR0
{
    return float4(1,1,1,1);
}

technique ColorShaded
{
    pass P0
    {
        VertexShader = compile vs_1_1 Vertex_Shader_Transform();
        PixelShader  = compile ps_1_4 Pixel_Shader();
    }
}
```

There are two different types of input to this effect. The first is the world-view-projection matrix declared in the start as a float4x4 type (a 4x4 matrix of floating-point values), which is called an *effect parameter*. The effect parameters usually remains constant for each vertex and pixel processed, and they need to be set by the application itself. They hold data like transformation, light and material. The second input type is the varying data that is different for each execution. In the example above the vPosition argument holds the position of the vertex being processed. This data comes from the vertex buffer.

As discussed in Section 5.2.3, a vertex shader needs at the very minimum to return a transformed position. The position received from the vertex buffer is the position of the vertex in model-space (local to the model), so by multiplying that position with the world-view-projection matrix we get the screen-coordinates of the pixel representing that vertex.

In the example above the pixel shader does not take any inputs and simply returns a color white, as such everything drawn with this effect is completely white. However it could have gotten more information about the vertex color and position to create different effects or to simply return the interpolated vertex color instead.

The same effect file may contain several techniques that renders differently. One technique might include lighting, while the other renders everything a shade of red. The techniques can also contain several passes so the shader gets executed several times for each vertex and pixel to allow more complex functions.

```
//Initialize the parameter
Effect exampleEffect = content.Load<Effect>("ExampleEffect");
EffectParameter worldViewProjParameter =
  exampleEffect.Parameters["mWorldViewProj"];

Matrix worldViewProj = Matrix.Identity *  //world transform
  Matrix.CreateLookAt(                  //view transform
      new Vector3(0f, 0f, -10f),
      Vector3.Zero,
      Vector3.Up) *
  Matrix.CreatePerspectiveFieldOfView(  //projection transform
      MathHelper.PiOver4,
      1.333f,
      0.1f,
      100f);

//Set the world-view-projection matrix
worldViewProjParameter.SetValue(worldViewProj);
```

The example above is XNA code that shows how we can set the value of the effect parameter for the world-view-projection matrix in the effect example. First we load the effect file, then we create an instance of the EffectParameter class where we give the effect parameter by name, and then we use this instance to set the value of our game's world-view-projection matrix.

5.3.3 3D Models

Suppose we have a ready 3D model created with some 3D modeling tool that we want to use in our XNA game. First we would need to export this model to the .fbx or .x file format. When that is done we can easily add the model to our game project as an asset. If the filename were "ship.fbx" the asset name would be "ship". XNAs content manager manages all the content such as models, effects and textures, and by using its load method we can easily load our 3D model from file to XNA's model class by specifying its asset name.

```
Model gameShip;
gameShip = Content.Load<Model>("ship");
```

The model class stores all the data we need for drawing the model on screen. As discussed in Section 5.2.1, this includes vertex positions, normal data, color info, and texture coordinates. More specifically the XNA model class is made up of a vertex buffer, an index buffer and multiple model mesh objects (see Figure 5.6). If the model is a human body, then one model mesh could be an arm while another a nose. The model meshes contain references to the index buffer indicating the vertices it consists of, one or more effects and, texture coordinates, if any [21].

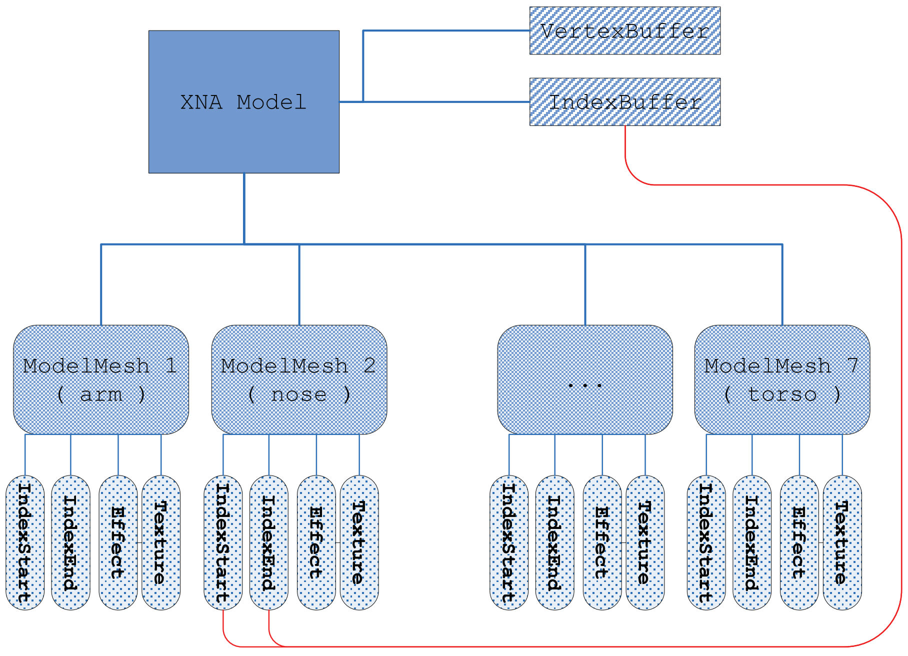


Figure 5.6: Illustration showing the components of an XNA model.

In order to actually render the model on screen in XNA, we do not need to know exactly how all this is put together. XNA supports the use of a basic effect called *BasicEffect* that includes lighting, fog and a single texture that makes rendering real easy. We can pretty much draw every model we have using the same basic method. Doing so will not exactly make our game look fantastic and realistic, but this approach lets us see some results quickly without the need to study all sorts of game graphics topics for months first. Later we can start adding custom effects to polish the game, when we feel it is time to start learning more about rendering models.

```
public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
  Matrix[] transforms = new Matrix[model.Bones.Count];
  model.CopyAbsoluteBoneTransformsTo(transforms);

  foreach (ModelMesh mesh in model.Meshes)
  {
    foreach (BasicEffect effect in mesh.Effects)
    {
      effect.EnableDefaultLighting();

      effect.View = view;
      effect.Projection = projection;
      effect.World = transforms[mesh.ParentBone.Index] *
                     Matrix.CreateScale(scale) *
                     Matrix.CreateRotationX(Rotation.X) *
                     Matrix.CreateRotationY(Rotation.Y) *
                     Matrix.CreateRotationZ(Rotation.Z) *
                     Matrix.CreateTranslation(Position);
    }
    mesh.Draw();
  }
  base.Draw(gameTime, spriteBatch);
}
```

Above is the simple draw method we have implemented in XQUEST. As we can see the only things one need to input to this method is a *model* object, a *view* and a *projection* matrix and some game logic data like the model's *position*, *scale* and *rotation* in the world.

The benefit of XNA here is that the knowledge of what is going on behind the curtain is not a pre-requisite for displaying the model. We can learn it later on if we so choose or we can leave it and focus on other aspects of the game such as architecture and game logic.

5.3.4 Rendering Objects on Screen

As discussed in Section 5.2.2 regarding the rendering pipeline, there are a lot of processes to go through if we have a 3D model and want to display it on screen. Traditionally we had to know all about these processes and implement many of them ourselves using graphics APIs like DirectX and OpenGL. This is usually a huge bottleneck for students and hobbyists who may have great ideas for a game, but not necessarily the math and programming skills to create it themselves [22]. XNA handles many of these rendering processes for us, making the job easier.

Say we have a 3D model and we have loaded it into our game project using XNA's content manager. Now we need to set up a sort of camera to render the model. The following code is actually all we need in order to render a model.

```
private void DrawModel(Model m)
{
  Matrix[] transforms = new Matrix[m.Bones.Count];
  float aspectRatio = graphics.GraphicsDevice.Viewport.Width/
                      graphics.GraphicsDevice.Viewport.Height;
  m.CopyAbsoluteBoneTransformsTo(transforms);

  // sets up the projection-matrix
  Matrix projection =
    Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(45.0f),
                                        aspectRatio, 1.0f, 10000.0f);

  // sets up the view-matrix
  Matrix view =
    Matrix.CreateLookAt(new Vector3(0.0f, 10.0f, 10f), Vector3.Zero, Vector3.Up);

  foreach (ModelMesh mesh in m.Meshes)
  {
    foreach (BasicEffect effect in mesh.Effects)
    {
      effect.EnableDefaultLighting();

      effect.View = view;
      effect.Projection = projection;

      // creates world-matrix
      effect.World = Matrix.CreateRotationY(MathHelper.Pi/2)*
                     transforms[mesh.ParentBone.Index]*
                     Matrix.CreateTranslation(new Vector3(2, 5, -3));
    }
    mesh.Draw();
  }
}
```

In the code above, we use two important methods of XNA to create the view and the projection matrices. As we know from Section 5.2.2 they play a major role in the process of rendering. Creating and using them in XNA is pretty straight forward as long as we know what kind of input parameters they expect.

CreatePerspectiveFieldOfView sets up the frustum that defines the viewing volume of the camera. Its inputs are; the angle called field of view that defines how wide a view the camera can see, the aspect ratio of the game window, and the distance to the near- and far-plane.

CreateLookAt sets up the view matrix, and in order to do that it needs to know a position in the world that the camera is looking directly at, the position of the camera, and what is considered *up* in the world (usually the y-axis in the world pointing up).

The third matrix, the world matrix, we can create with a static functions of the *Matrix* class like *createRotationY* which returns a matrix that rotates a vertex around the y-axis. In the code above we place the model in the world with a rotation of 180 degrees around the Y axis, and with its center at coordinates (2, 5, -3).

The matrices are passed along to the effect *BasicEffect* which is a shader program that handles the rest for us.

Part III
Survey of the Student Projects

In this part, we present a survey conducted on the basis of the execution of the student projects in the software architecture course. More information about the student projects can be found in Section 1.2. The survey has two parts; a general part in which we evaluate the execution of the XNA and robot student projects using a comparative approach, and an XQUEST specific part where we evaluate the usage of XQUEST in the XNA game projects.

6. Goals and Purpose

The survey will be used to answer our first two research questions. We want to find out to which degree students who chose the XNA game project managed to achieve the learning goals of the course and compare with the students who chose the robot simulation project (**RQ1**). We also want to measure the usefulness and usability of XQUEST (**RQ2**). Thus, the goals of the survey are as follows:

- **G1** Compare the students who chose the XNA game project and the students who chose the robot simulation project with regards to:
    - **G1.1** To which degree the COTS influenced
        - the requirement gathering and specification.
        - the design of the architecture.
        - the ATAM evaluation.
        - the architectural drivers.
        - the quality attribute focus.
    - **G1.2** The relationship between time spent on technical matters and architectural matters.
    - **G1.3** How difficult it was to
        - integrate known architectural and design patterns.
        - learn the necessary prerequisite skills to be able to develop programs.
    - **G1.4** How much the students feel they have learned about software architecture through the project.
    - **G1.5** The overall happiness with the project.

- **G2** Find out the usefulness and usability of XQUEST.

- **G3** Find out how the students used XQUEST in their project.

7. Method

The survey is based on the survey method of conducting an empirical investigation as described in Section 2.2.3. However, in addition to using questionnaires to gather data, we have also been able to collect qualitative data through our positions as student assistants in the Software Architecture course.

We have applied recognized methods combined with our own subjective assessments to the implementation of the survey. This chapter describes these methods.

7.1 Likert Items

Measureable items on the questionnaires have been formed as Likert[23] items. These are not questions, but statements that the respondents respond to by specifying their agreement to the statements. We have used 5-level Likert items, where the levels of agreement are:

- Strongly Disagree

- Disagree

- Neither Agree Nor Disagree

- Agree

- Strongly Disagree

7.2 System Usability Scale

The System Usability Scale[24], hereafter SUS, is a usability questionnaire consisting of ten generic Likert items. Responses to the questionnaire result in a score, called the SUS score, a single number between 0 and 100 indicating the overall usability of the system being studied. We have incorporated the SUS items in our XQUEST questionnaire. We provide an explanation of how a SUS score is calculated in Section 11.3.

7.3 Subjective Assessments

The items from the questionnaires that are not part of the SUS will be assessed subjectively. This means that a non-formal explanation will be given to the results of these items with regards to the goals of the survey.

8. Strategy and Setup

We have used two questionnaires. The first questionnaire consists of general questions about the course project that can be responded to by all students regardless of choice of project. This will form the basis for a comparative evaluation. In addition, we wanted the students who chose to use XQUEST in their XNA game project to answer a separate questionnaire. This questionnaire incorporates the ten generic SUS items in addition to several other items used to query the usefulness of XQUEST.

9. Participants and Environment

The participants of our survey are the students of the course TDT4240 Software Architecture. Like every other student of NTNU they use the online e-learning platform it's learning more or less every day to keep up to date on the different courses they are participating in. In total there were 93 registered students to the course. However, six of them did not participate in a group that made a final project delivery at the end of the course, making the number of students that were eligible as respondents to our questionnaires 87.

Since all students had access to it's learning, this proved an ideal place for us to publish our questionnaires. The time for publishing the questionnaires is an important issue. We found that the questions are best asked after the students have finished the implementation phase, because in the next and last part of the project the students have to conduct a mandatory post mortem analysis. During these days the students have to reflect on their project, and we are more likely to get better and more sincere responses from them. We were also allowed to make the questionnaires a mandatory part of the project, to reduce the risk of not having enough data to reasonably evaluate the student projects and XQUEST. To further reduce this risk, we decided that we would publish the questionnaires as a single questionnaire, but with different sections to answer depending on which project the student chose and whether or not he or she used XQUEST. The primary concern was that of students not willing to answer several questionnaires. We fear some students might overlook one questionnaire had we split them into several separate questionnaires. Besides that, to make sure we would get a sufficient amount of data, we would have had to make all questionnaires mandatory to answer, something that would be impossible since, for example, the robot students were not supposed to answer the XNA/XQUEST questionnaire.

We published our questionnaires on it's learning using the survey functionality on the 21st of April, three days after the delivery deadline for the projects. The questions were divided into three sections within the same survey, and each question was prefixed with a context name indicating which section it belonged to. We added a note in the description box, instructing the students to only answer the part of the survey that was eligible to them. The three sections of the survey were as follows:

- A general part that was to be answered by all students.
- An XNA-specific part that was to be answered by students who worked on an XNA game project.
- An XQUEST-specific part that was to be answered by students who in any way used XQUEST in their XNA game project.

This meant that the robot students only had to answer the general part of the survey, while the XNA students, depending on whether or not they used XQUEST, had to answer a higher percentage of the questions.

The participants and environment is authentic in the sense that the students of the course are the intended users of XNA and XQUEST.

10. The Questionnaires

For reference purposes, this chapter presents the questionnaires we developed. The items on the questionnaires were derived from the goals that we identified in Chapter 6.

10.1 General Questionnaire

This questionnaire consisted of twelve general items that were to be responded to by both the robot and XNA students. Table 10.1 lists the twelve general items and the alternative choices associated with them.

Table 10.1: The twelve general items labeled Q1-Q12.

Item	**Alternatives**
Q1: My group worked with the following COTS[9]	1. XNA 2. Robot
Q2: My group focused on the following quality attribute	1. Testability 2. Modifiability 3. Safety (robot only)
Q3: I found it hard to come up with good requirements	(5-level Likert item)
Q4: I think the COTS did not hinder the design of a good architecture	(5-level Likert item)
Q5: I found it difficult to evaluate the other group's architecture in the ATAM	(5-level Likert item)
Q6: I think the COTS made it easier to identify architectural drivers	(5-level Likert item)
Q7: I found it difficult to focus on our assigned quality attribute	(5-level Likert item)
Q8: I found it easy to integrate known architectural or design patterns	(5-level Likert item)
Q9: I spent more time on technical matters than on architectural matters	(5-level Likert item)
Q10: I spent too much time trying to learn the COTS in the start	(5-level Likert item)
Q11: I have learned a lot about software architecture during the project	[Yes/No]
Q12: I think I would have chosen the other project if I could go back in time	[Yes/No]

[9] "COTS" corresponds to either the XNA API or the robot simulator API.

10.2 XNA and XQUEST-specific Questionnaire

The XNA-specific items were to be answered by all XNA students. The XQUEST-specific items were to be answered only by those who in any way used XQUEST in their XNA game project. This included the ten generic items from the SUS. Table 10.2 shows the XNA-specific items, while Table 10.3 shows the XQUEST-specific items and the ten SUS items.

Table 10.2: The two XNA items labeled X1-X2.

Item	**Alternatives**
X1: My game was a 3D game	1. True 2. False
X2: I spent too much time developing the gameplay and not enough time on the architecture	(5-level Likert item)

Table 10.3: The eight XQUEST-specific items (labeled XQ1-XQ8) plus the ten generic SUS items (labeled XQSUS1-XQSUS10).

Item	**Alternatives**
XQ1: I used the following components of XQUEST	• Sprite/AnimatedSprite • InputManager • AudioManager • GameObjectManager • TextOut • TextureStore
XQ2: I found the XQUEST documentation lacking	(5-level Likert item)
XQ3: I found that I could use XQUEST as it is without modifications	(5-level Likert item)
XQ4: I spent too much time looking into the XQUEST code	(5-level Likert item)
XQ5: I think XQUEST helped me focus more on architectural matters, and less on technical matters	(5-level Likert item)
XQ6: I think XQUEST saved me a lot of time and effort by providing components and functionality that I otherwise would have had to create myself	(5-level Likert item)
XQ7: I think there was components missing, that most students could have benefitted from	(5-level Likert item)
XQ8: If you felt there were components missing, which ones would you like to see in a future version of XQUEST?	(Open question)
XQSUS1: I think that I would like to use this system frequently	(5-level Likert item)
XQSUS2: I found the system unnecessarily complex	(5-level Likert item)
XQSUS3: I thought the system was easy to use	(5-level Likert item)
XQSUS4: I think that I would need the support of a technical person to be able to use this system	(5-level Likert item)
XQSUS5: I found the various functions in this system were	(5-level Likert item)

well integrated	
XQSUS6: I thought there was too much inconsistency in this system	(5-level Likert item)
XQSUS7: I would imagine that most people would learn to use this system very quickly	(5-level Likert item)
XQSUS8: I found the system very cumbersome to use	(5-level Likert item)
XQSUS9: I felt very confident using the system	(5-level Likert item)
XQSUS10: I needed to learn a lot of things before I could get going with this system	(5-level Likert item)

11. Results

This chapter presents the results from the survey. In Chapter 12, we present an in-depth evaluation of the results with regards to the stated goals and expectations.

The survey was conducted as a single questionnaire answering, where the questionnaire was divided into three sections, each section with items to be responded to by a selection of students depending on which project they had chosen to work with and whether or not they had used XQUEST. As such, the data gathered from this single questionnaire was split into three bulks before extracting the statistics.

- A robot bulk, consisting of responses from those students who did the robot project.
- An XNA bulk, consisting of responses from those students who did the XNA game project.
- An XQUEST bulk, consisting of responses from those students who had used XQUEST in their XNA game project.

By separating out the robot and XNA data, we enabled the evaluation of the responses to the general questionnaire on a comparable level, and put ourselves in a situation to be able to study any differences in the perception of the robot and XNA projects.

11.1 General Questionnaire

This questionnaire included ten general items to be responded to by all students, regardless of their choice of project. We received a total of 67 responses to the general questionnaire.

Figure 11.1 to Figure 11.24 presents the results from the general questionnaire.

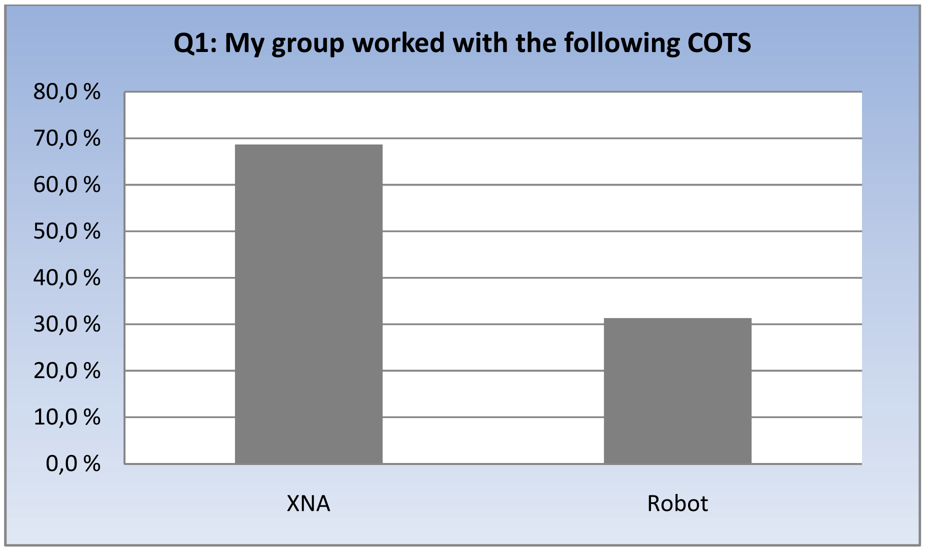


Figure 11.1: Column diagram showing results from Q1.

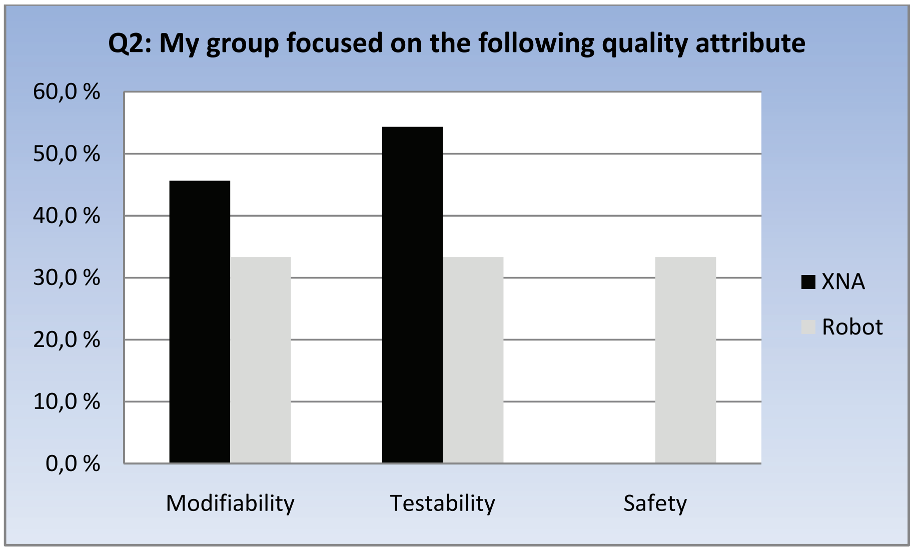


Figure 11.2: Column diagram showing results from Q2.

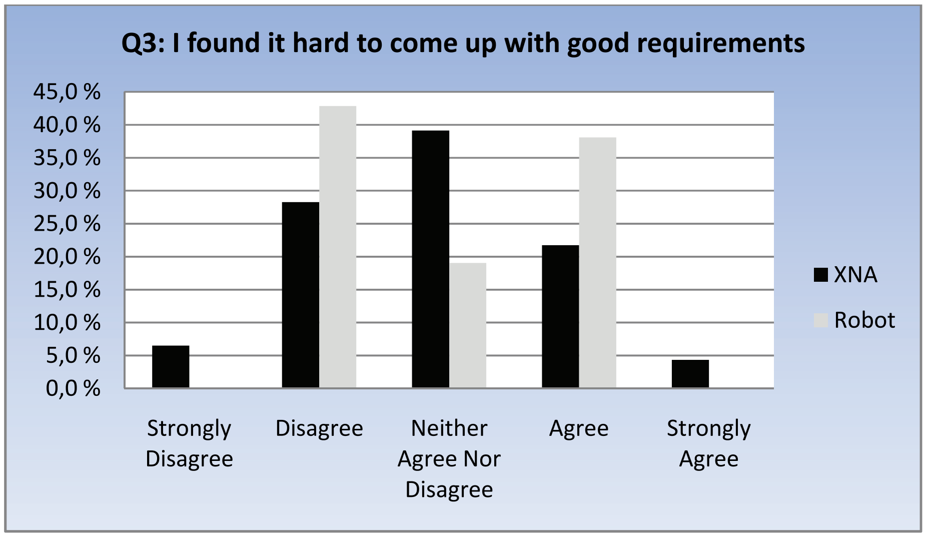


Figure 11.3: Column diagram showing results from Q3.

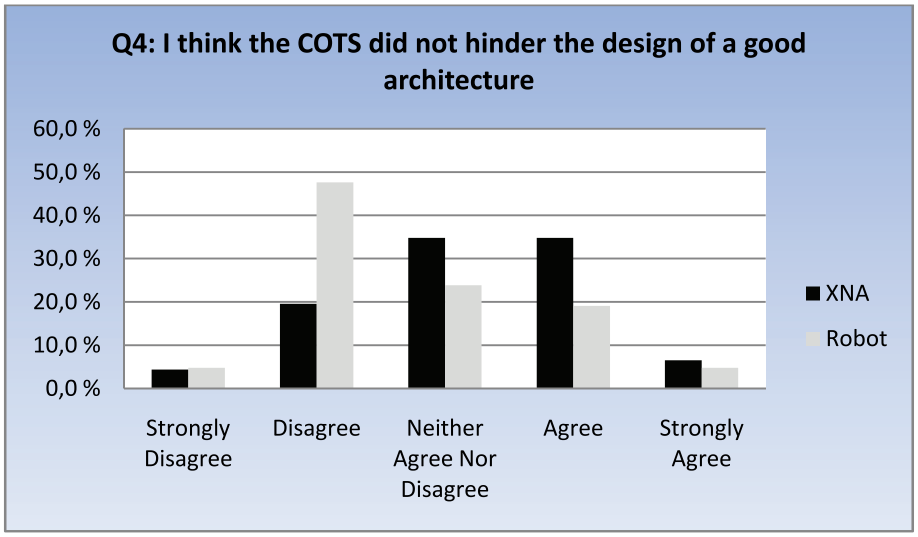


Figure 11.4: Column diagram showing results from Q4.

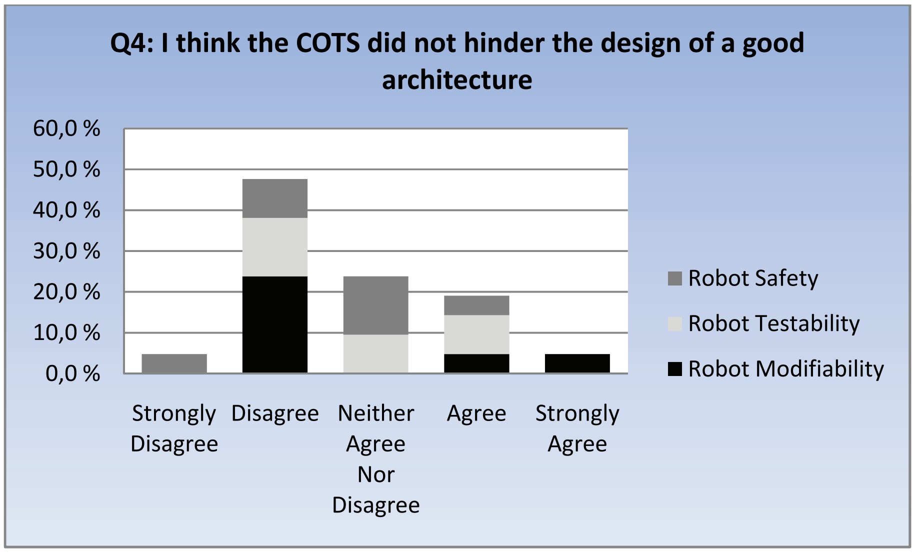


Figure 11.5: Stacked column diagram showing the quality attribute distribution for robots from Q4.

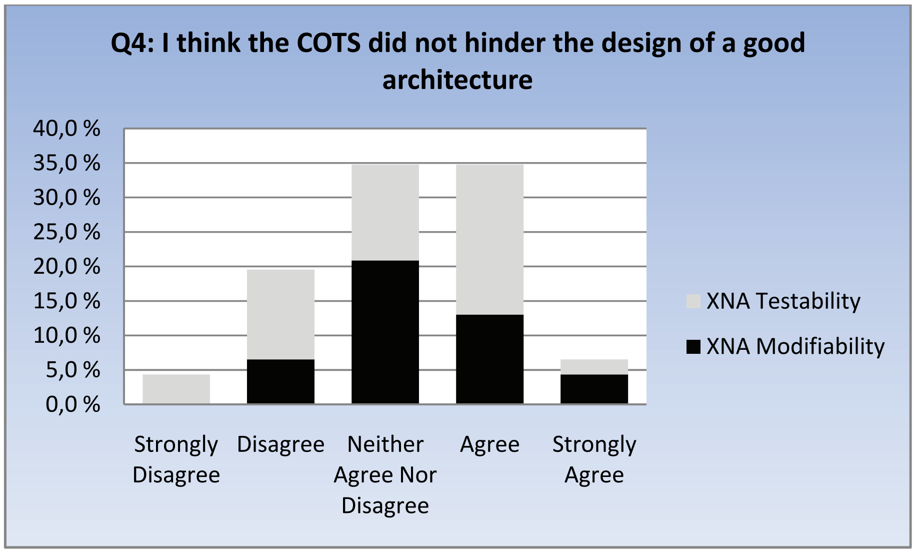


Figure 11.6: Stacked column diagram showing the quality attribute distribution for XNA from Q4.

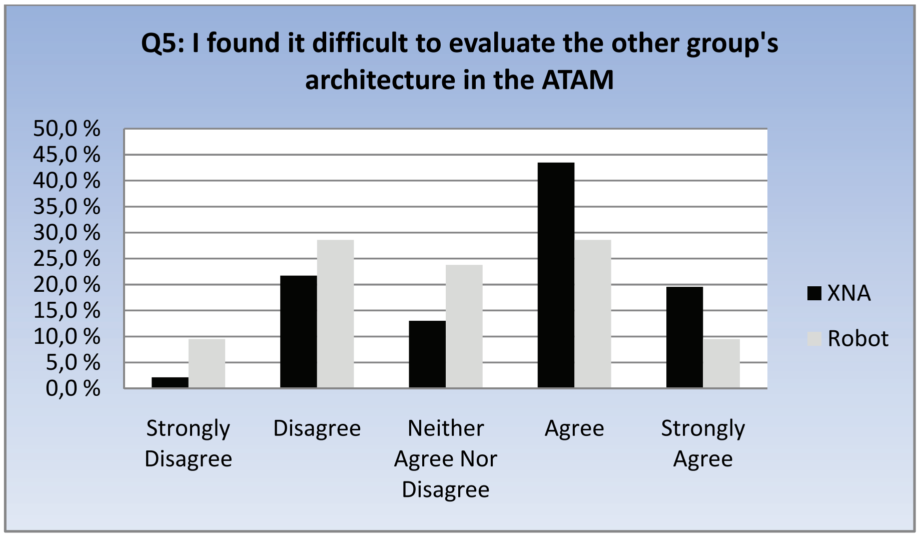


Figure 11.7: Column diagram showing results from Q5.

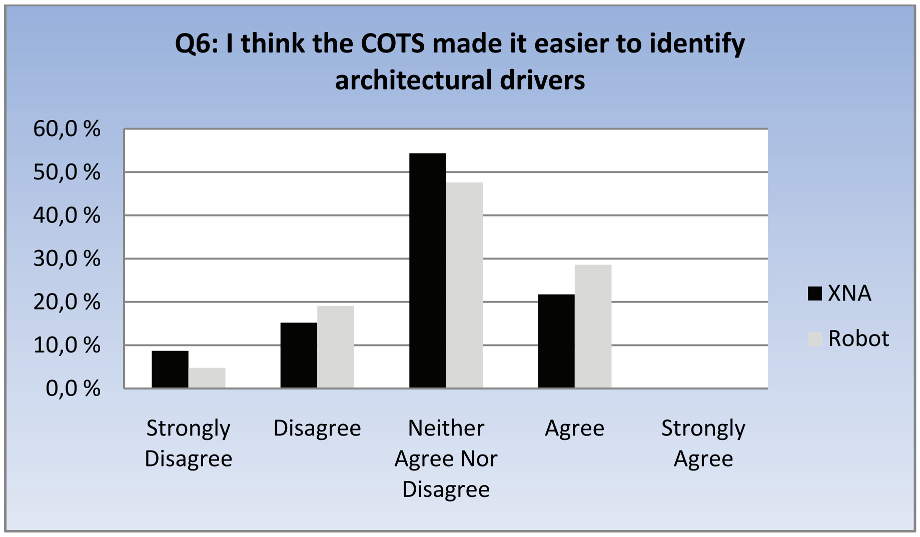


Figure 11.8: Column diagram showing results from Q6.

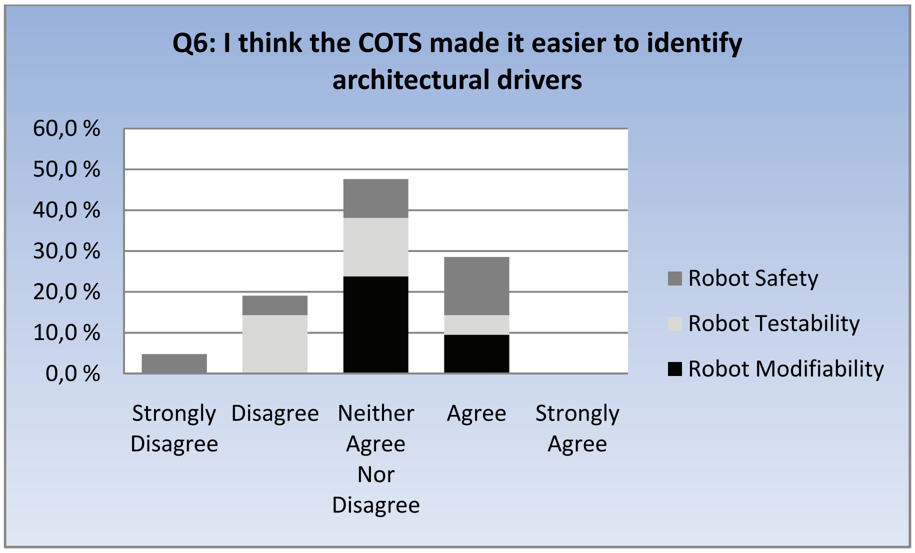


Figure 11.9: Stacked column diagram showing the quality attribute distribution for robot from Q6.

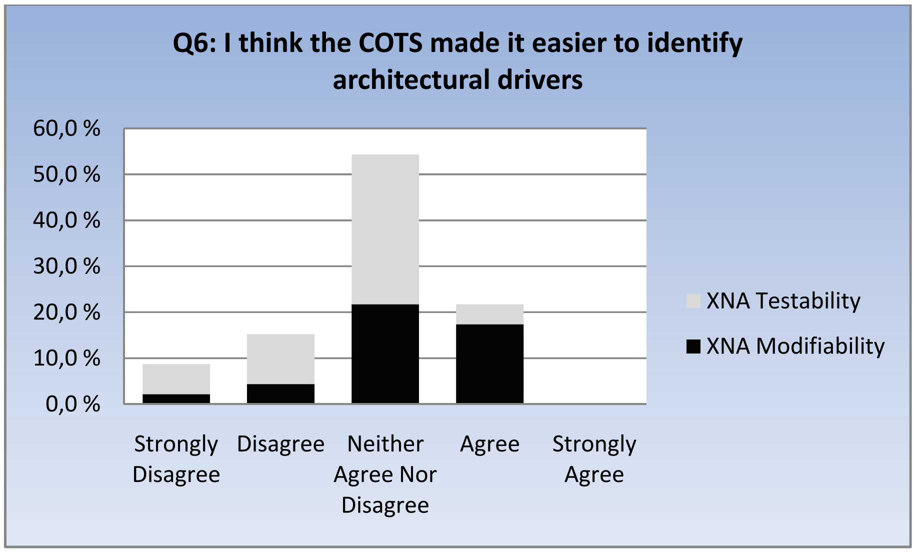


Figure 11.10: Stacked column diagram showing the quality attribute distribution for XNA from Q6.

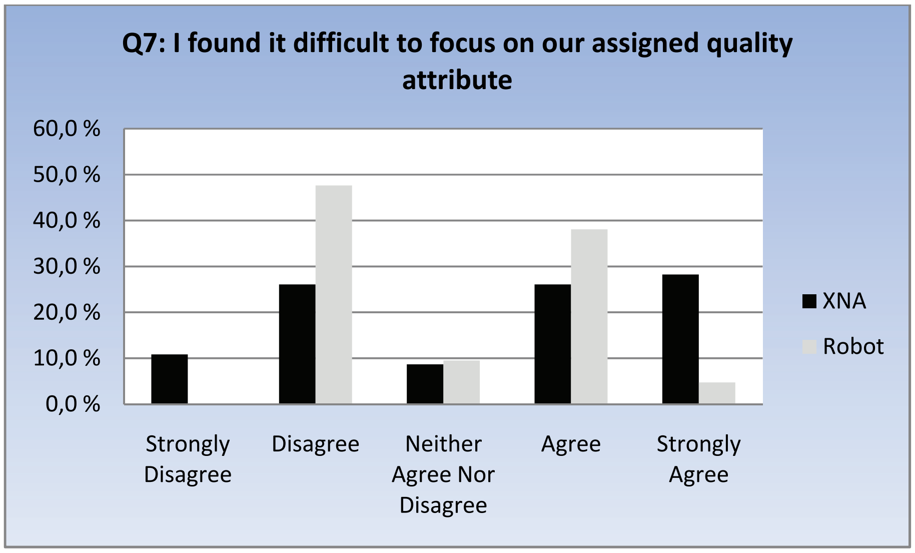


Figure 11.11: Column diagram showing results from Q7.

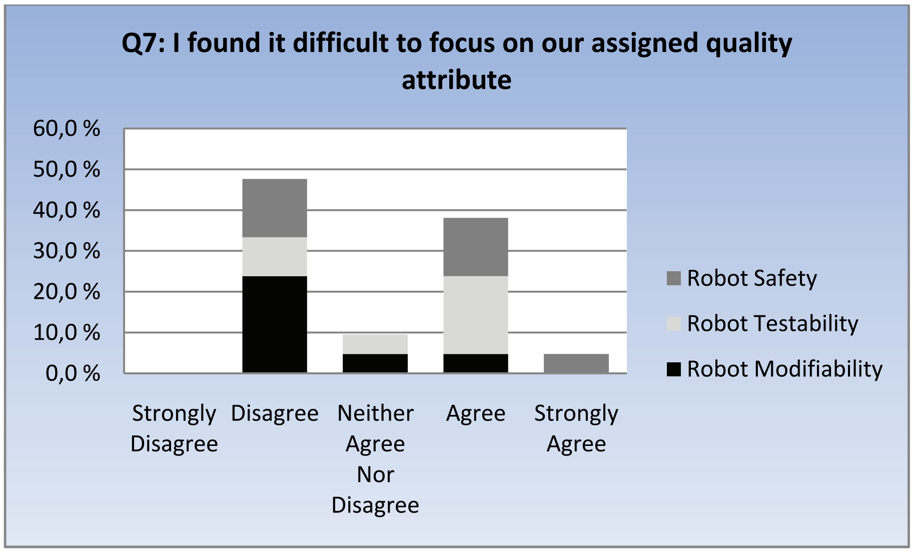


Figure 11.12: Stacked column diagram showing the quality attribute distribution for robot from Q7.

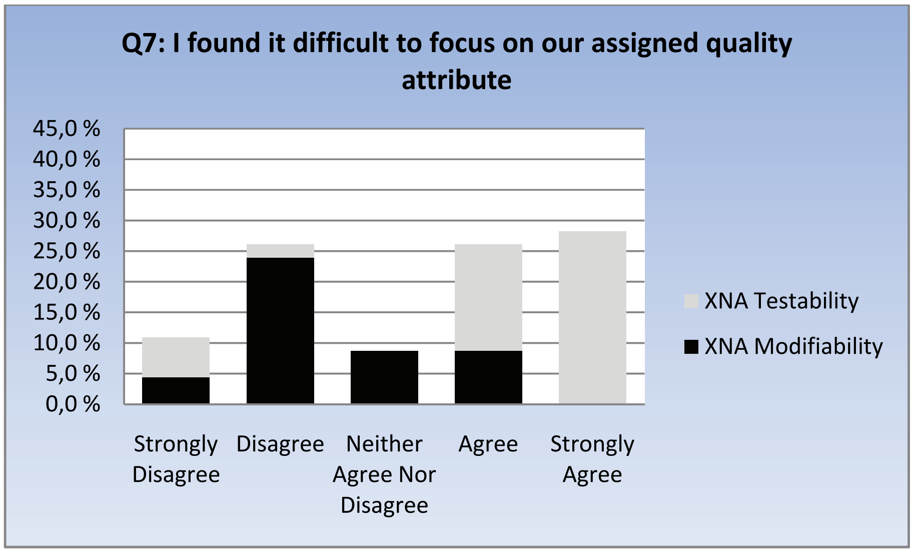


Figure 11.13: Stacked column diagram showing the quality attribute distribution for XNA from Q7.

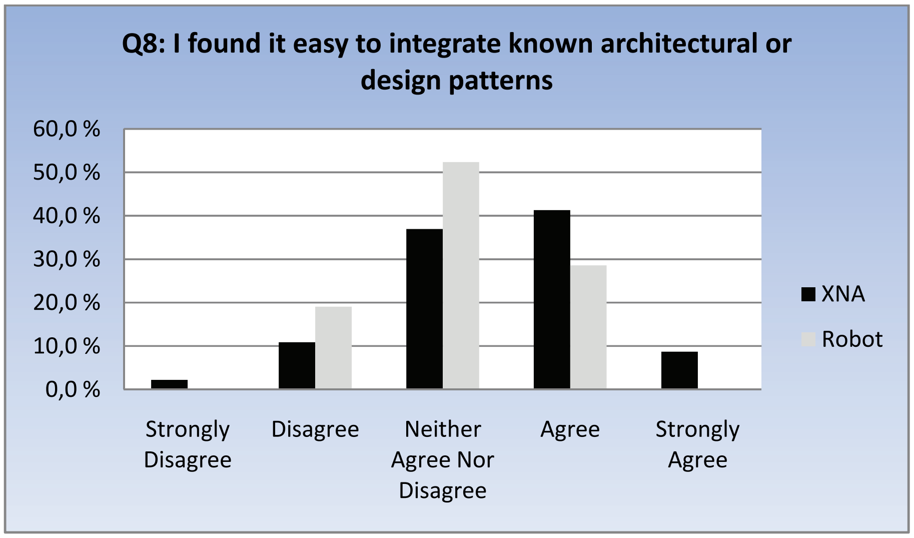


Figure 11.14: Column diagram showing results from Q8.

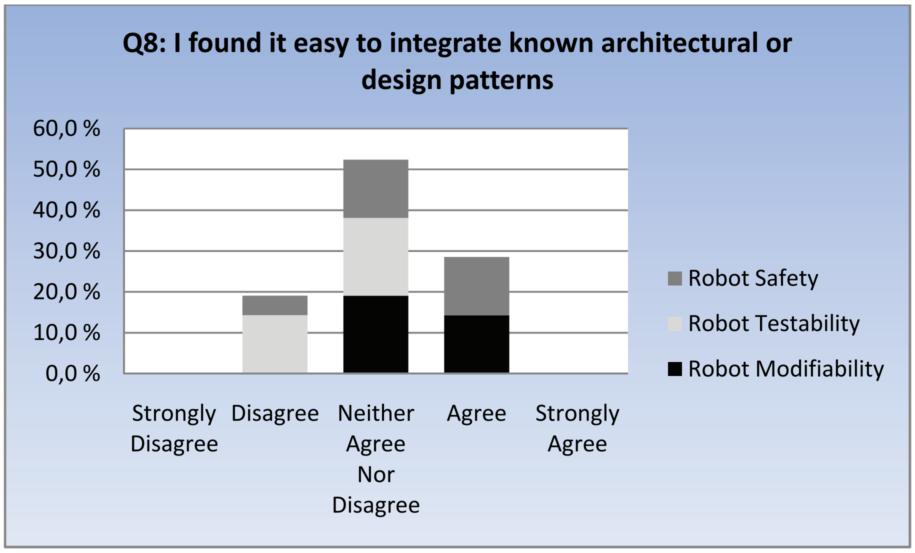


Figure 11.15: Stacked column diagram showing the quality attribute distribution for robot from Q8.

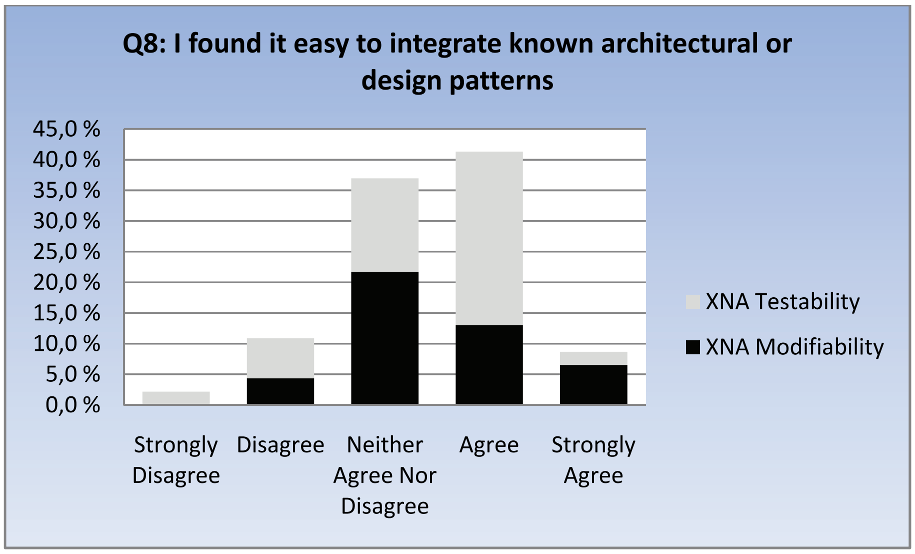


Figure 11.16: Stacked column diagram showing the quality attribute distribution for XNA from Q8.

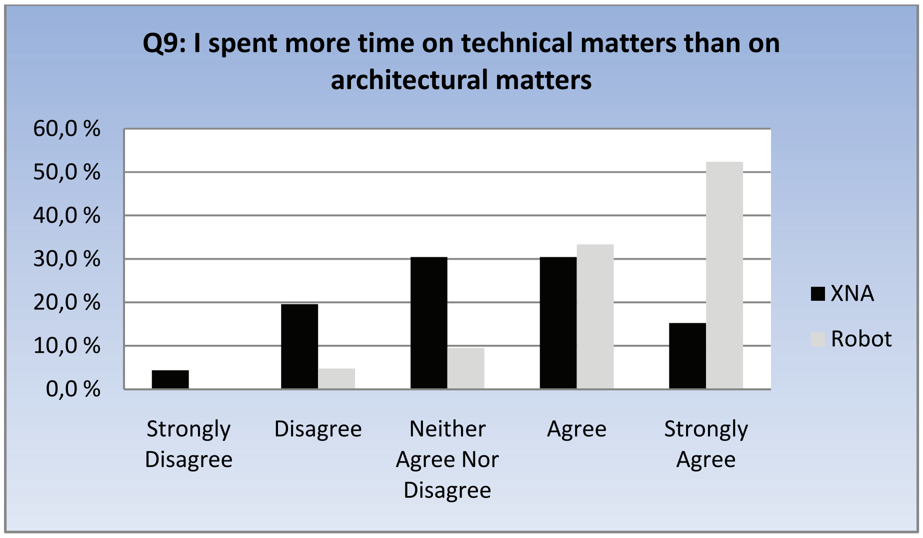


Figure 11.17: Column diagram showing results from Q9.

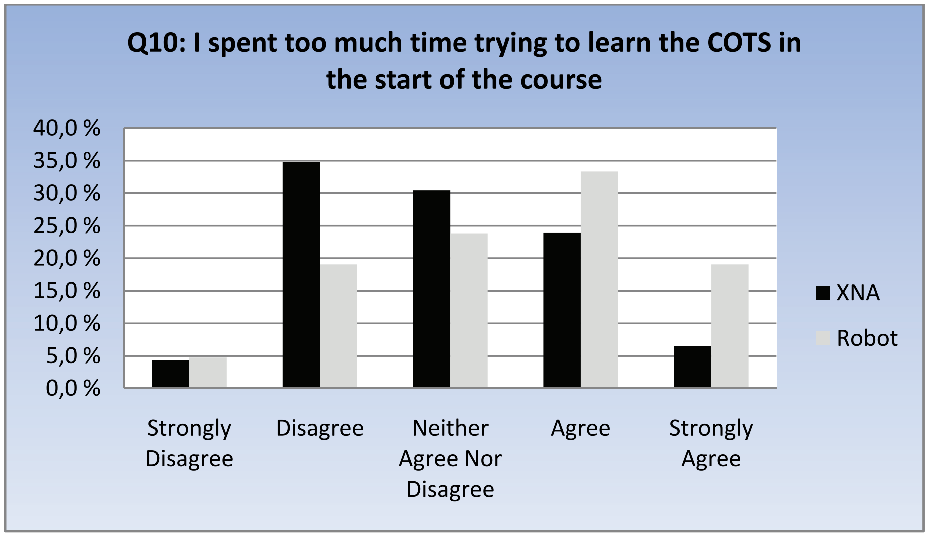


Figure 11.18: Column diagram showing results from Q10.

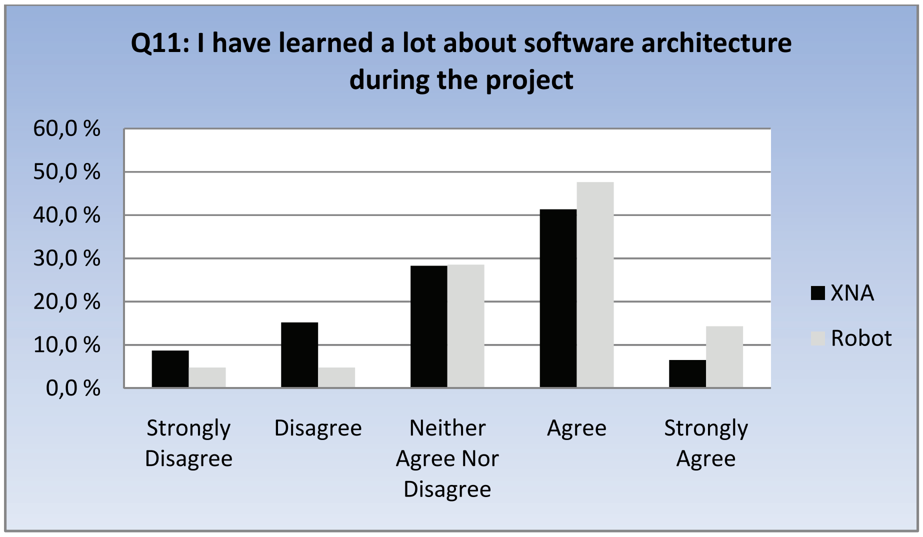


Figure 11.19: Column diagram showing results from Q11.

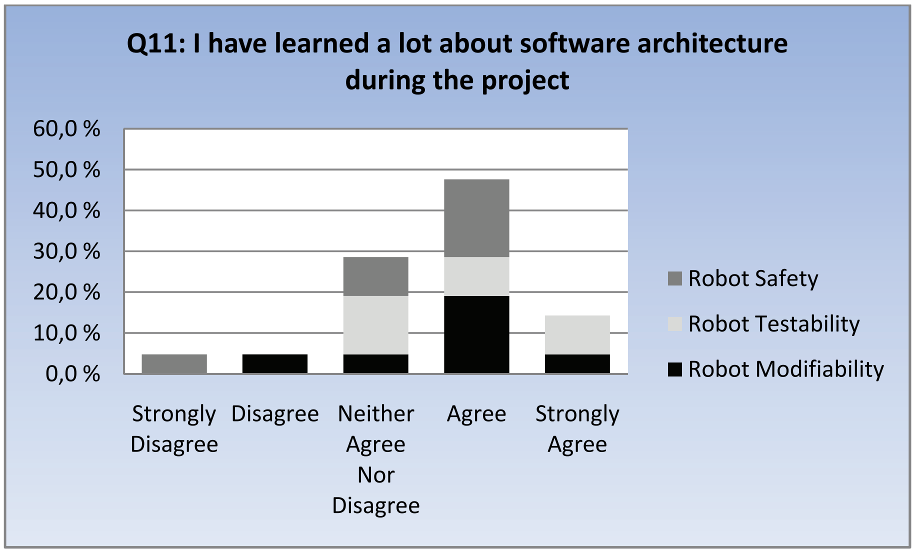


Figure 11.20: Stacked column diagram showing quality attribute distribution for robot for Q11.

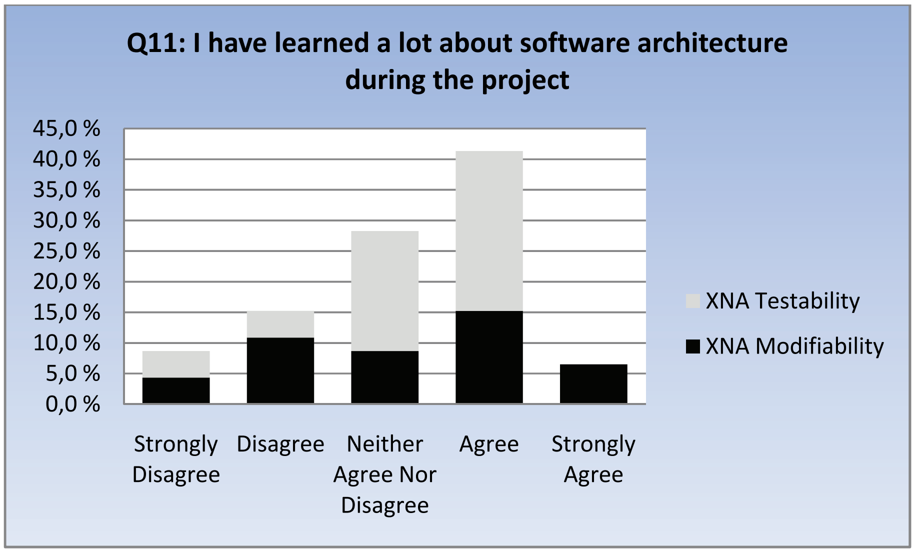


Figure 11.21: Stacked column diagram showing quality attribute distribution for XNA from Q11.

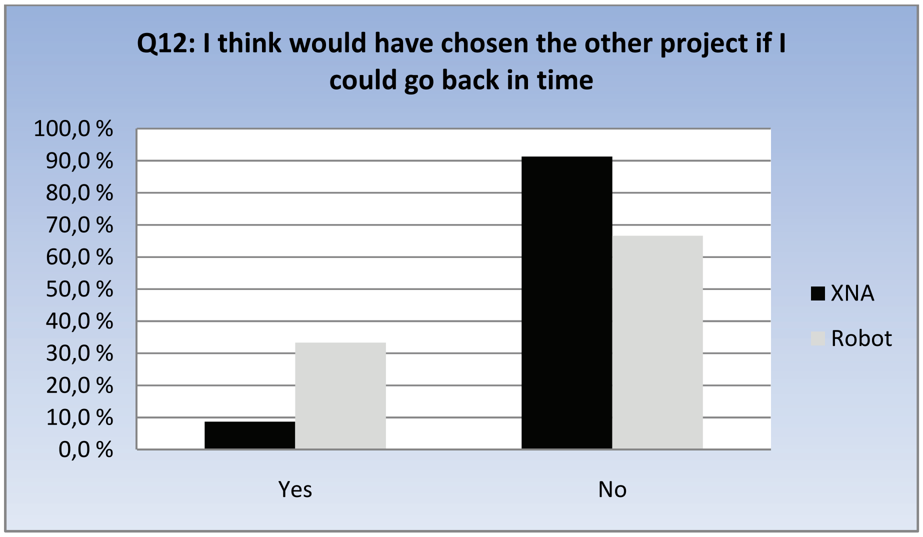


Figure 11.22: Column diagram showing results from Q12.

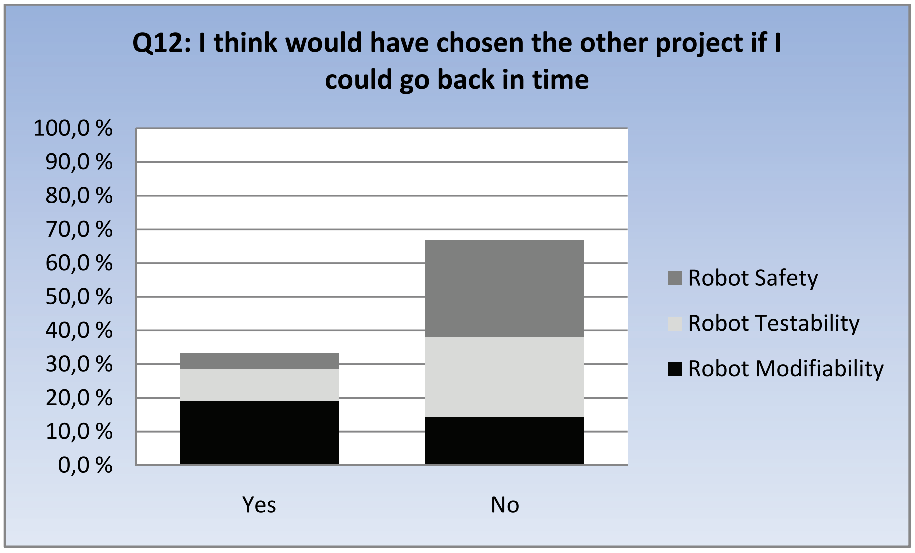


Figure 11.23: Stacked column diagram showing quality attribute distribution for robot from Q12.

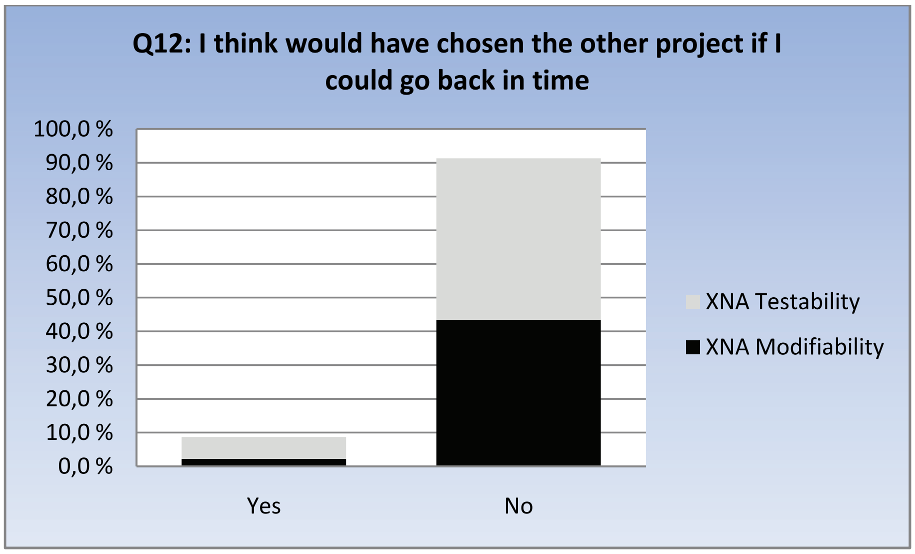


Figure 11.24: Stacked column diagram showing quality attribute distribution for XNA from Q12.

11.2 XNA and XQUEST-specific Questionnaire

This questionnaire was only to be responded to by the students who chose the XNA game project. For the first two items, X1 and X2, we received a total of 46 replies. The rest of the items were supposed to be responded to only by those students who had used XQUEST in their XNA game project. We received a total of 19 responses to those items.

Figure 11.25 to Figure 11.33 presents the results from the XNA and XQUEST-specific questionnaire.

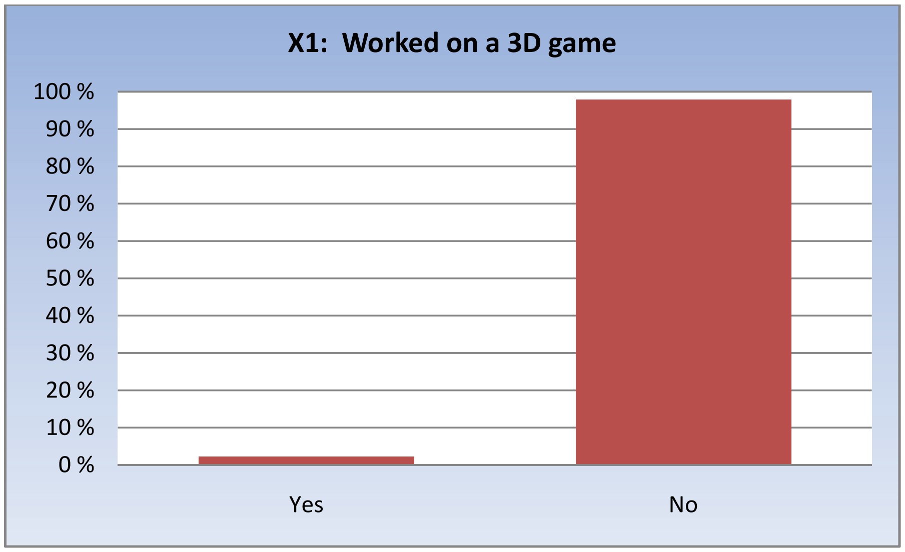


Figure 11.25: Column diagram showing results from X1.

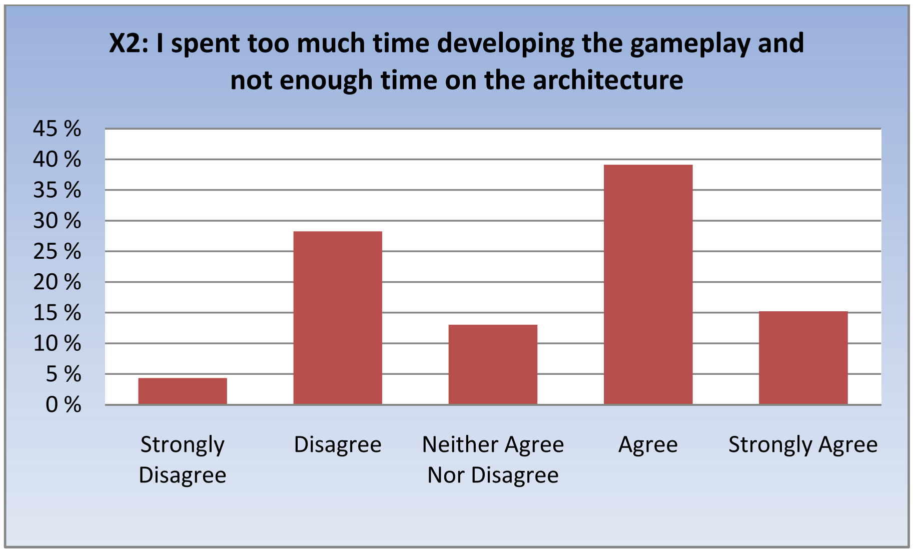


Figure 11.26: Column diagram showing results from X2.

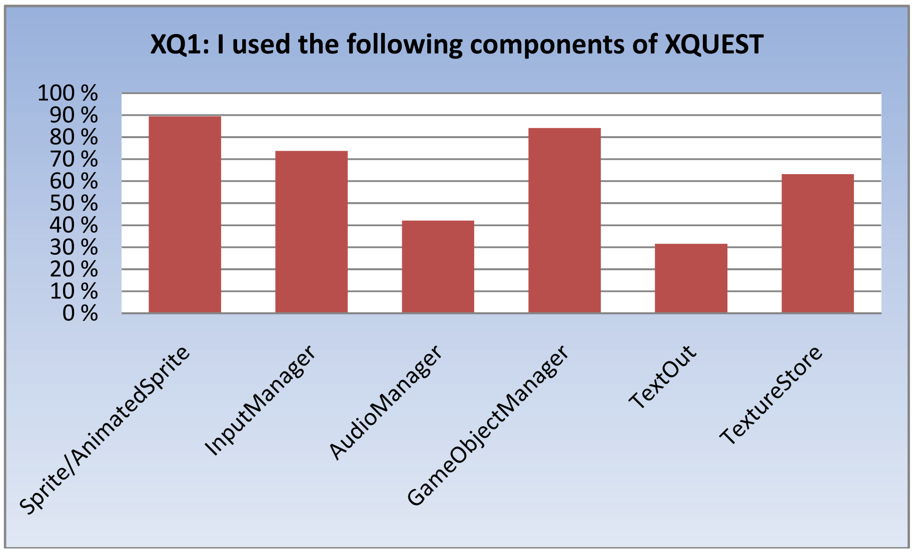


Figure 11.27: Column diagram showing results from XQ1.

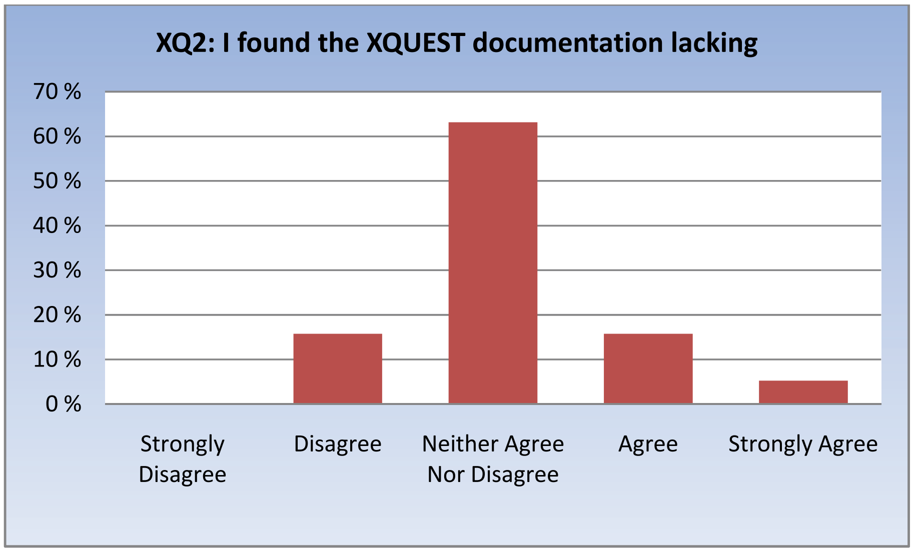


Figure 11.28: Column diagram showing results from XQ2.

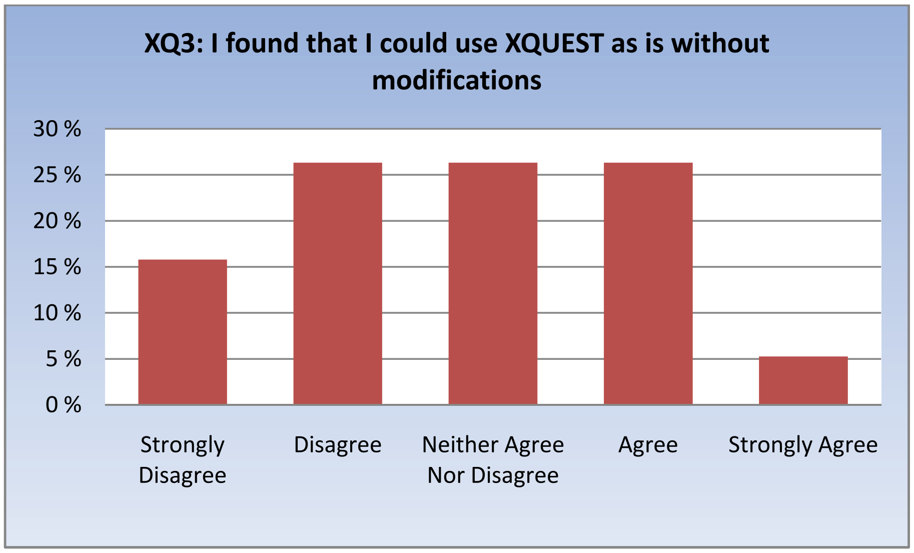


Figure 11.29: Column diagram showing results from XQ3.

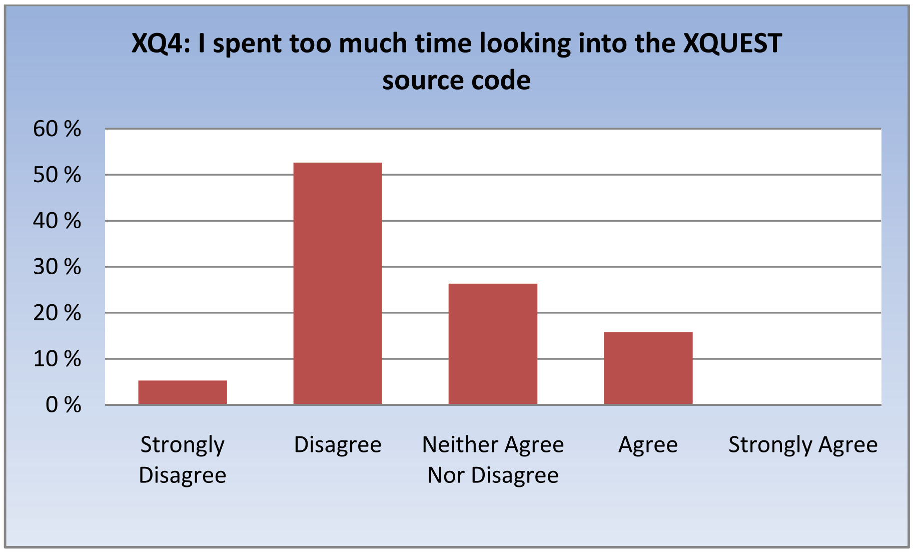


Figure 11.30: Column diagram showing results from XQ4.

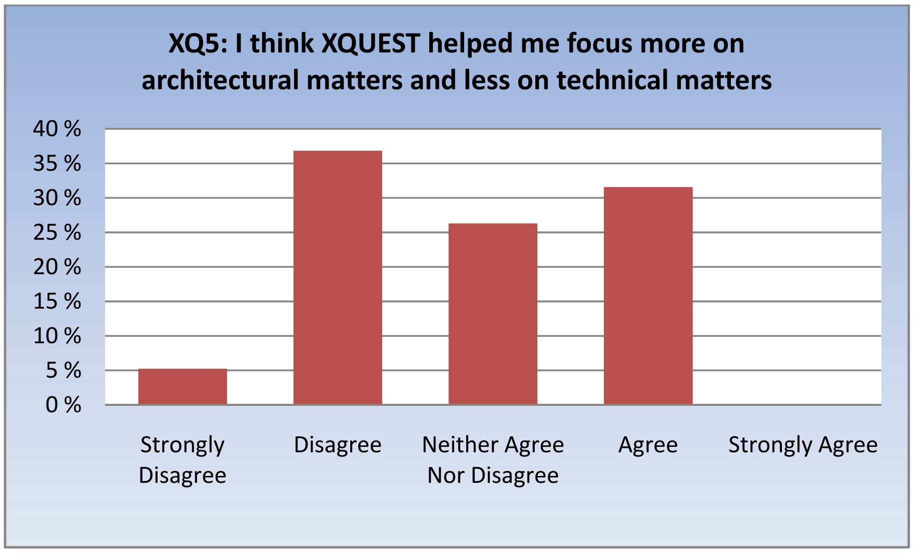


Figure 11.31: Column diagram showing results from XQ5.

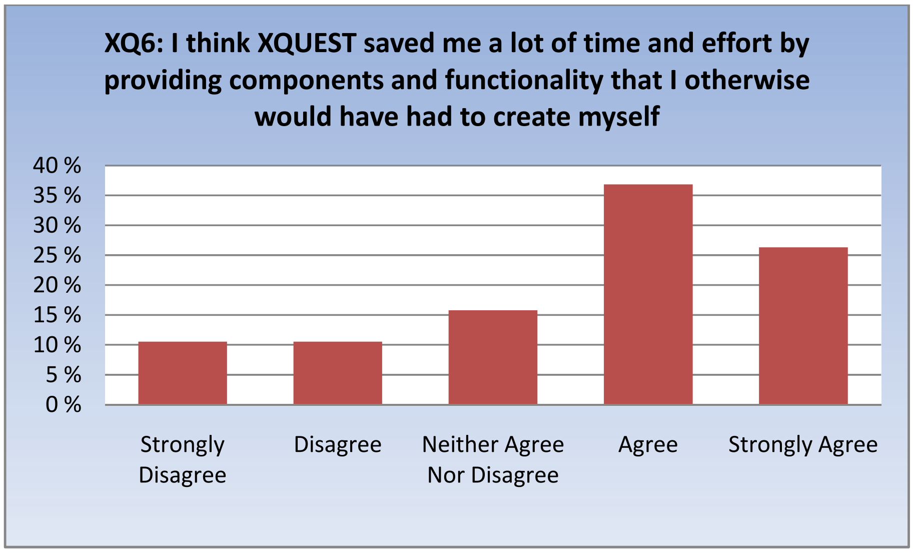


Figure 11.32: Column diagram showing results from XQ6.

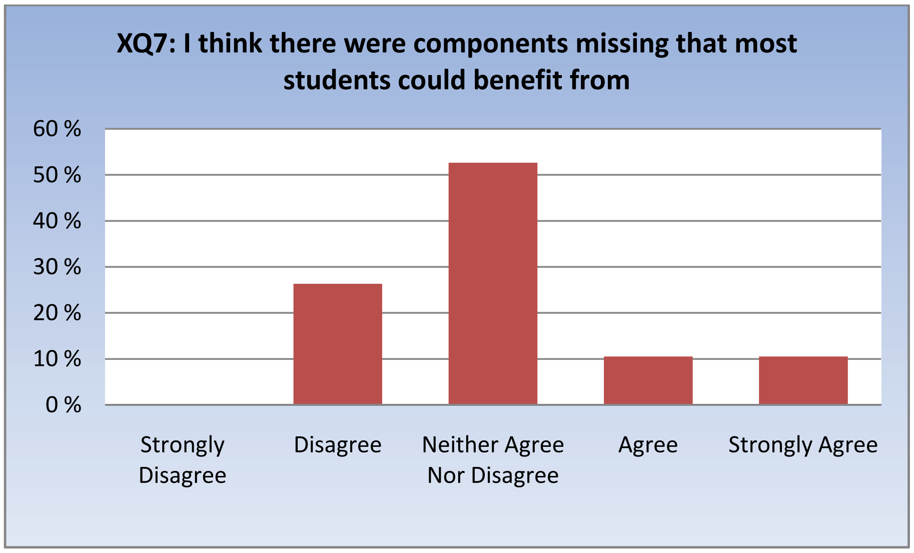


Figure 11.33: Column diagram showing results from XQ7.

XQ8: If you felt there were components missing, which ones would you like to see in a future version of XQUEST?

This was an open question, and out of the 17 people who responded to the XQUEST question, 4 people (24%) entered something here.

1. Sprite layers and pixel collision detection.
2. More flexible BasicGameObject, allowing non-sprite objects.
3. Not using enum for states. Make it easier to add new states.
4. Pixel-based collision detection, system for being called with certain intervals, better modifiability.

11.3 XQUEST SUS

Here we present the results from the SUS part of the XQUEST questionnaire. The SUS is used for subjectively measuring usability of a system on a high level. The outcome of a SUS questionnaire is a score within the range of 0 to 100, where higher values indicate a higher measured usability of the system. Each item in the SUS is responded to by assigning a scale value from 1 to 5, where 1 indicates strong disagreement and 5 indicates strong agreement.

To calculate the SUS score, we first sum together the score contributions for each question. Each question's score contribution is a number in the range 0 to 4. For odd-numbered questions (1, 3, 5, 7, 9), the score contribution is given by the scale position minus 1. For even-numbered questions (2, 4, 6, 8), the score contribution is 5 minus the scale position. The sum of the score contributions are then multiplied by 2.5 and divided by the number of replies to the survey to obtain the final SUS score.

Table 11.1: Results from the SUS.

	Question	**Sum score contribution**
1	I think that I would like to use this system frequently.	35
2	I found the system unnecessarily complex.	52
3	I thought the system was easy to use.	50
4	I think that I would need the support of a technical person to be able to use this system.	55
5	I found the various functions in this system were well integrated.	47
6	I thought there was too much inconsistency in this system.	48
7	I would imagine that most people would learn to use this system very quickly.	44
8	I found the system very cumbersome to use.	45
9	I felt very confident using the system.	40
10	I needed to learn a lot of things before I could get going with this system.	44
	Sum:	**460**
	SUS Score: 460 * 2.5 / 19 =	**60.53**

Our system achieved a SUS score of **60.53** out of a possible 100.

12. Evaluation of Survey Results

Here we present an evaluation of the results from the survey presented in Chapter 11. In addition to commenting the results from the survey, we have also based the evaluation on insights we gained while assessing the student project deliveries as student assistants.

Data gathered from the general part of the survey will in this chapter be assessed with a comparative approach. We will go through the results and emphasize any differences between the robot and XNA projects.

12.1 General

The general part of the survey consisted of twelve items that were to be responded to by every student, regardless of their project and assigned quality attribute focus. In total we received 46 responses from students who worked with an XNA game project and 21 answers from students who worked on a robot project (a total of 67 responses). One third of the students who worked on a robot project had modifiability as their assigned quality attribute. Similarly one third had testability and the last third had safety. For the XNA game project students, approximately 45% had modifiability and 55% had testability as their assigned quality attribute (see Table 12.1).

Table 12.1: Overview of how the answers were divided by project and assigned quality attribute.

67 Responses in Total	46 XNA	45 % Modifiability
		55% Testability
	21 Robot	33% Modifiability
		33% Testability
		33% Safety

The even split among quality attributes enabled us to query the results with regards to quality attributes and look at differences on that level as well, in addition to the overall differences between the robot and the XNA projects.

In the general part with mainly wanted to query for differences between the two projects to answer our first research question (RQ1). In the following, we will evaluate the results presented in the previous chapter against the stated goals of the evaluation.

G1: Compare the students who chose the XNA game projects and the students who chose the robot simulation project with regards to:

- **G1.1** To which degree the COTS influenced:

  The requirements gathering and specification

  We wanted to find out if the students felt their choice of COTS would make it easier or harder to come up with good requirements, reflected in **Q3**. The results did not give any indication that the COTS made a significant impact on the requirements phase of the project. As seen in Figure 11.3 there were many students who thought it was hard, and there were many who thought it easy to come up with good requirements for both robot and XNA projects.

  The design of the architecture

  Another phase of the project to look at was the architecture design phase. To determine if the COTS influenced the design of the architecture in any way, we asked the students in **Q4** if they thought the COTS prevented them from coming up with a good architecture. The results were very interesting. As seen in Figure 11.4 most of the XNA students agreed that the COTS did not hinder design of a good architecture, while the majority of the robot students on the other hand disagreed.

  When we organized the results by quality attributes (see Figure 11.5) we noticed how about half of the robot students who disagreed had modifiability as their focus. The reason we think is due to the constraints in how the robot simulator works, because everything needs to evolve around one main function called *doWork( )*, which limits what the students can do with regards to architecture.

  There was no clear similar indication when looking at the quality attributes distribution for the XNA students (see Figure 11.6).

  The ATAM evaluation

  We wanted to find out if the difficulty of evaluating a group's ATAM was affected by the COTS of the evaluated group. In **Q5** we asked the students if they thought it was difficult to evaluate the other group's ATAM.

  All robot groups evaluated an XNA group, but since there were more XNA groups than robot groups, some of the XNA groups had to evaluate another XNA group instead of a robot group. As such, our results cannot give a clear indication to whether or not the COTS influenced the difficulty of evaluation. In retrospect, we should have asked the students which kind of group they evaluated as well. That being said, the results did show that most of the XNA students found it difficult to evaluate another group's ATAM document (see Figure 11.7).

Architectural Drivers

Q6 asked the students if they thought the COTS made it easier to identify architectural drivers. At first glance there seems to be little to no difference between the two sides. There is roughly the same amount that disagrees as agrees for both the XNA projects and the robot projects, and in both cases there is a very large amount of people who neither agree nor disagree (see Figure 11.8).

When looking at the quality attribute distribution for this item (see Figure 11.9 and Figure 11.10), we see that students who focused on modifiability tend to agree more that the COTS made it easier to identify architectural drivers, while students who focused on testability tend to disagree more. For the robot safety quality attribute the answers were evenly spread.

It seems safe to say that the COTS does not influence the difficulty of identifying architectural drivers.

Quality Attribute Focus

Q7 asked the students if they found it difficult to focus on their assigned quality attribute. The results showed that most students had an opinion on this, but the opinions varied widely between agreeing and disagreeing. About as many agreed as disagreed for robot, and for XNA a few more agreed than disagreed (see Figure 11.11). There is little to say about this regarding the COTS, so we looked at the quality attribute distribution (see Figure 11.12 and Figure 11.13) and found that generally students with modifiability focus found it easy and students with testability focus found it difficult in both cases of robot and XNA. For the robot safety attribute more students found it difficult than easy.

We think the reason why modifiability seems to be easier to focus on is because the students have a much better understanding of what exactly modifiability is. Much of the computer programming education given through school emphasize on topics that enhance modifiability, as opposed to testability, which from our own experience is taught very little. The idea of creating a program that is highly modifiable is nothing new to most students, but creating one that makes testing easier is a whole new area. Also, robot safety students might have found the robot simulator very hard to work with because it does not always behave consistent.

The results indicate that choice of COTS does not have much influence on difficulty of focusing on the assigned quality attribute.

- **G1.2** The relationship between time spent on technical matters and architectural matters.

  When we asked the students if they felt they spent more time on technical matters than on architectural matters in **Q9**, almost all of the students with robot projects responded that they thought they did. For XNA, the answers were a bit more even, but still the majority agreed (see Figure 11.17).

  From our own experience, we can tell that the robot simulator is not very good, and requires a lot of hard work to get working, while the XNA environment is much more user friendly. We believe this to be the main reason that the students who chose the robot project had to spend more time on technical matters, even though many of the XNA students were completely new to both C# and XNA.

  Clearly the choice of COTS here influences the time the students have at their disposal to focus on architectural matters.

- **G1.3** How difficult it was to:

  Integrate known architectural and design patterns

  We asked the students if they found it easy to integrate known architectural or design patterns in **Q8**. Generally the majority of students, for both XNA and robot, thought it easy. XNA had the largest majority (see Figure 11.14). Also, we found when looking at the quality attribute distribution for the robot projects (see Figure 11.15) that no modifiability students thought it hard, while none of the testability students thought it easy. We think the differences here lies in the fact that most of the patterns presented for the robot students are for modifiability. For XNA there was not that many suitable patterns presented, and most of the student groups chose to use the Model-View-Controller pattern.

  Clearly more learning material can and will be created with time as the XNA game project matures.

  Learn the necessary prerequisite skills to be able to develop programs

  We wanted to compare the robot and XNA projects, with regards to if they think they wasted too much time learning the COTS in the start of the project. In **Q10** we ask them about this, and the results show that slightly more XNA students disagree that they spent too much time learning the COTS, while a good majority of the robot students agreed to the same (see Figure 11.18).

We were a bit surprised that a lot less students who chose XNA felt they spent too much time learning C# and XNA because these topics are fairly new to most students. The robot simulator on the other hand requires java, which should be well known to every student attending a 4th grade computer science course. Again, we think the faulty and time consuming robot simulator is the reason for this result.

Also, from our own experience as student assistants we received almost no requests for help from the XNA groups, but several from the robot groups.

- **G1.4** How much the students feel they have learned about software architecture through the project.

  From the results of **Q11** (see Figure 11.19), we see that most of the students feel they have learned a lot about software architecture throughout the project. However, a larger portion of the students with robot projects feel so than students with XNA projects. We think the reason why is because the robot project is more mature having run for several years, resulting in more available learning material for the students. Also, some of the XNA students might have become spellbound by the fun of creating a game, thus focusing more on gameplay than architecture. This is also backed up by the results from X2 (see Figure 11.26).

- **G1.5** The overall happiness with the project.

  On the last item in the general part, **Q12**, we ask the students if they would have chosen the other project if they could go back in time. Most would not have, but it is worth noting that as much as 33% of the robot project students would have, and only 8% of the XNA project students would have liked to work on a robot project instead (see Figure 11.22).

12.2 XNA/XQUEST 1.0

There were no groups that worked on a 3D game, although some groups used 3D techniques in their 2D game. One student responded that he or she was working on a 3D game (see **X1**, Figure 11.25). We believe that this student was simply confused or responded randomly.

When it comes to the relationship between time spent on developing gameplay and time spent on architecture, slightly more students agreed that they spent more time on gameplay than disagreed (see **X2**, Figure 11.26).

XQUEST achieved a SUS score of 60.53. This is a little above average usability. We lack comparison with other similar systems, so the score cannot be put into any context at this time. We observe however that the generality of the SUS questions makes the middle alternative extremely attractive in this case. Also, the SUS is normally used for feedback of running systems rather than frameworks such as XQUEST where the entry level is usually a bit higher[24].

G2: Find out the usefulness and usability of XQUEST.

Out of the 46 students who responded that they worked on an XNA game project, 19 (41%) responded they used XQUEST.

When it comes to the XQUEST documentation, four students agreed that they found it lacking, and one student even strongly agreed (see **XQ2**, Figure 11.28). On the other hand, three students disagreed. The rest (63%) had no opinion on the documentation and selected the middle alternative. We cannot really come to a conclusion about the documentation, since the numbers are spread on both sides of the scale, but the fact that there were more negative responses than positive indicates that the documentation could have been better. Personally, we feel that having more examples on the usage of the different components in XQUEST would greatly enhance the documentation, and is something we had in the back of our minds when developing XQUEST 2.0.

A trend we saw was that many of the groups who used XQUEST used it as a template rather than referencing it as a library. This gives them the added advantage of modifying the XQUEST code to better fit in with their game architecture and design. 16% strongly disagreed that they could use XQUEST without modifications, against 5% who strongly agreed (see **XQ3**, Figure 11.29). Although one of the design philosophies with XQUEST was to create it as abstract and reusable as possible, we expected people's need to modify it. After all, that is why the T in XQUEST stands for Template.

The students disagreed that they spent too much time looking into the source code of XQUEST (see **XQ4**, Figure 11.30). Only three students felt they spent too much time on this. Whether this positive result is because of good documentation or self-explanatory public interfaces we do not know, but we feel that XQUEST 1.0 sports good source code commenting that is helpful for people trying to understand the code. Using Visual Studio greatly benefits source code commenting in that these comments show up on the IntelliSense[25] tooltips that pop up while the programmer is typing in code. For example, when creating a new instance of a class, the IntelliSense will display any comments available for the different parameters that the constructor accepts.

Responses to **XQ5** (see Figure 11.31) indicate that XQUEST was not so useful for students for focusing more on architectural than technical matters. However, results from **XQ6** (see Figure 11.32) clearly show that XQUEST helped save time. This indicates that this time was spent on other technical matters in which QUEST could not provide any usefulness, rather than architectural matters. A more positive response to **XQ6** could surely be achieved if XQUEST could cover more ground and help in more areas. This is further backed up by the responses to **XQ7** (see Figure 11.33). Most of the students agree that there are components missing in XQUEST that most of them could benefit from.

G3: Find out how the students used XQUEST in their projects.

Responses to **XQ1** (see Figure 11.27) reveal that all components of XQUEST were taken into use, some more than others. The most popular component was the Animated Sprite Framework. Since all groups worked on 2D games, this is understandable since sprite rendering is the easiest way to output graphics in a 2D environment.

Another popular component was the Game Object Management component. When going through the XNA deliveries, we were surprised to find out that most groups did not create their own implementation of the *IGameObject* interface, but rather used the standard *BasicGameObject*. Using *BasicGameObject* limits you somehow, because it is tightly interwoven with the Sprite Animation Framework for representing the game object using sprites. This of course implies that groups that used this approach, also needed to use the Sprite Animation Framework. Looking at the high percentage of students who responded to having used both of these components, there is no doubt that the use of *BasicGameObject* is the main reason for this.

In third place comes the *InputManager* component. This is probably the most useful component in XQUEST 1.0, since every game needs to handle input in some form, and we really did expect this to be the most popular component. We can only speculate as to why not; it may have to do with the sheer size of the *InputManager* class that scared people away from using it. It contains a lot of methods for supporting all the XNA input devices such as keyboard, mouse, and up to four Xbox 360 game pads. By looking at the deliveries, we found that most games were single-player games played with a keyboard, or hot seat multiplayer games where each player used the same keyboard. Some games used the mouse as the primary input device, but very few implemented game pad support. This is of course understandable, since they did not have access to Xbox 360 game pads during the project unless they brought one themselves. The input needs may therefore have not been so great that it required a component like the *InputManager*.

The least used components were the *AudioManager* and *TextOut* components. Having music and sound effects in your game may not take the greatest priority in a school project where the evaluation criteria leans more towards software architecture and fulfilling an assigned quality attribute. For this reason, many groups decided not to implement audio features in their games, and hence no need for the *AudioManager* component. Still, almost half of the games that used XQUEST used this component. The *TextOut* component was a component we thought would be more popular. It is really simple to use, and has features that makes it very convenient for text display. It may be the fact that text display is so simple that the students did not see the need for using it. The standard way of displaying text with *SpriteBatch* may fulfill all the desired text rendering needs.

XQ8 was an open question where the students could put down anything they would like to see in a future version of XQUEST. Only four students replied, and the new features that were proposed are

- Pixel-perfect collision detection. Two of the four replies proposed this feature.
- More flexible *BasicGameObject*, allowing non-sprite objects.
- Sprite layers.
- More flexible system for game object states where states can be defined by the user.
- Better modifiability.

Pixel-perfect collision detection is a very performance-intensive operation that we described as not suitable for a multi-purpose game object system such as the one in XQUEST in our XQUEST 1.0 documentation. However, we decided to include support for it in XQUEST 2.0. It is disabled by default, but can be enabled on a per-object basis, meaning the user is in total control of how the collision detection should be executed for every object in the scene.

BasicGameObject is per definition not supposed to be flexible. The flexibility of the game object management system in XQUEST 1.0 lies in the *IGameObject* interface, of which *BasicGameObject* is an implementation. As expressed above, we were surprised that so few groups did not take advantage of this flexibility by providing their own implementation of the *IGameObject* interface. Doing so, they could have tailored it towards their game. Instead, they chose to use the standard implementation in *BasicGameObject*, which of course also constrained them to using the *Sprite* class for the graphical representation.

Part IV
Extensions of The XQUEST Game Library

13. Introduction

In our depth study, we developed a basic XNA game library with useful components that was meant to act as an aid to the students of the software architecture course in the spring semester. We named it XQUEST, which is short for XNA QUick and Easy Starter Template. In Chapter 12, we presented an evaluation of XQUEST 1.0. This evaluation is based on a survey study of the students who used XQUEST in their XNA game project. Because this evaluation had to be carried out late in the semester, after the students were finished with their projects, the results from the evaluation would not be interpreted into new development of XQUEST until the very last month of our project. Therefore, we decided to start early in the semester to improve and extend XQUEST based on what we felt was missing or needed to be integrated. The next several chapters will describe the development of XQUEST that was done both before and after the evaluation.

For the sake of clarity, we will refer to the work we did on XQUEST in our depth study as XQUEST 1.0, and the work we have done modifying and extending it as part of this master thesis XQUEST 2.0. When we mention XQUEST in general without pointing to a specific version, we will simply use XQUEST.

We presented an overview of XQUEST 1.0 in Section 4.1. The following will therefore assume the reader has either read this chapter, or read the relevant chapters in our depth study report[1] pertaining to XQUEST 1.0.

13.1 Motivation and Goals

It is safe to assume that many of the students of the software architecture course have little or no experience with XNA. In addition, although they should have knowledge of object oriented programming, they may not have any experience with the C# programming language or the .NET development environment at all. When we also take into consideration that game programming differs from “normal” programming in that it concerns real-time applications, we realize that it is a lot to grasp for the non-experienced student. All this implies that they will need to acquire quite a bit of knowledge before they can become productive. XQUEST aims to help out with the XNA bit. By providing usable game-related code as a resource to the students, the primary goal of the XQUEST game library is to help the students get up to speed with game programming using XNA.

13.2 Design Guidelines

The XNA framework provides a high-level object-oriented API whose design is heavily inspired by the .NET design standards. This represents something new in game programming. Historically, we are used to games that are written in C or C++, using a service API such as OpenGL or DirectX. What

Microsoft has done with XNA is bring game programming to the .NET platform. As such, they have in a way bridged the gap between game development and more traditional software development.

When we create a new XNA project in Visual Studio, we are presented with a code skeleton on which to base our code. We can start coding the game logic right away. This code skeleton is called the XNA Application Model, and it "provides functionality to accomplish common game development tasks"[26]. It manages the game loop and is among other things responsible for handling the graphics device and providing game timing. It is extensible by means of the concept of game components and game services. A game component is a custom-created piece of game functionality that can be plugged directly in to the Application Model. By definition, game components are highly reusable and should be as general as possible and only provide functionality for a distinct set of tasks. A game component can be exposed as a game service so that other game components can make use of it. Game services follow a publish-subscribe pattern. Game components can both publish themselves to the game and subscribe to other game components. Game services are globally available in the code through the *Game* class.

We wanted XQUEST to take full advantage of the Application Model by using it as a base for our own development. We can think of XQUEST as an extension of the Application Model, offering even more common game development tasks. XQUEST makes heavily use of game components and game services, offering highly reusable and adaptable components that can be used in any game project, regardless of genre and whether it is a 2D or 3D project. Furthermore, XQUEST is very flexible in that it can be used in many ways. The primary usage method is as a library. In this scenario, the users reference XQUEST in their game project, and use components without the need to look at the source code. The documentation should be enough to provide a self-explanatory usage of the components. However, we realize that for more complex games the need for control is much greater, and the users may want to modify or extend certain parts of XQUEST. In this scenario, XQUEST can be used as a template from which to build a game upon. The users grab the parts of the XQUEST source code they need, and puts them into their own project. This code can then be modified and adapted to fit in with the rest of the code.

13.3 Releasing XQUEST to the Students

We released XQUEST 1.0 to the students on February 13, 2008 through the it's learning e-learning platform. This was right before the startup of the student projects. Since the students would be using the 2.0 version of the XNA framework, we had to convert XQUEST 1.0, which was written for XNA 1.0, to use the new version of XNA before releasing it. This process went fairly smooth, as there were no big changes in XNA that affected XQUEST 1.0, with the exception of an improvement to the XINPUT API that let users enumerate buttons on a gamepad in the same way we could enumerate keys on a keyboard. We realized that with this improvement in the XNA 2.0 framework, we could make our *InputManager* class a lot more intuitive and user-friendly. Instead of having a method for each button we could press on a gamepad, like *IsAButtonDown*, *IsBButtonDown*, *IsButtonXDown*, and so on, we could now replace all these methods by a single method that accepted an enumerated value from the new *Buttons* enumeration like this:

```
public bool IsButtonDown(Buttons button)
{
  for (int i = 0; i < MaxInputs; i++)
  {
    if (IsButtonDown(button, (PlayerIndex) i))
      return true;
  }
  return false;
}

public bool IsButtonDown(Buttons button, PlayerIndex playerIndex)
{
  return currentGamePadStates[(int) playerIndex].IsButtonDown(button);
}
```

This helped shrink the size of the *InputManager* considerably. We included the fixed *InputManager* in the process of converting XQUEST 1.0 to an XNA Game Studio 2.0 Game Library project.

13.4 Source Code and Examples

The source code for XQUEST 2.0 can be found as an attachment to this report, along with several small demo applications showing how to use the different components. In Chapter 15, we present a brief description of each of these demos. We also include a video file showing the demos in action. If the attachment is not provided alongside this paper, it can be obtained by contacting our project supervisor, associate professor Alf Inge Wang (alfw@idi.ntnu.no).

14. XQUEST 2.0

This chapter presents the modifications and improvements we made to existing components as part of development of XQUEST 2.0. The library will be presented namespace by namespace.

14.1 Overview

Figure 14.1 shows an overview of the logical compartmentalization of XQUEST into namespaces.

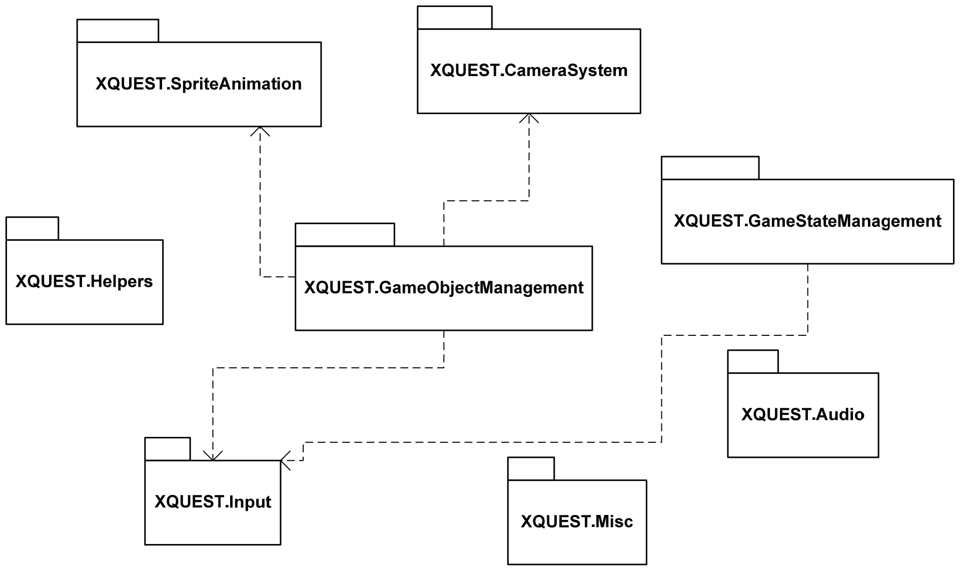


Figure 14.1: The namespaces in XQUEST and dependencies between them.

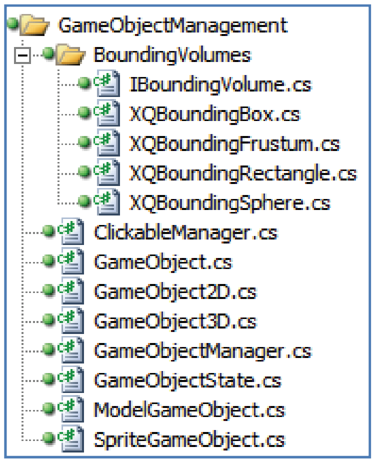
XQUEST functionality is split into eight namespaces. In Figure 14.1, we have put the *XQUEST.GameObjectManagement* namespace in the middle purposely to indicate its importance. It contains the game object system which is at the heart of XQUEST. Following is brief description of each namespace (Table 14.1). Each namespace will be explained in more detail in subsequent sections.

Table 14.1: Short description of each namespace.

Namespace	**Description**
XQUEST.GameObjectSystem	Contains the game object system, responsible for handling game objects such as players, enemies, powerups, etc. The object system allows for many different types of objects, 2D or 3D, and provides state tracking and a flexible collision detection system.
XQUEST.SpriteAnimation	Contains functionality for handling 2D sprites and sprite animation using sprite sheets.
XQUEST.CameraSystem	Provides functionality for allowing the set up of both 2D and 3D cameras to view a scene or track game objects in multiple perspectives.
XQUEST.GameStateManagement	Handles state and state transitions in the game. It uses the concept of game screens. A game screen can be a menu screen,

	an inventory screen, a combat screen, etc.
XQUEST.Audio	Handles audio-related functionality like playback of sound effects and music, adjustment of volume levels, grouping into categories, etc.
XQUEST.Misc	Contains miscellaneous components of utility that do not fit into any other namespace.
XQUEST.Input	Handles querying and interpretation of keyboard, mouse, and Xbox 360 game pad input.
XQUEST.Helpers	Contains convenient helper classes for common tasks.

14.2 XQUEST.GameObjectManagement


```
GameObjectManagement
  BoundingVolumes
    IBoundingVolume.cs
    XQBoundingBox.cs
    XQBoundingFrustum.cs
    XQBoundingRectangle.cs
    XQBoundingSphere.cs
  ClickableManager.cs
  GameObject.cs
  GameObject2D.cs
  GameObject3D.cs
  GameObjectManager.cs
  GameObjectState.cs
  ModelGameObject.cs
  SpriteGameObject.cs
```

The game object management part of XQUEST 1.0 was completely rewritten for XQUEST 2.0. There were several reasons for this decision. First, we needed to take extensions of XQUEST into consideration. The game object management system only supported 2D game objects. We wanted to support 3D game objects, so we knew we had to rethink the whole system. We decided to step away from the interface-based design; what we envisioned was a hierarchy of game object classes. The main idea was to have a base class that all other game object classes derived from, both 2D and 3D game object classes. Further, a 2D game object can be represented as a sprite, so a game object class based on a sprite representation was included. A 3D game object can be represented as a model, so a game object class based on a model representation was also included. Figure 14.2 shows the game object hierarchy that we came up with.

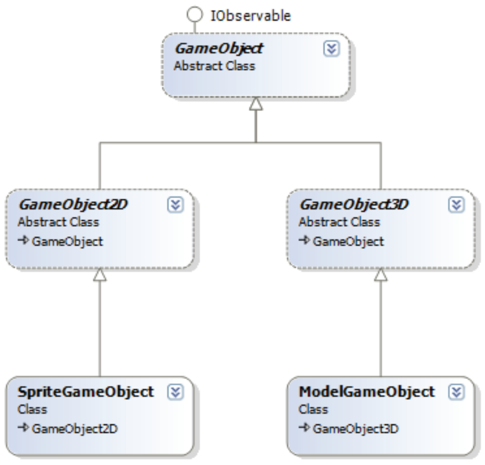


Figure 14.2: The game object hierarchy.

14.2.1 GameObject.cs

GameObject is the abstract base class that contains core functionality that every game object has in common.

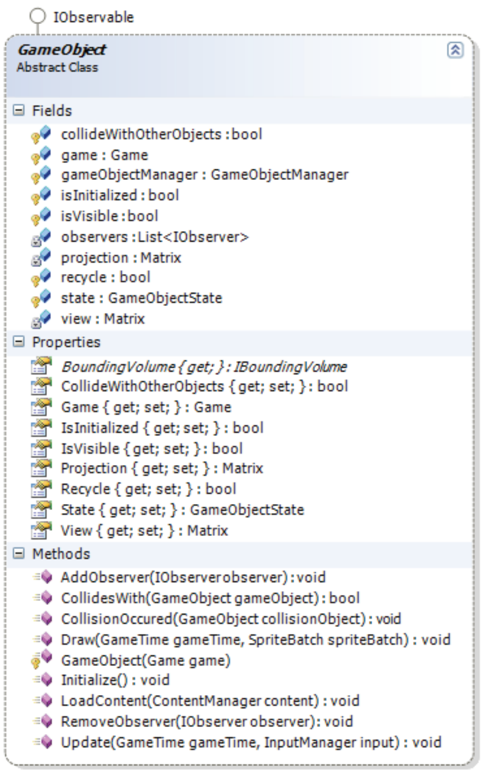

IObservable
GameObject
Abstract Class
Fields
collideWithOtherObjects : bool
game : Game
gameObjectManager : GameObjectManager
isInitialized : bool
isVisible : bool
observers : List<IObserver>
projection : Matrix
recycle : bool
state : GameObjectState
view : Matrix
Properties
BoundingVolume { get; } : IBoundingVolume
CollideWithOtherObjects { get; set; } : bool
Game { get; set; } : Game
IsInitialized { get; set; } : bool
IsVisible { get; set; } : bool
Projection { get; set; } : Matrix
Recycle { get; set; } : bool
State { get; set; } : GameObjectState
View { get; set; } : Matrix
Methods
AddObserver(IObserver observer) : void
CollidesWith(GameObject gameObject) : bool
CollisionOccured(GameObject collisionObject) : void
Draw(GameTime gameTime, SpriteBatch spriteBatch) : void
GameObject(Game game)
Initialize() : void
LoadContent(ContentManager content) : void
RemoveObserver(IObserver observer) : void
Update(GameTime gameTime, InputManager input) : void

Figure 14.3: The GameObject class.

In the following we will explain the fields, properties, and methods of the *GameObject* class. The fields include:

- *collideWithOtherObjects.* This field is a boolean value indicating whether or not the game object should be able to collide with other game objects. The default value is true. We included this field for efficiency reasons. Settings this field to false will exclude the game object from collision detection in the manager class.

- *game.* This is just a reference to the current XNA *Game* instance, which should be passed to the constructor of *GameObject*.

- *gameObjectManager.* This is a reference to the *GameObjectManager* instance responsible for managing the *GameObject*. The reference is retrieved from the *Game.Services* collection, which means a *GameObjectManager* instance will need to be created and added to *Game.Components* before creating any *GameObject*s.

- *isInitialized.* This is boolean value indicating whether or not the game object has been initialized, that is, their *Initialize* and *LoadContent* methods have been called. This flag is useful since we can prevent unnecessary reinitialization of a game object that has been recycled.

- *isVisible.* This is a boolean value indicating whether or not the game object should be visible. If this flag is set to false, the draw method will return immediately without drawing anything.

- *observers.* A game object can be observed by several observers, typically cameras. This paradigm follows the observer design pattern.

- *projection, view.* The current camera's projection and view matrices.

- *recycle.* This is a boolean value indicating whether or not the game object should be recycled after its "death". We will discuss recycling of game objects later in this chapter.

- *state.* This field controls the state of the game object. The *GameObjectState* enum is unchanged from XQUEST 1.0.

The properties of GameObject include:

- *BoundingVolume*. This is an abstract property defining the bounding volume of the game object. Possible bounding volumes include rectangles (2D only), boxes, spheres, or frustums. Derived classes should implement this property to define a bounding volume that most accurately approximates the shape of the game object.

The rest of the properties are access properties for the fields, and should be self-explanatory. The methods of the *GameObject* class include:

- *CollidesWith(GameObject).* This method is abstract and must be implemented in derived classes. It decides under what condition this type of game object collides with other game objects. For example, we will see later that *GameObject2D* uses a rectangle intersection test, while *GameObject3D* uses a bounding sphere intersection test to decided collisions. Note that this method is for internal use only, and should not be called by the user.

- *CollisionOccured(GameObject).* This method is a callback method that is called if a collision has occurred between this game object and another game object. The game object that collided is passed to the method.

- *Initialize, LoadContent, Update, and Draw.* These methods should be familiar. These are called by the manager at appropriate times during the game loop as dictated by the XNA Application Model.

- *AddObserver, RemoveObserver.* Adds or removes observers of the game object.

Before presenting a detailed description of the *GameObjectManager* and the subclasses of the *GameObject* class, a discussion about bounding volumes is in place.

14.2.2 Bounding Volumes

Collision detection in video games is an advanced subject. In XQUEST 2.0, we have opted for simple intersection-based collision detection using approximate bounding volumes. A bounding volume is a shape that defines the bounds of an object. The ideal bounding volume would be a volume that is perfectly shaped after the object. However, computing and checking for intersection between such volumes is expensive and thus inefficient. Instead, an approximation to the bounding volume using simple shapes is desirable.

Bounding spheres are widely used in 3D application to approximate the ideal bounding volume of an object. Checking for intersection between two spheres is easy and computationally inexpensive. It is a simple matter of checking whether the distance between the two centers is less than the sum of their radiuses. However, not all objects are suited for a sphere approximation. For example, using a sphere to approximate the shape of a banana will result in a lot of "wasted space", since the banana is much longer than it is wide. For such objects, a bounding box is better suited.

Conveniently, XNA provides algorithms for collision detection between several bounding volumes. The three structs *BoundingBox*, *BoundingSphere*, and *BoundingFrustum* can be used to represent such volumes. Each of them has a method *Intersect*, which can be used to check for intersection between all three types interchangeably:

```
BoundingSphere ballSphere = CreateBoundingSphere();
BoundingBox bananaBox = CreateBoundingBox();
BoundingFrustum cameraFrustum = CreateCameraFrustum();

bool ballIntersectsBanana = ballSphere.Intersects(bananaBox);
bool cameraCanSeeBanana = cameraFrustum.Intersects(bananaBox);
```

In the game object system, we wanted to have the game objects support any kind of bounding volume, and let the users choose for themselves the most appropriate one. However, abstracting the bounding volume proved difficult using the aforementioned structs since they do not inherit a common interface. The solution was to write wrappers around these structs and make them implement a common interface. Figure 14.4 shows this interface, and the four wrappers that implement it. Each wrapper struct is prefixed XQ, short for XQUEST, to make them distinct from the XNA structs. Using this pattern also allows for new bounding volumes to be defined without changing the functionality within the game object system.

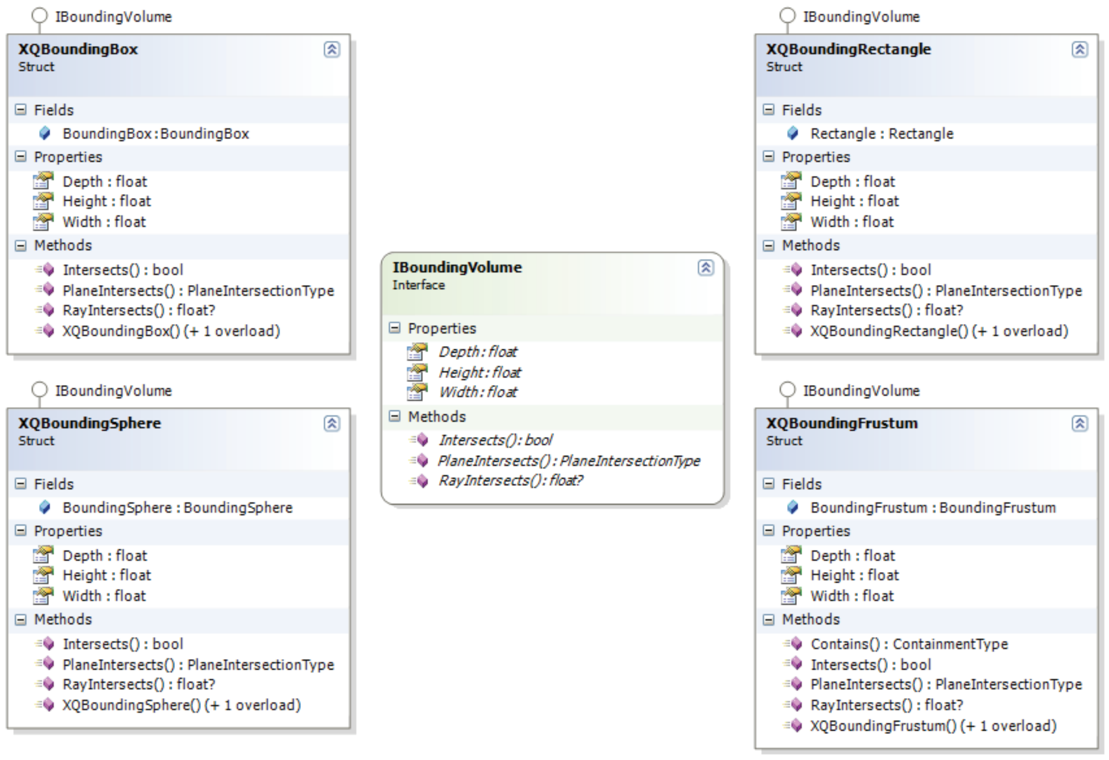
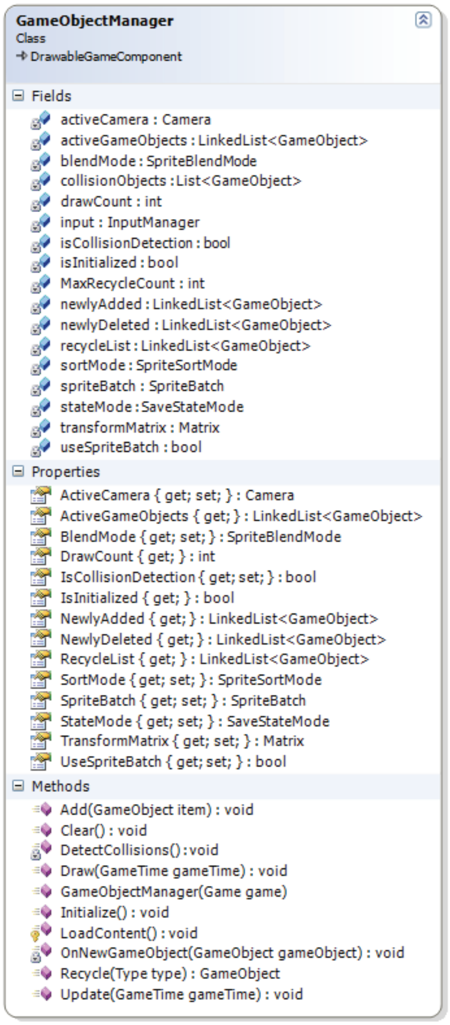


Figure 14.4: Bounding volume interface and implementations wrapping the XNA bounding volume structs.

An *IBoundingVolume* implementation must provide the size of the volume through the *Depth*, *Height*, and *Width* properties. Further, it must provide an *Intersect(IBoundingVolume)* method, and methods for checking intersections against planes and rays, respectively.

14.2.3 GameObjectManager.cs

The *GameObjectManager* class is a game component that manages game objects and provides features such as collision detection and recycling.



```
activeCamera : Camera
activeGameObjects : LinkedList<GameObject>
blendMode : SpriteBlendMode
collisionObjects : List<GameObject>
drawCount : int
input : InputManager
isCollisionDetection : bool
isInitialized : bool
MaxRecycleCount : int
newlyAdded : LinkedList<GameObject>
newlyDeleted : LinkedList<GameObject>
recycleList : LinkedList<GameObject>
sortMode : SpriteSortMode
spriteBatch : SpriteBatch
stateMode : SaveStateMode
transformMatrix : Matrix
useSpriteBatch : bool
```

```
ActiveCamera { get; set; } : Camera
ActiveGameObjects { get; } : LinkedList<GameObject>
BlendMode { get; set; } : SpriteBlendMode
DrawCount { get; } : int
IsCollisionDetection { get; set; } : bool
IsInitialized { get; } : bool
NewlyAdded { get; } : LinkedList<GameObject>
NewlyDeleted { get; } : LinkedList<GameObject>
RecycleList { get; } : LinkedList<GameObject>
SortMode { get; set; } : SpriteSortMode
SpriteBatch { get; set; } : SpriteBatch
StateMode { get; set; } : SaveStateMode
TransformMatrix { get; set; } : Matrix
UseSpriteBatch { get; set; } : bool
```

```
Add(GameObject item) : void
Clear() : void
DetectCollisions() : void
Draw(GameTime gameTime) : void
GameObjectManager(Game game)
Initialize() : void
LoadContent() : void
OnNewGameObject(GameObject gameObject) : void
Recycle(Type type) : GameObject
Update(GameTime gameTime) : void
```

Figure 14.5: The *GameObjectManager* class.

The central idea of the *GameObjectManager* is to have a list of game objects that it manages. The class plugs into the XNA Application Model as a *DrawableGameComponent*. To use the *GameObjectManager* in a game, we first need to initialize it in the *Game* class, typically in the *Initialize* method.

```
GameObjectManager gom = new GameObjectManager(this);
Components.Add(gom);
```

The XNA Application Framework will now handle the *GameObjectManager* instance since we added it to the *Components* collection, which means calling its *Initialize*, *LoadContent*, *Update*, and *Draw* methods automatically.

The *GameObjectManager* uses several lists to manage game objects:

- *activeGameObjects.* This list contains the game objects that are currently alive and should be drawn and updated.

- *collisionObjects.* This list contains game objects that will be checked against each other for collisions.

- *newlyAdded.* This list contains game object that have been added since the last frame. Since there can be many game objects created every frame, we need to store these in a list before putting them all in one go to the *activeGameObjects* list.

- *newlyDeleted.* This list contains game object that have "died" since the last frame. These will be removed in one go in the next frame. If a dead game object's recycle flag is set to true, it will not be deleted, but put into the recycle list.

- *recycleList.* This list contains game objects that have died but have their recycle flag set to true. This means that these game objects can be reused without loading any new content along with them, since they are already initialized. Of course, if we keep adding newly deleted game objects to this list forever, it will grow out of control, so we limit the size of it to a default value of 100.

By looking at *GameObjectManager's Update* method, we can see how these lists are used:

```
public override void Update(GameTime gameTime)
{
  foreach (GameObject gameObject in activeGameObjects)
  {
    switch (gameObject.State)
    {
      case GameObjectState.Alive:
      case GameObjectState.Dying:
        gameObject.Update(gameTime, input);

        // Update the game object's view and projection matrices.
        if (activeCamera != null)
        {
          gameObject.View = activeCamera.View;
          gameObject.Projection = activeCamera.Projection;
        }

        if (isCollisionDetection && gameObject.CollideWithOtherObjects)
          collisionObjects.Add(gameObject);
        break;
      case GameObjectState.Dead:
        newlyDeleted.AddLast(gameObject);
        if (gameObject.Recycle)
        {
          // Maintain capacity.
          while (recycleList.Count >= MaxRecycleCount)
            recycleList.RemoveFirst(); // Remove oldest.

          // Add to recycle list.
          recycleList.AddLast(gameObject);
        }
        break;
    }
  }

  // Remove dead game objects.
  foreach (GameObject gameObject in newlyDeleted)
    activeGameObjects.Remove(gameObject);

  // Add newly added game objects.
  foreach (GameObject gameObject in newlyAdded)
    activeGameObjects.AddLast(gameObject);

  DetectCollisions();

  newlyDeleted.Clear();
  newlyAdded.Clear();
  collisionObjects.Clear();

  base.Update(gameTime);
}
```

We first loop through the *activeGameObjects* list and check the state of the game objects. If a game object is alive or dying, its *Update* method is called. Then, if the *isCollisionDetection* flag is enabled and the game object can collide with other objects, it is added to the *collisionObjects* list. If, on the other hand, the game object's state is set to *GameObjectState.Dead*, it is added to the *newlyDeleted* list. In addition, if its recycle flag is enabled, it will be added to the *recycleList* list. We also check to see that the *recycleList* does not grow too big, in which case we remove the oldest item from this list. After having looped through all active game objects, we remove dead game objects and add new game objects to the *activeGameObjects* list. Next, we call a method called *DetectCollision* (shown below) that does the actual collision detection. The last thing we do is to clear the intermediary lists *newlyDeleted*, *newlyAdded*, and *collisionObjects* so that we start with fresh ones next frame.

```
private void DetectCollisions()
{
  for (int i = 0; i < collisionObjects.Count; i++)
  {
    GameObject go1 = collisionObjects[i];
    for (int j = i + 1; j < collisionObjects.Count; j++)
    {
      GameObject go2 = collisionObjects[j];
      if (go1.CollidesWith(go2))
      {
        go1.CollisionOccured(go2);
        go2.CollisionOccured(go1);
      }
    }
  }
}
```

The way we do collision detection is simple, but not very efficient. That is why we have included the *collideWithOtherObjects* flag in the *GameObject* class, so that game objects that do not need to be involved in collision detection can set this flag to false and save precious processing time.

The rest of the fields of the *GameObjectManager* class are as follows:

- *stateMode*, *blendMode*, and *transformMatrix.* These fields are passed as parameters to *SpriteBatch.Begin*[10] in the *Draw* method. They control how the sprite batch should draw sprites.

- *input.* This is a reference to an *InputManager* instance. This one is passed to the *Update* method of the game objects. When the *GameObjectManager* initializes, it looks for an *InputManager* service. If one is not found, a new one is created.

- *spriteBatch.* A *SpriteBatch* instance used for drawing sprites.

- *isInitialized.* This field indicates whether or not the GameObjectManager is initialized, that is, its *Initialize* and *LoadContent* methods have been called.

- *isCollisionDetection.* This field indicates whether or not to do collision detection.

- *MaxRecycleCount.* This constant field indicates the maximum size of the recycle list, and is set to 100.

- *useSpriteBatch.* Flag indicating whether or not to use a sprite batch for drawing game objects. A 3D game that does not use 2D game objects can safely set this flag to false to avoid the overhead of calling *SpriteBatch.Begin* and *SpriteBatch.End*.

- *drawCount.* Mostly for debugging purposes, this field tells how many game objects were drawn last frame.

The methods of the *GameObjectManager* class are pretty straightforward. We will mention the *Recycle* method only. This method returns a game object of the specified type that currently sits in the recycle list. As an example, consider a game where the player can shoot bullets. A bullet is modeled as a game object, and we may have a *Bullet* class that derives from *GameObject*. Obviously, we want to recycle bullets as much as we can instead of creating new *Bullet* objects each time the player shoots, so we set the recycle flag to true. Now, whenever a bullet hits something, its state is set to *GameObjectState.Dead* and the *GameObjectManager* places it in the recycle list. When the player shoots, we request a recycled *Bullet* instance from the *GameObjectManager*. If we found one, we just saved ourselves from creating a new *Bullet* object, so we just add the recycled object to the *GameObjectManager* again. If not, we have to create a new object. The following code snippet shows what happens when the player shoots:

```
if (isTimeToShoot)
{
  Bullet bullet = (Bullet) gameObjectManager.Recycle(typeof (Bullet));
```

[10] http://msdn2.microsoft.com/en-us/library/microsoft.xna.framework.graphics.spritebatch.begin.aspx

```
  if (bullet == null)
    bullet = new Bullet(Game);

  gameObjectManager.Add(bullet);
}
```

First, we call the *Recycle* method requesting an object of type *Bullet*. If no Bullet object was available for recycling, the *Recycle* method returns null, and we have to create a new *Bullet* object. Lastly, we add the bullet to the *GameObjectManager*. The *Recycle* method is shown below. We simply loop through the *recycleList* and stop at the first game object that matches the specified type. We then remove it from the *recycleList*, reset the state to *GameObjectState.Alive*, and return it.

```
public GameObject Recycle(Type type)
{
  // Find the first game object of type type.
  GameObject gameObject = null;
  foreach (GameObject go in recycleList)
  {
    if (go.GetType() == type)
    {
      gameObject = go;
      break;
    }
  }

  // If we found one, remove it from the recycle list
  // and set its state to Alive.
  if (gameObject != null)
  {
    recycleList.Remove(gameObject);

    // Return the game object back to life.
    gameObject.State = GameObjectState.Alive;
  }

  return gameObject;
}
```

14.2.4 GameObject2D.cs

GameObject2D is an abstract base class for 2D game objects. It uses Vector2 to represent the position, velocity, and acceleration of the game object.

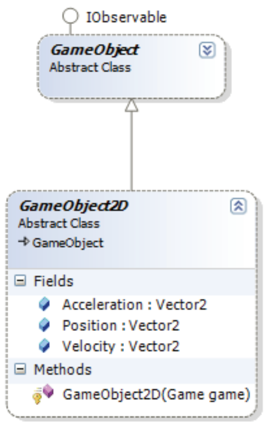


Figure 14.6: The GameObject2D class.

14.2.5 SpriteGameObject.cs

SpriteGameObject is a *GameObject2D* that uses a *Sprite* object for its graphical representation. We have built per-pixel collision detection into *SpriteGameObject*, which can be enabled by setting the *IsPerPixelCollisionDetection* property to true. Per-pixel collision only works on *SpriteGameObject*s, because a sprite texture is needed for extracting the pixel data. Therefore, only per-pixel collision detection between 2D game objects using the *SpriteGameObject* class is supported in XQUEST.

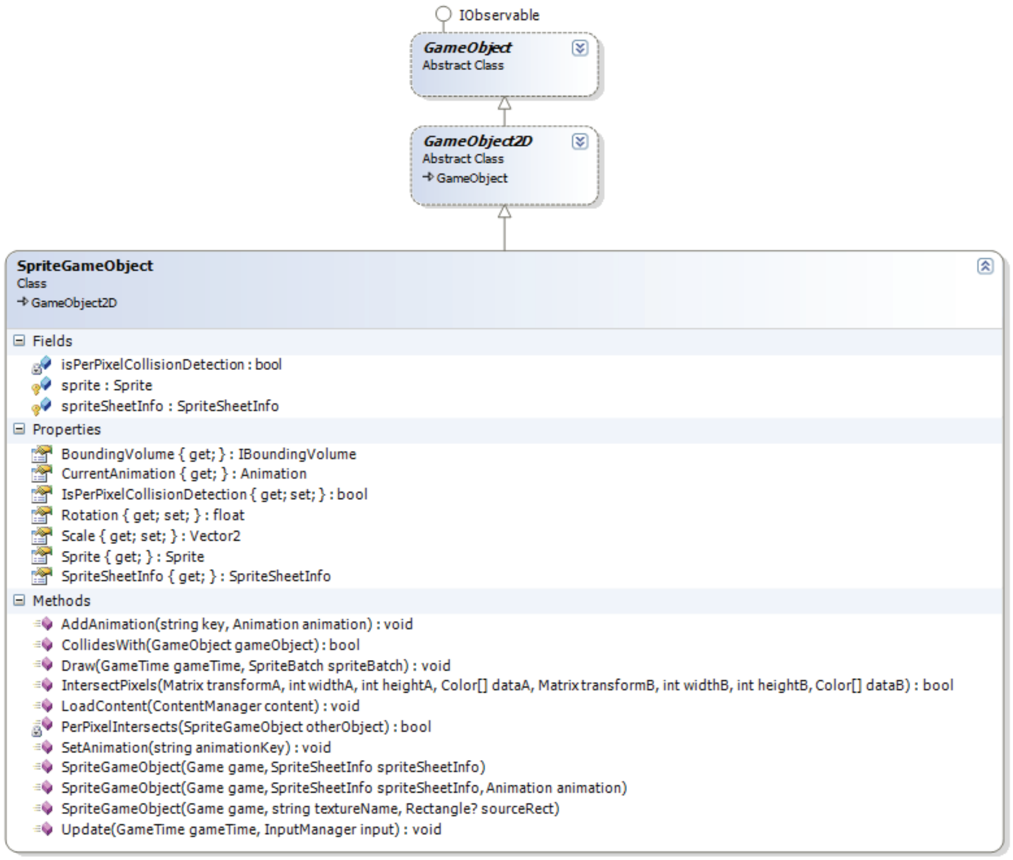

IObservable
GameObject
Abstract Class
GameObject2D
Abstract Class
GameObject
SpriteGameObject
Class
GameObject2D
Fields
isPerPixelCollisionDetection : bool
sprite : Sprite
spriteSheetInfo : SpriteSheetInfo
Properties
BoundingVolume { get; } : IBoundingVolume
CurrentAnimation { get; } : Animation
IsPerPixelCollisionDetection { get; set; } : bool
Rotation { get; set; } : float
Scale { get; set; } : Vector2
Sprite { get; } : Sprite
SpriteSheetInfo { get; } : SpriteSheetInfo
Methods
AddAnimation(string key, Animation animation) : void
CollidesWith(GameObject gameObject) : bool
Draw(GameTime gameTime, SpriteBatch spriteBatch) : void
IntersectPixels(Matrix transformA, int widthA, int heightA, Color[] dataA, Matrix transformB, int widthB, int heightB, Color[] dataB) : bool
LoadContent(ContentManager content) : void
PerPixelIntersects(SpriteGameObject otherObject) : bool
SetAnimation(string animationKey) : void
SpriteGameObject(Game game, SpriteSheetInfo spriteSheetInfo)
SpriteGameObject(Game game, SpriteSheetInfo spriteSheetInfo, Animation animation)
SpriteGameObject(Game game, string textureName, Rectangle? sourceRect)
Update(GameTime gameTime, InputManager input) : void

Figure 14.7: The *SpriteGameObject* class.

There are three constructors. The first one takes as a parameter a *SpriteSheetInfo* object. If this constructor is used, *SpriteGameObject* instantiates the sprite field to an empty AnimatedSprite. Any animations will have to be defined and added using the *AddAnimation* method after the *SpriteGameObject* has been created, or by passing an already existing animation to the second constructor. The third constructor takes as parameters a texture name and a rectangle that defines the portion of the texture to use for the sprite. In the case of using this constructor, *SpriteGameObject* instantiates the sprite field to a simple *Sprite* object with no animation. In this case, calling the methods *AddAnimation* or *SetAnimation* will have no effect.

SpriteGameObject implements the *BoundingVolume* property, defining the bounding volume of the object as a bounding box.

```
public override IBoundingVolume BoundingVolume
{
  get
  { // Calculate bounding box.
    Vector3 min = new Vector3((int)(Position.X - sprite.Origin.X),
                              (int)(Position.Y - sprite.Origin.Y), 0.0f);
    Vector3 max = min + new Vector3((int)(sprite.Width * sprite.Scale.X),
                              (int)(sprite.Height * sprite.Scale.Y), 0.0f);

    return new XQBoundingBox(min, max);
  }
}
```

We use a bounding box without depth, essentially stripping out the third dimension.

SpriteGameObject also supports per-pixel collision detection, enabled by setting the *IsPerPixelCollisionDetection* property to true (it is false by default). While per-pixel collision detection is more accurate than the default bounding-based collision detection, it is considerably more processing intensive, so use of it should be regulated.

14.2.6 GameObject3D.cs

GameObject3D is the abstract base class for game objects that live in a 3D world.

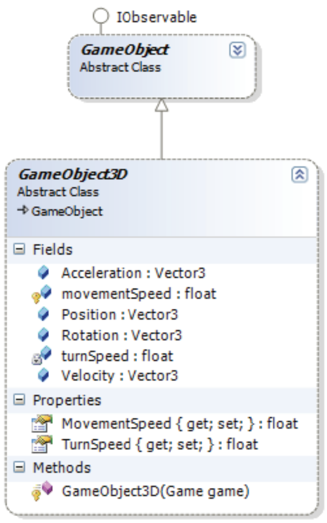


Figure 14.8: The GameObject3D class.

It uses Vector3 to represent the position, velocity, and acceleration. In addition, a rotation vector keeps track of the rotation of the object in all three dimensions. The two fields *movementSpeed* and *turnSpeed* specifies how fast the object moves, and how fast it rotates, respectively.

14.2.7 ModelGameObject.cs

ModelGameObject is a *GameObject3D* that uses an XNA *Model* object for its graphical representation. A *Model* is a mesh hierarchy where each mesh defines geometric data. A model is typically loaded from an external resource created in a modeling tool such as 3D Studio Max or Maya (see Section 5.2).

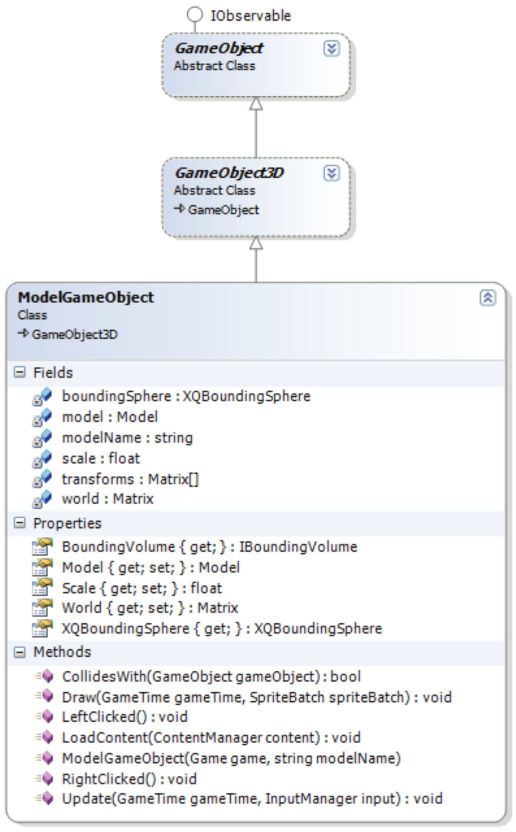

IObservable
GameObject
Abstract Class
GameObject3D
Abstract Class
GameObject
ModelGameObject
Class
GameObject3D
Fields
boundingSphere : XQBoundingSphere
model : Model
modelName : string
scale : float
transforms : Matrix[]
world : Matrix
Properties
BoundingVolume { get; } : IBoundingVolume
Model { get; set; } : Model
Scale { get; set; } : float
World { get; set; } : Matrix
XQBoundingSphere { get; } : XQBoundingSphere
Methods
CollidesWith(GameObject gameObject) : bool
Draw(GameTime gameTime, SpriteBatch spriteBatch) : void
LeftClicked() : void
LoadContent(ContentManager content) : void
ModelGameObject(Game game, string modelName)
RightClicked() : void
Update(GameTime gameTime, InputManager input) : void

Figure 14.9: The *ModelGameObject* class.

The bounding volume is by default approximated as a bounding sphere. The *BoundingVolume* property is implemented to simply return the field *boundingSphere*, which is calculated in the *Update* method:

```
public override void Update(GameTime gameTime, InputManager input)
{
  Matrix scaleMatrix = Matrix.CreateScale(scale);
  Matrix rotationMatrix = Matrix.CreateRotationX(Rotation.X)*
                          Matrix.CreateRotationY(Rotation.Y)*
                          Matrix.CreateRotationZ(Rotation.Z);
  Matrix translationMatrix = Matrix.CreateTranslation(Position);

  world = scaleMatrix*rotationMatrix*translationMatrix;

  // Create combined bounding sphere that encompasses
  // the entire model.
  BoundingSphere sphere = Model.Meshes[0].BoundingSphere;
  for (int i = 1; i < Model.Meshes.Count; i++)
  {
    BoundingSphere meshBoundingSphere = Model.Meshes[i].BoundingSphere;

    BoundingSphere.CreateMerged(ref sphere,
                                ref meshBoundingSphere,
                                out sphere);
  }

  boundingSphere =
    new XQBoundingSphere(sphere.Transform(scaleMatrix * translationMatrix));

  base.Update(gameTime, input);
}
```

Noteworthy fields and methods include:

- *scale.* Determines the scale of the model.
- *world.* A model-wide transformation matrix that transforms the model from local space to world space.
- *transforms.* An array of transformation matrices, one for each mesh in the model. These are used together with the *world* matrix to move the model into world space.
- *LeftClicked(), RightClicked().* These are virtual methods that, when combined with the *ClickableManager*, can be overridden to implement functionality responding to a mouse click on the model.

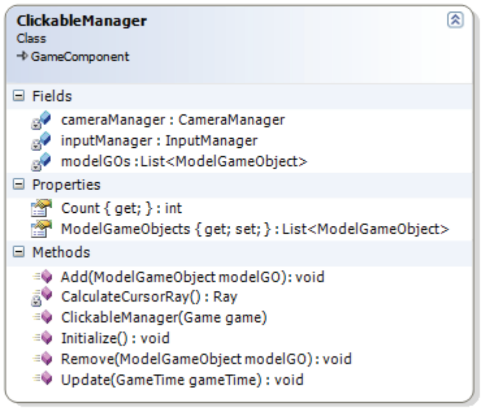
ModelGameObject provides an implementation of the *Draw* method that is based on *BasicEffect*. For applying custom shader effects to the game object, the *Draw* method will need to be overridden, and a call to the base method must be omitted.

```
public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
  transforms = new Matrix[model.Bones.Count];
  model.CopyAbsoluteBoneTransformsTo(transforms);

  foreach (ModelMesh mesh in model.Meshes)
  {
    foreach (BasicEffect effect in mesh.Effects)
    {
      effect.EnableDefaultLighting();

      effect.View = View;
      effect.Projection = Projection;
      effect.World = world*transforms[mesh.ParentBone.Index];
    }
    mesh.Draw();
  }
  base.Draw(gameTime, spriteBatch);
}
```

The *Draw* method demonstrates a simple way to draw a model. First, the bone transforms are copied over to the *transforms* array. Then each mesh is drawn, using *BasicEffect*. Notice that the *World* effect parameter is set to the *world* matrix multiplied by the bone transformation matrix for each mesh. This puts the model into world space.

14.2.8 ClickableManager.cs

The *ClickableManager* class (Figure 14.10) is used to handle 3D game objects that can be "selected" or clicked with the mouse.



ClickableManager
Class
→ GameComponent

Fields
- cameraManager : CameraManager
- inputManager : InputManager
- modelGOs :List<ModelGameObject>

Properties
- Count { get; } : int
- ModelGameObjects { get; set; } : List<ModelGameObject>

Methods
- Add(ModelGameObject modelGO): void
- CalculateCursorRay() : Ray
- ClickableManager(Game game)
- Initialize() : void
- Remove(ModelGameObject modelGO) : void
- Update(GameTime gameTime) : void

Figure 14.10: The *ClickableManager* class.

The class manages a collection of *ModelGameObject3D* objects. When the mouse is clicked, the manager will calculate which object in the managed collection, if any, is clicked. It does so by a technique called picking[12]. The problem of picking is to translate the 2D coordinates of the mouse cursor into 3D coordinates within the game world. We solve this by calculating a cursor ray starting at the camera's near plane and ending at the far plane:

```
private Ray CalculateCursorRay()
{
  Matrix projection = cameraManager.ActiveCamera.Projection;
  Matrix view = cameraManager.ActiveCamera.View;

  Vector3 rayStart =
    Game.GraphicsDevice.Viewport.Unproject(
      new Vector3(inputManager.Mouse.PositionX, inputManager.Mouse.PositionY,
                  0.01f),
      projection, view, Matrix.Identity);

  Vector3 rayEnd =
    Game.GraphicsDevice.Viewport.Unproject(
      new Vector3(inputManager.Mouse.PositionX, inputManager.Mouse.PositionY,
                  0.99f),
      projection, view, Matrix.Identity);

  return new Ray(rayStart, Vector3.Normalize(rayEnd - rayStart));
}
```

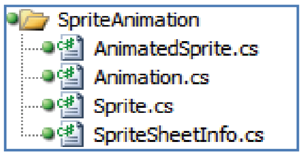
We can then use this ray to check for intersection against any of the objects managed by *ClickableManager* using those objects' bounding volumes:

```
Ray cursorRay = CalculateCursorRay();
foreach (ModelGameObject modelGO in modelGOs)
{
  float? distance = modelGO.BoundingVolume.RayIntersects(cursorRay);

  if (distance != null)
  {
    // The ray hit the object.
    ...
```

Depending on which mouse button was pressed, the appropriate method is called in the *ModelGameObject* (*LeftClicked*/*RightClicked*).

14.3 XQUEST.SpriteAnimation

From XQUEST 1.0, this part consisted of three classes as shown in Figure 14.11.



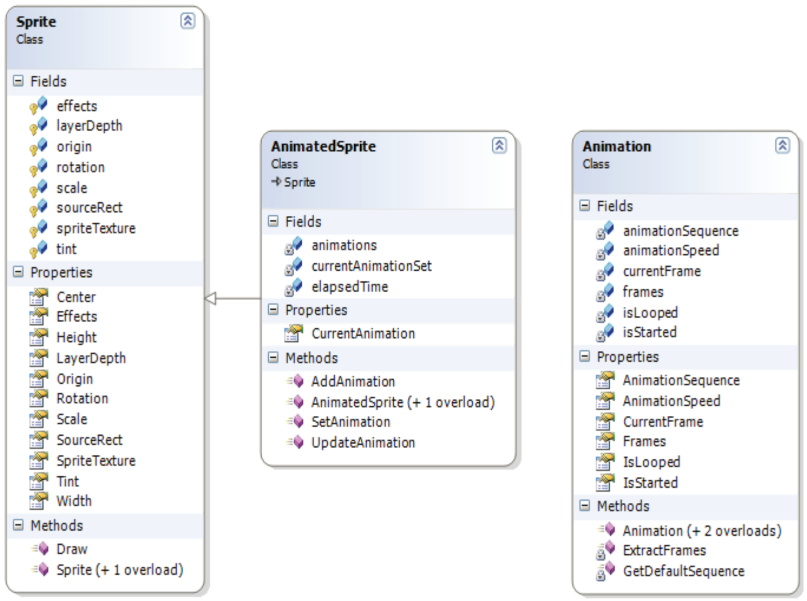


Figure 14.11: The classes of the sprite animation framework in XQUEST 1.0.

14.3.1 Sprite class

The Sprite class used to take in a reference to a *Texture2D* object in its constructor.

```
Texture2D texture = Content.Load<Texture2D>("texture");
Sprite sprite = new Sprite(texture);
```

The problem with this approach was that we had to have such a reference before we could create an instance of the sprite class. This can be problematic in situations where we want game objects to load themselves and not be dependent on other objects to load the content for them. So, in an effort to make the sprite class be more self-contained we changed the constructor to take in a string indicating the name of the texture to load.

```
public Sprite(string textureName)
```

This way, we do not have to load any texture before creating an instance of the *Sprite* class. We simply pass the **name** of the texture to load, and then the texture can be loaded by an external manager class when appropriate, for instance when the graphics device reset, or at any other occurrence that causes the content to be lost and in need for a reload. This is done by calling the new method *LoadTexture* of the *Sprite* class, passing it a reference to the *ContentManager* from which to load the texture.

```
Sprite sprite = new Sprite(“cow”);
sprite.LoadTexture(Content);
```

The main advantage of the new approach is that we can better design game objects that use sprite instances to fit in with the XNA Application Model, which expects every resource to be loaded in the *LoadContent* method.

14.3.2 Animation

We removed the constructor of the *Animation* class that accepts information about a sprite sheet. Instead, we created a structure called *SpriteSheetInfo* that contains all this information. Using the example from our depth study report, this was the way we created an animated sprite from the dragon sprite sheet:

```
Texture2D spriteTexture = content.Load<Texture2D>(@"content\dragonsheet");

Animation flyAnimation = new Animation(spriteTexture.Width,
    spriteTexture.Height, 104, 52, 0, 0, 0.1f, true);
flyAnimation.AnimationSequence = new int[] { 0, 1, 2, 3, 4, 3, 2, 1 };
sprite = new AnimatedSprite(spriteTexture, flyAnimation);
```

Using the new and improved *Animation* class along with the new *SpriteSheetInfo* struct, the code for creating the animated sprite is now:

```
SpriteSheetInfo ssi = new SpriteSheetInfo(@”content\dragonsheet”, 104, 52, 5, 1, 0, 0);
Animation flyAnimation = new Animation(ssi.GetFrames(), 0.1f, true);
sprite = new AnimatedSprite(ssi, flyAnimation);
```

As we can see, we have separated concerns in the *Animation* class; the sprite sheet information and the animation information are now in separate classes. The code is cleaner and easier to use.

Figure 14.12 shows the updated classes of the sprite animation framework.

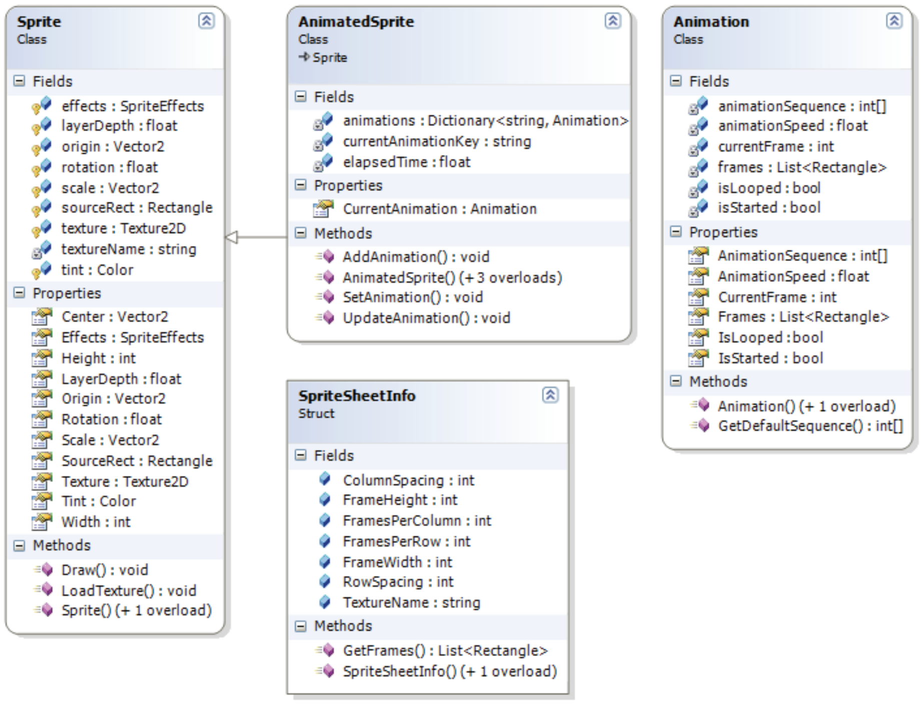
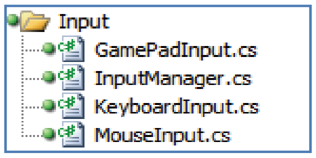


Figure 14.12: The classes of the modified sprite animation framework in XQUEST 2.0.

14.4 XQUEST.Input



This namespace provides functionality for input querying and processing from keyboards, mice, and Xbox 360 game pads. The central class is *InputManager*, which provides a convenient interface for the most common input querying tasks. The other classes (*KeyboardInput*, *GamePadInput*, and *MouseInput*) each manage the state of keyboards, game pads, and the mouse, respectively. Each of these classes maintains a *current* and a *previous* state (see Figure 14.13). This is so we can distinguish between a key or button being *held down*, and a key or button that was *just pressed*. If the current state differs from the previous state, we know that a key or button was *just pressed*. The difference is that a key or button *down* event will happen as long as that key or button is being held down, while a key or button *pressed* event only occurs once. The importance of differentiating between these two events can be emphasized through an example. Imagine a game wherein the player shoots bullets by pressing the space bar on the keyboard. The code for checking would be something like this:

```
if (inputManager.Keyboard.IsKeyDown(Keys.Space))
  player.ShootBullet();
```

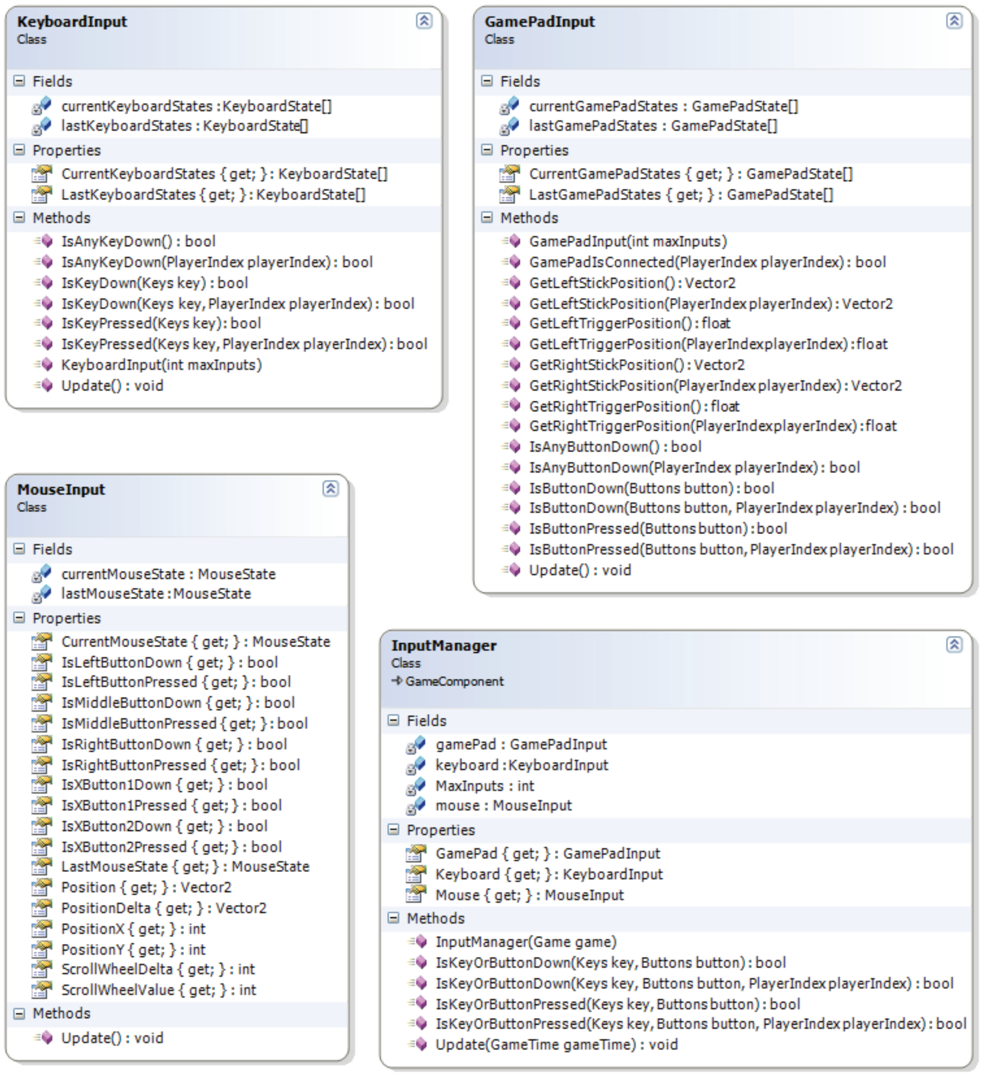
However, the *IsKeyDown* method will return true as long as the space bar is being held down, resulting in a bullet being fired each time this code is reached in the game loop. If say, the game runs at 60 frames per second, the player will fire 60 bullets every second as long as he or she holds the space bar down. Obviously, the desired effect is to only shoot one bullet each time the player presses the space bar. This is achieved by using the *IsKeyPressed* method instead:

```
if (inputManager.Keyboard.IsKeyPressed(Keys.Space))
  player.ShootBullet();
```

Now, the player will shoot a bullet every time the space bar is pressed down, and he or she must release it and press it again to shoot another. The difference between the *Is*Down* and *Is*Pressed* methods is shown below using the keyboard as an example.

```
public bool IsKeyDown(Keys key, PlayerIndex playerIndex)
{
  return currentKeyboardStates[(int)playerIndex].IsKeyDown(key);
}
```

```
public bool IsKeyPressed(Keys key, PlayerIndex playerIndex)
{
  return
    currentKeyboardStates[(int)playerIndex].IsKeyDown(key) &&
    lastKeyboardStates[(int)playerIndex].IsKeyUp(key);
}
```



```
KeyboardInput
Class

Fields
  currentKeyboardStates : KeyboardState[]
  lastKeyboardStates : KeyboardState[]
Properties
  CurrentKeyboardStates { get; } : KeyboardState[]
  LastKeyboardStates { get; } : KeyboardState[]
Methods
  IsAnyKeyDown() : bool
  IsAnyKeyDown(PlayerIndex playerIndex) : bool
  IsKeyDown(Keys key) : bool
  IsKeyDown(Keys key, PlayerIndex playerIndex) : bool
  IsKeyPressed(Keys key) : bool
  IsKeyPressed(Keys key, PlayerIndex playerIndex) : bool
  KeyboardInput(int maxInputs)
  Update() : void
```

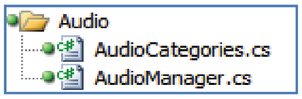
```
GamePadInput
Class

Fields
  currentGamePadStates : GamePadState[]
  lastGamePadStates : GamePadState[]
Properties
  CurrentGamePadStates { get; } : GamePadState[]
  LastGamePadStates { get; } : GamePadState[]
Methods
  GamePadInput(int maxInputs)
  GamePadIsConnected(PlayerIndex playerIndex) : bool
  GetLeftStickPosition() : Vector2
  GetLeftStickPosition(PlayerIndex playerIndex) : Vector2
  GetLeftTriggerPosition() : float
  GetLeftTriggerPosition(PlayerIndex playerIndex) : float
  GetRightStickPosition() : Vector2
  GetRightStickPosition(PlayerIndex playerIndex) : Vector2
  GetRightTriggerPosition() : float
  GetRightTriggerPosition(PlayerIndex playerIndex) : float
  IsAnyButtonDown() : bool
  IsAnyButtonDown(PlayerIndex playerIndex) : bool
  IsButtonDown(Buttons button) : bool
  IsButtonDown(Buttons button, PlayerIndex playerIndex) : bool
  IsButtonPressed(Buttons button) : bool
  IsButtonPressed(Buttons button, PlayerIndex playerIndex) : bool
  Update() : void
```

```
MouseInput
Class

Fields
  currentMouseState : MouseState
  lastMouseState : MouseState
Properties
  CurrentMouseState { get; } : MouseState
  IsLeftButtonDown { get; } : bool
  IsLeftButtonPressed { get; } : bool
  IsMiddleButtonDown { get; } : bool
  IsMiddleButtonPressed { get; } : bool
  IsRightButtonDown { get; } : bool
  IsRightButtonPressed { get; } : bool
  IsXButton1Down { get; } : bool
  IsXButton1Pressed { get; } : bool
  IsXButton2Down { get; } : bool
  IsXButton2Pressed { get; } : bool
  LastMouseState { get; } : MouseState
  Position { get; } : Vector2
  PositionDelta { get; } : Vector2
  PositionX { get; } : int
  PositionY { get; } : int
  ScrollWheelDelta { get; } : int
  ScrollWheelValue { get; } : int
Methods
  Update() : void
```

```
InputManager
Class
→ GameComponent

Fields
  gamePad : GamePadInput
  keyboard : KeyboardInput
  MaxInputs : int
  mouse : MouseInput
Properties
  GamePad { get; } : GamePadInput
  Keyboard { get; } : KeyboardInput
  Mouse { get; } : MouseInput
Methods
  InputManager(Game game)
  IsKeyOrButtonDown(Keys key, Buttons button) : bool
  IsKeyOrButtonDown(Keys key, Buttons button, PlayerIndex playerIndex) : bool
  IsKeyOrButtonPressed(Keys key, Buttons button) : bool
  IsKeyOrButtonPressed(Keys key, Buttons button, PlayerIndex playerIndex) : bool
  Update(GameTime gameTime) : void
```

Figure 14.13: The classes in the XQUEST.Input namespace.

14.5 XQUEST.Audio


Audio
AudioCategories.cs
AudioManager.cs

The Audio namespace contains two classes; *AudioCategories* and *AudioManager*. The *AudioCategories* class (Figure 14.14) is responsible for managing the audio categories that have been specified in the XACT tool. Each category can have a volume level assigned to them. Also, a global volume scale can be applied to all category volumes. The *AudioCategories* class is used by the *AudioManager* class (Figure 14.15). Since there is no way of loading the categories from the XACT audio project, the audio categories must be specified using the *SpecifyCategories* method. This method accepts string arguments, each specifying a category.

The *AudioManager* provides methods for pausing, resuming, stopping, and playing cues that are defined in the XACT audio project. It exposes the XNA audio classes *WaveBank*, *SoundBank*, and *AudioEngine* should the user want to manipulate these directly.

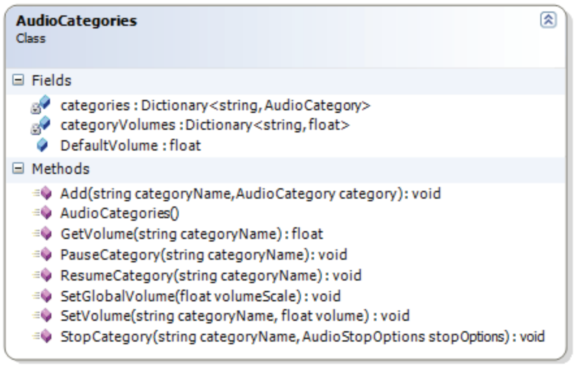

```
AudioCategories
Class

Fields
  categories : Dictionary<string, AudioCategory>
  categoryVolumes : Dictionary<string, float>
  DefaultVolume : float
Methods
  Add(string categoryName, AudioCategory category) : void
  AudioCategories()
  GetVolume(string categoryName) : float
  PauseCategory(string categoryName) : void
  ResumeCategory(string categoryName) : void
  SetGlobalVolume(float volumeScale) : void
  SetVolume(string categoryName, float volume) : void
  StopCategory(string categoryName, AudioStopOptions stopOptions) : void
```

Figure 14.14: The *AudioCategories* class.

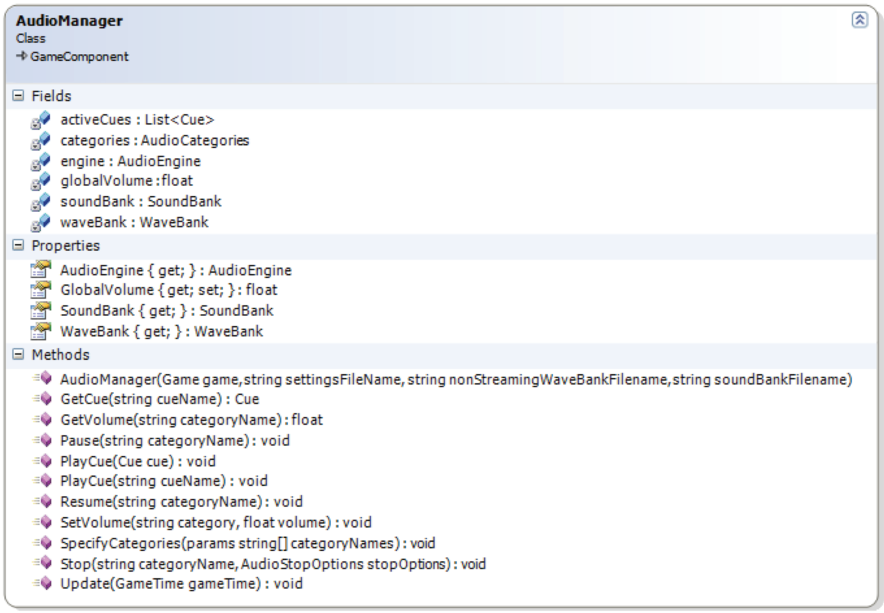
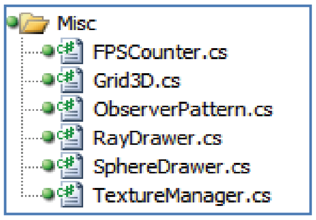

```
AudioManager
Class
→ GameComponent

Fields
  activeCues : List<Cue>
  categories : AudioCategories
  engine : AudioEngine
  globalVolume : float
  soundBank : SoundBank
  waveBank : WaveBank
Properties
  AudioEngine { get; } : AudioEngine
  GlobalVolume { get; set; } : float
  SoundBank { get; } : SoundBank
  WaveBank { get; } : WaveBank
Methods
  AudioManager(Game game, string settingsFileName, string nonStreamingWaveBankFilename, string soundBankFilename)
  GetCue(string cueName) : Cue
  GetVolume(string categoryName) : float
  Pause(string categoryName) : void
  PlayCue(Cue cue) : void
  PlayCue(string cueName) : void
  Resume(string categoryName) : void
  SetVolume(string category, float volume) : void
  SpecifyCategories(params string[] categoryNames) : void
  Stop(string categoryName, AudioStopOptions stopOptions) : void
  Update(GameTime gameTime) : void
```

Figure 14.15: The *AudioManager* class.

14.6 XQUEST.Misc

The *Misc* namespace contains functionality that does not fit into any other namespace. These classes are utility or debug classes that are used by other parts of XQUEST. A brief description of these classes is provided in Table 14.2.

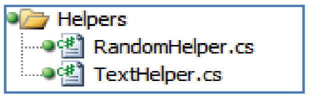


Misc
FPSCounter.cs
Grid3D.cs
ObserverPattern.cs
RayDrawer.cs
SphereDrawer.cs
TextureManager.cs

Table 14.2: Description of the classes in the XQUEST.Misc namespace.

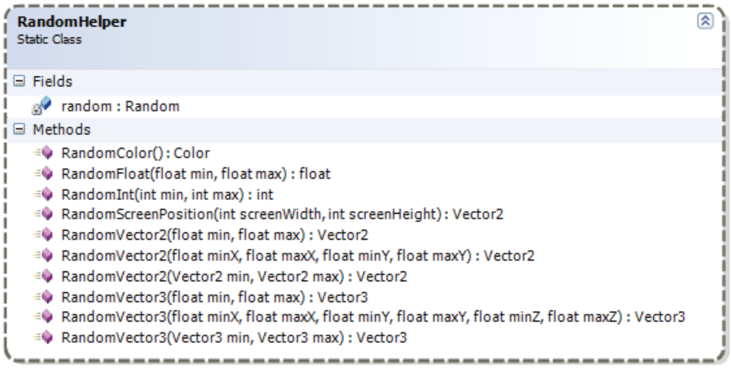
Class	**Description**
FPSCounter	A *DrawableGameComponent* that prints out the current frames per second. This class is useful for measuring performance of a game. Note however that it is pointless to count the frames per second if the game runs in fixed timestep mode, since in this mode the game always runs at 60 fps.
Grid3D	A *DrawableGameComponent* that draws a grid in 3D space. The size and appearance of the grid can be specified through the properties of the class.
ObserverPattern	An implementation of the observer design pattern. We use this pattern in the game object system to allow game objects to be observed by cameras.
RayDrawer	A *DrawableGameComponent* that visualizes an XNA framework *Ray* struct.
SphereDrawer	A *DrawableGameComponent* that visualizes an XNA framework *BoundingSphere* struct.
TextureManager	A static class that keeps track of textures in a dictionary for easy look-up anywhere in the code.

14.7 XQUEST.Helpers



This namespace contains helper classes that provide shortcut utility to common tasks. It contains two classes; *RandomHelper* (Figure 14.16) and *TextHelper* (Figure 14.17). *RandomHelper* provides convenient shortcut methods to getting random numbers in several formats. The class is static and can therefore be used from anywhere. It can generate random colors, random floating point number, random integers, random positions on the screen (taking the current screen resolution into account), random two-dimensional vectors, and random three-dimensional vectors.

TextHelper provides functionality for drawing 2D text using sprite fonts. It contains methods for drawing text centered around a point and for drawing text with shadows. The class is static, meaning the methods are accessible from any scope. However, a *SpriteFont* and *SpriteBatch* instance is required to be set on the class before any text drawing can occur.

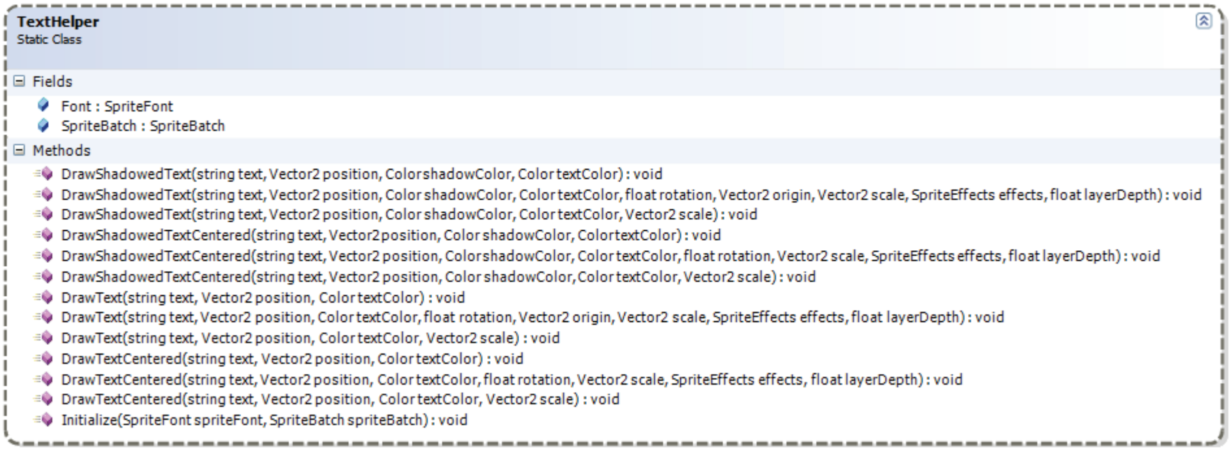


RandomHelper
Static Class

Fields
- random : Random

Methods
- RandomColor() : Color
- RandomFloat(float min, float max) : float
- RandomInt(int min, int max) : int
- RandomScreenPosition(int screenWidth, int screenHeight) : Vector2
- RandomVector2(float min, float max) : Vector2
- RandomVector2(float minX, float maxX, float minY, float maxY) : Vector2
- RandomVector2(Vector2 min, Vector2 max) : Vector2
- RandomVector3(float min, float max) : Vector3
- RandomVector3(float minX, float maxX, float minY, float maxY, float minZ, float maxZ) : Vector3
- RandomVector3(Vector3 min, Vector3 max) : Vector3

Figure 14.16: The *RandomHelper* class.



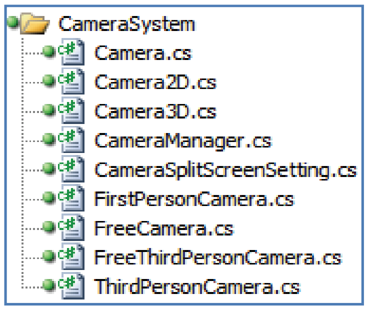
TextHelper
Static Class

Fields
- Font : SpriteFont
- SpriteBatch : SpriteBatch

Methods
- DrawShadowedText(string text, Vector2 position, Color shadowColor, Color textColor) : void
- DrawShadowedText(string text, Vector2 position, Color shadowColor, Color textColor, float rotation, Vector2 origin, Vector2 scale, SpriteEffects effects, float layerDepth) : void
- DrawShadowedText(string text, Vector2 position, Color shadowColor, Color textColor, Vector2 scale) : void
- DrawShadowedTextCentered(string text, Vector2 position, Color shadowColor, Color textColor) : void
- DrawShadowedTextCentered(string text, Vector2 position, Color shadowColor, Color textColor, float rotation, Vector2 scale, SpriteEffects effects, float layerDepth) : void
- DrawShadowedTextCentered(string text, Vector2 position, Color shadowColor, Color textColor, Vector2 scale) : void
- DrawText(string text, Vector2 position, Color textColor) : void
- DrawText(string text, Vector2 position, Color textColor, float rotation, Vector2 origin, Vector2 scale, SpriteEffects effects, float layerDepth) : void
- DrawText(string text, Vector2 position, Color textColor, Vector2 scale) : void
- DrawTextCentered(string text, Vector2 position, Color textColor) : void
- DrawTextCentered(string text, Vector2 position, Color textColor, float rotation, Vector2 scale, SpriteEffects effects, float layerDepth) : void
- DrawTextCentered(string text, Vector2 position, Color textColor, Vector2 scale) : void
- Initialize(SpriteFont spriteFont, SpriteBatch spriteBatch) : void

Figure 14.17: The *TextHelper* class.

14.8 XQUEST.CameraSystem


CameraSystem
Camera.cs
Camera2D.cs
Camera3D.cs
CameraManager.cs
CameraSplitScreenSetting.cs
FirstPersonCamera.cs
FreeCamera.cs
FreeThirdPersonCamera.cs
ThirdPersonCamera.cs

The camera system is a hierarchy of cameras that includes several different common types of cameras, both for 2D and 3D games. It includes a manager class for use in scenarios where a game supports switching between several types of cameras (e.g. first person and third person views). The cameras can be used in a split screen setting by specifying the region of the screen to point it at through the *CameraSplitScreenSetting* enumeration (Figure 14.19). *Fullscreen* indicates that the camera should present the scene on the entire screen, *TopHalf* that it should use the top half, *BottomHalf* that it should use the bottom half, and so on.

Figure 14.18 shows the camera hierarchy.

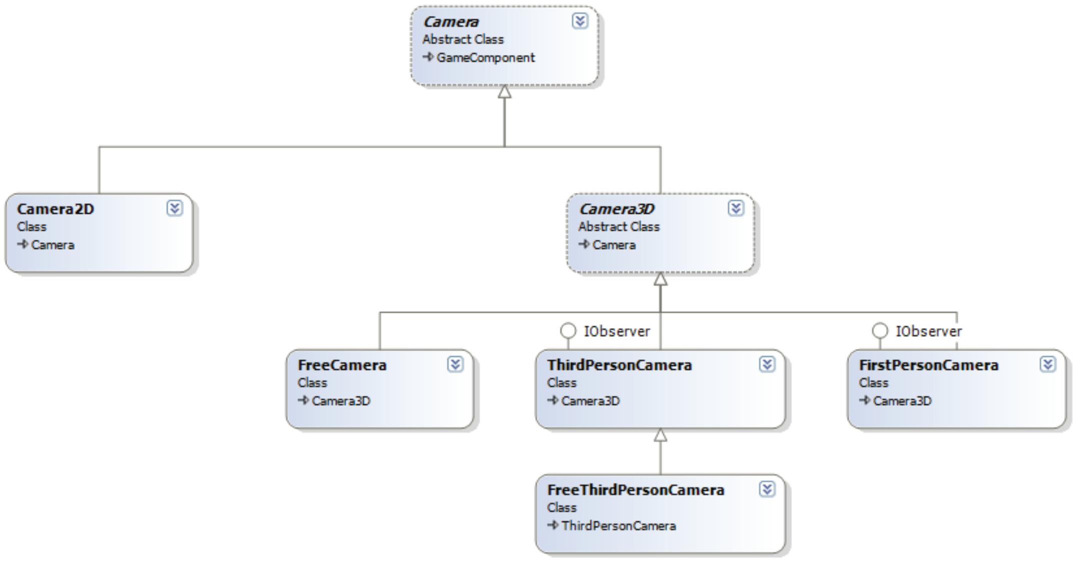
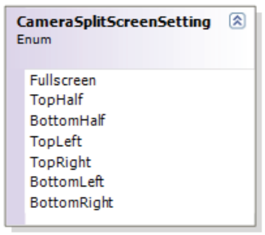


Figure 14.18: The camera hierarchy.

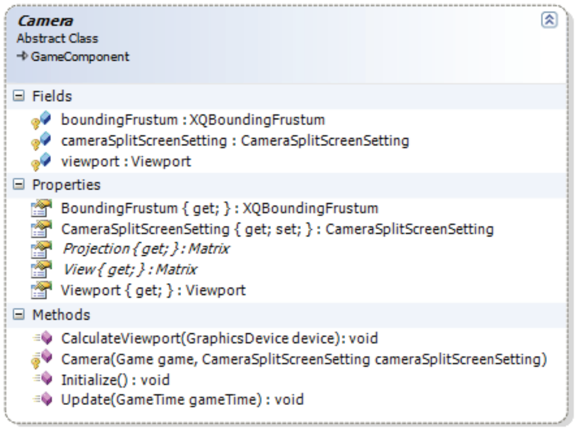

CameraSplitScreenSetting
Enum

Fullscreen
TopHalf
BottomHalf
TopLeft
TopRight
BottomLeft
BottomRight

Figure 14.19: The *CameraSplitScreenSetting* enum.

14.8.1 Camera.cs

All cameras derive from the abstract base class *Camera* (Figure 14.20). Itself derives from *GameComponent* and a *CameraSplitScreenSetting* enum value should be passed to its constructor, indicating the portion of the screen the camera should be set up to view. *Camera* defines two abstract properties that need to be implemented in derived classes; *View* and *Projection*. These are matrices that define the view and orientation of the camera, and how the camera projects its view onto the scene, respectively.



Camera
Abstract Class
GameComponent
Fields
boundingFrustum : XQBoundingFrustum
cameraSplitScreenSetting : CameraSplitScreenSetting
viewport : Viewport
Properties
BoundingFrustum { get; } : XQBoundingFrustum
CameraSplitScreenSetting { get; set; } : CameraSplitScreenSetting
Projection { get; } : Matrix
View { get; } : Matrix
Viewport { get; } : Viewport
Methods
CalculateViewport(GraphicsDevice device) : void
Camera(Game game, CameraSplitScreenSetting cameraSplitScreenSetting)
Initialize() : void
Update(GameTime gameTime) : void

Figure 14.20: The base class *Camera*.

Further, the *Camera* class includes a bounding frustum that encompasses the area in 3D space currently visible from the camera's point of view. It is calculated every frame in the *Update* method like so:

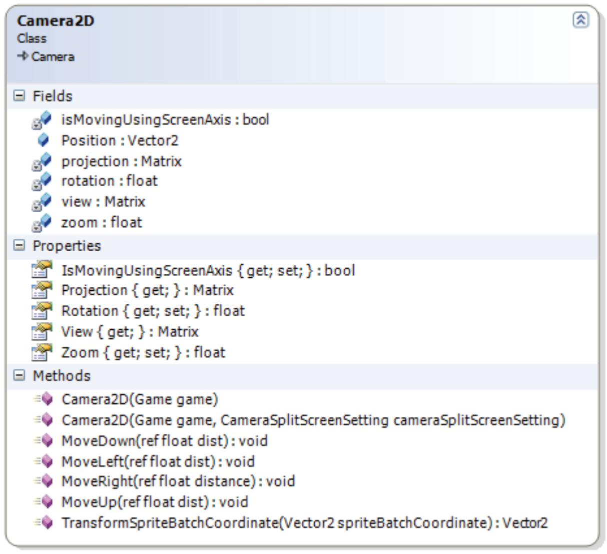
```
boundingFrustum = new XQBoundingFrustum(View*Projection);
```

The bounding frustum is used in the *GameObjectManager* to decide whether or not to draw game objects. If a game object is not inside the camera's view, there is no point in drawing it – this technique is called frustum culling. Since the bounding frustum of the camera is an *IBoundingVolume* (see Section 14.2.2), we can check for intersection against a game object's *IBoundingVolume* to determine whether the game object is visible for the camera. Frustum culling is very efficient if the scene contains a lot of game objects.

The *CalculateViewport* method sets up the viewport field. It needs access to the graphics device in order to retrieve the current screen resolution. The method sets the viewport based on the split screen setting.

14.8.2 Camera2D.cs

Camera2D (Figure 14.21) is a concrete camera implementation for use in 2D games where the game world exceeds the limits of the screen resolution. Games that typically need such cameras are scrolling games like horizontal-scrolling platform games or vertical-scrolling shoot-em-up's.



Camera2D
Class
→ Camera

Fields
isMovingUsingScreenAxis : bool
Position : Vector2
projection : Matrix
rotation : float
view : Matrix
zoom : float

Properties
IsMovingUsingScreenAxis { get; set; } : bool
Projection { get; } : Matrix
Rotation { get; set; } : float
View { get; } : Matrix
Zoom { get; set; } : float

Methods
Camera2D(Game game)
Camera2D(Game game, CameraSplitScreenSetting cameraSplitScreenSetting)
MoveDown(ref float dist) : void
MoveLeft(ref float dist) : void
MoveRight(ref float distance) : void
MoveUp(ref float dist) : void
TransformSpriteBatchCoordinate(Vector2 spriteBatchCoordinate) : Vector2

Figure 14.21: The *Camera2D* class.

Unlike the coordinate system that *SpriteBatch* uses, Camera2D uses a coordinate system with the origin in the middle of the screen as shown in Figure 14.22. The reason for this is to support zooming. With the coordinates spread evenly over the screen, natural zooming is easier to achieve. It uses the origin as the reference point and scales everything in the game world based on their offsets. Users of *Camera2D* can transform a *SpriteBatch* coordinate to a *Camera2D* coordinate with the *TransformSpriteBatchCoordinate* method. This is useful when using *Camera2D* together with *SpriteGameObject*s.

If the *isUsingMovingScreenAxis* field is set to true, the camera will pan relative to the screen axis when the camera is rotated. Otherwise, it will move to one direction regardless of the current rotation.

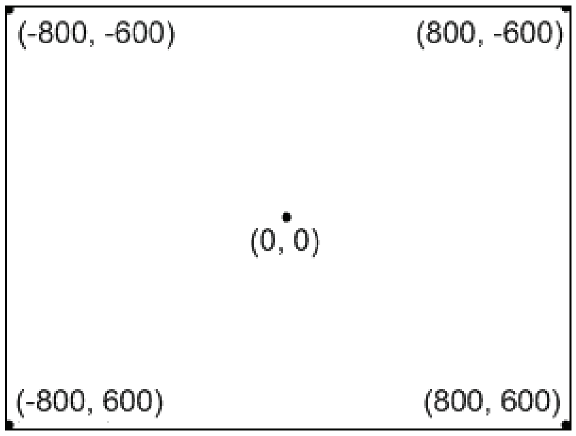
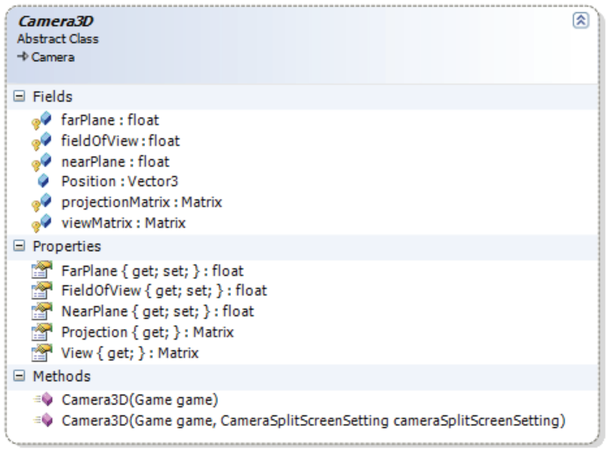


Figure 14.22: Screen coordinate layout used by *Camera2D* illustrated in a 1600x1200 screen resolution.

14.8.3 Camera3D.cs

Camera3D (Figure 14.23) provides a base class for 3D cameras that view a game world set in 3D space.



Camera3D
Abstract Class
→ Camera

Fields
- farPlane : float
- fieldOfView : float
- nearPlane : float
- Position : Vector3
- projectionMatrix : Matrix
- viewMatrix : Matrix

Properties
- FarPlane { get; set; } : float
- FieldOfView { get; set; } : float
- NearPlane { get; set; } : float
- Projection { get; } : Matrix
- View { get; } : Matrix

Methods
- Camera3D(Game game)
- Camera3D(Game game, CameraSplitScreenSetting cameraSplitScreenSetting)

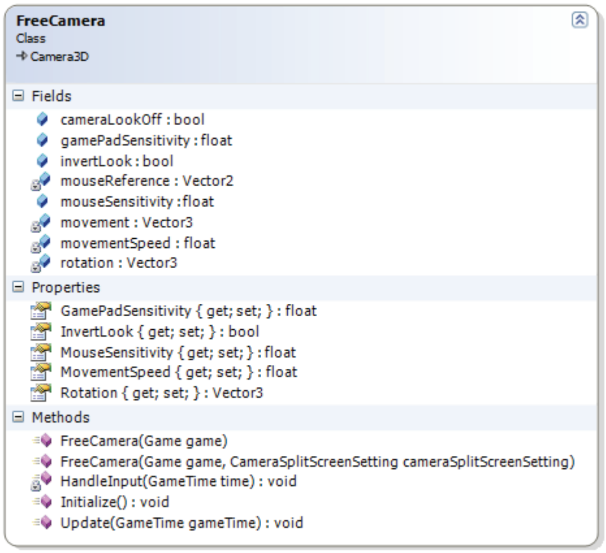
Figure 14.23: The *Camera3D* base class.

It provides a default implementation of the *View* and *Projection* properties where the view matrix is calculated using the *position* field along with pointing the camera to look at the origin of the coordinate system. The projection matrix used is a perspective projection matrix based on a field of view. The matrices are calculated using the static helper methods on the XNA *Matrix* struct:

```
viewMatrix = Matrix.CreateLookAt(Position, Vector3.Zero, Vector3.Up);
projectionMatrix =
      Matrix.CreatePerspectiveFieldOfView(fieldOfView, viewport.AspectRatio,
                                          nearPlane, farPlane);
```

14.8.4 FreeCamera.cs

FreeCamera (Figure 14.24) is a concrete implementation of a 3D camera, and it can be moved around freely using a combination of the keyboard and mouse, or a game pad. We use the term *free* to indicate that this camera is not stationary, does not follow any predetermined path, or is not set up to follow any game object around. The mouse rotates the camera around both the y-axis and z-axis, while holding the right mouse button will move the camera in direction it is currently looking. The view and projection matrices are calculated based on this input.


FreeCamera
Class
→ Camera3D

Fields
cameraLookOff : bool
gamePadSensitivity : float
invertLook : bool
mouseReference : Vector2
mouseSensitivity : float
movement : Vector3
movementSpeed : float
rotation : Vector3

Properties
GamePadSensitivity { get; set; } : float
InvertLook { get; set; } : bool
MouseSensitivity { get; set; } : float
MovementSpeed { get; set; } : float
Rotation { get; set; } : Vector3

Methods
FreeCamera(Game game)
FreeCamera(Game game, CameraSplitScreenSetting cameraSplitScreenSetting)
HandleInput(GameTime time) : void
Initialize() : void
Update(GameTime gameTime) : void

Figure 14.24: The *FreeCamera* class.

14.8.5 ThirdPersonCamera.cs and FreeThirdPersonCamera.cs

ThirdPersonCamera (Figure 14.25) is a concrete camera implementation that views an object in the game world from a third person view. It implements the *IObserver* interface, which means it can be attached to any *GameObject3D*. A game object that has this camera as an observer will call the camera's *Notify* method when its position changes. This way, the camera can update itself to maintain the third person view whenever the game object moves. The *Notify* method is show below.

```
public void Notify(object someObject)
{
  if (!(someObject is GameObject3D))
    return;

  GameObject3D subject = someObject as GameObject3D;

  Vector3 size =
    new Vector3(subject.BoundingVolume.Width, subject.BoundingVolume.Height,
                subject.BoundingVolume.Depth);

  // Camera should be a little behind and a little up from the subject.
  defaultCameraPosition.Z = 3*size.Z;
  defaultCameraPosition.Y = 2*size.Y;

  Position.Z = defaultCameraPosition.Z + deltaZ;
  Position.Y = defaultCameraPosition.Y + deltaY;

  heightChangeSpeed = 0.1f*size.Y;
  zoomChangeSpeed = 0.1f*size.Z;

  maxCameraPositionY = 5*size.Y;
  minCameraPositionY = 0;
  maxCameraPositionZ = 5*size.Z;
  minCameraPositionZ = size.Z;

  // Camera's position behind game object
  Vector3 thirdPersonReference = new Vector3(0, Position.Y, -Position.Z);

  // Camera should be rotated along with the subject.
  Matrix rotationMatrix = Matrix.CreateRotationY(this is FreeThirdPersonCamera ?
cameraYaw : subject.Rotation.Y);

  // Create a vector pointing the direction the camera is facing.
  Vector3 transformedReference =
    Vector3.Transform(thirdPersonReference, rotationMatrix);

  // Calculate the position the camera is looking from.
  Vector3 cameraPosition = transformedReference + subject.Position;

  // Set up the view matrix
  viewMatrix =
    Matrix.CreateLookAt(cameraPosition, subject.Position,
                        new Vector3(0.0f, 1.0f, 0.0f));

  // Set up the projection matrix
  projectionMatrix =
    Matrix.CreatePerspectiveFieldOfView(fieldOfView, viewport.AspectRatio,
                                        NearPlane, FarPlane);
}
```

The *deltaY* and *deltaZ* variables contain the displacement from the default camera position, which is a little behind and up from the game object. This displacement is decided from input in the *UpdateCameraInput* method. The camera can be moved closer or farther away from the game object using the scroll wheel on the mouse, and it can be moved up and down by holding the Alt key and pressing the arrow keys up and down.

FreeThirdPersonCamera (Figure 14.26) differs from *ThirdPersonCamera* in that it does not follow the rotation of the observed game object, but determines this based on mouse input. This type of camera is used in popular MMORPG games such as Everquest and World of Warcraft. Holding the right mouse button while moving the mouse in any direction causes the camera to orbit the game object, but will still follow it around.

The virtual method *UpdateCameraInput* in *ThirdPersonCamera* is responsible for querying input from the keyboard and mouse. This method is overridden in *FreeThirdPersonCamera* to allow a different input scheme. The calculations of the view and projection matrices remain the same in both classes however.

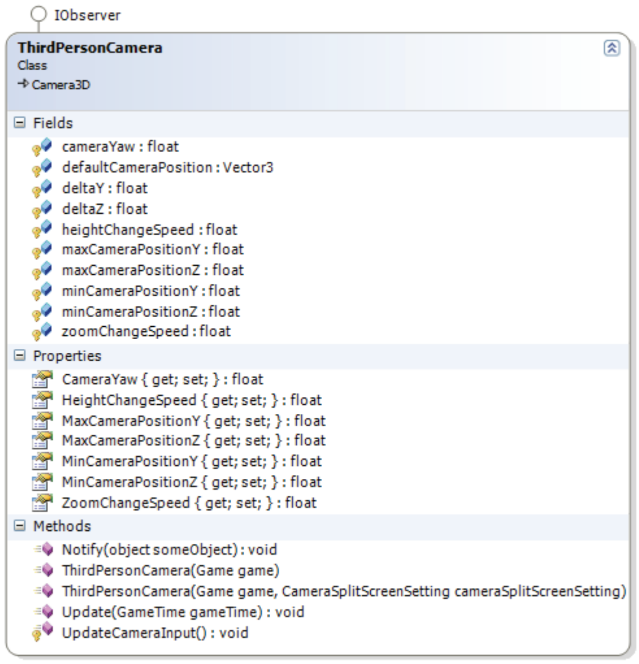

IObserver
ThirdPersonCamera
Class
Camera3D
Fields
cameraYaw : float
defaultCameraPosition : Vector3
deltaY : float
deltaZ : float
heightChangeSpeed : float
maxCameraPositionY : float
maxCameraPositionZ : float
minCameraPositionY : float
minCameraPositionZ : float
zoomChangeSpeed : float
Properties
CameraYaw { get; set; } : float
HeightChangeSpeed { get; set; } : float
MaxCameraPositionY { get; set; } : float
MaxCameraPositionZ { get; set; } : float
MinCameraPositionY { get; set; } : float
MinCameraPositionZ { get; set; } : float
ZoomChangeSpeed { get; set; } : float
Methods
Notify(object someObject) : void
ThirdPersonCamera(Game game)
ThirdPersonCamera(Game game, CameraSplitScreenSetting cameraSplitScreenSetting)
Update(GameTime gameTime) : void
UpdateCameraInput() : void

Figure 14.25: The *ThirdPersonCamera* class.

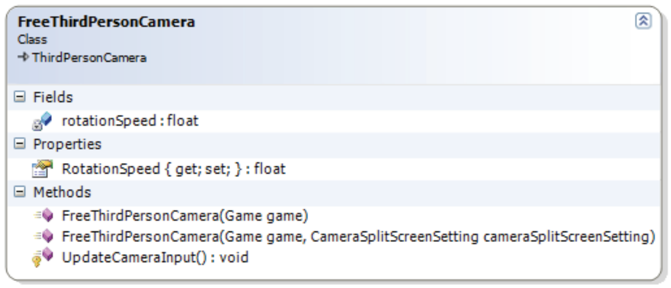

FreeThirdPersonCamera
Class
ThirdPersonCamera
Fields
rotationSpeed : float
Properties
RotationSpeed { get; set; } : float
Methods
FreeThirdPersonCamera(Game game)
FreeThirdPersonCamera(Game game, CameraSplitScreenSetting cameraSplitScreenSetting)
UpdateCameraInput() : void

Figure 14.26: The *FreeThirdPersonCamera* class.

14.8.6 FirstPersonCamera.cs

FirstPersonCamera (Figure 14.27) implements a 3D camera that views the world through the eye of the observed game object. It differs from *ThirdPersonCamera* only in that it positions itself in the same position (plus a little up) as the game object rather than behind it. The camera cannot be controlled by input.

Like *ThirdPersonCamera*, it implements the *IObserver* interface so it can observe a game object. The *Notify* method is shown below.

A *head offset* can be specified, which indicates how far up (from the center of the game object) the camera should be positioned.

```
public void Notify(object someObject)
{
  if (!(someObject is GameObject3D))
    return;

  GameObject3D subject = someObject as GameObject3D;

  // Head position almost at top of the object
  headOffset = new Vector3(0, subject.BoundingVolume.Height*0.90f, 0);

  // Create rotation matrix based on subject's current rotation.
  // Not interested in the rotation about the z-axis, since this is not
  // natural for a first person view.
  Matrix rotationMatrix = Matrix.CreateRotationY(subject.Rotation.Y);

  // Transform the head offset by the rotation matrix
  Vector3 transformedHeadOffset = Vector3.Transform(headOffset, rotationMatrix);

  // Set camera Position to the subject's "head".
  Position = subject.Position + transformedHeadOffset;

  // Transform the camera direction by the rotation matrix.
  Vector3 transformedReference =
    Vector3.Transform(cameraReference, rotationMatrix);

  // Calculate the position the camera is looking at.
  Vector3 cameraLookat = transformedReference + Position;

  // Create view matrix.
  viewMatrix = Matrix.CreateLookAt(Position, cameraLookat, Vector3.Up);

  // Create projection matrix
  projectionMatrix =
    Matrix.CreatePerspectiveFieldOfView(fieldOfView, viewport.AspectRatio,
                                        nearPlane,
                                        farPlane);
}
```

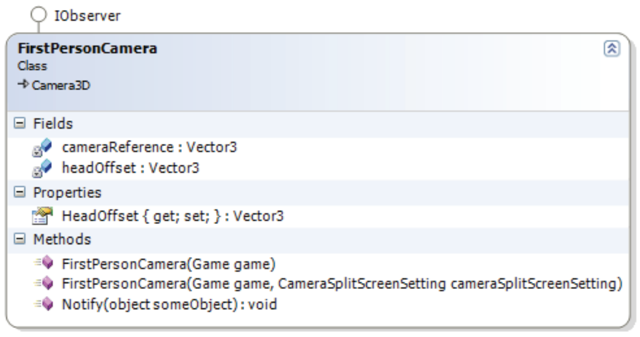

IObserver
FirstPersonCamera
Class
Camera3D
Fields
cameraReference : Vector3
headOffset : Vector3
Properties
HeadOffset { get; set; } : Vector3
Methods
FirstPersonCamera(Game game)
FirstPersonCamera(Game game, CameraSplitScreenSetting cameraSplitScreenSetting)
Notify(object someObject) : void

Figure 14.27: The *FirstPersonCamera* class.

A first-person camera is mostly used in first-person shooter games like Halo, Half-Life, Quake, Doom, and so on.

14.8.7 CameraManager.cs

The *CameraManager* (Figure 14.28) manages a collection of cameras. It is useful for games that allow multiple views of the game world, for example in a game where the player character can be controlled through either a first-person or third-person view. One of the cameras managed by the *CameraManager* can be set as the *active* camera. The active camera can be set through the

ActiveCamera property, or by calling *ActivateNextCamera*, in which the active camera is set to the next camera in the managed collection.

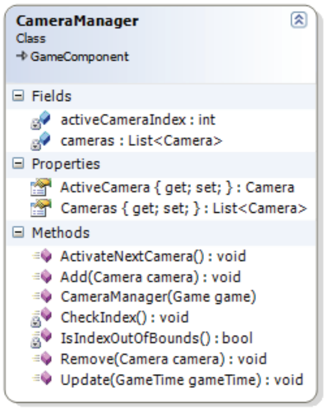
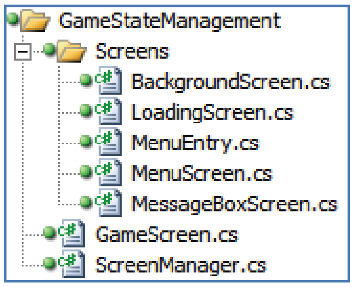

CameraManager
Class
GameComponent
Fields
activeCameraIndex : int
cameras : List<Camera>
Properties
ActiveCamera { get; set; } : Camera
Cameras { get; set; } : List<Camera>
Methods
ActivateNextCamera() : void
Add(Camera camera) : void
CameraManager(Game game)
CheckIndex() : void
IsIndexOutOfBounds() : bool
Remove(Camera camera) : void
Update(GameTime gameTime) : void

Figure 14.28: The *CameraManager* class.

14.9 XQUEST.GameStateManagement



This namespace contains a modified version of the game state management sample from the Creators Club web site[27]. As such, we do not take credit for any of the concepts this namespace brings with it. However, we did modify the sample to fit in with the rest of XQUEST. Among other things, we replaced the input code in the sample with the *XQUEST.Input.InputManager* game component. We decided to include this namespace after discovering that many of the student groups used the sample in their games.

The game state management system is based on the concept of *game screens*, for which it provides an abstract base class *GameScreen*. Each game screen represents a view of the game in a certain state, for example the main menu screen, the inventory screen, the map screen, or the character screen. *ScreenManager* is the class that manages the game screens. It allows multiple screens to stack on top of each other, for features such as pop-up screens giving the player a message.

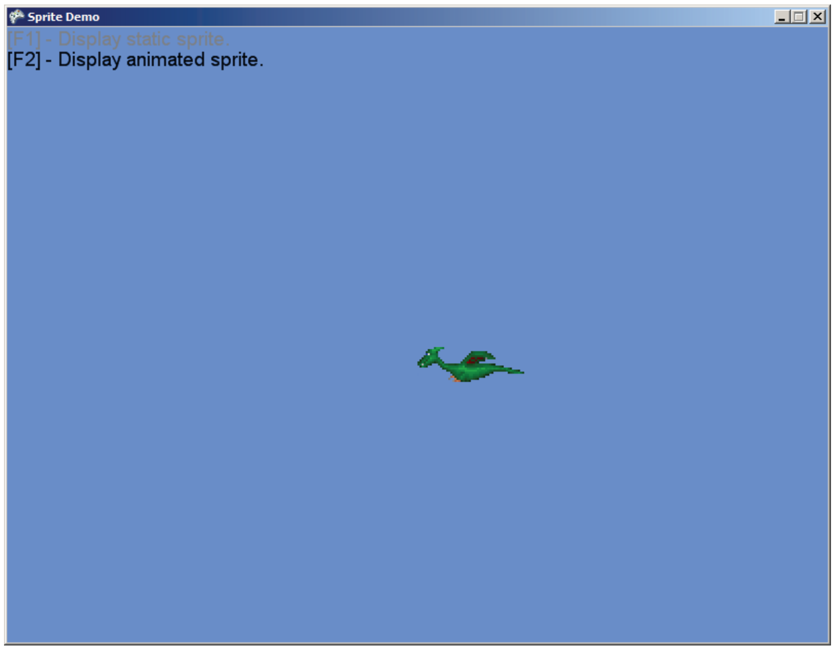
We have included some generic game screen classes. These are *MenuScreen*; representing a screen where the user can navigate and selected between several menu items, *LoadingScreen*; a screen that can be used when the game needs to load content that takes a while to load, *BackgroundScreen*; as screen showing a background image, and *MessageBoxScreen*; a pop-up screen showing a message to the player.

15. Demos

In order to provide better documentation for XQUEST, we have developed a number of demos that shows practical use of XQUEST. These demos are small programs that highlight the usage of the functionality of XQUEST. The source code for the demos can be found in the attachment, alongside a video showing the demos in sequence.

15.1 Sprite Demo

The sprite demo shows how to use both static and animated sprites in XQUEST. A dragon is displayed in the middle of the screen using a) a texture containing a single frame, and b) a sprite sheet texture containing five animation frames. The user switches between showing a static sprite and an animated sprite with the F1 and F2 keys, respectively (see Figure 15.1).



[F1] - Display static sprite.
[F2] - Display animated sprite.

Figure 15.1: The sprite demo.

15.2 Input Demo

The input demo shows how to read and interpret input from the keyboard, mouse, and Xbox 360 game pads.

In the demo we see an animated sprite that can be moved around on the screen using the arrow keys, mouse clicks or the game pad. Also on the top part of the game window the buttons and mouse clicks that we recently pressed are displayed in text. At the lower part of the game window the mouse cursor's coordinates are displayed (see Figure 15.2).

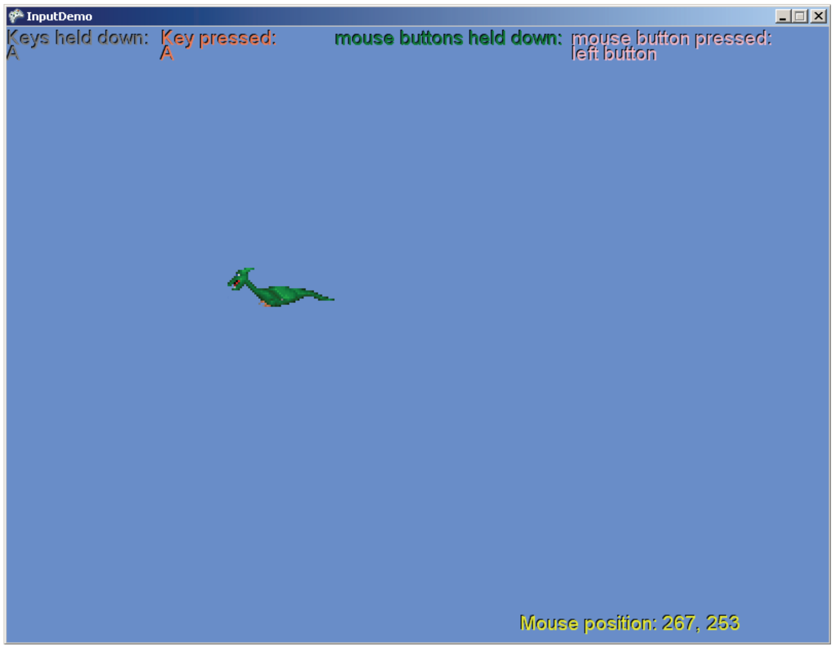
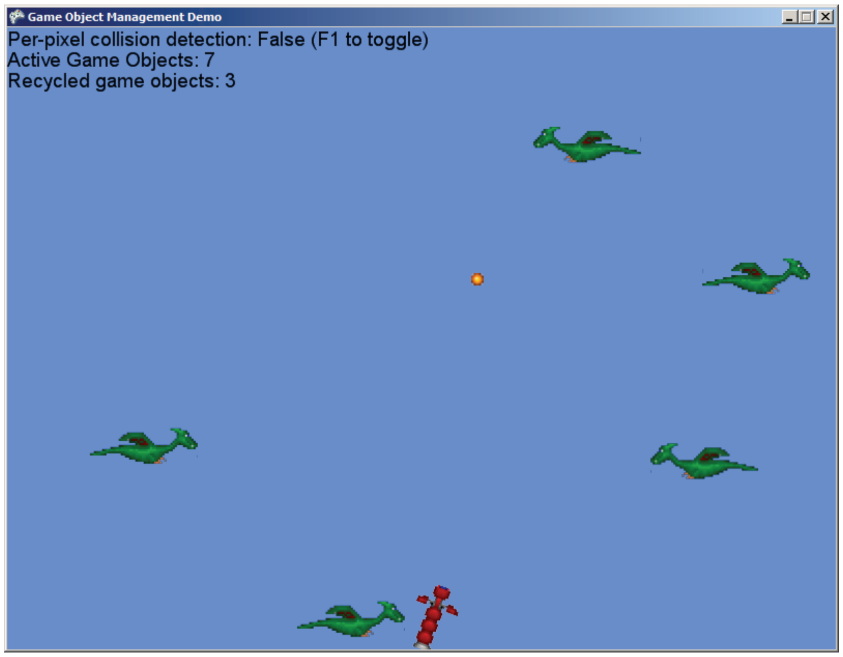


Figure 15.2: The input demo.

15.3 2D Game Object Management Demo

This demo shows how to use *GameObjectManager* to manage game objects in a 2D world using sprites. A cannon located at the center bottom of the screen is moved using the arrow keys, and the space keys sends bullets out of it. The demo demonstrates collision detection between the bullets and the dragons, and the recycling feature of GameObjectManager. Whenever a bullet hits a dragon or goes out of the screen, the bullet is recycled for later use. Bullets like these are the perfect example of the need of a recycling system in order to minimize memory allocations. The text in the upper-left corner shows whether per-pixel collision detection is enabled (toggled with the F1 key), the number of active game objects (those that are displayed on the screen), and the number of recycled game objects (those that are ready to be reused) (see Figure 15.3).



Per-pixel collision detection: False (F1 to toggle)
Active Game Objects: 7
Recycled game objects: 3

Figure 15.3: The game objects management demo.

15.4 3D Game Objects/Camera Demo

This demo shows how to use *GameObjectManager* to manage game objects in a 3D world using models, and how to use the different supported camera types in XQUEST 2.0 (see Figure 15.4).

In the demo, a duck can be moved around using the keyboard. When the duck collides with one of the boxes, the box disappears. This shows how to use the game object system to detect and handle collisions.

When the F9 key is pressed, we cycle through four different camera modes, which illustrate how we can use the camera system to view a scene in different ways.

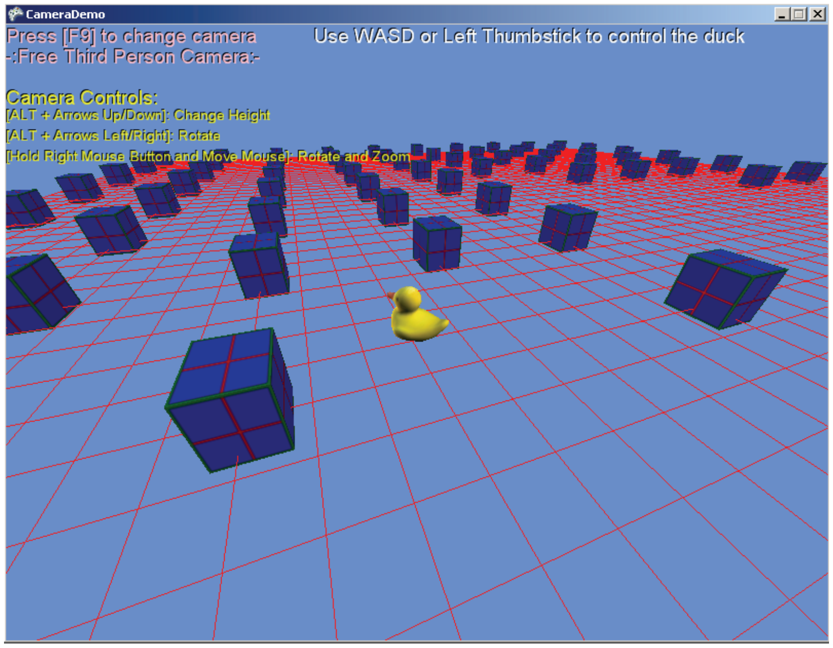

CameraDemo
Press [F9] to change camera
-:Free Third Person Camera:-
Use WASD or Left Thumbstick to control the duck
Camera Controls:
[ALT + Arrows Up/Down]: Change Height
[ALT + Arrows Left/Right]: Rotate
[Hold Right Mouse Button and Move Mouse]: Rotate and Zoom

Figure 15.4: The 3D Game Objects/Camera Demo.

15.5 Audio Demo

The audio demo shows how to incorporate sound effects and music into a game using the *AudioManager* XQUEST game component (see Figure 15.5).

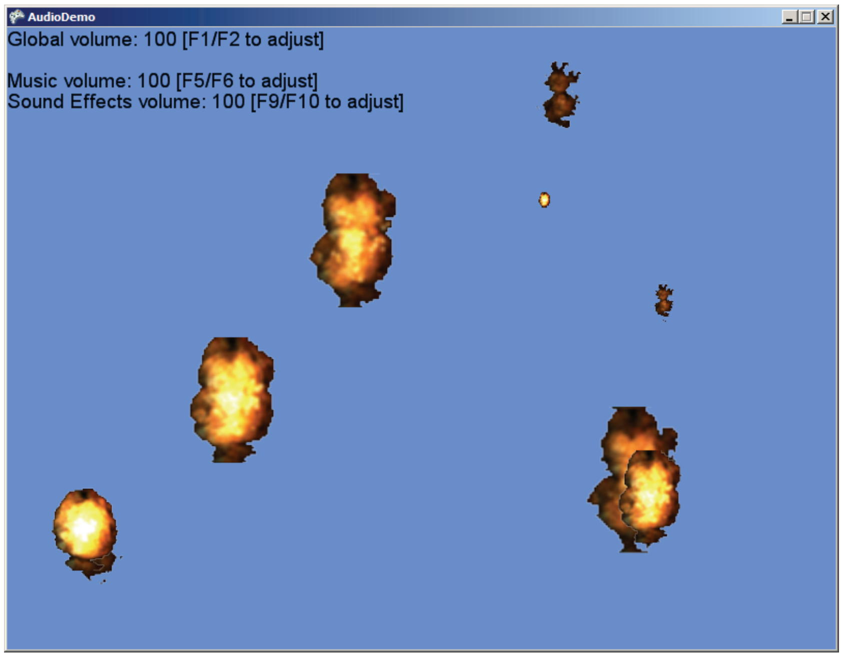

AudioDemo
Global volume: 100 [F1/F2 to adjust]
Music volume: 100 [F5/F6 to adjust]
Sound Effects volume: 100 [F9/F10 to adjust]

Figure 15.5: The audio demo.

The demo features two wave files; one is categorized in the “Music” category and its playback is looped throughout the demo. The other one is categorized in the “Sound Effects” category and will playback whenever a random explosion occurs on the screen.

The demo demonstrates how an XACT audio project is loaded through the *AudioManager*, how to specify custom categories, and how to adjust volume levels of individual categories. It also shows how to set and adjust a global volume level across all categories.

15.6 Text Rendering Demo

This demo shows how to use the *TextHelper* class to display text using sprite fonts (see Figure 15.6).

In the demo we see text displayed in different ways and in different positions. Some have shadows and some have been centered on a screen position. We can also switch between three different font types.

The demo also shows how we can create different effects with the text, like having it follow an object or making it shake.

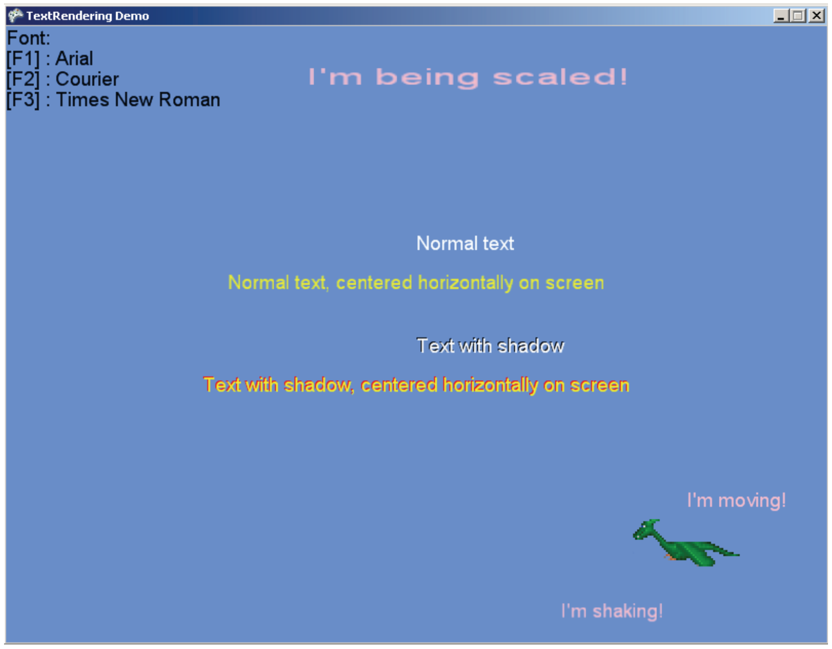

TextRendering Demo
Font:
[F1] : Arial
[F2] : Courier
[F3] : Times New Roman
I'm being scaled!
Normal text
Normal text, centered horizontally on screen
Text with shadow
Text with shadow, centered horizontally on screen
I'm moving!
I'm shaking!

Figure 15.6: The text rendering demo.

15.7 Pointing Demo

In the demo we see three ducks facing us. When we click one of them, it will start rotating and in the top left corner of the game window we will see the count of clicks increase. The demo shows how we can use our *ClickableManager* and *GameObjectManager* together to make objects clickable with the mouse (see Figure 15.7).

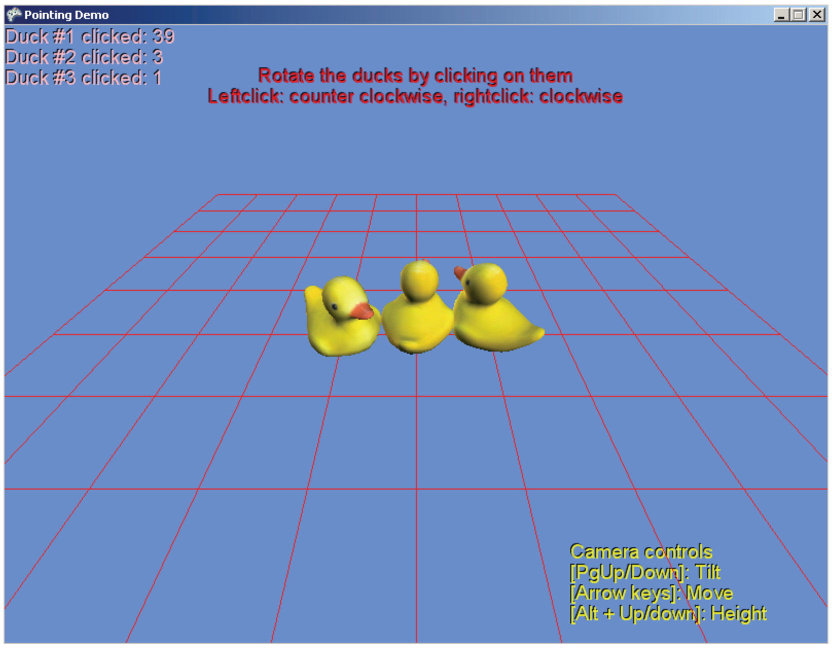


Figure 15.7: The pointing demo.

16. Summary and Discussion

Going beyond the 2D-only focus of XQUEST 1.0, we improved and extended XQUEST as part of this master thesis. We based the development of XQUEST 2.0 partly on our desire to support 3D, partly on our own subjective opinions of useful features, and partly on the feedback from the students in the software architecture course and the survey.

We substantially rewrote the game object management system. The main motivation was to support 3D objects and collisions in 3D space. We also recognized that students who used XQUEST 1.0 in their projects did not use this component to its fullest potential. This resulted in the introduction of an inheritance hierarchy of game object classes. At the root level, the *GameObject* class contains functionality common to all game objects. On the second level, *GameObject2D* and *GameObject3D* contain functionality common to 2D game objects and 3D game objects respectively. At the bottom level, a concrete 2D game object implementation, *SpriteGameObject*, and a concrete 3D game object implementation, *ModelGameObject*, exist to provide users with a graphical representation of the game object in addition the logic. We also enabled support for per-pixel collision detection in *SpriteGameObject* (as requested by many of the students), and support for picking in *ModelGameObject* (allowing objects to be selected by the mouse). We also incorporated into the game object management system a recycling scheme; allowing more efficient game object handling with regards to memory consumption, flexible collision detection; enabling the users to define bounding volumes and collision detection method on a per-game object basis, and game object monitoring; allowing the game objects to be observed by external entities, such as cameras, other game objects, or other entities interested in tracking a game object through the game world.

Further, we developed a flexible and extensible camera system. A 2D camera useful for 2D scrollable games and several 3D cameras were implemented. *FreeCamera* can be controlled by the keyboard and allows free movement in every direction in a 3D world. *ThirdPersonCamera* allows the observing of a 3D game object. This camera will view the game object from behind, in a third person style. *FreeThirdPersonCamera* extends *ThirdPersonCamera* by allowing free look in any direction. Lastly, *FirstPersonCamera* will view the scene through the eyes of the game object in a first person style. A *CameraManager* game component is included, which lets users manage multiple cameras more easily. In addition, each camera can be assigned a portion of the screen to use for viewing, something that is useful in split screen games.

We incorporated the game state management sample from the Creators Club website into XQUEST 2.0. We discovered that many of the students found this sample useful, and used it in their own projects. We adapted the sample to fit in with XQUEST. The sample came with an input handling mechanism that we replaced with the already existing input handling in XQUEST.

The sprite animation namespace only received minor changes in XQUEST 2.0. We added a struct *SpriteSheetInfo* to separate the sprite sheet data and the frames and animation data. The *SpriteSheetInfo* has a method *GetFrames*, which is useful for extracting all the frames from a sprite sheet and to pass to the constructor of *Animation*.

The input handling was improved. An improvement in the XNA 2.0 API enabled us to cut down the number of methods for checking game pad input in the *InputManager* class substantially. We also separated the input handling for the keyboard, game pad, and mouse into separate classes named *KeyboardInput*, *GamePadInput*, and *MouseInput*, respectively. This made the interface a lot cleaner, and makes access to input querying more logical.

We improved the audio support by adding additional features such as support for user-defined audio categories, separate volume levels for each category, and a global volume level across all categories. We also improved the efficiency and lowered the memory consumption of the *AudioManager* class by keeping track of active cues, stopping, pausing, resuming, and playing them at will.

Lastly, we made several other improvements and additions.

Through working with XQUEST and extending it to include 3D we have come to better understand the effort required by 3D game programming. Not only did we have to learn more math to be able to program and understand the functionality of XQUEST, but we also had to find 3D content to work with.

In Chapter 5, we presented our prestudy regarding math and 3D concepts. Most of the topics were familiar to us from earlier mathematical courses in linear algebra, such as vectors, matrices and transformations. What is nice about XNA is that we are able to use its 3D features before fully understanding what exactly is going on behind the curtains. It can be compared to baking bread. We follow the recipe and include baking powder because we know this makes the bread rise. We may also know that this is due to carbon dioxide being released, but the chemical formula for why the bread will rise is not necessary for us to know of in order to bake good tasting bread.

Another discovery was that we did not need to learn any new ways of programming. Using the 3D features of XNA was just straightforward C# and not that different from what we were used to from working with the 2D side of XNA. However, in our prestudy we also looked into shader programming which indeed was new to us, but since XNA provides a basic shader effect already this is not an absolutely needed prerequisite to start creating 3D games with XNA.

On the other hand, having 3D content to work with is more or less required. When we were working with 2D we only had to search for images which there are plenty of on the web. Animations were a little tougher to find, but since hobbyists have been creating 2D games for many years now, there were plenty of good and free resources for these too. Also, we always had the option to create the 2D graphics ourselves and even though our artistic skills may not be so good, we can usually create something recognizable. However, that option disappears when working with 3D games. In Section 5.2, we mention some 3D modeling tools and say that they easily manage all the triangles that make up a 3D model. This is partly true in that they do make it a lot easier to create 3D models, but for a student new to them they are very hard to use and time-consuming to learn. As such, finding content to 3D games is not an easy task due to several reasons. The people who are skilled at creating models, spend much time making them, and naturally want to get compensated. Often they only want their models to be used in projects that they are part of themselves as well. Like with 2D graphics, there exist resources on the web with free 3D models too, but they are fewer and harder to come by.

The difficulties with finding 3D content for games could be remedied by cooperation with 3D modeling courses at other schools, since as of today there are no courses at NTNU that teach 3D modeling aimed at the entertainment industry. However, a game development education is taking shape at NTNU [28], and it would be natural to introduce courses that teach 3D modeling at NTNU in the future.

Part V
Conclusion

17. Conclusions

We had several goals with this master thesis. We wanted to evaluate the introduction of XNA game development projects in a software architecture course at the Norwegian University of Science and Technology (NTNU). Also, we wanted to evaluate and improve our existing XNA game library that has been used in these projects. Lastly, we wanted to investigate the effort required to learn the necessary 3D concepts and background math necessary to produce 3D content and create 3D games in XNA.

We had four main research questions. The research conducted in this master thesis leads to answers of the following questions:

> **RQ1: What are the pros and cons of using XNA in the student projects as opposed to the traditional robot simulation?**

We compared the students who chose XNA game projects and those who chose the robot project through conducting a survey, which was presented in Part III. By comparing the two sides with regards to the different phases of the project, execution of the project, in addition to results and other feedback, we were able to determine some important differences.

We found that game programming might steal some focus with respect to the learning goals of the software course, but that this can be sorted out with time by creating more learning material and resources aimed specific towards the XNA project.

Further we found that most of the students who chose the XNA project were happy with their project. This was not the case with the robot simulation project, where as much as one third of the students wished they had chosen the XNA project instead. Coupled with the fact that most of the robot groups struggled a lot, and were frustrated with the robot simulator, we believe this indicates a higher motivation among the XNA students.

We also wanted to identify whether or not the XNA students would require more assistance, since most of the students did not have much experience with C# and XNA. As it turned out, the students who chose XNA spent less time learning the prerequisites in the start than the robot students.

Lastly, we looked into how the COTS influenced the different architectural sides of the project, and found that the choice of COTS did not matter much except for when it came to the design of the architecture. It was clear that the robot simulator put some constraints on the architecture, while the XNA environment was more flexible.

> **RQ2: How useful is our XNA game library for the students?**

Through the survey, we found that roughly 40% of the students used XQUEST in their XNA projects. Generally, the students found it useful. Of the components they especially found useful was the sprite and sprite animation components and the game object system. More than half of the students caught themselves spending too much time on the gameplay rather on the architecture of the game. At the same time, they did not think that XQUEST helped them focus more on architectural matters, but that it instead helped them save time by providing functionality that they otherwise would have had to implement themselves. This indicates that this saved time was rather spent on other non-architectural matters for which XQUEST could not provide any help. This indication is further backed up by the fact that the students agree that there were components and functionality missing from XQUEST, which most of them could benefit from. Further, we discovered that 20% of the students were not happy with the documentation and found it lacking. Also, about 40% of the students made their own modifications to XQUEST in order to adapt certain features into their own code. The most requested new feature was per-pixel collision detection. This was added in XQUEST 2.0.

We employed the SUS to measure the usability of XQUEST. XQUEST achieved a SUS score of 60.53 out of 100, which indicates a little above average usability. However, we lack a comparison with other similar systems, and acknowledge that the SUS may not give a good measure of usability on frameworks such as XQUEST where the entry level is a bit higher than typical end-user applications.

RQ3: How can our game library be extended and improved?

Based partly on our own subjective assessment on the functionality that should go into such a game library, we also based the extension of XQUEST partly on the results from the survey. The game object system was rewritten from scratch to support both 2D and 3D game objects. Two concrete game object classes were implemented; one for 2D based on sprites, and one for 3D based on models. A recycling scheme was implemented to allow more efficient handling of game objects with regards to memory consumption. A new highly flexible collision detection system was incorporated into the game object system. Collision checking and collision response can now be customized for every individual game object. Further, an extensible camera system was added, sporting both a 2D camera for scrolling 2D games and several 3D cameras for viewing the game world from different perspectives. The 3D cameras *FirstPersonCamera* and *ThirdPersonCamera* can be set up to observe a game object from a first person and third person perspective, respectively.

In addition to these extensions, several improvements were made to existing components to enhance modifiability and reusability, and to provide additional functionality.

Based on the results from the survey and evaluation of XQUEST 1.0, we decided to include a few additional features in XQUEST 2.0. The "Game State Management Sample" from the Creators Club website was modified and adapted to fit into XQUEST. The decision to include this in the library was made when we discovered that many of the student groups used this sample in their games. Additionally, we implemented per-pixel collision detection support for 2D game objects in the game object system, a feature requested by many of the student groups. Lastly, we improved the documentation by creating several demos showing off most of the functionality of XQUEST 2.0. This was motivated by the survey results, where the students found the documentation lacking, and from their feedback requesting more information about how to use the functionality of XQUEST in their own games.

RQ4: What is the difficulty level/what kind of effort is required for learning the 3D concepts and production of 3D content needed to be able to create 3D games with XNA?

Through our own work with XQUEST we have been able to assess the difficulties of 3D game creation with XNA, regarding the requirements of programming and content creation.

We found that we already had all of the programming skills required, meaning we did not need to learn any new languages or ways of programming when we went from 2D to 3D. However, with 3D came some new methods and in order to understand how and why to use them we first had to refresh on some linear algebra including vectors, matrices and transformations. The math itself was nothing new and should not be for any student taking 4th grade Computer Science, but the connection between the math and 3D programming was new, however, fairly easy to understand. With XNA we can start creating a 3D game real fast with just some basic knowledge of 3D programming, and later we can expand on that knowledge if we so choose and start using the more advanced functionality of XNA.

Finding 3D content to use within the game is a tougher issue. We cannot expect the students of the software architecture course to learn 3D modelling fast enough to be able to create their own 3D graphics. The 3D content has to be acquired elsewhere, and that means pointing the students to free graphic resources on the web. Finding exactly what we need of 3D content on the web can be pretty hard and sometimes impossible. To remedy these difficulties we propose cooperation with 3D modelling courses, preferably at NTNU, but other schools might be a possibility as well.

18. Further Work

Our research has focuses around empirical evaluations and development of support software. While we feel that we have achieved results that will be helpful for the software architecture course, further research into software architecture specific areas is desirable.

18.1 Investigation of Software Architecture in Game Development

An investigation into how video games can be architected using knowledge from software architecture can prove valuable. Such a study would look into the differences between traditional software development and game development, as well as identifying different architectural and design patterns that are useful for game development. Also, portraying the challenges of designing and implementing game architectures could be proved useful for determining the scope of such an endeavor. Very little has been published on the subject of software architecture in game development, although some attempts have been made [29-31]. However, these attempts fail to deliver a general high level presentation of software architecture topics, and tend to focus more on the design and implementation of software modules that are common in games.

18.2 How to Reflect the Quality Attributes in a Game Architecture

Quality attributes are the driving force behind every big decision in the development process, and have to be considered at all times. In our depth study, we identified four quality attributes that were most relevant for game development: modifiability, testability, availability, and usability. However, what we did not investigate was how to actually reflect these attributes in a game architecture. As this fell outside the scope of this master thesis, we recommend a study of this in future projects. The study should look at structures and patterns that underline certain quality attributes, and how to use these elements in a game architecture.

Part VI
Appendices

Appendix A. List of Figures

Appendix B. List of Tables

References

[1] J.-E. Strøm, T. B. Kvamme, and V. Heggdal, "Exploration of the XNA Game Programming Framework and its Suitability for Teaching Software Architecture," Trondheim: Norwegian University of Science and Technology, 2007.

[2] A. Djupdal and L. Natvig, "Age of computers II - an improved system for game based teaching," in *Norsk Informatikk Konferanse 2004 (NIK)* Stavanger, Norway, 2004.

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2 ed.: Addison-Wesley, 2003.

[4] A. I. Wang and T. Stålhane, "Using Post Mortem Analysis to Evaluate Software Architecture Student Projects," in *Conference on Software Engineering and Training 2005*, 2005.

[5] "it's learning Web Site", http://www.itslearning.no. Retrieved April 2, 2008.

[6] V. R. Basili, "The Experimental Paradigm in Software Engineering," in *Dagstuhl Workshop*. vol. Experimental Software Engineering Issues: Critical Assessment and Future Directives, H. D. Rombach, V. R. Basili, and R. W. Selby, Eds. Dagstuhl Castle, Germany: Springer-Verlag, 1992, pp. 3-12.

[7] M. V. Zelkowitz and D. R. Wallace, "Experimental Models for Validating Technology," *IEEE Computer,* vol. 31, pp. 23-31, May 1998.

[8] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering An Introduction*: Kluwer Academic Publishers, 2000.

[9] N. Landry, "Microsoft XNA: Ready for Prime Time?," in *CoDe Magazine*. vol. Sept/Oct, 2007.

[10] Microsoft; "Zune.net", http://www.zune.net/. Retrieved April 24, 2008.

[11] Microsoft; "XNA Creators Club Online", http://creators.xna.com/. Retrieved May 21, 2008.

[12] F. Luna, *Introduction to 3d Game Programming with Direct X 9.0c*. Plano: Wordware Publishing, Inc, 2006.

[13] T. Miller, "Managed DirectX 9 Graphics and Game Programming," Sams Publishing, 2004.

[14] Microsoft; "XACT Orientation", http://msdn.microsoft.com/en-us/library/bb172306(VS.85).aspx. Retrieved May 16, 2008.

[15] Microsoft; "Application Model", http://msdn.microsoft.com/en-us/library/bb203871.aspx. Retrieved October 6, 2007.

[16] Microsoft; "Overview of the Application Model", http://msdn2.microsoft.com/en-us/library/bb203873.aspx. Retrieved October 6, 2007.

[17] C. Carter, "Microsoft XNA Unleashed: Graphics and Game Programming for Xbox 360 and Windows," in *Unleashed*: Sams, 2007.

[18] Microsoft; "SpriteBatch.Draw Method", http://msdn2.microsoft.com/en-us/library/microsoft.xna.framework.graphics.spritebatch.draw.aspx. Retrieved November 6, 2007.

[19] Microsoft; "Shader Stages (Direct3D 10)", http://msdn.microsoft.com/en-us/library/bb205146(VS.85).aspx. Retrieved May 26, 2008.

[20] Microsoft, "Shader Series Primer: Fundamentals of the Programmable Pipeline in XNA Game Studio Express," 2007. http://creators.xna.com/downloads/?id=128

[21] R. Grootjans; "Riemers XNA Tutorial > Loading a Model", http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series2/Loading_a_Model.php. Retrieved May 26, 2008.

[22] C.-w. Xu, "Why and How to Teach Game Programming," in *Proceedings of the 2006 International Conference on Frontiers in Education: Computer Science Computer Engineering FECS 2006* Las Vegas, Nevada, USA: CSREA Press, 2006, pp. 215-220.

[23] R. Likert, *A technique for the measurement of attitudes*. New York: [s.n.], 1932.

[24] J. Brooke, "SUS - A quick and dirty usability scale," in *Usability Evaluation in Industry* London: Taylor and Francis, pp. 189-194.

[25] Wikipedia; "IntelliSense", http://en.wikipedia.org/w/index.php?title=IntelliSense&oldid=208720089. Retrieved May 7, 2008.

[26] Microsoft; "Overview of the Application Model", http://msdn2.microsoft.com/en-us/library/bb203873.aspx. Retrieved October 6, 2007.

[27] Microsoft; "XNA Creators Club Online - game state management", http://creators.xna.com/en-us/samples/gamestatemanagement. Retrieved May 25, 2008.

[28] "Datateknikk at IDI NTNU", http://www.idi.ntnu.no/education/datateknikk_studieprogram.php?spraak=norsk&menu=teknologi. Retrieved December 15, 2007.

[29] A. Rollings and D. Morris, *Game Architecture and Design*, 2 ed.: New Riders Publishing, 2003.

[30] D. Eberly, *3d Game Engine Architecture*. Amsterdam: Morgan Kaufman Publishers, 2005.

[31] R. Rucker, *Software Engineering and Computer Games*. Boston: Addison-Wesley, 2003.