MINUETO, AN UNDERGRADUATE TEACHING DEVELOPMENT FRAMEWORK

by
Alexandre Denault

School of Computer Science
McGill University, Montreal

April 2005

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Master of Science

Copyright © Alexandre Denault, 2005. All right reserved.

Abstract

In the last decade, the video game industry has shown unparalleled growth, both in revenue and in development complexity. Modern video games are built using powerful game development tools. However, given their complexity and scope, these tools are ill-suited for the undergraduate academic environment.

Minueto is a platform independent game development framework specifically designed for undergraduate students. Originally developed to help students of the course COMP-361 for their game programming project, Minueto offers a simple object-oriented API that is ideal for any 2D game development. Minueto's simple design allows students to quickly learn the framework and avoid the overhead typically associated with graphic programming. Extra effort was spent writing multiple types of documentation, each of them corresponding to different learning techniques. Minueto was first implemented in Java and used by the students of the Winter 2005 term. There was an impressive increase in the overall quality of the 2005 projects if compared to the 2004 projects. A significant decrease in the number of graphic programming complaints from the students was also noticed. This indicates that Minueto successfully reduced the graphic programming burden from the students, and allowed them to focus on the content of the game.

Résumé

Dans la dernière décennie, l'industrie du jeu vidéo a subit une croissance inégale, autant pour ses revenus que pour la complexité de ses jeux. Les jeux vidéo modernes sont bâtis avec des puissant outils de développement. Cependant, étant donné leurs complexités et leurs portées , ces outils sont mal adaptés pour les étudiants du premier cycle universitaire.

Minueto est un cadre de développement de jeu pour divers plateforme, spécifiquement conçu pour les étudiants du premier cycle universitaire. Quoique Minueto a été originalement conçu pour aider les étudiants du cour COMP-361 dans leur projet de programmation, la simple architecture objet-orientée de Minueto est idéale pour tout projet de développement de jeu 2D. La structure simple de Minueto permet aux étudiants de rapidement apprendre à utiliser le cadre et réduire le fardeau de la programmation d'une interface graphique. Beaucoup d'effort a été investi dans l'écriture de plusieurs types de documentation, chaque type correspondant à une technique d'apprentissage différente. Minueto a été implémenté en Java et utilisé pour la première fois par les étudiants de la session d'hiver 2005. Quand l'on compare avec les projets de l'année précédente, une augmentation impressionnante dans la qualité des projets de la session 2005 a été remarquée. Une diminution signifiante des plaintes reliées au développement d'une interface graphique a également été constatée. Ceci indique que Minueto à réussi à réduire le fardeau des étudiants pour la programmation de leur interface graphique, tout en leur permettant de se concentrer sur le contenu du jeu.

Acknowledgments

I would sincerely like to thank my supervisor, Prof. Jörg Kienzle, for his guidance, assistance and support. I am extremely graceful for the freedom and flexibility I had in exploring the different possibilities for Minueto. I also truly appreciate his patience, given the strong opinions I sometimes have.

I would also like to thank Marc Lanctot for being my first official tester. His comments were invaluable in refining and tuning Minueto. As well, I would like to thank the Winter 2005 COMP-361 class for being my first official users. I was truly touched by the high number of students that chose to use Minueto. More specifically, I would like to thank Ziad Abou-Saab, Benjamen Azan, Jeremy Claude, Daniel Crittenden, Michel A. Hawker, Adam Hooper, Tolga Kart, Jean-Sébastien Légaré, Vicent Leung, Denis Lebel, David Ly-Gagnon, Seth Marinello, Frédéric Menu, Marc Ovidiu, Will Stevens and Razvan Taranu for providing me valuable feedback and helping me improve the framework.

I'd like to thank the Java Game Development Community, especially the talented programmers that donate their valuable time to answering questions on the Java Game forums. Their help was critical in understanding the nuance of Java Game development.

Finally I'd like to thank my parents, brother, family, and the rest of my friends. Their patience with me was inspiring, as I spent long hours working on Minueto. Their encouragement was a critical component of this project, and for this, I thank them.

Contents

Abstract ii

Résumé iii

Acknowledgments iv

Contents v

List of Figures xi

List of Tables xii

1 Introduction 1

2 Fundamentals of Game Programming 3

2.1 Game Programming Concepts 3

2.1.1 Frame Rate 3

2.1.2 Double Buffering 4

2.1.3 Game Loop 5

2.2 Game Design Concepts 7

2.2.1 Dimensionality of Games 7

2.2.2 Single vs. Multiplayer 8

2.2.3 Real Time vs. Turn Based 10

I Framework Rationale 11

3 Comp 361 : System Development Project 12
3.1 Course Description 12
3.2 Course Goals 13
3.3 Previous Framework 14
3.4 Student Projects and Feedback 15

4 Improving the Course Infrastructure 18
4.1 Defining the Goals of the Infrastructure 18
4.2 Current Game SDKs on the Market 20
4.2.1 DirectX 20
4.2.2 SDL : Simple Media Layer 21
4.2.3 OpenGL 21
4.2.4 Java 21
4.2.5 Summary of Evaluation 22

II Minueto 24

5 A Simple Object-Oriented Framework 25
5.1 Architecture 25
5.2 2D Graphics Engine 26
5.2.1 The Window 26
5.2.2 The Images 26
5.2.3 Abstraction Classes 28
5.2.4 Drawing 28
5.3 Keyboard / Mouse Listener 30
5.3.1 Traditional solutions 30
5.3.2 Queuing System 31
5.3.3 Events From Expansion Modules 33
5.4 Possible Expansion Modules 33

5.4.1 MinuetoSound 33
5.4.2 MinuetoNetworkMessaging 34
5.4.3 MinuetoNetworkSync 34
5.4.4 MinuetoXML 35
5.4.5 MinuetoTileMap 36
5.4.6 MinuetoUI 38

6 An implementation in Java 39

6.1 Why Java? 39
6.2 Performance Analysis 40
6.2.1 Benchmarks 40
6.2.2 Benchmark Results 41
6.2.3 Other Performance Analysis 43
6.3 Programming Java Graphics 45
6.3.1 Types of Images 45
6.3.2 Double Buffer Strategy 46
6.4 Documentation Strategy 47
6.4.1 Tutorials 47
6.4.2 Examples 48
6.4.3 Specifications 48
6.5 Current Limitation 49
6.5.1 Single Window Restriction 49
6.5.2 Apple JVM 50
6.5.3 Keyboard Input on Linux 51
6.5.4 Fullscreen Mode Under Linux 51
6.6 Programming Practices 52
6.6.1 Coding Standards 52
6.6.2 Automated Build System 52
6.6.3 Regression Testing 53
6.6.4 Source Control 53
6.6.5 New Releases 53

6.7 Additional Benchmarks . . . 54
6.7.1 Goals and Methodology . . . 54
6.7.2 Analysis . . . 55

III Case Study: Winter 2005 59

7 Student Evaluation 60
7.1 Student's Platform Choices . . . 60
7.2 Student's Improvements To Minueto . . . 61
7.2.1 Alpha Channels . . . 62
7.2.2 Swing Integration . . . 63
7.2.3 Active Queue . . . 63
7.2.4 Keyboard Handler . . . 64
7.3 Observation of Student Progression . . . 65

8 Learning from 2004 and 2005 69
8.1 General Conclusions . . . 69
8.2 Impact Analyst . . . 69
8.3 Course Improvements . . . 71
8.3.1 User Interface Design . . . 71
8.3.2 Usability Testing . . . 72
8.4 Minueto's Evolution . . . 73
8.4.1 Change Policies for the Core Component . . . 73
8.4.2 The Role of Expansion Modules . . . 74
8.4.3 The Next Minueto . . . 75
8.5 Looking to the Future . . . 75

Appendices

A Minueto SDK Readme 76
A.1 Minueto Readme . . . 76

A.1.1 System Requirement 77
A.1.2 Installation 77
A.1.3 Samples 78
A.1.4 Documentation 78
A.1.5 Compiling 78
A.1.6 Contact 79

B Minueto Code Samples 80

B.1 CircleDemo 80
B.2 CropDemo 80
B.3 FramerateDemo 80
B.4 HandlerDemo 81
B.5 HandlerDemo2 81
B.6 HandlerDemo3 81
B.7 Helloworld 81
B.8 ImageDemo 81
B.9 LineBenchmark 82
B.10 LineDemo 82
B.11 LoadingFileDemo 82
B.12 RectangleDemo 82
B.13 RotateDemo 82
B.14 RotateDemo2 82
B.15 RotateDemo3 83
B.16 ScaleDemo 83
B.17 ScaleFlipDemo 83
B.18 TextDemo 83
B.19 TextDemo2 83
B.20 TriangleRover 83
B.21 TriangleRover2 84

C Minueto Tutorials **85**
C.1 How do I run the samples? 85
C.2 How do I compile and run an application with Minueto? 85
C.3 How do I create a game Window? 86
C.4 How do I load and draw an Image? 86
C.5 How do I draw a line? 86
C.6 How do I draw a rectangle or a circle? 86
C.7 How do I write text on an image or the screen? 86
C.8 How do I get input from the keyboard? 87
C.9 How do I get input from the mouse? 87
C.10 How do I measure time using Minueto? 87
C.11 How do I scale/rotate an image? 87

D Glossary **88**

Bibliography **91**

List of Figures

2.1 Content of buffers in a double buffering strategy. 5
2.2 Example of a simple game loop. 6
2.3 Example of a dual threaded game loop. 6
2.4 Pong features one degree of player's motion, two degrees of world's motion and zero degrees of viewer's motion. Image Source: Wikipedia 9

3.1 BattleShip Framework provided to Winter 2003 COMP–361 students. 15
3.2 Naval Battle, as implemented by Team 01 using the provided GUI . . 16

5.1 UML diagram of the 2D graphic engine component (part 1) 27
5.2 UML diagram of the 2D graphic engine component (part 2) 29
5.3 UML diagram of the input component 32
5.4 Example of a game map stored as a XML file. 36
5.5 Mappy Win32, a tile map editor. 37

6.1 The first benchmark, BlackWizardGrass 41
6.2 The second benchmark, TownMap 42
6.3 Frame rate achieved while running BlackWizardGrass demo. 43
6.4 Frame rate achieved while running TownMap demo. 44
6.5 Test Implementation of Strategic Conquest 45

7.1 Strategic Conquest, team Purple Monkey Screwdrivers 65
7.2 Strategic Conquest, team Golden Balaika Mariachis 66
7.3 Strategic Conquest, team Muff Busters 67

List of Tables

3.1 Programming Languages and Software Development Kits used by students during the Winter 2004 term. 17

4.1 Student feedback on most difficult task during the Winter 2004 project 18
4.2 Evaluation summary of the various game development kits available on the market 22

6.1 Frame rate (in frames per second) achieved while running BlackWizardGrass demo and TownMap demo in windowed mode (continued on next table). 55
6.2 Frame rate (in frames per second) achieved while running BlackWizardGrass demo and TownMap demo in windowed mode. 56
6.3 Frame rate (in frames per second) achieved while running BlackWizardGrass demo and TownMap demo in fullscreen mode. 57

7.1 Programming Languages and Software Development Kits used by student during the Winter 2004 & 2005 term. 61

8.1 Student feedback on most difficult task during the Winter 2005 project 70

Chapter 1
Introduction

What do academia and video games have in common? Twenty years ago, the answer to this question would have been very little. Video games were simple and could be developed by a small team of programmers. Today, video games represent a multi-billion dollars industry. Consumer expectations are high and developers must reinvent themselves with each new game. The evolution of the video game industry has resulted in numerous innovation in the field of computer science, most notably in the fields of computer graphics and computer audio.

Electronic Arts, one of the largest game developers in North America, hires over 1,000 new employes each year, and it would like to fill 75% of these positions with university graduates [Pau04]. Ubisoft will double its staff at its Montreal studio, adding 1,000 jobs by 2010 [Moo05]. The video game industry is growing at a tremendous rate and is looking to universities as research partners and recruiting centers.

The University of Alberta Games Group, in collaboration with Bioware, is researching new intelligent path-finding techniques and scripting languages for computer role-playing games [oAGG05]. Carnegie Mellon's Entertainment Technology Center is developing, jointly with the Walt Disney Imagineering VR Studio, Panda 3D, an open source game and simulation engine [ETC05]. The Institute for Creative Technologies (part of the University of Southern California) is supplying cutting-edge artificial intelligence capabilities to the award winning army simulation game Full Spectrum Warrior [fCT05].

To contribute to the video game industry, a university needs specialized researchers and graduate students with game programming knowledge and experience. Thus the importance of introducing game programming concepts at the undergraduate level. This represents a non-trivial tasks, since most game programming tools are geared towards the industry. For undergraduate students to properly understand and experience the game development cycle, universities need the proper teaching tools.

This thesis defines Minueto, a game development framework targeted for the computer science undergraduate population. It allows students to rapidly develop non-trivial games by simplifying several game-programming concerns such as graphics, sound, networking and player input. Thus, professors can ask the students to implement a game-related project knowing that students can focus on the behavior of the game.

The thesis is split into three main parts. Part 1 (chapter 3-4) explains the rational behind this game development framework. Chapter 3 describes a course where such a framework is needed and analyses the tools previously used in that course. Chapter 4 outlines the framework's goal and describes some game development tools available on the market. A review of common game programming and game design concepts can also be found in the next chapter (chapter 2).

Part 2 (chapter 5-6) introduces Minueto's architecture. Chapter 5 describes this platform independant framework, defining both the core components and possible expansion modules. The details of the Java implementation of the framework can be found in Chapter 6. This includes a performance analysis of the Java2D engine and the documentation/programming pratices used during the construction of the framework.

Part 3 (chapter 7-8) relates how this framework was used in a class setting and proposes possible evolutions for the framework. Chapter 7 focuses on the various choices students made using this framework and compares their results to previous classes. Chapter 8 concludes with various suggestion for improving both the course and the framework.

Chapter 2
Fundamentals of Game Programming

This chapter reviews fundamental game programming and game design concepts, focusing mostly on the vocabulary used in this thesis. Note that this review should not be viewed as a complete definition of those concepts, but as a summary to establish a common understanding with the reader.

2.1 Game Programming Concepts

Like many other professional disciplines, game programming uses a vocabulary that is shared by many industries. However, terms used by various industries can often have subtle differences.

The following section describes some key terms used by this document, focusing on the game programming context for these terms.

2.1.1 Frame Rate

Animation in video games is achieved by the subsequent drawing of time-varying images depicting motion in the game. Each of these image is called a frame and the speed at which these images are drawn is known as the frame rate.

To achieve the illusion of motion, a minimum 16 of these frames must be drawn every second. If a frame speed of less than 16 fps is used, our eyes will notice the

consecutive drawing of the different images. Perfectly smooth animations requires a minimum frame rate of 30 fps. At this speed, the human eyes cannot notice the sequential drawing of the different images.

Animations of less than 24 fps might be considered jerky and unrealistic. At this speed, our eyes cannot see the individual images, but they notice that the animation is missing some key frames. This phenomena is known as temporal aliasing and mostly occurs when fast moving objects are drawn. This problem can be solved by increasing the temporal resolution (increasing the number of frames per second) of the animation.

The performance of a video game can be measured by the frame rate that can be achieved with a given hardware configuration. It is the programmer's responsibility to assure that the video game runs at an acceptable frame rate on the minimum required hardware. Although a higher frame rate is always desirable, the maximum frame rate of a video game is limited by the monitor used to display that game.

The refresh rate of a monitor describes how often the image on the monitor is drawn. A typical television screen will have a refresh rate of 60 Hz, thus the screen is redrawn every 60 seconds. Computer screens are designed to support much higher refresh rates, usually 70Hz to 90Hz. If a video games achieves a frame rate higher than the refresh rate of the monitor, the viewer will not benefit from the additional frames because the monitor is not fast enough to draw these additional frames.

2.1.2 Double Buffering

Modern computers (and game consoles) store the image currently displayed on the monitor in video memory. Programmers can alter the displayed image by modifying the content of the video memory. However, precautions must be taken to prevent changes in the video memory while the monitor is drawing the current frame. Such a situation would allow the viewer to see the different immediate states of the frame as they are being drawn.

A simple solution to this problem is double-buffering in which an additionnal buffer in memory is created [FvDFH97]. The first buffer contains the image currently

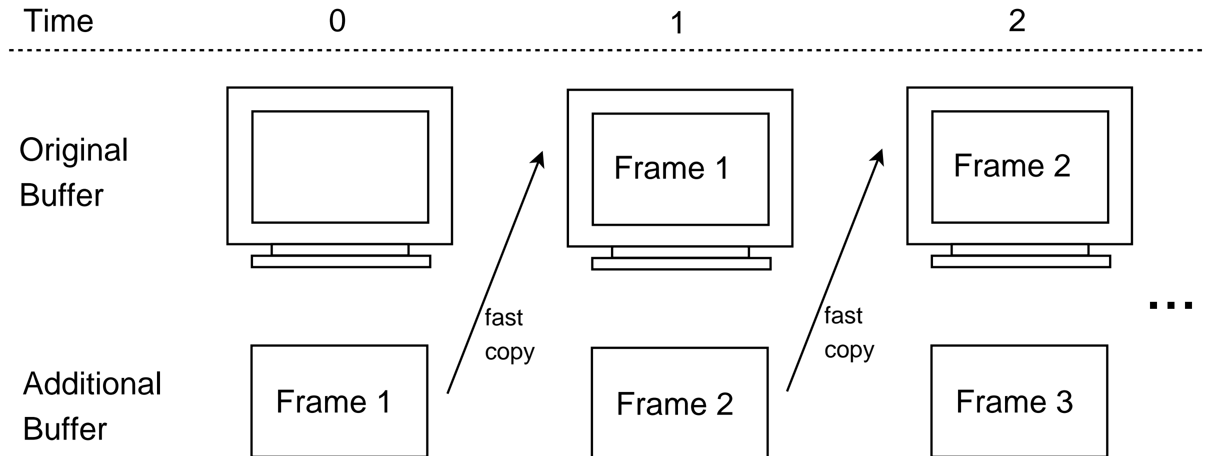

Figure 2.1: Content of buffers in a double buffering strategy.

displayed while the additional buffer is used to prepare the next frame (see figure 2.1). Given that the two buffers are independent, the next frame can be prepared without altering the content of the screen.

When the next frame is ready to be displayed, the content of the additional buffer is copied to the original buffer using a near-instantaneous copy function. Subsequently, the application uses the additional buffer to prepare the the next frame.

Other buffering techniques, such as triple-buffering, can also be used, but are beyond the scope of this document.

2.1.3 Game Loop

The game loop contains the core implementation of a game. It consists of the three following continuous tasks :

- The system monitors its inputs for changes.
- The system recomputes its state given the detected input.
- The system updates its outputs given the changes in state.

These task should be executed concurrently since they are independent. The state of a game is continually changing, whether the user enters input or not. Furthermore,

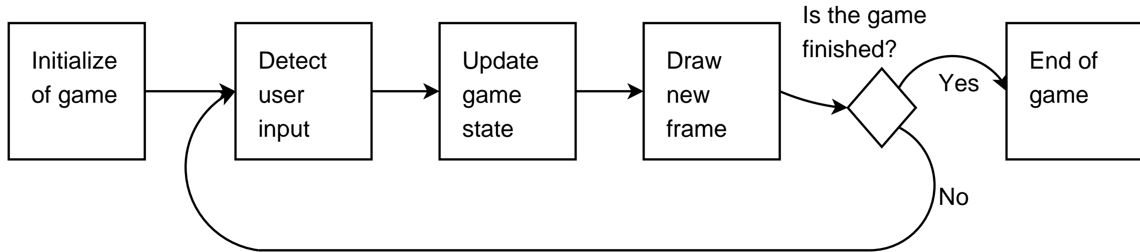

Figure 2.2: Example of a simple game loop.

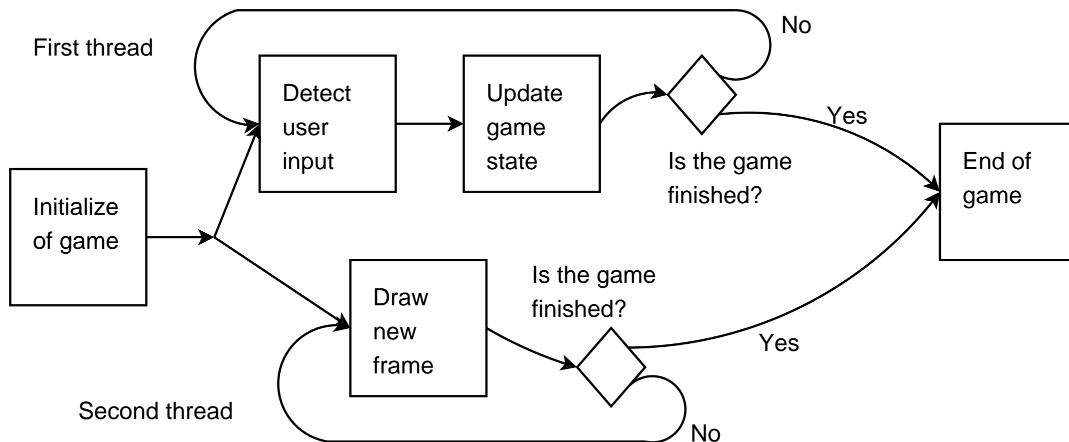

Figure 2.3: Example of a dual threaded game loop.

since the state is continuously updated, the system cannot wait for an exact state to update its output. However, given that most gaming machine are single-processor machines, a video game must create the illusion of these three tasks simultaneously executing.

Game loops are usually single or dual threaded [DS03] (see figure 2.2, 2.3). If the three tasks require very little time to complete, only a single threaded loop is required. However, when recomputing the state requires a large amount of time, a dual threaded strategy is preferable. Splitting the state computation and the output rendering allows a game to compute its state without affecting the frame rate. In other words, more time can be spent on game related computation (such as an A.I. planning its next move) without reducing the number of updates to the monitor.

Unfortunately, the dual threaded strategy introduces complexities since it requires some synchronization mechnanism between the two threads. Even though the two threads might be operating at the same speed, precautions must be taken so that the display thread does not read the game state while the other thread is updating it. Failure to do so could cause the display thread to draw an inconstant state of the game.

2.2 Game Design Concepts

Most established industries, like finance, health or sports, have a vocabulary to call their own. However, the field of video game design suffers from the lack of a common vocabulary [Chu99], borrowing design methodologies mostly from the movie industry. Unfortunately, these methodologies fail to address the interactive nature of video games.

The following section describes some key terms specific to this interactive media.

2.2.1 Dimensionality of Games

Video games are commonly described as two-dimensional or three-dimensional. Two-dimensional games allow the player to move on two axis (up/down and left/right) while three-dimensional games favor movement on three axis. Though this notion describes the degrees of freedom a player is given, it gives very little information on the environment and how the player perceives it.

The dimensionality of games can be broken down into three categories: the dimensionality of a player's motion, the dimensionality of the world's motion and the dimensionality of the viewer's motion [Ruc03].

We have already covered the first notion of dimensionality while discussion the degrees of freedom a player is given in a game. It is important to mention that some games, such as Pong, only provide one degree of freedom (up/down) (see figure 2.4). Other games provide partial degrees of freedom. For example, Tetris allows a player to move their blocks left, right and down. Such a game is said to have 1.5 degrees of

freedom. This notion of partial degrees of freedom can be found in all three types of dimensionality.

The dimensionality of the world's motion describes how the environment interacts with the player. Our previous example, Pong, has two degrees of world freedom since the ball is allowed to move on the X axis and the Y axis. If we consider a modern first-person shooter such as Half-Life 2, the world's motion has three degrees of freedom since opponents can use changes in elevation against a player (e.g. jumping from a building to surprise a player).

The dimensionality of the viewer's motion describes how the viewpoint of a player can change. Our classic example of Pong has zero degrees of viewer motion since the game play area never scrolls. A traditional strategy game, such as Warcraft 2, where a player can scroll the game map on the X and Y axis has two degrees of viewer motion. If the player could freely zoom in/out of the map in the same strategy game, then that game would have three degrees of viewer motion.

2.2.2 Single vs. Multiplayer

The scenario and technical challenges of designing a single or a multiplayer game are very different. These issues must be taken into careful consideration, especially if a game should support both single and multi player.

People play video games because of the stories they tell and the stories they build playing them. In a single player game, the story gives the player goals and tasks. Players interact with the story line in their unique fashion, thus creating their game play experience. In a multiplayer game, the goals and tasks can also be defined by the story, but they are heavily influenced by the various interactions between the players. This is especially true with massively multiplayer online games (MMOG). Thus the single player scenario and the multiplayer scenario for the same game should have different goals.

Single and multiplayer games also have different technological focus. As previously mentioned, single player games often have a more elaborate story line. As such, they require special components to relate the story and allow the player to progress through

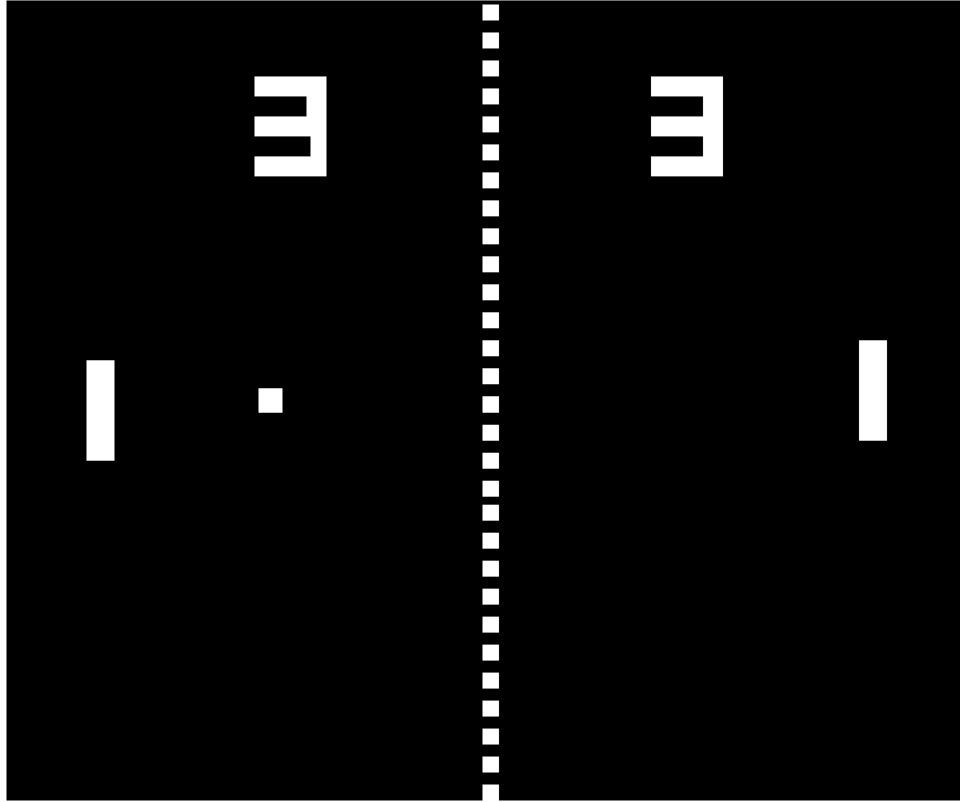

Figure 2.4: Pong features one degree of player's motion, two degrees of world's motion and zero degrees of viewer's motion. Image Source: Wikipedia

it. An example of such a component would be a cinematic engine, which allows a game to have cinematic scenes that relate the story. These story telling components are often not needed in a multiplayer game.

In a multiplayer game, every player must have a consistent view of the game state. Time delays in the communication between the player's machines can make this a very difficult challenge. Failure to properly update the global state can result in paradoxical situation, such as two competitors believing they each won the same race. Techniques, such as dead-reckoning [PW02], where missing information is estimated, try to address this problem.

2.2.3 Real Time vs. Turn Based

The concept of real time and turn based games can be better understood by looking at the differences between the two: real time games exist in a continual continuum of time as opposed to turn based games which exist in a discrete continuum of time. This difference can be observed from the player's point of view and the programmer's point of view.

From the player's perspective, turn based games mostly focus on tactics while real time games focus on strategy [Wal02]. In real time games, players have a tendency to focus on short-term goals and winning battles. Turn based games allows players to reflect on long term goals and focus on winning the war. The artificial intelligence in turn based games offers a stronger opponent, mostly because more CPU cycles can be spared to the computer opponents. In contrast, a player's ability to successfully complete a real-time game often relies on his reflexes and his ability to make quick decisions.

From a programmer's perspective, real time games are more difficult to develop since every task must be completed under a time-constrained condition [DS03]. In other words, player input must be processed and game state must be updated in a timely fashion. This can be very difficult, especially in a multiplayer game where, in addition to the tasks mentioned above, the game engine must maintain a synchronized game state on all player machines. This requires a distributed environment which can be updated and synchronized in continuous time. Turn based games are much easier to develop since they remove the time-constrained condition and allow the programmer to allocate the necessary CPU cycles to each task. Building a distributed environment that synchronizes itself across all nodes in discrete time is also significantly easier.

Part I

Framework Rationale

Chapter 3
Comp 361 : System Development Project

3.1 Course Description

In Fall 2002, the School of Computer Science at McGill University introduced a new Software Engineering program. This program, under the administration of the Faculty of Science, offers a practical approach to learning Computer Science and focuses on various software-related topics such as object-oriented software design, software processes and programming techniques [oCS05]. To create this new software-focused program, several new course were needed. Among them was COMP-361: Systems Development Project.

The official course description can be found in the undergraduate course calendar [ARA04].

> Practical issues in systems programming including: inter-process communication, task scheduling, special purpose systems, multi-processor systems. Implementation of a large body of software to illustrate core concepts and provide substantial hands-on experience.

Professor Jörg Kienzle was given the task of designing a course that would fit these requirements. The obvious solution to this problem would have been to design a course that requires students to design and implement components of an operating

system. Already having a background in game programming, Professor Kienzle however decided that a networked turn-based game project was an ideal topic for this course. If properly designed, a game programming project would allow to evaluate the students abilities to work on a large piece of software in groups, while at the same time provide a challenge that might stimulate some students to go beyond the minimum requirements.

3.2 Course Goals

The Systems Development Project course was introduced into the Software Engineering program to give students the opportunity to work in teams on a large-scale project early into their Bachelors degree. Here are some of the course goals that were declared important early in the planning of the course.

- Students work in teams of three to four students, thus gaining some valuable teamwork experience. They also learn how to use tools related to teamwork such as a version control system.

- The requirements for the games will be incomplete. Thus, students learn how to make design decision. Given that they implement the game they design, they acquire valuable insight on how their design decisions affect their project.

- The game is networked based. This gives students an opportunity to work on a project that requires interprocess communication and concurrent programming skills.

- Students must use an object-oriented programming language. Although students should have the freedom to choose which tools they can use to complete the project, students should also get some experience in building large projects using object-orientation.

Game programming projects can be very complicated and cover a broad range of topics. Thus, we defined topics that should not be covered in the course:

- Students should not have to learn about graphic programming issues (such as transparency masks, color depth and palettes, etc).
- Students should not have to learn about audio programming. Audio is outside the scope of this course will not be required for the project.
- The game should not be real-time. Building the state synchronization algorithm needed for a real-time game is a difficult challenge and is best left to a distributed computing course.

Given the course goals and the current game development tools on the market, it was decided that some kind of game development framework would have to be provided to the students. This framework would allow students to focus on the course goals while hiding the complexities of game programming.

3.3 Previous Framework

The Systems Development Project course, COMP-361, was offered for the first time in Winter 2004. Students were required to implement a turn-based strategy game, Naval Battle. Unfortunately, there was not enough time to develop a proper game programming framework for the start of the term. Instead, Sheriff Shaker developed a game-specific graphical user interface (GUI) and students could program the game behavior into that GUI, thus avoiding graphic programming issues.

The game-specific GUI provided a library and a template for students to make their game. The library provided a complete set of API that would allow students to draw the necessary elements of the game. Students could use methods such as *drawBattleShip* or *drawTorpedo* to draw the various elements of the game. A layout for the game grid and necessary buttons was also provided.

To implement the behavior of the game, students were provided with a template file where they could add their implementation code. Students where also free to re-implement the template file, as long as their new template had the required functions.

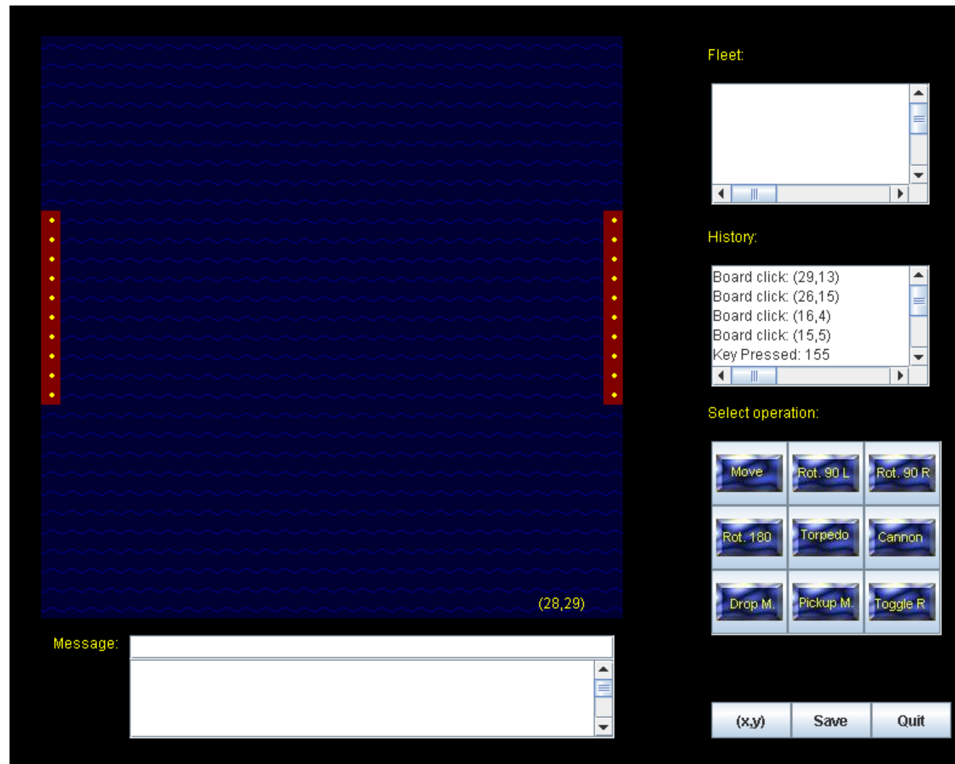

Figure 3.1: BattleShip Framework provided to Winter 2003 COMP–361 students.

Unfortunately, given the tight time frame available to develop this GUI, very little documentation was written on how to use the it. Although a tutorial was given to teach students how to use templates, students had to study the source code to understand how to properly use the GUI.

3.4 Student Projects and Feedback

In Winter 2004, the COMP-361 students were grouped into 10 teams of 4 students. Although some students had an engineering background, most students were from the faculty of science.

As previously mentioned, students were required to implement Naval Battle, a BattleShip-like game 3.2. However, this game was far more complicated than the traditional version of Battleship since it added the notion of radars, firing range,

Figure 3.2: Naval Battle, as implemented by Team 01 using the provided GUI

islands and several new ship types. The game was turn based.

Students were given free choice in the technology they used for their project. Students were also provided with the game-specific GUI described in the previous section. This allowed students uncomfortable with graphic programming to avoid building a GUI and focus on the game play instead. However, the variety of technologies used by the students to complete the project was surprising (see table 3.1).

A quick analysis of the class demographics allowed us to conclude that students with an engineering background favored C++ while students from the Faculty of Science preferred to use Java. Although it might be argued that engineering students prefer C++ because they are exposed to it early in their program, Computer Engineering and Software Engineering students take the same introduction classes as Computer Science students. Further study is required to determine if the students'

Programming Language	Software Development Kits	Number of teams
Java	Swing	5
Java	Naval Battle GUI	1
C++	DirectX	1
C++	DirectX (XBox)	1
C++	SDL	1
Python	PyGame (SDL)	1

Table 3.1: Programming Languages and Software Development Kits used by students during the Winter 2004 term.

choice is motivated by familiarity with the language or the performance difference between C++ and Java.

It should be noted that only one team out of 10 chose to use the provided GUI. Discussions with the students revealed that the GUI was not chosen for the following reasons:

- Some students felt that the framework was too restrictive. By providing the graphic interface for the game, students felt that the GUI lacked the flexibility to allow them to customize features in the game.
- The GUI had very little documentation. Thus, students knew very little about it and could not easily determine if it would fit their needs.

Netherless, more than half of the groups chose to do their project using Java. Many of those groups were heavily influenced by the GUI and opted to recycle some of its source code.

To summarize, the Winter 2004 term of COMP-361 made it clear that some students do require a tool to help them complete the project. However, this tool is not accepted accepted by the students if it does not provide them with adequate flexibility and proper documentation. Students do also not hesitate to recycle existing components from a tool.

Chapter 4
Improving the Course Infrastructure

One defining characteristic of the Systems Development Project course is that students are allowed to choose the technology they use to complete their project. Given that only 10% of the class chose to use the provided GUI, we can conclude that it did not properly address the students' needs. During final grading of the project, students were asked what was the most difficult tasks of the project. The results (see table 4.1) indicate that graphics programming and GUI design is one of the most difficult tasks of the project, thus emphasizing the need for a proper development framework.

Most difficult task	Number of teams
Coding a large scale project	3
Properly designing and planning the project	3
Building the GUI and programming the graphics	4

Table 4.1: Student feedback on most difficult task during the Winter 2004 project

4.1 Defining the Goals of the Infrastructure

It was obvious that the students needed a flexible game programming framework, rather than just a GUI catered towards a specific game. A careful review of the tools

provided by similar courses [Ruc03, SW04] and discussion with the students allowed us to build a list of goals for such a new framework.

- The framework should not be game specific. A major drawback of the game-specific GUI offered in Winter 2004 was the lack of freedom. Methods such as *drawMap* and *drawBattleShip* were too high level and forced the students to develop their project using a predefined design. Lower level methods such as *drawImage* should be provided instead.

- The framework should be easy to learn. Frameworks with an extensive feature set are difficult to learn, mostly because of the sheer number of options offered to the programmer. Most game development kits on the market suffer from this problem, since they target commercial game companies. The framework to be developed should have a limited feature set, focusing on the features students need. A simple API should be easier to learn.

- The framework should provide fast results. Some game development frameworks require hundreds of lines of code before a programmer can open a window and draw on it. Although this approach might provide additional flexibility, it becomes a frustrating learning tool since an important amount of effort is required to get an initial result. The framework to be developed should allow students to get fast results, thus motivating them to learn more about the tool.

- The framework should be properly documented. This was the second major drawback of the game-specific GUI: lack of documentation. Even though the framework provided a complete API to build the game, the API was not documented, thus difficult to learn. Students wanting to use the framework had to dive into the source code.

- The framework should be reasonability fast. If the students want to include real time animations into their game, the provided framework must be able to run at a reasonable frame rate.

- The framework should be easy to install. Although the School of Computer Science provides all the resources students require to complete the project, some students prefer to work from their home. Thus, the framework should be easy for them to install and use. A single file installation would be preferable.

4.2 Current Game SDKs on the Market

In information technology, the expression "Re-inventing the wheel" symbolizes the absurdity of developing software for a specific task when such software is already available. Thus before proposing to build a new game development tool, we surveyed the game development tool currently available.

The following evaluation criteria were used:

- Portability: On which platform can it be used?
- Performance: How fast/optimized is it?
- Ease of use: How easy is it to learn and use?
- Ease of installation: How easy is it to install?

4.2.1 DirectX

By far, DirectX is the most popular software development kit freely available. The API allows developers to draw on the screen (both in 2D and in 3D), play sounds and music, capture input from the keyboard/mouse/joystick and create multiplayer network games. It is used by numerous large game programming companies and is supported by Microsoft, one of the largest software companies in the world. However, DirectX suffers from two major flaws: it is only available on the Microsoft operating systems and can be very difficult to learn for people with little prior game programming knowledge.

DirectX would be the ideal choice if the framework was only needed in a Microsoft environment. However, the School of Computer Science favors a Unix environment.

4.2.2 SDL : Simple Media Layer

SDL is a relatively new member to the game programming community. Created by Sam Lantinga, a software engineer for Blizzard Entertainment, this API includes modules to interact with video, audio and input components. Code written in SDL is very portable. Even though Windows, Linux, MacOSX and BeOS are the only officially supported operating systems, games written in SDL are reported to work on many other OS with little or no modification. Several commercial projects were built using SDL, including the Linux port of Civilization 3. However, SDL is a tool with no commercial backing.

SDL is the ideal software development kit when working in a platform independent environment. However, given the lack of commercial support, it is somewhat less user friendly to install and to use.

4.2.3 OpenGL

Before being used for game development, OpenGL was the defacto standard for CAD and simulation development. It is well known for its speed, performance and portability. Unfortunately, it is also one of the most complicated APIs to learn and offers no sound support. It does, however, have a large backing in the industry and a tremendous amount of documentation.

Like SDL, OpenGL is a good choice for portability and performance. However, it is very low level and does not offer all the required components. For those reasons, it is used in combination with another development kit such as SDL or DirectX.

4.2.4 Java

Because of its interpreted nature, Java is often considered too slow for game development. However, newer versions of Java (such as 1.4.2 and 1.5) feature video hardware acceleration, allowing Java games to run at an impressive frame rate. Unfortunately,

Software Development Kit	DirectX	SDL	OpenGL	Java
Portability:	Windows Only	Windows, MacOSX, Linux, etc	Windows, MacOSX, Linux, etc	Any platform with a good JVM
Performance:	Fast	Fast	Fast	Medium
Ease of use:	Hard to learn, but gets easier	Not too hard	Very hard, obscure sometimes	Easy to use, harder to use properly
Ease of installation:	Very Easy	Somewhat complicated	Easy	Very Easy

Table 4.2: Evaluation summary of the various game development kits available on the market

properly using these features can be a very tricky task, especially since the documentation is scattered across many books and web sites. Java has the backing of numerous large companies such as IBM and Sun. It provides all the necessary tools to control video, sound and input components. Java is also known for its ease of use and portability.

Java would be the ideal game development platform if it wasn't slow. Although the hardware acceleration features do help improve performance, the proper use of these functionalities is outside the scope of an undergraduate project course.

4.2.5 Summary of Evaluation

None of the game software development kits that have been evaluated matched the goals outlined in the previous section. The tools presented have a high learning curve and would not be appropriate for an undergraduate course. However, both Java and SDL have enormous potential, and therefore might be used to build the new

framework.

Benchmark and test applications were built for both SDL and Java. However, Java was found to be the ideal implementation candidate for the framework (see section 6.1). In addition to being easy to use and install, the benchmark results (see section 6.2) indicated that Java's graphic performance was more than sufficient.

Part II

Minueto

Chapter 5
A Simple Object-Oriented Framework

Minueto is a multi-platform framework who's goal is to simplify the game development process by encapsulating complex programming tasks such as graphics, audio and keyboard/mouse programming into simple to use objects. The framework was originally designed as a game development framework for COMP-361, the Systems Development Project course at McGill University. Students using this infrastructure are only in their second year of undergraduate studies, and therefore Minueto is designed to offer an easy to learn API. However, Minueto is a complete framework and can be of use in any 2D game development project.

5.1 Architecture

Minueto is designed to be a modular framework. Its core components include a 2D graphics engine and a keyboard/mouse listener. Additional components are planned to extend Minueto's core functionalities. Minueto's modular infrastructure allows a programmer to provide additionnal services by creating expansion modules, such has sound support or networking.

5.2 2D Graphics Engine

Minueto's core component includes a 2D graphic engine designed for raster/bitmap graphics. The core classes of this module are *MinuetoWindow* and *MinuetoImage*. A *MinuetoWindow* object corresponds to a window displayed by the operating system. Images are stored in *MinuetoImage* objects and displayed on the screen when drawn in the *MinuetoWindow*.

5.2.1 The Window

The *MinuetoWindow* object (see figure 5.1) is the canvas for the game. Images representing the game are drawn on the *MinuetoWindow*, which is displayed to the user. At the operating system layer, a *MinuetoWindow* is a hardware accelerated drawing surface.

The desired size of this drawing surface is declare in the *MinuetoWindow* constructor. Once a *MinuetoWindow* is created, it cannot be resized. If a *MinuetoWindow* of a difference size is required, the current window can be closed and a new one can be created.

A *MinuetoWindow* can be declared as fullscreen. A fullscreen *MinuetoWindow* has no title bar or window border and completely covers the screen. Minueto will also change the resolution of the screen to match the desired size of the window. If Minueto cannot, for any reason, enable fullscreen mode, an exception will be thrown.

5.2.2 The Images

MinuetoImages (see figure 5.2) are the basic blocks of a Minueto application. The different types of images available are sub-classes of the *MinuetoImage* class. The *MinuetoImageFile* class is used to load images from files. Various popular image formats, such as JPEG, GIF and PNG are supported, as are transparency layers found in several image formats. The *MinuetoRectangle* and *MinuetoCircle* classes are used to create images of rectangles and circles. An empty (transparent) image of specific size can also be created by using the *MinuetoImage* class directly. This can

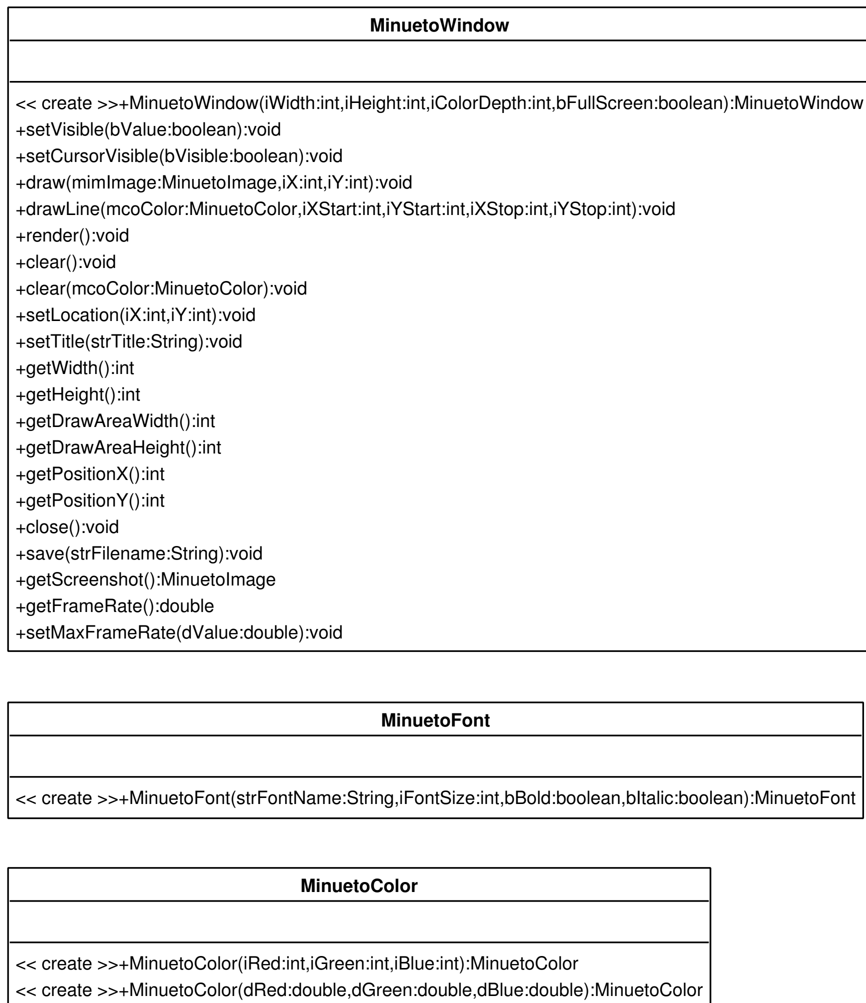

Figure 5.1: UML diagram of the 2D graphic engine component (part 1)

be useful to create more complex images by compositing multiple images.

To draw a string on a *MinuetoWindow*, the string must first be converted to a *MinuetoImage*. Given a string, a *MinuetoFont*, a font size and a *MinuetoColor*, a *MinuetoText* object representing the string of the given color and font can be created. This object, which is a sub-class of *MinuetoImage* can be drawn and manipulated like any other images loaded or created in Minueto.

Minueto includes functions to *scale*, *rotate* or *crop* images. These functions are built into the *MinuetoImage* class and return new transformed copies of the image. The original image, however, remains unchanged. Complex new images can be created dynamically by drawing several simple images on a blank *MinuetoImage*.

5.2.3 Abstraction Classes

Minueto uses two small classes to define the concept of colors and fonts (see figure 5.1). These classes allow Minueto to have its own abstraction of these concepts and thus avoid directly using components of the implementation platform.

MinuetoFont is a small class that contains a reference to a font. The name of the font is declared when constructing the object. Common fonts for the implementation platform should be available as static objects stored within *MinuetoFont*.

MinuetoColor is a small class that contains an RGB (red, green, blue) color value. Common colors, such a red, green, blue, yellow, black and white can be accessed as static objects within *MinuetoColor*. Other colors can be created by passing the RGB color value to the constructor.

5.2.4 Drawing

Both the *MinuetoImage* and *MinuetoWindow* class have *draw* methods. Although their function prototypes are identical, the two functions have two very different goals. The first *draw* method can be used to draw *MinuetoImages* in other *MinuetoImages*, as previously discussed. The second *draw* method, found in *MinuetoWindow*, allows *MinuetoImages* to be drawn on the screen. It should be noted that changes to the *MinuetoWindow* are not immediately shown on screen.

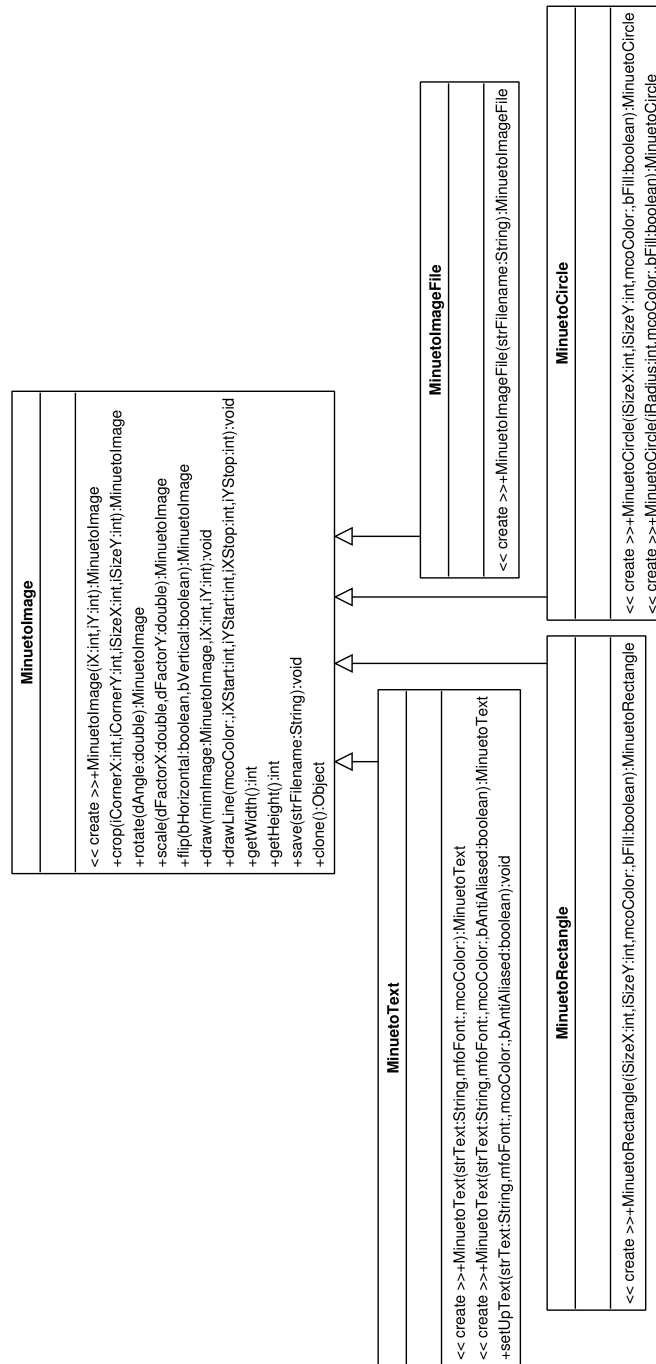

Figure 5.2: UML diagram of the 2D graphic engine component (part 2)

Synchronizing screen updates with the refresh rate of the computer monitor is a difficult task. Failure to do so can result in noticeable graphic glitches. A common solution to this problem is to use double buffering. Frames are prepared/drawn in an alternate space in video memory and the content of this alternate buffer is copied to the active buffer when the next frame must be displayed. *MinuetoWindow* contains a *render* method which shows the next frame by displaying the content of the alternate buffer on the screen.

5.3 Keyboard / Mouse Listener

Keyboard and mouse input handling in Minueto is done differently than in standard user interface frameworks. The following section describes why traditional input gathering techniques, which are usually based on callbacks, were not appropriate for Minueto. In addition, Minueto's proposed solution is also outlined.

5.3.1 Traditional solutions

In a traditional game, keyboard/mouse input is handled with a massive switch statement. Keyboard input is typically stored in a fixed length buffer. A switch statement must test the first character in the buffer to determine which button was pressed. Given that one case statement is required for each key, these switch statements typically contain hundreds of case blocks that, although can be implemented very efficiently, are often very difficult to maintain.

In a traditional GUI application, keyboard/mouse input is handled by an event-based system. When a keyboard/mouse event is detected, a new thread is created to handle that event and execute the associated code. A programmer defines the behavior of a GUI component by implementing a particular interface and registering it with the component. Applications with a high rate of input, such as games, often suffer a performance degradation, because a high number of short lived threads have to be created to handle the input.

Proper software engineering and modularity requires avoiding the massive switch

statement solution. Handlers are a good idea, but they require additionnal threads. Ideally, the number of threads required to run a game should be minimized, since threads introduce concurrency and subsequently, additionnal complexity.

5.3.2 Queuing System

Minueto uses the *MinuetoEventQueue* object (see figure 5.3) to record input from the keyboard and the mouse. The events stored in the queue can then be processed using the handle method found in the queue.

In the backgroud, invisible to the user, events are added in real time to the queue by a thread monitoring the inputs devices. On the user side, events can be processed by the game thread one at a time by calling the handle method.

Minueto uses handler interfaces to process events stored in the queue. To deal with a specific type of input, a programmer must implement a predefined interface and register it with the *MinuetoEventQueue*. When an event is processed, the appropriate method in the registered handler is called. For example, if a keyboard event, such as a key press is processed, the handle method will execute the *keyPress* method in the keyboard handler.

The implementation of Minueto's core component requires the following types of event handler:

- Keyboard input (key press / release)
- Mouse button input (mouse button press / release)
- Mouse motion input (moving the mouse)
- Window events (minimize, maximize, get focus, lose focus)

It is important to note that if no keyboard handlers are registered, then no keyboard events will be recorded on the *MinuetoEventQueue*. Only one handler of a given type can be registered at a time. If a second handler of the same type is registered, the first handler is erased for efficiency reasons and the second one is used.

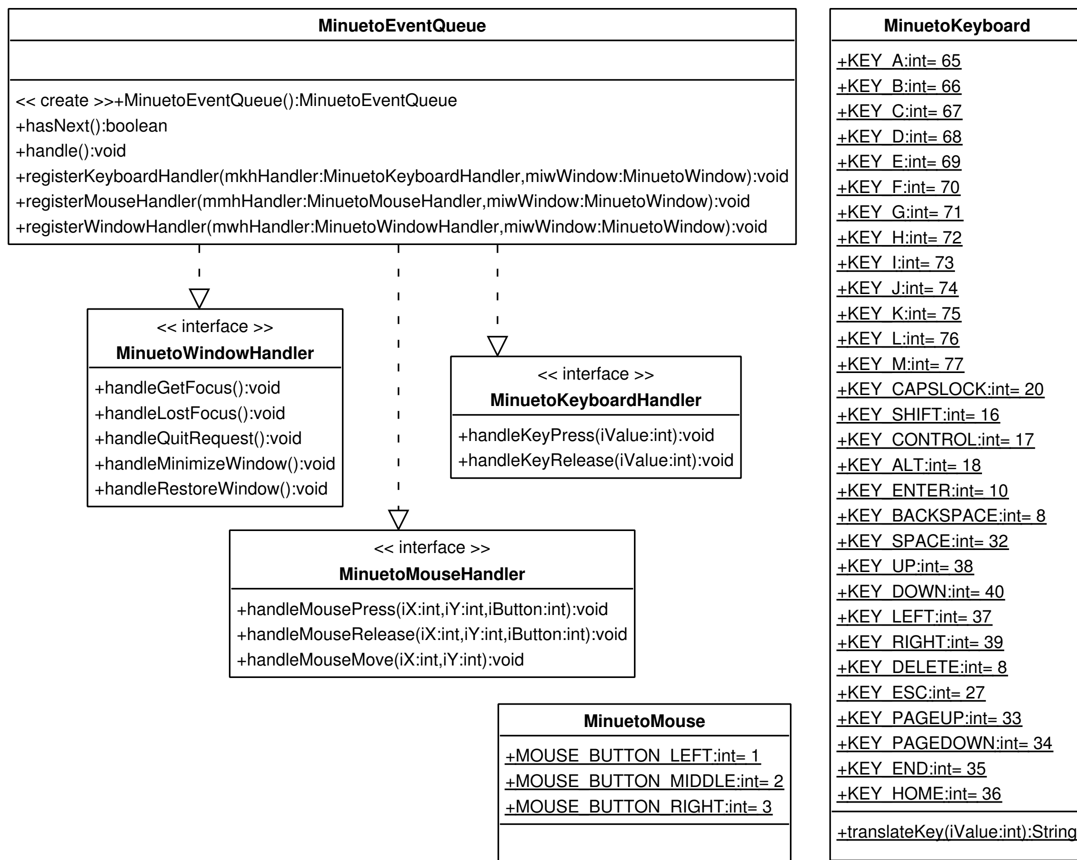

Figure 5.3: UML diagram of the input component

Depending on the type of application, switching handlers depending on the context or mode might be a good idea. For example, a game might have a handler to handle input on the game menu and a second handler for game controls.

5.3.3 Events From Expansion Modules

When possible, other modules should take advantage of the Minueto queue system. For example, since sound playback is non-blocking (see section 5.4.1) the *MinuetoSound* module could add a message to the queue when a sound has been successfully played.

It is important to note that each type of event requires its own handler.

5.4 Possible Expansion Modules

Currently, only the core component of Minueto has been designed and implemented. However, the following section presents components that could be developed to expand Minueto's feature set.

5.4.1 MinuetoSound

The MinuetoSound component allows sound playback. Although functions activating sound playback are non blocking, an optional *MinuetoSoundHandler* can notify the application when the sound playback is completed. The playback of minimum 16 simultaneous channels should be supported. Built-in functions should allow arbitrary changes in the playback position of the file (rewind, forward, reset, etc.).

MinuetoSound should support both sampled sounds and MIDI playback. Supported sampled sound format should include uncompressed wave and at least one popular compressed formats (such as MP3 or OGG). Given the amount of resources available on modern computers, MinuetoSound does not need to support playback of sounds generated on-the-fly (FM synthesis).

5.4.2 MinuetoNetworkMessaging

The MinuetoNetworkMessaging component simplifies network communication by creating an abstraction layer over the TCP/IP socket implementation of the operating system. Messages sent are user-created sub-classes of the *MinuetoNetworkMessage* class. Messages are delivered on the recipient's registered *MinuetoEventQueue* and handled with a *MinuetoNetworkHandler* object.

MinuetoNetworkMessaging uses a client-server architecture. Clients connect to the server by providing the server's IP address and port number. Upon connection, the server returns to the client a player id number which acts as the address of the client. Messages sent from one player to another are routed through the server. The id 0 is reserved for the server (so that clients always know the server id) and the id -1 is a multicasting address which broadcast a message to every player and the server.

The limitations imposed on the format of messages are determined by the implementation language. Ideally, this module should allow the sending and receiving of any serializable object.

In addition, a peer-to-peer version of this module could be implemented. One major difficulty in creating this component would be implementing an efficient routing algorithm for the peer-to-peer network.

5.4.3 MinuetoNetworkSync

The MinuetoNetworkSync component is a highly-concurrent network object store. It abstracts network communication between different applications. Applications just share a number of objects.

At first glance, this module might seem similar to the Common Object Request Broker Architecture (CORBA) or to Remote Method Invocation (RMI). However, MinuetoNetworkSync is simpler than both models because it has a unique centralized server which stores all shared objects and removes the need for a naming server.

Shared objects are created by a client and registered with the server. A unique name is assigned to each object. Other clients can use this name to retrieve the object from the object store. Depending on the features available on the implementation

language, objects registered with clients can be designed in different ways:

- If the implementation language supports reflection, then any objects can be registered with the MinuetoNetworkSync module. The module can use reflection to read/write directly into any registered object.
- If the implementation language supports aspects, then any objects can be registered with the module. The code to allow MinuetoNetworkSync to read/write registered objects could be weaved in at compile time. To simplify the identification of shared objects at compile time, shared objects should implement an empty predefined interface.
- If the implementation language does not support reflection or aspects, then shared objects could implement an interface which defines read/write operations on the object. These functions could then be used by the module to monitor and update registered objects.

5.4.4 MinuetoXML

Computer games have various long-term storage needs. Such needs include map level data (see figure 5.4), saved games, NPC dialogues, etc. Simple text files are a popular solution to this long-term storage need, but they are typically complicated to parse and inflexible. The MinuetoXML component provides an alternative for long-term storage by allowing the previously mentioned items to be stored in an XML file. XML provides a well-structured and flexible format, where data can be saved in an hierarchical tree. Data can then be easily retrieved by parsing the tree. In addition, if the structure of the tree needs to be changed, only slight modification of the parser is required.

The XML tree can be initialized by creating a *MinuetoXMLTree* object (which will also acts as the root node). Nodes can be added to the tree through the use of *MinuetoXMLNode* objects. Attributes can be associated to nodes by creating objects of the *MinuetoXMLAttribute* type. The *MinuetoXMLTree* also provides functions to load and save the tree in an compressed or uncompressed text format. Information

```
<?xml version="1.0" encoding="UTF-8"?>
<Map>
  <Size Height="50" Width="50"/>
  <Data>
    <Tile x="0" y="0" type="2" imagekey="15"/>
    <Tile x="0" y="1" type="2" imagekey="15"/>
    ...
    <Tile x="4" y="21" type="1" imagekey="0">
      <City name="Terrebonne" type="0" owner="0" production="3">
        <UnitProd id="16" type="3" owner="0" buildstat="1"/>
      </City>
    </Tile>
    ...
    <Tile x="49" y="48" type="1" imagekey="0"/>
    <Tile x="49" y="49" type="1" imagekey="0"/>
  </Data>
</Map>
```

Figure 5.4: Example of a game map stored as a XML file.

can be read from the XML tree by traversing the various nodes and examining their attributes.

5.4.5 MinuetoTileMap

Early 2D video games had important memory restrictions. First generation consoles had very little video memory, and storage memory in game cartridge was very expensive. This limitation could be overcome by the use of a technique known as tile maps. This technique allows developers to use a limited number of tiles to create large worlds for players to explore. These square tiles can easily be loaded in video memory. The game can then consult an array containing the level data to determine where each tile should be drawn.

The MinuetoTileMap component would allow fast development of games using a tile-based background. The module should support square tiles of a user-defined size

Figure 5.5: Mappy Win32, a tile map editor.

(16x16, 32x32, 64x64, 128x128) and maps of arbitrary sizes. MinuetoTileMap could allow attributes to be assigned to coordinates, such as the name of a city, the id of an event triggered by stepping on the tile, etc.

The data stored in the map (both tile information and additional attribute) could be saved in XML format. MinuetoTileMap could also be compatible with existing tile map editors, such as Mappy Win32 (see figure 5.5). The module would provide the necessary functions to load and save a map. When provided with the *MinuetoImages* of the tiles to display, MinuetoTileMap could render a section of the map and return it as a *MinuetoImage*. A rudimentary level editor should be provided to serve both as a tool to make the maps and example code to understand how MinuetoTileMap can be used.

5.4.6 MinuetoUI

Visual development tools, such as Visual Basic, TkInter and Swing provide reusable user interface (UI) components such as text boxes, drop down boxes, buttons, etc. The behavior of these UI components is often multi-threaded, one thread being created each time a user interacts with these components. This can lead to an important performance impact if the user heavily interacts with the application, as mentioned in section 5.3.1.

MinuetoUI could provide reusable user interface components that integrate with the *MinuetoEventQueue* system. The behavior of these components would be defined by handlers. Events produced by the components, such as clicking and editing, would be stored in the *MinuetoEventQueue* and handled when the handle method is called, much like keyboard and mouse input.

This module could be particularly difficult to implement because of the subtleties in the interaction between this module and the keyboard/mouse input module. For example, when a keyboard key is pressed, Minueto must determine if the keyboard input should be sent to a text box. This means Minueto must track mouse clicks to determine which, if any, text box has focus.

Chapter 6
An implementation in Java

6.1 Why Java?

The first implementation of Minueto was developed using C++ and the SDL framework. The C++ programming langage was chosen because it is favored by game developers. The SDL development framework was chosen because of its platform independence and impressive community support.

However, after a few weeks of development, it became obvious that using a game development framework in C++ would require a skill-set that McGill Computer Science undergraduate students did not have. Teaching this skill-set in Comp-361 would have greatly increased the burden on students without adding to the required teaching goals.

The first year of undergraduate studies at McGill in Computer Science heavily focuses on object-oriented programming in Java. However, we had reservations about game development in Java, given its interpreted nature and the slow performance of Swing. Before committing Minueto to Java, we needed to test the performance of the Java 2D engine to determine if it was suitable for our needs.

The initial benchmarks for simple demos running in a window of 800x600 pixels on the school's Linux workstation provided results of 150 frames per second. This result was well above our initial expectation. However, more tests were required to determine if a sufficient frame rate could be maintained in a complete game. In the

end, inefficient programming of the game demo logic resulted in a greater impact to the frame rate than the increased burden on the Java 2D engine. The engine could easily achieve the required performance on a moderately fast computer (1Gz CPU with a graphic card that supports hardware acceleration). More information on the benchmark programs and results can be found in the next section.

Given its platform independence and sufficient performance, we chose Java as the implementation language for Minueto. More importantly, we chose Java because we knew that students would have sufficient background to properly use the framework.

6.2 Performance Analysis

As previously mentioned, before selecting Java as our first implementation platform, we needed to study the performance and limitations of the Java 2D engine. The following section describes some of the initial benchmarks that were created for this purpose, and the analysis of the data we gathered.

6.2.1 Benchmarks

The first goal of the benchmarks were to determine if the Java 2D engine had all the necessary features to implement Minueto. The second goal was to evaluate its performance and ascertain if a sufficient frame rate could be achieved. The first round of tests used two small benchmark applications.

The first benchmark, BlackWizardGrass, features a small character sprite that can be moved over a field of grass (see figure 6.1). The field of grass is a tile map composed of 32x32 pixels grass tiles. The field is redrawn, tile-by-tile, at every frame. The sprite character is composed of eight 32x32 pixels tiles, each of them representing a different orientation or step in the walk cycle of the character. Only one of these tiles is displayed at each frame, depending on the character's current orientation and step. The benchmark runs in a 800 by 600 pixel window.

The second benchmark, TownMap, is very similar to the first one (see figure 6.2). It features three small character sprites walking over a slightly more complex tile

Figure 6.1: The first benchmark, BlackWizardGrass

map. The tile map is composed of several different 32x32 pixels tiles, some of them depicting the edge of a small cliff. The user can control one of the characters. The two other characters move randomly. Like in the previous demo, each of the characters have eight animation tiles, each one depicting a different step or orientation in the walk cycle of the character.

Both benchmarks can be launched from a small launcher application. This application allows the user to run the benchmark in either windowed mode or fullscreen mode. The user can also chose to use either double buffering techniques discussed in section 6.3.2.

6.2.2 Benchmark Results

There were no problems implementing both benchmarks using the Java 2D engine. Given the simplicity of both applications, efforts could be concentrated on tuning

Figure 6.2: The second benchmark, TownMap

both benchmarks to maximize performance. The minimum acceptable frame rate was 30 fps for both benchmarks.

The first demo, BlackGrassWizard achieved impressive performance (see figure 6.3). In windowed mode, the benchmark broke the 150 fps barrier on the workstations found in the student's labs. In fullscreen mode (which was available only on Windows XP at the time), the frame rate capped at 70 fps, which corresponds to the refresh rate of the monitors. A quick look at the results allowed us to conclude that the Windows version of the Java2D engine (which relies on DirectX for its acceleration) outperformed Linux (which relies directly on X11 for its acceleration). Finally, even on modest hardware, the target goal of 30 fps was achieved. The mediocre performance of MacOS X's JVM is explained in section 6.5.2.

Results from the second benchmark (see figure 6.4) are very similar to the first benchmark. This is not surprising since both benchmarks have similar graphics. The

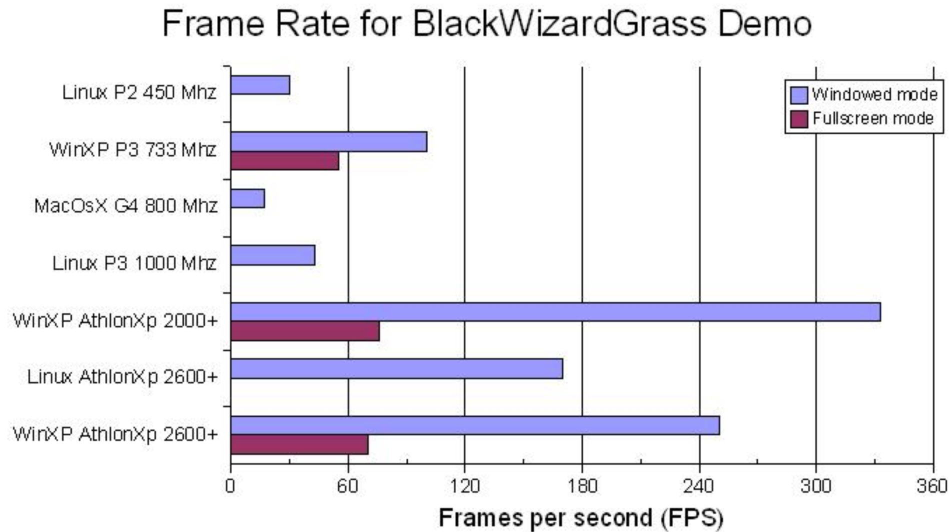

Figure 6.3: Frame rate achieved while running BlackWizardGrass demo.

small loss of speed in the second benchmark is caused by the calculations used to move the two random characters.

Given that we were able to build our two benchmarks and run them at an acceptable frame rate, we can conclude that the Java2D engine can be used to implement Minueto.

6.2.3 Other Performance Analysis

Both benchmarks described in the previous section show that the Java 2D engine is a good candidate for implementing Minueto. However, the simplicity of both benchmarks prevents us from drawing any solid conclusion. Thus, a third benchmark is required.

At the beginning of this project, the game for the 2005 winter session of COMP-361, Strategic Conquest, had already been chosen. This game was an ideal benchmark since Minueto would be first used by the students to implement this game. It was decided that if a simple yet complete version of the game could be developed using the Java 2D engine, then we would definitively choose Java as an implementation

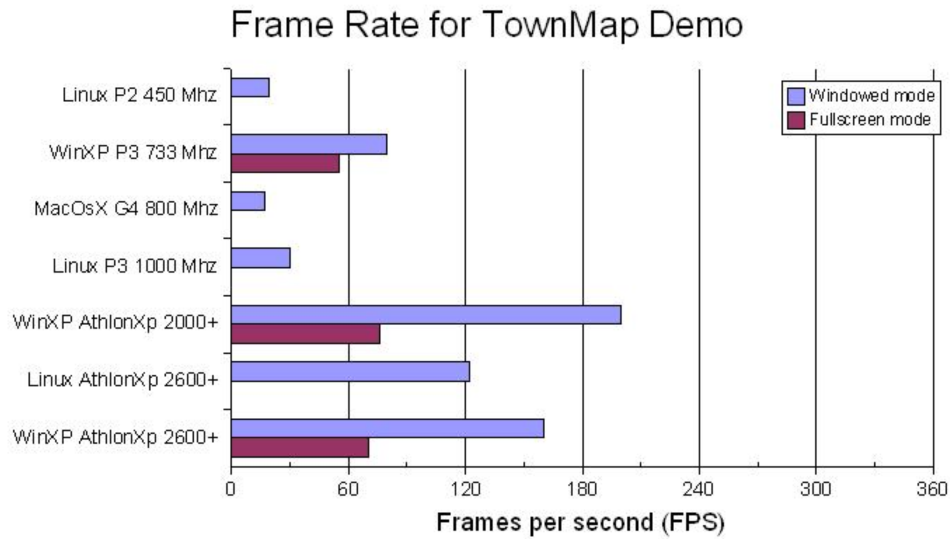

Figure 6.4: Frame rate achieved while running TownMap demo.

language for Minueto.

It took three weeks to create this benchmark (see figure 6.5). Given the complexity of Strategic Conquest's rule set and the required client/server architecture, this was not a surprise. On a modern computer (1.5 Ghz processor or faster), the game runs at a constant 30 fps on both Windows and Linux. On a slower computer, or a computer using MacOs X, an average frame rate of 15 fps was achieved.

Testing of the game revealed that the Java 2D engine was not the display bottleneck in the application. Careful profiling revealed that the game architecture, inefficient due to the rapid game development, caused a larger performance impact than rendering to the screen. When a player moves a unit, the frame rate drops as much as 30%. Synchronizing both clients and the server turned out to be a very ressource expensive task in a poorly defined architecture.

Given the performance achieved by all three benchmarks, it was decided that the Java would be the ideal candidate to implement Minueto. The Java 2D engine would be used to render the graphics and AWT's listeners could be used to gather user input.

Figure 6.5: Test Implementation of Strategic Conquest

6.3 Programming Java Graphics

The Java implementation of Minueto is pretty small, less than 5000 lines of code. One of the important challenge in building the Java implementation of Minueto was learning how to properly harness the acceleration features of the Java 2D engine.

6.3.1 Types of Images

In Java 1.4, Sun introduced the notion of volatile images, accelerated images whose content is stored in video memory. Unfortunately, on certain platforms, the content of the video memory can be replaced by another application. Thus, the volatile image must be reloaded from the original file every time the content of the video memory is lost.

Also introduced in Java 1.4 was the concept of managed images. This type of

image is administered by the Java Virtual Machine. When possible, the JVM tries to accelerate the drawing speed of the image by creating an accelerated copy in video memory. The JVM also deals with the "content lost" problem found in volatile images. However, managed images can be difficult to use because several methods (including methods that modify the image at the pixel level) can make a managed image lose its acceleration benefits.

Chet Haase, an engineer on the Java2D team, documented no less than 38 different methods to create an image with the Java 1.4 API. The different methods produce images with various characteristics. The key to successful image acceleration is to create an image with the same characteristics as the current display. For example, if the current display is switched to RBGA 32-bit color mode, then all images should be created, loaded and/or converted to this color mode. Images stored using different color modes are converted when drawn to the screen, causing tremendous slowdowns.

Minueto avoids this problem by creating and/or loading all images at the same color mode as the current display. To achieve this, Minueto requires that the *MinuetoWindow* object be initialized first. Images are created and/or loaded as managed images and their acceleration is managed by the JVM. These managed images cannot lose their accelerated status since the methods causing the loss of acceleration are not used by the framework and are not available to the users.

6.3.2 Double Buffer Strategy

Early in the implementation phase, it was decided that Minueto would use a double buffer strategy. It was also decided that the implementation details of the double buffer would be hidden from the students.

The initial version of Minueto provided two double buffering techniques: one that manually implemented double buffering and one that was automated by the Java Virtual Machine (JVM). Having two methods allowed us to measure the effectiveness of the JVM's double buffering capabilities.

Minueto could manually create a double buffer by using a hardware accelerated image as the back buffer. All draw operations were done on the back buffer and a

render method copied the content of the back buffer to the screen. Double buffering can also be managed by the JVM by using the createBufferStrategy method.

After several tests, it was obvious that a manual solution was not needed since the automated solution outperformed the manual one in most situations. Thus, Minueto was implemented with the JVM's double buffering technique.

6.4 Documentation Strategy

It has been shown that the difference between a novice and an expert chess player is the fact that the latter has thousands of board configuration stored in his long time memory [HK73]. Expert players can use these memorized board configurations to derive their next move without having to rely too much on their limited working memory. Further research has shown that problem solving relies more on stored memories than complex reasoning [W.94]. This theory can easily be extended to programmers, where the difference between a novice and an expert is years of problem solving experience [GS02]. An experienced programmer can draw upon years of previous programming challenges to find similarities between previous and current problems.

The main purpose of Minueto is to provide a tool that is easy to learn and to use. Students with little or no game programming background do not have the required experiences to deal easily with problems commonly associated with game programming. Minueto's documentation must address that fact and allow the programmer to learn about various problematic situations and their solutions.

Minueto's documentation is available in three different formats to better address the different learning habits of students. The following sections describe these formats and their learning strategies.

6.4.1 Tutorials

On the Minueto website, students can find a collection of *Howto* documents. These short documents are tutorials that cover different specific aspects of Minueto. By

reading these tutorials, students can gain insight on how to achieve basic tasks with Minueto and how to solve common problems. These documents can be read in any order, although some dependencies do exist (to understand X, you must first read Y).

This type of documentation targets students who prefer learning by reading books and manuals.

A complete list of the tutorials available on the Minueto website can be found in Appendix C.

6.4.2 Examples

Some students prefer to learn by example. Minueto is distributed with over 20 examples that cover a wide range of specific topics. Each example is designed to solve exactly one problem. This allows the code for the examples to be as short as possible, thus making it easier for students to understand. By playing around with the examples, the student can gain hands-on experience with the Minueto framework and improve on future problem solving with Minueto.

Given their specific nature, the examples also play an important role in regression testing, as described in section 6.6.3.

A complete list of the examples distributed with the Minueto software development kit can be found in Appendix B.

6.4.3 Specifications

Early in their Bachelor program, students are encouraged to learn how to use the Java API documentation. This documentation is an essential tool for both novice and experienced programmers. The Java API describes the various classes available in the common class libraries provided by the JDK. For example, instructions on how to use linked lists, random number generators, strings and so on are stored in this API documentation.

Given that Minueto is a framework, an API documentation is also an essential tool. Minueto's API documentation is built using the standard JavaDoc tool and follows the documentation guidelines for Java applications as outlined by Sun Microsystem

[Mic00]. The wording and formatting of Minueto's documentation follows as closely as possible those used by the Java API documentation. This allows students to use previous knowledge of the Java API documentation to navigate and understand Minueto's documentation,

6.5 Current Limitation

Although the Java implementation of Minueto was successfully used by several students to create impressive game projects, it still has a few unresolved issues.

6.5.1 Single Window Restriction

Currently, the Java implementation of *MinuetoWindow* is a singleton. This design decision was heavily influence by Java 2D and the need to prevent the loss of hardware acceleration.

Hardware acceleration in the Java 2D engine is provided on a "as-available basis". Although Java provides several methods to create images, only a few of those methods will create an accelerated image. Certain operations, such as pixel manipulations, can cause an image to lose its acceleration status. In the current version of the Sun JDK, once the acceleration status is lost, it cannot be regained.

The method that creates images using hardware acceleration is keyed to a window. That means an image is accelerated as long as it is drawn to that window (or a window that shares similar characteristics). Since we do not fully understand the performance impact of drawing on a window an image that was not keyed to that window, we removed the ability to created multiple windows.

Further testing revealed that certain JVMs crash if two double buffered windows are created. This is a problem with the JVM itself and should be solved in future JVM releases.

This restriction should be lifted in Minueto 2.0 after more experimentation has been done on multi-windowed application environments. It should be noted that students found creative workarounds.

6.5.2 Apple JVM

The most predominant Java Virtual Machine (JVM) for Windows and Linux is developed by Sun Microsystems (subsequently referred to as Sun). Apple develops its own JVM for the MacOs X operating system. Although Sun shares its JVM source code with Apple, the integration of new features into the Apple JVM requires much more time. In addition, Apple computers use a processor architecture (PowerPC) significantly different from those supported by Sun (x68, Sparc). This forces Apple to re-write platform specific components of the JVM.

Currently, there exists a 18 month delay between the Sun release of a new Java Development Kit (JDK) and the Apple JDK. Given that Minueto is a cross platform development framework, its behavior must be similar across all platforms. Thus, new features added in the Java class libraries can only be used by Minueto once Apple integrates these feature into their JDK. This creates an important technology delay for the framework.

One current example of this delay is the addition of nano-second precise timers in Java 1.5. Java 1.4's timers are only precise to the milli-second. Given that the screen is updated several times per second, the precision of timers is paramount. However, to conserve Minueto's constant behavior across all platforms, the nano-second precise timers can only be used once Apple releases a JVM that supports this feature. To have timers of different precision on different operating systems would create abnormal behavior when switching from one operating system to another.

It is also important to note that the current Apple JVM does not have a hardware accelerated Java2D engine. This is a major limitation of the JVM, since games using Minueto without the hardware acceleration rarely achieve a frame-rate of over 25 fps. The newest version of the Apple Java runtime (1.5), which is supposed to have a hardware accelerated Java2D engine, should be release by the time this document is published. However, users wanting to take advantage of this new Java runtime will have to buy an upgrade for their operating system (MacOs x 10.4) since the newest runtime will not work on MacOs X 10.3 and below.

6.5.3 Keyboard Input on Linux

Keyboard input in Minueto is handled with a queue that stores both key press and key release events. These events are generate using AWT's *KeyboardListener* class. The Java API specifies that a key release event will only be generated if a key on the keyboard is released by the user. However, in Linux, under X11, when a key is pressed and not released, multiple key release events are generated, each one immediately followed by a key press event. These additional key release events are generated by X11 which handles a long key press event as a combination of multiple key press and key releases. Other operating systems also produce numerous key press events, but only one key release event.

The additional key release events are problematic when a game measures the length of time keys are pressed. This is impossible to do under Linux because long key press/release are reported as multiple key press/release. Minueto does have some built-in features that try to minimize the amount of extra key press/release events, but this workaround is not always perfectly effective.

This problem cannot be completely fixed until Sun changes the interaction of its JVM with X11. Sun claims that this is a X11 bug.

There exist two different partial solutions to this problem. Some window managers (such as KDE), have configuration options that allow to control the behavior of key repeat events. However, this becomes impractical when distributing the game to a large number of users. The second solution is to resolve user input once all keyboard events are processed. Since false key release events are always followed by a matching key press event, we can usually discard a false key release event if a matching key press event is found in the queue. An example of this can be found in the TriangleRover sample code, found in the Minueto SDK.

6.5.4 Fullscreen Mode Under Linux

Currently, Sun's JVM does not support screen mode changes under Linux. This means that although a window can be put in fullscreen mode (by putting the window in the upper corner, stretching it to screen size and hiding the top menu bar), the

resolution and the color-depth of the screen cannot be changed. This means fullscreen mode is useless under Linux.

Sun had been aware of this problem for a long time and did not fix it in its latest major release of Java (1.5). However, Sun does have a bug ticket opened and promises to fix this in the next major release (1.5.1 or 1.6).

6.6 Programming Practices

Minueto is a project of the McGill Software Engineering Lab. As such, multiple programmers are expected to contribute to improving/fixing Minueto over a long period of time. As in all projects involving multiple developers, standards must be established early in the project to promote an homogeneous development environment.

6.6.1 Coding Standards

Minueto was coded using Sun Microsystem's official Java coding standards [Mic00]. Comments are required on all variable declarations and on any non-intuitive piece of code. The format of the header is standard and largely inspired by those found in Sun Microsystem's JavaDoc documentation [Mic99]. Since the API documentation is auto-generated from the source code, when adding a class or a method, the proper heading must also be added.

6.6.2 Automated Build System

The building of debug versions and release packages of Minueto is automated through the use of *Ant*. The "ant debug" command allows developers to build a debug version of Minueto. It also places all the examples in the same build directory for easy access. Final release packages of Minueto can be created using the "ant all-jdk14" or "ant all-jdk15" commands. Those commands build the packages for use with Sun's/Apple's JDK 1.4.x and Sun's JDK 1.5.x respectively.

6.6.3 Regression Testing

As mentioned previously, the code samples found in Minueto's documentation can also be used for regression testing. Given their specific nature, developers can use the samples to test individual aspects of Minueto. When a new feature is added to Minueto, the developers adding this feature must write a new sample. This gives developers a permanent tool to test that feature and provides Minueto users with sample code that allows them to better understand the new feature. As explained in the following section, all the regression tests must be successfully completed before changes are submitted to the source control system or released on the Internet. Unfortunetaly, common regression tools such as JUnit cannot be used since Minueto's tests application have a graphic output which cannot be easily validated.

6.6.4 Source Control

The latest version of Minueto's source code can be found in a Subversion directory on a server hosted by the McGill School of Computer Science. Access is only granted to active developers. New features to Minueto are verified and approved by the project leader before they are committed to the main source tree. Programmers contributing code to the Minueto source tree must regularly run all the regression test on their development machine. Updates having a potential large impact require the programmer to run the regression tests on multiple platforms and Java SDK. Programmers are also required to note changes to Minueto in the "changelog.txt" document.

6.6.5 New Releases

New versions of Minueto are released when a major feature is implemented or when several bugs are fixed. The new Minueto SDK is announced on the main page and the download page is updated. Before a new SDK is released, the regression tests are executed on all supported platforms and all supported Java SDK. Minueto packages must be created using the automated build system.

6.7 Additional Benchmarks

Although numerous benchmarks were done to test the performance of Java 2D, no benchmarks were used during the development of Minueto. Since Minueto focuses on usability and stability, there was very little need for performance measurements. However, after a year of development, Minueto is nearing a final release and benchmarking could provide important optimization information.

6.7.1 Goals and Methodology

As previously mentioned, some benchmarking was done in the beginning of the project. However, these benchmarks were limited to determining if Java 2D could obtain a sufficient frame rate. Newer benchmarks should help qualify the relation between processor and frame rate. In addition, the benchmarks can be used to establish a relation between video hardware and frame rate, allowing us to better understand the minimum hardware required to use Minueto.

The first benchmarks were run only on one MacOs X machine. This proved to be a serious oversight since a non-negligible number of students use MacOs X as their main development platform. In addition, Apple has released a new version of its Java development kit (1.5) which promises performance improvements for Java 2D.

The Java 2D benchmarks were completed using two demo applications, BlackWizardGrass and TownMap. Converting these two applications to Minueto was not difficult, considering the simple nature of the demos. A complete description of these applications can be found in section 6.2.1.

To gather the required data, 15 computers were chosen, all equipped with a variety of processor, operating system and video hardware. Both demos were run on each machine, both in fullscreen and in windowed mode. When first started, all the demos showed large instabilities in frame rate. However, the frame rate was only recorded after a few minutes, once it had stabilized.

6.7.2 Analysis

The first two tables of this section (6.1 and 6.2) contain the frame rate achieved by both demos in windowed mode. The third table (6.3) also contains the frame rate achieved by both demos, but in fullscreen mode. Please note that fullscreen mode is not available under Linux, so fullscreen performance data is unavailable for that operating system.

Benchmark machine	Black Wizard Grass	Town Map
Imac, G4 800 Mhz, 512 Mb, Geforce 4 MX, MacOs X, JVM 1.4.2	30	27
IBook, G4 800 Mhz, 640 Mb, ATI Radeon 9200, MacOs X, JVM 1.4.2	33	31
Powerbook, G4 1200 Mhz, 2048 Mb, Radeon 9700, MacOs X, JVM 1.5.0	66	64
G5 Tower, Dual 2500 Mhz, 3072 Mb, Geforce 6800 Ultra, MacOs X, JVM 1.4.2	133	124

Table 6.1: Frame rate (in frames per second) achieved while running BlackWizardGrass demo and TownMap demo in windowed mode (continued on next table).

Several conclusions can be drawn from the collected results :

- frame rate for fullscreen results are capped at 60 or 75 fps. This suggests that the frame rate for an application in fullscreen mode is limited by the refresh rate of the screen. This is not a problem since a frame rate higher than the refresh rate of the screen is useless.
- The Athlon XP 2000+ equipped with the S3 Savage 2000 video card suffers from poor performance, especially when compared to its Geforce 4 TI counterpart. This would suggest that poor video hardware can have a significant impact on

Benchmark machine	Black Wizard Grass	Town Map
Pentium 2 333 Mhz, 384 Mb, ATI Rage, Linux, JVM 1.4.2	22	23
Pentium 3 733 Mhz, 512 Mb, Geforce 4 MX (unaccelerated drivers), Linux, JVM 1.4.2	73	72
Athlon XP 2000+, 512 Mb, Geforce 4 TI, Linux, JVM 1.4.2	225	195
Pentium 4 2660 Mhz, 512 Mb, Geforce 4 TI, Linux, JVM 1.4.2	255	255
Athlon 64 3000+ , 1024 Mb, Geforce 5800 (unaccelerated drivers), Linux, JVM 1.5.0	89	86
Celeron 500 Mhz, 512 Mb, Geforce 4 MX, Windows, JVM 1.4.1	75	72
Pentium 3 733 Mhz, 512 Mb, Geforce 4 MX, Window, JVM 1.4.2	108	105
Athlon XP 2000+, 512 Mb, Geforce 4 TI, Windows, JVM 1.5.0	400	360
Athlon XP 2000+, 512 Mb, S3 Savage 2000, Windows, JVM 1.4.2	127	122
Athlon 64 3000+, 1024 Mb, Geforce 5800, Windows, JVM 1.5.0	425	400

Table 6.2: Frame rate (in frames per second) achieved while running BlackWizard-Grass demo and TownMap demo in windowed mode.

Benchmark machine	Black Wizard Grass	Town Map
Imac, G4 800 Mhz, 512 Mb, Geforce 4 MX, MacOs X, JVM 1.4.2	30	28
IBook, G4 800 Mhz, 640 Mb, ATI Radeon 9200, MacOs X, JVM 1.4.2	31	29
Powerbook, G4 1200 Mhz, 2048 Mb, Radeon 9700, MacOs X, JVM 1.5.0	42	39
G5 Tower, Dual 2500 Ghz, 3072 Mb, Geforce 6800 Ultra, MacOs X, JVM 1.4.2	60	59
Celeron 500 Mhz, 510 Mb, Geforce 4 MX, Windows, JVM 1.4.1	60	57
Pentium 3 733 Mhz, 512 Mb, Geforce 4 MX, Window, JVM 1.4.2	73	72
Athlon XP 2000+, 512 Mb, Geforce 4 TI, Windows, JVM 1.5.0	60	60
Athlon XP 2000+, 512 Mb, S3 Savage 2000, Windows, JVM 1.4.2	48	48
Athlon 64 3000+, 1024 Mb, Geforce 5800, Windows, JVM 1.5.0	60	59

Table 6.3: Frame rate (in frames per second) achieved while running BlackWizardGrass demo and TownMap demo in fullscreen mode.

Minueto. A Geforce 4 MX video card, which is a fairly inexpensive, is required to effectively use Minueto.

- The Atlhon64 3000+ using unaccelerated drivers provided a very bad performance under Linux. However, the Pentium 3 733 Mhz, using the same drivers, offered a similar performance. This suggests that the performance of the Linux unaccelerated drivers is capped.
- There seems to be a linear relation between frame rate and processor speed for the MacOs X computers. This would suggest that the poor performance obtained by the 800 Mhz MacOs X computers is caused, in part, by the processor. Although it is already known that the Apple JVM is slower than the Sun JVM, faster MacOS X computers do offer reasonable frame rates when using Minueto.
- The 1.2 Ghz Powerbook presented a surprisingly high (for MacOS X) frame rate. This suggests that the Apple's JVM 1.5 does offer an important increase in Java2D's speed.

Part III

Case Study: Winter 2005

Chapter 7
Student Evaluation

At the time of writing, the students of the Winter 2005 System Development Project (COMP-361) class have successfully completed their game project and are enjoying a well-deserved summer vacation. The results for this year's game, Strategic Conquest were very impressive. Although Strategic Conquest is different from last year's Naval Battle game, the increase in the overall quality of the projects is easily noticed.

7.1 Student's Platform Choices

The Java implementation of Minueto was completed at the end of December 2004, just in time for the Winter 2005 term. Minueto was presented to the COMP-361 students during the third lecture, the two first lectures of the term being reserved for the course introduction and the game description. Although the students were free to use the technology of their choice, they were warned that technical support would only be provided for Java. This was a practical decision, since actively supporting multiple game development platforms would have been a heavy burden on both the professor and his assistant.

In Winter 2004, only one team chose to use the provided game-specific GUI. Among the other teams, half of them chose to use Java and the other half used various other technologies (C++, Python, etc). This year, the adoption rate for the

Programming Language	Software Development Kits	Teams 2004	Teams 2005
Java	Swing/AWT	5	1
Java	Naval Battle GUI	1	N/A
Java	Minueto	N/A	8
C#	DirectX	0	1
C++	DirectX	1	1
C++	DirectX (XBox)	1	0
C++	SDL	1	0
Python	PyGame (SDL)	1	1

Table 7.1: Programming Languages and Software Development Kits used by student during the Winter 2004 & 2005 term.

provided framework was astonishing (see table 7.1). A total of 8 teams chose to use Minueto, a dramatic increase if compared to last year's GUI. One other team chose to build their custom Java graphic library, borrowing from Minueto's code but adopting a greater degree of compatibility with Swing. Two other teams chose to develop their game using C++ and C#. Finally, as in 2004, one team chose Python as their game development platform. Special mention should be given to one of the Minueto teams, who also chose to use Python to develop the game server.

Minueto's high adoption rate suggests that the framework successfully answers students' needs when developing their project.

7.2 Student's Improvements To Minueto

The 2004 Naval Battle GUI did not properly address the students' needs. The GUI itself was poorly documented, so students had difficulties adapting it to their needs. Although several teams did borrow code from the GUI to design their own interface,

most teams rewrote their game interface from scratch. This would suggest that students prefer building their own GUI rather than adapting an existing one to their needs.

Our experiences with Minueto and the 2005 class were very different. Among the eight groups that chose to use Minueto, three of the groups used a modified version of Minueto. Since Minueto's source is commented and distributed with the SDK, students can easily modify Minueto's core components to better answer their needs. The following subsections present some of the modifications that were done by the students.

7.2.1 Alpha Channels

An image is composed of pixels, and each pixel has three color channels : red, green, blue. The value of these three channels determines the color of that pixel. A pixel can also have a fourth channel, the alpha channel, which determines the transparency of that pixel. Graphic engines use this value to create partially transparent ghost-like effects. Since the value of the alpha channel can vary from one pixel to another, different parts of an image can have different transparency values. Because of performance concerns, the alpha transparency model was not implemented in Minueto.

Minueto uses a bitmask transparency model. In other words, a pixel is either opaque or completely transparent. Although ghost-like effects are impossible to achieve with this technique, this transparency model is much faster to draw. Even though Direct3D and OpenGL can be used to accelerate 2D drawing in Java (including alpha transparencies), support for hardware acceleration is still experimental and occasionally crashes the virtual machine.

Two teams modified the Minueto source code to enable the alpha transparency model. Although performance under Windows was adequate, the frame rate on Linux dropped to 1 fps when an OpenGL compatible graphic card was not available. In addition, one of the teams had problems closing their MinuetoWindow when 3D acceleration was enabled. However, both teams were rewarded with beautiful transparency effects.

Further examination of the performance impact of transparency models is required to determine if/how alpha transparencies could be safely integrated into Minueto.

7.2.2 Swing Integration

Swing is a component of the Java Foundation Classes that simplifies the development of applications with graphical user interfaces (GUI). It provides easy to integrate and reusable UI components, such as text boxes, buttons, panels, etc. It is currently impossible to insert a Swing component (such as a button) into a MinuetoWindow.

The Swing library is too slow to be used for game development. Even though Minueto does internally use several Swing components as drawing surfaces, their painting engine is disabled. In other words, Minueto replaces Swing's painting engine with its custom drawing engine. Although Swing components can be added to a MinuetoWindow, they will not be painted properly. Further testing on this problem also revealed compatibility problems between Swing and Java's double buffering components.

The implementation of Strategic Conquest requires several UI components such as buttons and text boxes. Currently, Minueto provides no tool to easily generate and manage these. Some resourceful students avoided Minueto/Swing interaction problems by placing the Swing components in a separate window. In addition, hooks were added to MinuetoWindow so that they could interact with the underlying Swing components. Another team tried to encapsulate MinuetoWindow's drawing surface into a JPanel and add it to a regular Swing window. That modification didn't work too well.

For the moment, Minueto and Swing just don't work together. However, future projects will most likely contain other UI components. Coding these by hand is an unnecessary chore. More research is needed to either create generic UI components for Minueto or improve Minueto/Swing compatibility.

7.2.3 Active Queue

As described in the previous chapter, Minueto uses a special queuing system that allows it to record keyboard/mouse events in real-time and delegate their processing

to the main thread. This avoids the need for small short-lived individual threads to handle each input event. However, a group of students decided, given the low amount of input in their game, that they preferred an active input system. Thus, they modified the mouse and keyboard handlers so that events would be processed as they occured by individual threads.

Although the Minueto architecture discourages such kind of event handling, programmers with GUI development experience may be more comfortable with this kind of event handing. A possible expansion to the Minueto queuing system would be to create a single dedicated thread to continuously deal with input events. However, this idea should be discouraged since it introduces concurrent programming concerns.

7.2.4 Keyboard Handler

When the keyPress or keyRelease methods from a MinuetoKeyboardHandler are called, an integer representing the key (a key ID) is passed as a function parameter . However, to translate the key ID to a character representation (from 59 to 'a'), a large inefficient switch statement is required. This was discovered when students started implementing text boxes.

Ideally, the MinuetoKeyboardHandler should provide both the key ID and the character representation of that key, if applicable. Currently, this change cannot be done without breaking compatibility since the MinuetoKeyboardHandler interface would need to be changed. Subsequently, this change is best left to the next major version of Minueto (2.0?), where backward compatibility will be broken anyways.

To solve this problem in the current version of Minueto, some students modified the MinuetoKeyboardHandler to return a Java KeyEvent object instead of a key id. This allowed them to retrieve both the key id and the character representation when a keyboard event was generated. Unfortunately, this modification cannot be applied to the core component since it breaks Minueto's encapsulation by incorporating Java elements in the public interface.

As a temporary solution, a translate key function was added to the MinuetoKeyboard class. This function will translate a key id into a character representation and

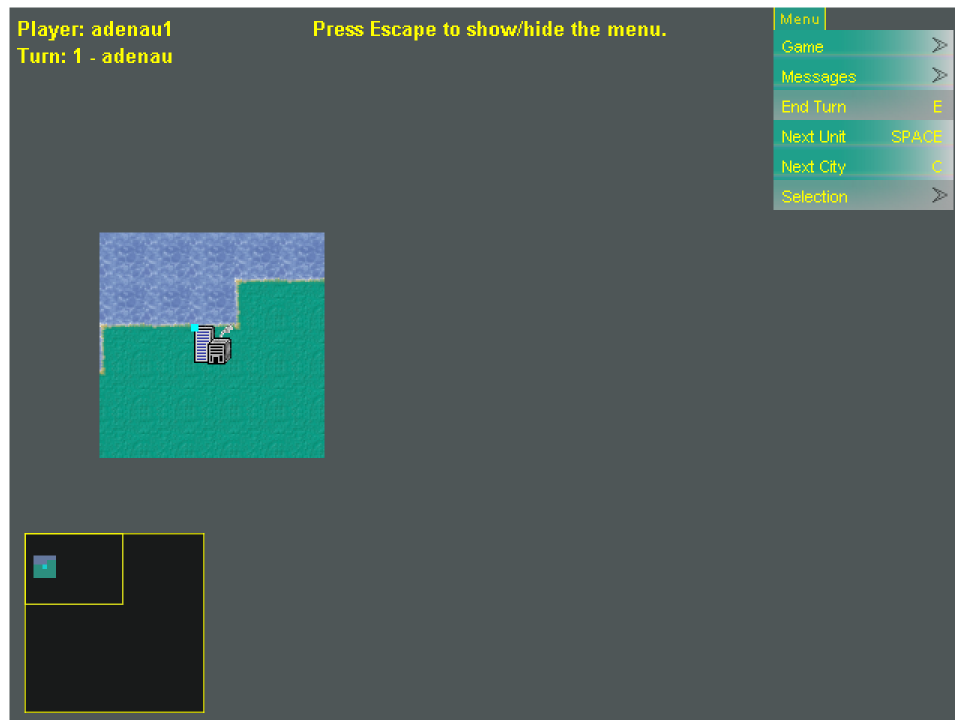

Figure 7.1: Strategic Conquest, team Purple Monkey Screwdrivers

send that character back as a string. Although a bit slow, this function automates the character transformation, thus avoiding the need to modify the handler.

7.3 Observation of Student Progression

Comp-361 students are required to attend weekly meetings with the class professor so their progress can be monitored. Three months into the development of their projects, students are called to a special meeting to present a "demo" version of their project. During this "demo" meeting, students present a working version of their game with a minimal set of features. Students then meet the professor a month later to present the final version of their game.

Although the Minueto framework was introduced during the third lecture, students started using the tool after the first lecture already. Only a few days later, a

Figure 7.2: Strategic Conquest, team Golden Balaika Mariachis

small Minueto demo application had already been posted on the class bulletin board by a student. After the class presentation of Minueto, the number of groups using Minueto doubled. Students also started sending comments and questions. This was concrete proof that students were using the framework.

An important change in the weekly meetings was also noticed. In the previous year, students often complained about the difficulty of developing their custom GUI. Problems using Java/Swing were a big concern for the students. This is understandable, since Swing was never meant to be used for game development. The number of technical complaints and problems dramatically decreased for the class of Winter 2005. Although some Minueto concerns did come up during the meetings, most were diagnosed as incorrect or inefficient uses of Minueto and quickly solved. The only unresolved issue students encountered was the keyboard repetition problem described in section 6.5.3.

Figure 7.3: Strategic Conquest, team Muff Busters

An increase in the quality of the projects was also noticed during the "demo" presentations; the games looked more professional and polished. More importantly, all the student's projects met the minimum requirements for the demonstration. Last's year demonstration had no minimum requirements and some teams did not even manage to present a working project. Hence, the 2005 students were able to develop a better working product given the same amount of time. Given that the 2004 students devoted much time to developing and debugging their GUI, we can deduce that students using Minueto were able to concentrate their efforts on other tasks.

The final project presentations were a learning experience for the Minueto development team. The creativity of some of the student's projects was amazing. Several of Minueto's features had been used in ingenious ways, expanding our vision of Minueto's capabilities (see figure 7.1, 7.2, 7.3). Although the top projects from 2005 were not necessarily superior to the top projects from the previous year, the quality of the

average projects had greatly increased. Furthermore, although some games did not meet the final minimum requirements for this project, all the game projects did have a functional user interface.

Although the student of Winter 2005 COMP-361 were free to use the tool of their choice to make their game, a high percentage of the groups chose to use Minueto. Very early into the projects, students sent questions and comments by email to the Minueto development team. Although the development team could not accommodate every feature requested by the students, the team was very helpful in explaining how students could modify Minueto so it would better fit their needs. In addition, several students sent emails to Minueto's developers, thanking them for the large amounts and the quality of the documentation that was available.

Chapter 8
Learning from 2004 and 2005

8.1 General Conclusions

Minueto was successfully tested by the students of the Winter 2005 COMP-361 Systems Development Project class. In addition, it was used by Marc Lanctot to develop a data gathering tool/game for his research [Lan05]. Furthermore, at least two other games have been implemented using Minueto.

Minueto was originally designed to reduce the learning curve of game development and allow students to focus on the behavior of the game. The following sections summarize Minueto's impact on COMP-361 and offer some suggestions on improving the course. Also, Minueto's future is discussed; topics include change policies, the role of expansion modules and future implementations of Minueto.

8.2 Impact Analyst

As previously mentioned in chapter 4, students from the 2004 course were asked what were the most difficult tasks for the project (see table 4.1). Two thirds of the class answered that most problems were related to project design and project management. Given that the project is designed to evaluate software engineering and team-work skills, this result was expected and desired. The other third of the class answered

GUI development as their most difficult task. This indicates a flaw in the design of the course, since students should focus on the behavior of the game, not the interface.

Most difficult task	Number of teams
Implementing the submarine unit	1
Network programming	3
Teamwork and code mergers	2
Building the GUI and programming the graphics	2

Table 8.1: Student feedback on most difficult task during the Winter 2005 project

The same question was asked to the students of the 2005 class (see table 8.1). This year, network programming was considered one of the most difficult tasks of the project. This is surprising since the network code required for Strategic Conquest should have been fairly similar to the code required for last year's Naval Battle project. However, inter-application communication is one of the course goals. As such, it is normal that the design of the network component for the game be a challenging task.

Teams were also asked which development tools they used for their project. Apart from the traditional programming IDE tools such as Visual Studio and Eclipse, there was an impressive increase in the number of teams that used a source revision control system such as Subversion. Some teams also used a bug tracking system or a Wiki to simplify communication between team members. This increase in collaborative tools could explain the why so few 2005 groups reported problems with teamwork and merging their code.

Only two groups answered that graphic programming was the most difficult part of the project. This represents an important improvement over last year's results. However, a deeper analysis is required to determine if the difficulties were caused by the tool they used, or by the complexity of their GUI.

One of these two teams decided not to use Minueto. Instead, they tried to create a Minueto/Swing hybrid where Minueto-like components could be used to draw images and Swing components could be used for the GUI interface. The team experienced

several problems, many of them similar to those experienced by Minueto's developers early in the project. They were unable to complete this Minueto/Swing engine because of lack of time and resources. Although they did have a working prototype, their engine was a bit unstable and functioned only under the Windows operating system.

The other team created an artistic marvel that showcased Minueto's possibilities. It is easy to understand that graphics was their most difficult task, given the shear complexity of their animation system. Although their problem is not caused by Minueto, better infrastructure (such as tutorials or expansion modules) could be created to help students build and manage complex animation systems.

8.3 Course Improvements

Although the students of the 2005 class were all successful in implementing Strategic Conquest, the final testing of the game revealed that focus should be added on two software engineering topics : user interface design and usability testing.

After playing each team's implementation of the game for twenty minutes, it was obvious that several teams had not properly tested or played their games. Although the game themselves were not flawed, their user interface made the game-play slow and unintuitive. Some teams had complex menu systems where several clicks were required to accomplish basic tasks. Other teams required players to know a large amount of keyboard shortcuts, without any built-in reminder/help system in the game.

Many of these problems could have been avoided if user interface design and usability testing had been taught in class.

8.3.1 User Interface Design

An application must provide means for users to interact in ways that are intuitive and natural [Woo97]. Although building the user interface for a game can be a daunting experience, it is important that the interface be designed from a player's perspective,

not a software engineer's. Students should not forget that the video game industry has decades of experience with user interfaces that work. For many types of games, there are traditional user interfaces that players have come to expect.

Even if each game requires a distinct user interface, students should analyze the user interfaces used by similar games. Strategy games such as Warcraft 3, Ages of Empires, Total Annihilation and Red Alert all have similar user interfaces that would be well adapted for Strategic Conquest. Although the students should not feel constrained to build their game using the same kind of interface, they should be aware that variations from a proven user interface require additional usability testing. A one hour course on the different user interfaces of video games would be a valuable asset to the students.

8.3.2 Usability Testing

Steve Krug's first law of usability is "Don't make me think!" [Kru00]. He believes that a well design interface should be self-evident to a user. Although this goal might seem unrealistic for a video game, many video games can be played without having to read the game manual. The usability of a game can be evaluated through usability testing.

The people working on a project are usually worst qualified to evaluate the usability of their software. Their intricate knowledge of the project prevents them from determining if their application is self-evident. Ideally, usability testing is done using a small random group of people with no prior knowledge of the application. Students can usually achieve this by asking a group of their friends to play the game. Students can then observe how the testers use their application and learn how to improve it. Proper usability testing (both the interactions with the testers and interpreting the results) can be tricky and requires some training. Students would benefit greatly from a one hour course on usability testing.

8.4 Minueto's Evolution

Minueto is a game development architecture that is currently implemented in Java. Given that the Minueto name is used to describe both the platform independent architecture and the Java implementation, discussion about changes in Minueto can be confusing. Because there are currently no plans to port Minueto to another programming language, changes and improvements on the architecture are directly carried to the Java implementation. However, the roles and policies described in the following sections should be interpreted at the architecture level.

In the following sections, particular emphasis is placed on the difference between Minueto's core component (which only handles keyboard/mouse input and drawing on the screen) and the expansion modules (that provide all other functionalities). These two components have different roles, and thus, two very different design philosophies.

8.4.1 Change Policies for the Core Component

If compared to other game development frameworks, Minueto's core component has a surprisingly small number of features. This design choice is motived by the desire to keep the framework as simple as possible. Thus, students are able to learn Minueto faster given the relatively small size of the API.

Such a design philosophy introduces some interesting design challenges: new features must add functionality while minimizing the increase in complexity of the framework. The challenge lies in determining which changes are worth the increase in complexity.

Minueto is not a game specific framework. Its core components focus on gathering input and displaying output. As such, changes that apply to specific types of games should be avoided. One common change that is requested by the students is the addition of a Sprite class. However, sprites are game specific since their implementation varies from one type of game to another.

Before adding new features to Minueto's core component, the following questions must be considered:

- Is this change specific to a particular type of game?
- Is the change specific to a particular implementation?
- Do the changes affect components other than the core?
- Will this change benefit a specific group of users?
- Can the new functionalities be achieved with the existing functionalities?
- Will this change affect existing Minueto applications?
- Will this change decrease Minueto's performance?

A "yes" answer to any of these questions indicates that the change should not be implemented on Minueto's core component.

8.4.2 The Role of Expansion Modules

Since Minueto's expansion modules provide all other functionalities beyond what is provided by the core component, they have a crucial role in Minueto's future. As we have already mentioned, extreme measures are taken to keep the core component as simple as possible. However, the expansion modules are bound by no such limitations; they can be very complex or game specific. They are designed to simplify tasks commonly found in specific types of games. Section 5.4 describes several components which could greatly extend Minueto's functionalities.

Expansion modules are never bundled with the core SDK. As such, they do not affect Minueto's initial learning curve since they need be installed separately. Expansion modules should be available as a separate download, although bundling several them into an expansion pack is acceptable. Installation of expansion modules should be simple and, ideally, similar to the installation of the core component.

8.4.3 The Next Minueto

Most of the changes required in Minueto can be found in Section 6.4. Some of these changes occur at the implementation level, like removing the singleton restriction on the MinuetoWindow class. Other changes, such as the much needed modification to MinuetoKeyboardHandler, affect Minueto at the architecture level.

However, the most important new feature is the addition of efficient and reusable GUI components (buttons, text boxes, etc.). This could be achieved by either integrating Swing components or building the components with Minueto objects. The latter solution would be preferable because it would be implementation independent. This feature would greatly improve student productivity since students spend a lot of time coding these components themselves.

A non-essential, but useful improvement, would be to automate the regression testing infrastructure. Manually running all the regression tests requires several hours, especially when running them on multiple virtual machines and on multiple platforms.

8.5 Looking to the Future

During Summer 2005, Minueto 1.0 will be released to the Java Game Development community. It will also be announced on various open-source software news sites such as Freshmeat.net. At the same time, developers will be working on the second version of Minueto, addressing concerns outlined in the previous section. Minueto will also be introduced to the academic community as a game development learning tool for Java-centric environments. It is our hope that Minueto contributes to both the game development community and the Computer Science undergraduate community, making game development more accessible.

Appendix A
Minueto SDK Readme

The following is the content of the Readme file provided with the Minueto software development kit.

A.1 Minueto Readme

Minueto - The Game Development Framework
Copyright (c) McGill University, 2004. All right reserved.
3480 University Street, Montreal, Quebec H3A 2A7

Minueto is a multi-platform 2D Graphic API which is easy to learn and use. Minueto also addresses other game programming concerns such as Input from Keyboard/Mouse..

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

Minueto is available in two different packages. The current version of Minueto is 0.6.0.

- MinuetoSDK_0.6.0_jdk1.4.2.zip : The Minueto software development kit (examples, source, doc) with the compiled runtimes for JDK 1.4.2.
- MinuetoSDK_0.6_jdk1.5.zip : The Minueto software development kit (examples, source, doc) with the compiled runtimes for JDK 1.5.

A.1.1 System Requirement

- Minueto has been tested using JDK 1.4.2 and JDK 5.0 (1.5).
- Minueto runs on the Windows XP, MacOs X and Linux platform.
- A 500 Mhz machine is sufficient for development purposes.
- A 1 Ghz machine is required to run a game that requires a fast framerate.

A.1.2 Installation

Download the Minueto SDK (either the 1.4.2 or 1.5 version, depending on which JDK version you use). Unzip the Minueto package using your favorite zip program. On Linux, you can use the following command :

```
unzip MinuetoSDK\_0.6.0\_jdk1.4.2.zip
```

Once you unzip the full package, you'll find the following directory structure.

```
 |- readme.txt       : This file
 |- license.txt      : Minueto's licence (LGPL)
 |- changelog.txt    : Changes in Minueto
```

```
|- todo.txt          : Things I should do in the near futur
|- build.xml         : Allows you to compile Minueto
|---\ lib            : Minueto's runtime jar
|---\ api            : Minueto's api documentation
|---\ samples        : Multiple examples to get you started
|---\ Minueto        : Minueto's source
```

If you want to start developping with Minueto, don't forget to add Minueto.jar in the lib directory to your classpath. You can find more information on changing your classpath at :

```
http://www.dynamic-apps.com/tutorials/classpath.jsp
```

A.1.3 Samples

You will find in the sample directory several examples on how to use Minueto. Each example has a make.sh and a make.bat shellscript that you can use to build and run the sample. Please note that both scripts expect the javac and java command to be in your path.

A.1.4 Documentation

The API to use Minueto can be found in the API directory. You can also find a lot of information on Minueto at

```
http://www.cs.mcgill.ca/~adenau/minueto/
```

A.1.5 Compiling

You can compile a new Minueto.jar runtime using Ant. From the sdk root directory:

```
ant all-jdk14
```

recompiles Minueto for use with JDK 1.4.2.

```
ant all-jdk15
```

will recompile Minueto for use with JDK 1.5.

A.1.6 Contact

Minueto is a project of the McGill Software Engineering Lab.

```
<http://www.cs.mcgill.ca/~joerg/sel/sel.html>
```

Questions related to Minueto should be directed to Alexandre Denault.

```
<alexandre.denault@adinfo.qc.ca>
```

Appendix B
Minueto Code Samples

The following appendix contains a description of the samples provided with the Minueto software development kit. Since samples are also used for regression testing, it can be assumed that a sample used to demonstrate a specific feature should also be used to test that same feature.

B.1 CircleDemo

This sample demonstrates how to build circle objects. The circles have different sizes and can be either empty or filled.

B.2 CropDemo

This sample demonstrates how the crop function can be used to create new images by copying a portion of an existing image.

B.3 FramerateDemo

This sample demonstrates how timers can be used to measure the frame rate of a MinuetoWindow.

B.4 HandlerDemo

This sample demonstrates how input from the keyboard and mouse can be captured. Captured events are printed to the console.

B.5 HandlerDemo2

This sample also demonstrates how input from the keyboard and mouse can be captured. However, this sample allows the user to hide/show the cursor and enable/disable the MouseHandler. Also, a box and a large X is drawn in the window to determine if Minueto is correctly calculating the window margins.

B.6 HandlerDemo3

This sample also demonstrates how input from the keyboard and mouse can be captured. However, special focus is given to the mouse move event. A cross is drawn at the location of the cursor and moves when the cursor is moved.

B.7 Helloworld

This sample demonstrates how text can be drawn in the exact center of the screen.

B.8 ImageDemo

This sample demonstrates how several small images can be composited together to form a large, more complex image.

B.9 LineBenchmark

This sample calculates how many lines Minueto can draw on screen while maintaining an average framerate of 30 fps. This sample can be used to benchmark the implementation of Java's internal drawing functions.

B.10 LineDemo

This sample demonstrates how several lines can be drawn over a blank MinuetoImage.

B.11 LoadingFileDemo

This sample demonstrates how an image can be loaded from disk and drawn on screen.

B.12 RectangleDemo

This sample demonstrates how to build rectangle objects. The rectangles have different sizes and can be either empty or filled.

B.13 RotateDemo

This sample demonstrates how an image can be loaded from disk and continually rotated as it is drawn on screen.

B.14 RotateDemo2

This sample demonstrates that the center of rotation of an image is in the center of that image. This sample was written to test if images were properly rotated and centered.

B.15 RotateDemo3

This sample demonstrates that the size of an image might increase when rotated. This sample was written to test if the size of the image was properly increased when it was rotated.

B.16 ScaleDemo

This sample demonstrates how an image can be loaded from disk and continually scaled as it is drawn on screen.

B.17 ScaleFlipDemo

This sample demonstrates the flip function, which is internally a subcase of the scale function.

B.18 TextDemo

This sample demonstrates the difference between normal text and anti-aliased text. It is also a good example to learn how to draw text.

B.19 TextDemo2

This sample demonstrates how text input can be handled. Although simple, this example provides a good starting block to implement a text box.

B.20 TriangleRover

This sample demonstrates how to implement 2D movement. This sample also demonstrates a work-around for the keyboard repeat bug under Linux.

B.21 TriangleRover2

This sample demonstrates how simple it is to improve the visuals of the first TriangleRover sample by loading images.

Appendix C
Minueto Tutorials

The following appendix contains a description of the tutorials (also called Howtos) found on the Minueto website.

C.1 How do I run the samples?

The Minueto SDK package contains a collection of over 20 samples. This tutorial teaches a student how to use the provided make scripts to compile and run these samples.

C.2 How do I compile and run an application with Minueto?

To compile an application that uses the Minueto framework, the Minueto.jar library file must be declared in the Java Classpath. This tutorial teaches a student various ways Minueto.jar can be inserted into the Classpath.

C.3 How do I create a game Window?

In Minueto, the Window is the canvas where images are drawn. This tutorial teaches the student how to create a window and how to display it.

C.4 How do I load and draw an Image?

An image is a basic construction block of a Minueto application. This tutorial teaches the student how to create a MinuetoImage from an existing file and how to draw that image on the screen.

C.5 How do I draw a line?

In Minueto, lines can be arbitrarily drawn over any image. This tutorial teaches the student an efficient technique for using the drawLine method.

C.6 How do I draw a rectangle or a circle?

With Minueto, a new MinuetoImage can be created with the image of a rectangle or a circle. This tutorial teaches the student how to create these images and how to draw them.

C.7 How do I write text on an image or the screen?

To write a string on the MinuetoWindow, an image of that string must first be created. This tutorial teaches the student how to create the image of a string and how to display it on the screen.

C.8 How do I get input from the keyboard?

Minueto uses a queue-based messaging system with event handlers. This tutorial teaches the student how to implement the keyboard event handler interface and how to register that implementation with the event queue.

C.9 How do I get input from the mouse?

Capturing input from the mouse is fairly similar to capturing input from the keyboard. This tutorial teaches the student how to implement the necessary interface and how to register it.

C.10 How do I measure time using Minueto?

Minueto application can measure time using a provided timer class. This tutorial teaches the students how timers can be used to measure the time required to draw lines on the backbuffer.

C.11 How do I scale/rotate an image?

MinuetoImages can be scaled or rotated using pre-defined methods. This tutorial teaches the student how to cache the results of a rotated image and how to smoothly rotate an image.

Appendix D
Glossary

- 2D : 2 Dimensions : Defines a 2 dimensional environment composed of height and width.
- 3D : 3 Dimensions : Defines a 3 dimensional environment composed of height, width and depth.
- AI : Artificial Intelligence : Intelligence exhibited by any manufactured (i.e. artificial) system.
- API : Application Programming Interface : Set of rules (functions) that define how a piece of software can be used by other softwares.
- AWT : Abstract Windowing Toolkit : Java's platform-independent toolkit for graphic user environments (GUI).
- CORBA : Common Object Request Broker Architecture : Standard that allows components/services written in different languages/platforms to interact.
- CPU : Central Processing Unit : Central unit of a computer, carries out all the computations.
- FPS : Frames per second : Unit of measure, represents the number of updates sent to the monitor every second.

- Ghz : Giga Hertz : Unit of measure, represents 1,000,000 hertz (Hz)
- GIF : Graphics Interchange Format : Bitmap image format, allows for transparencies, color palettes and animation.
- GNU : GNU's Not Unix : Recursive acronym, software license for free software.
- GUI : Graphical User Interface : Interface to interact with computers thought the use of icons and images.
- Hz : Hertz : International unit of measure for frequency, represents number of events occurring per second.
- IDE : Integrated Development Environment : Complete suite of software tools that allows a programmer to develop new applications.
- JDK : Java Development Kit : Collection of tools distributed by Sun Microsystems to allow programmers to create Java applications.
- JPEG : JPEG File Interchange Format : Image format for photographic images that uses a lossy compression format.
- JVM : Java Virtual Machine : Software or hardware machine that runs Java code.
- KDE : K Desktop Environment : Desktop environment for Unix and unix-like systems.
- MIDI : Musical Instrument Digital Interface : Communication standard for musical notes that allow music instruments and computers to exchange information.
- MMOG : Massively Multiplayer Online Game : Video/computer game with several thousands of players sharing a persistent state environment.
- MP3 : MPEG-1 Audio Layer 3 : Encoding format for music using a lossy compression format.

- NPC : Non-Player Character : Character in a game whose actions are not controlled by a player.
- OGG : Ogg : Open source multimedia container format for video and audio.
- PNG : Portable Network Graphics : Open source bitmap image format, created to replace GIF (which requires a patent license to use).
- RGB : Red, Green, and Blue : Color model where red, green and blue light can be combined to create new colors.
- RGBA : Red, Green, Blue and Alpha : Color model similar to RGB, but adds an Alpha component that determines how transparent (ghost-like) a color is.
- RMI : Remote Method Invocation : Java programming interface that allows programmers to call methods on remote objects. Similar to CORBA.
- SDK : Software Development Kit : Domain specific collection of tools that allow programmers to create software for that domain.
- TCP/IP : Transmission Control Protocol / Internet Protocol : Communication protocol that allow two computers to interchange data. TCP/IP is the main protocol used by the Internet.
- X11 : X Window System : Window system for Unix and unix-like operating system. Provides basic windowing mechanism and is further enhanced by desktop environments such as KDE.
- XML : Extensible Markup Language : Flexible language that allows the definition of a special purpose language for long-term storage or transmission of data.

Bibliography

[AL00] James H. Andrews and Hanan Lutfiyya. Experience report: A software maintenance project course. *Thirteenth Conference on Software Engineering Education and Training*, page 8, March 2000.

[AM03] Maria Isabel Alfonso and Francisco Mora. Learning software engineering with group work. *Sixteenth Conference on Software Engineering Education and Training*, March 2003.

[ARA04] Admissions, Recruitment, and Registrar's Office (ARR). *Undergraduate Course Calendar*. McGill University, 2004.

[Chu99] Doug Church. Formal abstract design tools, game developer. *http://www.gamasutra.com/features/19990716/design_tools_01.htm*, 1999.

[DHZS00] Birgit Demuth, Heinrich Hussmann, Steffen Zschaler, and Lothar Schmitz. A framework-based approach to teaching oot: Aims, implementation, and experience report: A software maintenance project course. *Thirteenth Conference on Software Engineering Education and Training*, page 11, March 2000.

[DS03] Crespo Dalmau Daniel Sanchez. *Core Techniques and Algorithms in Game Programming*. New Riders, 2003.

[ETC05] CM Entertainment Technology Center. Current projects. *http://www.etc.cmu.edu/project_subpages/projects_listing.html*, 2005.

[FBK+01] Theodore Frick, Elizabeth Boling, Kyong-Jee Kim, Daniel Oswald, and Todd Zazelenchuk. Software developers' attitudes toward user-centered design. *24th National Convention of the Association for Educational Communications and Technology*, November 2001.

[fCT05] Institute for Creative Technologies. Ict games project. *http://www.ict.usc.edu/disp.php?bd=proj_games*, 2005.

[FvDFH97] Foley, van Dam, Feiner, and Hughes. *Computer Graphics Principles and Practice, Second Edition in C.* Addison Wesley, 1997.

[GKPS03] Michael Gnatz, Leonid Kof, Franz Prilmeier, and Tilman Seifert. A practical approach of teaching software engineering. *Sixteenth Conference on Software Engineering Education and Training*, page 9, March 2003.

[GS02] Garner and Stuart. Reducing the cognitive load on novice programmers. *Association for the Advancement of Computing in Education (AACE)*, page 7, June 2002.

[Haa04a] Chet Haase. Buffered image as good as butter. *http://weblogs.java.net/blog/chet/archive/2003/08/bufferedimage_a.html*, 2004.

[Haa04b] Chet Haase. Buffered image as good as butter, part ii. *http://weblogs.java.net/blog/chet/archive/2003/08/bufferedimage_a_1.html*, 2004.

[HK73] Simon H. and Gilmartin K. A simulation of memory for chess positions. *Cognitive Psychology*, 1973.

[IH02] Toru Iiyoshi and Michael J. Hannafin. Cognitive tools and user-centered learning environments: Rethinking tools, functions, and applications. *Association for the Advancement of Computing in Education*, June 2002.

[III00] Richard Rouse III. Designing design tools, gamasutra. *http://www.gamasutra.com/features/20000323/rouse_pfv.htm*, 2000.

[Kre02] Bernd Kreimeier. The case for game design patterns, gamasutra. *http://www.gamasutra.com/features/20020313/kreimeier_pfv.htm*, March 2002.

[Kru00] Steve Krug. *Don't Make Me Think: A Common Sense Approach to Web Usability*. New Riders Press, 2000.

[Lan05] Marc Lanctot. Adaptive virtual environments in modern multi-player computer games. Master's thesis, McGill University, 2005.

[Mic99] Sun Microsystems. Java code conventions. *http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html*, 1999.

[Mic00] Sun Microsystems. How to write doc comments for the javadoc tool. *http://java.sun.com/j2se/javadoc/writingdoccomments/*, 2000.

[Moo05] Lynn Moore. Ubisoft expansion to create 1,000 jobs, quebec offers $22 million intax credits grants. *The Gazette, Montreal*, February 2005.

[oAGG05] The University of Alberta GAMES Group. Scriptease, a scripting language for computer role-playing games. *http://www.cs.ualberta.ca/ script/*, 2005.

[oCS05] McGill School of Computer Science. Undergraduate academic programs. *http://www.cs.mcgill.ca/undergraduate/programs.html*, 2005.

[Pau04] Randy Pausch. An academic's field guide to electronic art, observations based on a residency in the spring semester of 2004. *Carnegie Mellon University*, 2004.

[PW02] Lothar Pantel and Lars Wolf. On the suitability of dead reckoning schemes for games. *First Workshop on Network and System Support for Games (NetGames2002)*, 2002.

[Ruc03] Rudy Rucker. *Software Engineering and Computer Games.* Addison Wesley, 2003.

[SW04] Scott Schaefer and Joe Warren. Teaching computer game design and construction. *Computer-Aided Design*, 36(14), December 2004.

[W.94] Caroll W. Using worked examples as an instructional support in the algebra classroom. *Journal of Education*, 1994.

[Wal02] Mark H. Walker. Strategy gaming: Part v – real-time vs. turn-based, gamespy. *http://archive.gamespy.com/articles/february02/strategygames05/*, February 2002.

[Woo97] Larry E. Wood. *User Interface Design: Bridging the Gap from User Requirements to Design.* CRC Press, 1997.