Raptor: Sketching Video Games With a Tabletop Computer

by

John David Smith

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
August 2009

Copyright © John David Smith, 2009
Abstract

Game sketching is used to identify enjoyable designs for digital games without the expense of fully implementing them. This thesis presents Raptor, a novel tool for sketching games. Raptor shows how tabletop interaction can effectively support the ideation phase of interaction design by permitting small collocated groups to participate in the design and testing process together. Raptor relies heavily on efficient gesture-based interaction, mixed-reality interaction involving physical props and digital artifacts, Wizard-of-Oz demonstration gameplay sketching, and fluid change of roles between designer and tester. An evaluation of Raptor using seven groups of three people showed that a sketching tool based on a tabletop computer indeed supports ideation and collaboration among collocated groups better than a more traditional PC-based tool.
Acknowledgments

I would first like to thank my supervisor, Dr. T.C. Nicholas Graham for his guidance, patience, and support throughout the course of my Ph.D. Nick gave me the freedom to pursue my degree in my own way, and was always there when I needed a helping hand. For the past four years, Nick has been the biggest supporter of my work. I would also like to thank all my labmates in the EQUIS Group for keeping the lab a fun, productive place to work. And finally I wish to thank my family for their love and support. In particular, I would like to thank my wife, Karina. It was her love that saw me through to the end of this project, and I am forever indebted to her.
Contents

Abstract i

Acknowledgments ii

List of Tables vi

List of Figures vii

Chapter 1 Introduction 1
  1.1 Motivation .............................................. 2
  1.2 Tabletop Computers ................................. 4
  1.3 Problem Statement ................................. 5
  1.4 Contributions ........................................ 6
  1.5 Organization ........................................ 6

Chapter 2 Video Game Sketching 7
  2.1 The Game Development Process ..................... 7
    2.1.1 Pre-production .................................. 8
    2.1.2 Development .................................... 9
    2.1.3 Post-Production ................................ 10
  2.2 Sketching User Interfaces ......................... 11
    2.2.1 Sketching vs. Prototyping ...................... 12
    2.2.2 Sketching Techniques ......................... 13
  2.3 Sketching Games ..................................... 17
    2.3.1 Sketching Games with Paper .................... 18
    2.3.2 Programming Tools .............................. 20
    2.3.3 Scene Design Tools .............................. 31
  2.4 Conclusions ........................................ 37

Chapter 3 Tabletop Computers 39
  3.1 Creating Tabletop Applications .................... 40
    3.1.1 Designing for Collaboration ................... 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5</td>
<td>Data and Statistical Analysis</td>
<td>93</td>
</tr>
<tr>
<td>6.6</td>
<td>Results</td>
<td>94</td>
</tr>
<tr>
<td>6.7</td>
<td>Discussion and Conclusions</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>Chapter 7 Summary and Conclusion</td>
<td>98</td>
</tr>
<tr>
<td>7.1</td>
<td>Summary of Contributions</td>
<td>99</td>
</tr>
<tr>
<td>7.2</td>
<td>Limitations</td>
<td>100</td>
</tr>
<tr>
<td>7.3</td>
<td>Future Work</td>
<td>100</td>
</tr>
<tr>
<td>7.4</td>
<td>Conclusion</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>103</td>
</tr>
</tbody>
</table>
List of Tables

6.1 Likert Item Responses .............................................. 94
6.2 Analysis of Likert Item Responses ............................... 95
List of Figures

1.1 Raptor ................................................. 3
2.1 Paper Prototyping ..................................... 14
2.2 Mock-up Prototyping .................................. 16
2.3 Game Sketching With a Desktop Computer ............ 17
2.4 GABIT .................................................. 22
2.5 UnrealScript ............................................. 25
2.6 Alice ..................................................... 27
2.7 Virtools ............................................... 30
2.8 Sketch3D ............................................... 33
2.9 3DPalette ............................................... 35
2.10 Declarative Modeling ................................ 37
3.1 A Tabletop Computer .................................. 40
3.2 Magic Land .............................................. 44
3.3 MicroMotoCross ........................................ 46
4.1 Molding Terrains ...................................... 53
4.2 Stamping Content ...................................... 54
4.3 Adding Input .......................................... 55
4.4 Collaboration With Mixed-Roles ....................... 57
4.5 Wizard-of-Oz Prototyping .............................. 58
4.6 Interactive Object Code Sample ....................... 63
4.7 Role-Playing Game .................................... 65
4.8 First-Person Shooter .................................. 68
5.1 Software Conceptual Architecture ..................... 78
5.2 Microsoft Surface ...................................... 82
5.3 Microsoft Surface Hardware Architecture ............. 83
5.4 Microsoft Surface Dominos ........................... 84
6.1 Desktop Condition ...................................... 87
6.2 Desktop Camera and Controller ....................... 88
Chapter 1

Introduction

Sketching is an emerging concept in user interface development that supports the early ideation phase of interaction design [27]. Ideation is the process of generating, developing, and communicating ideas that shape the user experience of an interactive system [23, 64]. When sketching, designers create rough interactive experiences that illustrate the main features of an interactive system. Iterative design and collaboration with end users has been shown to be beneficial when building interactive systems [39], and sketching has shown to be a useful component of the iterative design process [69]. Sketching is an inherently collaborative process, where designers, end-users, and developers can all work together to develop and communicate design concepts. Sketches are not intended to provide a reusable piece of software. Instead, they help explore user experience without dwelling on the details of implementation.

This thesis presents Raptor: a novel sketching tool for video games (Figure 1.1). Raptor addresses the problem that games now cost tens of millions of dollars to build, and involve teams often in excess of 100 people [86]. Given these costs, it becomes critically important to assess early in the development process whether the game will
actually be fun to play. Sketching provides a technique for rapidly enabling players to experience game ideas without creating fully functioning prototypes.

A key component of Raptor is a tabletop computer. Tabletop computers consist primarily of a large touch-screen display that covers the surface of a table, around which multiple users can work at the same time (Figure 1.1). A key difference between tabletop and desktop computers is support for groups of users. With a tabletop computer, multiple users can interact with the computer at one time, whereas a desktop computer typically supports input from a single keyboard and mouse.

This thesis presents an end-user study that examines the role the tabletop computer plays in the effectiveness of Raptor as a game sketching tool. I asked groups of users to perform a simple game design task using both a tabletop and desktop version of Raptor. I measured user feedback for both approaches and concluded that Raptor does in fact support collaboration among groups of users for game sketching and the tabletop computer provides a significantly better user experience.

This chapter begins by motivating video game sketching research and introducing tabletop computers. A problem statement and specific contributions of this work are then presented. The chapter concludes with the organization of the thesis.

1.1 Motivation

Over the past 10 years, the interactive entertainment industry has grown into a multi-billion dollar industry [86]. Games are a hit-driven medium, where the vast majority of titles lose money. According to the President of Nintendo of America, only 16 of 486 released titles (just over 3%) for the Wii game console have been profitable to date [97]. It is therefore critical to find ways of determining the fun of game designs
Figure 1.1: *Raptor* is a game sketching tool based on a tabletop surface. The tool helps groups of users to brainstorm early in the game design process.

before they are implemented at the cost of tens of millions of dollars [63].

Today, the development process for video games is heavily driven by gaming technologies. Participants in the process fall into three roles: programmers, artists, and designers. The tools that support this process are a collection of single-user desktop applications. Artists use robust 3D modeling environments to create virtual worlds, and programmers use integrated development environments (IDEs) to create computer code that applies behaviors to those worlds. Designers primarily use documents to describe design elements. Collaboration occurs primarily when participants exchange artifacts. For example, a designer may give a design document to the programming team to discuss the technical feasibility of implementing the design. Alternatively, the programming team may give the artists a running application
built on the game’s graphical components so the artists can tune their 3D models for optimum performance.

Because collaboration is performed in a “hand-off” fashion, games tend to be developed with a waterfall model. The design of the game is defined and at the beginning of the process with documents and playable prototypes, given to the artists and programmers to implement, then finally tested after the game is mostly built [14]. Therefore, it is not determined whether a game is actually “fun” until near the end of development, and fixing design problems can be costly.

Sketching is an ideal way of helping with this problem. Sketches permit rapid, low-fidelity testing of game ideas. Because sketching is fast and inexpensive, there is little resistance to throwing ideas away that do not appear to work. The most promising ideas generated from game sketching can then be carried forward for higher-fidelity prototyping.

1.2 Tabletop Computers

When designing Raptor, our principle goal was to demonstrate how a tabletop computer could support collaborative game sketching. Compared to desktop computers, tabletop computers provide a user interface that more closely mimics the way people interact with everyday physical objects. For example, mTable is a tabletop photo browser that allows users to manipulate a pile of digital photos with their hands [34]. People use the same natural gestures used to handle paper photos, such as dragging photos across the surface, rotating photos with two fingers, or tossing them from one side of the table to the other. Academic research on tabletop computers has focused on better understanding the fundamentals of surface-based application design.
However the technology used to implement tabletop computer systems has recently become inexpensive enough to create commercial tabletop computers [73, 119], and a need exists for identification and understanding of potential application domains.

This work demonstrates that a game sketching tool based on a tabletop computer improves over existing tools [5] by providing:

- *Tool transparency*, where sketches can be created and manipulated using gestures and physical props rather than pointing and clicking;

- *Engaging collocated collaboration*, where the physical layout of a table allows intimate communication, and where the table’s multi-touch input allows more than one person to interact with the sketch at a time;

- *Egalitarian design process*, where people without programming skills are not disadvantaged, and where roles can be fluidly set and changed.

### 1.3 Problem Statement

This work addresses two problems with current game sketching tools. First, today’s game sketching tools require domain-specific technical knowledge to operate. This provides a significant barrier of entry for those without technical training. Second, today’s tools are designed primarily for single users. Most tools are based on desktop computer interaction with a keyboard and mouse and do not support multiple concurrent users well. This work addresses these two problems by providing a collaborative user experience that requires no prior technical skills to sketch video games.
1.4 Contributions

This work advances the state-of-the-art in the following ways:

• I present a novel game sketching tool based on a tabletop computer. By using a tabletop computer, the tool naturally supports collaboration and provides a rapid means of developing game sketches. The tool illustrates a collection of interactions based on emerging trends in human-computer interaction research that provide a logical mapping between tabletop computing and game sketching.

• *Raptor* illustrates a novel, practical application of table computers. Tabletop computers have shown promise in the research lab, but applications for the technology have been thus far lacking.

• Tabletop computers are compared directly to desktop computers. While tabletop computers show promise as an emerging platform, little work has been done to compare them directly to desktop systems.

1.5 Organization

Chapter 2 discusses the current state-of-the-art in sketching tools and techniques. Chapter 3 introduces tabletop computers and presents a survey of their design and emerging applications. Chapter 4 describes *Raptor*: my new video game sketching tool based on a tabletop computer. Chapter 5 describes the implementation on Raptor. Chapter 6 presents a study performed to evaluate the usefulness of *Raptor* as compared to a desktop computer-based approach. The study is described in detail and results are presented and analyzed. Finally, Chapter 7 concludes the thesis.
Chapter 2

Video Game Sketching

Video game sketching is an emerging research topic in human-computer interaction. The term was first introduced by Agustin et al. [5] and illustrated with a tool for sketching linear narrative games. This chapter provides a context for game sketching within the disciplines of video game development and more general interactive system sketching. The chapter begins by describing the game development process and the tools used to support it. A discussion on sketching and the role it plays in interactive system design is then presented. A survey of game sketching tools and techniques follows, and the chapter concludes with a discussion on open problems in game sketching.

2.1 The Game Development Process

Game development is a challenging specialization of interactive system development [14]. By their nature, games are designed for entertainment. Entertainment value is difficult to quantify, which makes comparing the quality of different game designs
troublesome. Given the cost of developing a commercial game, the amount of risk developers are willing to accept in game design is shrinking [5, 14, 55]. Consequently, today’s game development process is focused on the production of the game rather than the design of the game [14, 5]. The game development process is divided into three phases: pre-production, development, and post-production [32, 14].

2.1.1 Pre-production

The pre-production phase is the first stage of development. The pre-production phase seeks to answer two questions [32]: “What are you going to build?” and “How are you going to build it?” Game design requires the coordination of design from numerous domains such as visual arts, sound engineering, story-telling, localization, and software engineering. Each of these has its own particular design challenges. Interdependencies between these challenges can create difficult design trade-offs. For example, visual design specifications might conflict with technical constraints, and therefore make a seemingly coherent visual design impractical. The pre-production phase produces two primary artifacts:

- **Design Documents** describe the design of a game in pictures and prose [32, 14]. Design documents are well suited for game design such as aesthetics and narratives, but do not illustrate the interactive components of games such as control schemes and animated behaviors.

- **Prototypes** are playable illustrations of design concepts [5, 32, 14]. Playable prototypes are useful for illustrating game concepts dealing with interaction. However development resources must be consumed to construct them and the cost of creating large, complicated prototypes can be significant.
2.1.2 Development

The development phase is where the bulk of the game is actually constructed. Construction is done with the coordination of several groups including game designers, artists, software developers, story writers, sound engineers, and musicians. The two largest groups are software engineers and artists. The artists create all the game assets such as 3D environments and character models, and the developers build the running game implementation with programming environments such as Microsoft Visual Studio [77]. Today, most games are built with a common set of reusable components that are assembled using a standard architecture. Some typical components include:

- **Graphics Libraries** provide tools for efficiently and accurately drawing visual scenes. Two de facto standards have emerged: OpenGL [89] and DirectX [72].

- **Audio Libraries** provide 3D spatial audio rendering and sound effects. Examples include OpenAL [37] and DirectSound [72].

- **Scene Management Tools** provide data structures that allow efficient and accurate management of virtual worlds. OGRE is a popular open-source example [111].

- **Networking Libraries** enable multi-player games to communicate over local area networks and the internet. For example, the networking components included in Microsoft XNA [79] provide a library that enables communication through Xbox and Windows Live! [71].

- **Artificial Intelligence (AI)** manages behavior of non-playable characters. Havok AI is a good example [88].
• *Dynamics Engines* simulate physical aspects of the virtual world such as gravity, collisions, and inertia. PhysX [87] and Havok Physics [110] are two popular examples.

• *Content Management Tools* provide models of entities that are present in the virtual world. For example, the Content Pipeline included with Microsoft XNA [79] provides tools for organizing and validating game assets at design-time and a programming interface for loading content at runtime.

These components are often tied together into a single software component called a “game engine”. Game engines provide the basic underlying technologies necessary for creating a game, and provide a means of deploying games across multiple platforms. The process of game development is budgeted in large part based on the reuse of a game engine for the creation of multiple games [32, 14]. Game engines provide a quick way to get started on a game design, but the features to be included in the game are restricted to the capabilities of the engine. Game engines are particularly well suited for use when the design closely resembles an existing game, such as a new entry in a running series.

### 2.1.3 Post-Production

Traditionally, games have been considered complete after development and delivery. However, with the introduction of online-enabled games, new content and maintenance releases can be delivered after a game’s release. The scope of the post-production process depends heavily on the type of game created. For example, Guitar Hero [1] is a music game that generates revenue by releasing new content in coordination with events in the music industry. However games such as Final Fantasy [106],
which tell a story with a clear beginning and end, primarily use post-production to deliver bug fixes.

### 2.2 Sketching User Interfaces

Sketching is most relevant to the pre-production phase of game development [5, 14]. Sketching is a specialized form of prototyping: a common topic in human-computer interaction research. In human-computer interaction, prototyping is a design activity used to encourage creativity among designers [12]. Prototypes can help generate and share ideas, gather requirements and feedback from users, and compare design alternatives. Beaudouin-Lafon and Mackay describe prototyping techniques along four dimensions [12]:

- **Representation** describes the form of the prototype, such as paper or digital.
- **Precision** describes the fidelity of the prototype, such as informal and rough or robust and polished.
- **Interactivity** describes the degree to which a user can actually interact with the prototype, ranging from “watch only” to fully interactive.
- **Evolution** describes the life cycle of the prototype, ranging from throw-away to iterative, where elements of the prototype may be included in the final system.

It is important to note that prototyping can serve a different purpose in other disciplines. For example, in engineering disciplines a prototype might evaluate the feasibility of a technical process or implementation. These would serve more as the apparatus in an experiment where a system is examined under various operating
conditions which are compared along a set of predefined metrics. In human-computer interaction, prototyping is used more for discovery and generation of new ideas rather than evaluation of existing ideas [12, 69]. This role more closely resembles the role of prototyping in other creative disciplines such as graphic design.

2.2.1 Sketching vs. Prototyping

In the context of game development, two distinct types of prototypes exist:

- **Low-fidelity** prototypes, sometimes called “early design” [14, 5] prototypes, are synonymous with a game sketch and serve the same purpose of rapidly examining a design. These prototypes are created early in the pre-production phase of game development. These prototypes can be built rapidly at little cost, although the precision of these prototypes is low and they have a short life span (i.e. are throw-away).

- **High-fidelity** prototypes, sometimes called “final” prototypes or “vertical slices” [14, 32, 5], are used to demonstrate that a game can actually be built. These are intended to resemble a finished game as closely as possible. For example, a high-fidelity prototype might be a full playable game level from a first-person shooter. Final prototypes have high precision, digital representation, are interactive, and can be recycled later for the production phase.

In game development, high-fidelity prototypes are used for more than only design illustrations [32]. High-fidelity prototypes are used by game studios to market new game concepts to prospective publishers [14, 32]. A prototype must convince the publisher that a game design has the potential to sell, and that the studio is capable
of developing the game. Therefore game prototypes must include both design and engineering elements. “Game sketching” is a term developed deliberately to avoid reference to these prototypes [5]. Sketching refers to low-fidelity prototypes, and focuses solely on the creative process of designing game experiences.

2.2.2 Sketching Techniques

Sketching techniques fall into two categories [12]: online and offline. Online techniques are created digitally and operate using a computer whereas offline techniques do not.

Offline Sketching Techniques

Offline sketches provide an easy way to think through design issues without worrying about the complexities associated with creating executable software. They typically have a short evolution and are used early in the design process. Offline techniques have the following advantages:

- **Offline prototypes can be built and maintained quickly.** Even with the best high level tools, fully functional online prototypes can take much longer to create than offline tools. Furthermore, once online prototypes are constructed, making changes to them through an iterative design process can be difficult. Offline prototypes, however, can be constructed in minutes and hours rather than days and weeks. This speed of development allows for more rapid progression through iterations of the design process, which means the design can be developed more fully before moving to the construction phase.

- **Offline prototypes take the focus away from “polish”**. Because online prototypes bear a resemblance to a finished product, they draw comments from evaluators
that address the presentation of the prototype than on the design idea the prototype is trying to express.

- **Offline prototypes are less buggy.** Software prototypes, particularly those requiring programming, tend to be complex compared to offline prototypes. Given that prototypes are often constructed under heavy time constraints, they are usually not subjected to the same testing regimen as finished software and are therefore more likely to contain bugs which can interrupt design evaluations.

The fastest way to produce an offline prototype is with paper (Figure 2.1) [96, 12]. Paper sketches can include various supplies commonly found in an office setting such as post-it notes and transparencies [83]. These items can be used to approximate common elements found in desktop user interfaces. For example, a post-it note with a list written on it can approximate a pop-up menu [12]. Similarly, a 3D “mock-up” can include prototypes of both the hardware and software of an interactive system (Figure 2.2) [21]. These systems can be made out of many low-cost materials such as
cardboard or foam and can include the same paper-based software sketching elements as a paper prototype. These prototypes are most useful when creating prototypes of systems with non-desktop hardware elements such as wall displays, tabletop computers, or handheld devices. “Wizard-of-Oz” prototyping enables users to experience interacting with a real, responsive interactive system even before it exists [65]. In a Wizard-of-Oz prototype, the user interacts with the system as though it is fully autonomous. However, located elsewhere, a developer assumes the role of “Wizard” by watching the interaction and operating the system. The technique is particularly useful in situations where the behavior of the system is not yet well defined. In these cases, the developer can experiment with different ideas simply by altering his/her responses to user input. However the technique is limited to systems where rapid responses to user input is not critical.

**Online Sketching Tools**

Online sketching tools provide higher precision sketches than offline tools that operate on an actual computer [12]. These sketches are useful for cases where it would be difficult to reproduce system behavior with physical objects, or where the layout and interaction of the system must be defined in detail. Additionally, online prototypes may reveal problems with the design that were not apparent in less precise offline prototypes. Additionally, digital sketches are often more easily distributed to other team members, particularly those located in remote locations.

Online tools come in three varieties, separated by level of interactivity [12]. Non-interactive tools such as animations and drawing toolkits give designers the ability to show the appearance of the interface, but do not allow users to actually interact
Figure 2.2: Mock-up prototyping extends paper prototyping to include support for the system hardware. Here, a hand-held mobile device is prototyped. Image acquired from [12].

with the system. Animated tools such as Macromedia Director [2] provide allow designers to demonstrate user interface behavior as a playable movie, where interface components respond to events determined with a time line. Other systems such as Microsoft Visio [74] provide a simple drawing tool where interface components are drawn with a simple drag-and-drop interface. More sophisticated sketches can also be created using interface builders common to integrated development environments (IDEs) such as Microsoft Visual Studio [77]. These enable rich, interactive sketches to be created, but often require programming expertise and experience with the toolkit.
2.3 Sketching Games

I know of only one system that is explicitly designed for creating sketches of games on a computer [5]. The system is a desktop computer interface that roughly resembles a first-person shooter game. The tool supports synchronous collaboration by providing views of a shared 3D environment over a local area network. To create a sketch, designers place a collection of textured primitive shapes into the 3D scene and assign one designer to be the “dungeon master”, which is effectively the “Wizard” in a Wizard-of-Oz prototype. To evaluate the system, the authors re-created some existing games and found that this type of tool is capable of creating many of the same complex rules and behaviors found in modern commercial games. However, collaboration on design issues beyond playtests is done on a single PC with the design team sharing a single desktop computer. The system also lacks a means of mapping user input to automated behaviors in the game world. Its scope is therefore limited to sketches where immediate response to user input is not required.
2.3.1 Sketching Games with Paper

Paper sketching is useful for users to roughly experience playing a game. This technique has shown to be useful for constructing game mechanic prototypes that explore rules, particularly in multi-player games [49]. The technique is similar to creating board games, and can even include components borrowed from existing board games [99]. A key advantage with paper prototyping is support for groups of users. Desktop tools, such as the programming tools described in the following subsection, are designed for a single user sitting at a desktop computer, while paper prototypes can be constructed collaboratively while sitting around a table. Additionally, a paper prototype requires little effort to create. This frees designers to try more ideas. However, the technique is useful only for games that can be approximated on paper. Role-playing games, puzzle games, and strategy games are particularly well suited for paper prototyping, but the technique is not as useful for games with rapid action such as sports simulations, first-person shooters, and racing games.

As an example, Fullerton et al. present a paper prototype of the popular game Battleship [49]. A simple multi-player prototype of the game can be constructed with four pieces of paper in a matter of minutes. To begin, each player receives two pieces of paper and draws a 10 x 10 grid on each. One of the grids represents a players “ocean”, where she places her own ships, and her “target”, where she marks the coordinates she is attacked. Players can then use a pen to place the ships in their “ocean”, mark shots on their “target”, and play a complete competitive game. Changes to the design of the game (the introduction of more ships, for example), is simple to enact when using paper, whereas a high-fidelity prototype might require a complicated software change such as a modification of the code or configuration.
Paper Prototyping Pervasive Games

Research has been particularly active in paper sketching of pervasive games. These games are defined as “a game that is always present, available to the player. These games can be location sensitive and use several different media to convey the game experience” [18]. These games are typically designed to associate locations in the real world with virtual locations in the game world. For example, REXplorer [11] is a pervasive game that is played by carrying a mobile device to locations throughout Regensburg, Germany. The system detects virtual “ghosts” that deliver a portion of the game’s narrative.

Koivisto et al. performed a formal evaluation of a paper prototype of Garden of Earthly Delights, a massively multi-player online game that incorporates pervasive gaming concepts [66]. To perform the testing, the evaluators adopted a version of the Wizard-of-Oz prototyping technique. Each trial required three evaluators: a “computer” to manage the user interface, a “storyteller” to read aloud the game narrative, and an “observer” to take notes on player behavior. Use of the technique was compared to simple focus group discussions and was found to reveal significantly more feedback. Later work compared the technique to both focus groups and software prototypes [67]. The authors placed the discoveries made from evaluations on the game design into three categories: gameplay, game usability, and pervasiveness. The technique was found to identify as many design problems as software prototypes, but constructed in less time, and evaluations were performed with fewer difficulties.
2.3.2 Programming Tools

Programming tools have been developed for the purpose of rapidly creating a playable online prototypes. This approach is well suited to industrial use because studios typically have a large amount of in-house programming expertise specific to game development. Additionally, assets created from previous games can be reused to create sketches. However, these sketches require a large degree of technical expertise to create and can be more time-consuming to build than a paper sketch.

The Experimental Gameplay Project is a popular example of the use of programming tools to produce public game sketches and prototypes [50]. The project began as a student organization at the Carnegie Mellon Entertainment Technology Center and has recently grown into an online community of amateur developers. The goal of the project to develop new forms of gameplay through experimentations and to attract attention from development studios to aspiring game designers. To accomplish this, the group organizes a series of game design contests. The rules of the contests require prototypes to be novel, built by a single person, and to be completed within seven days. Prototypes from the project must be playable, fully functioning games. The winning entries are then put on the project web site [50] and shared with sponsoring development studios who use the project as a recruitment tool. The project has also lead the the development of a custom toolset for rapidly developing games [50]. The toolkit is targeted for expert developers and provides a set of C++ libraries with graphics and sound components. In the past, project members developed games using Macromedia Flash [3].
High-Level Languages

High-level languages enable designers to create game prototypes more rapidly than with more generalized languages such as C++. High-level language tools are particularly useful for designers without the expertise or experience necessary to build a sketch with production-quality game development tools. However in professional design studios where development expertise is available, high-level languages can sometimes introduce an unnecessary training step. Typically these tools are specific to a genre and integrate game components directly into the language.

An early example of a high-level game prototyping language is GAMBIT [81]. GAMBIT was designed in the early 1980s as a language to be used specifically for game development. GAMBIT is an object-oriented language similar to SmallTalk and has a syntax borrowed from Pascal. The programming language and execution environment are designed to closely resemble the players’ perception of the game universe. To accomplish this, language-level support is included for game relevant structures such as location, space, and time. GAMBIT programs are composed of three basic structures: globals, classes, and messages. Globals define the virtual space in which the game is taking place (the force of gravity, for example), classes define objects that will participate in the game, and objects can pass messages to globals as well as each other. GAMBIT objects intrinsically have spatial properties such as location, height, and width. Figure 2.4 shows an example GAMBIT class representing a paddle in a Pong-like game [81].
CLASS Paddle
  --variable and behavioral definitions for any player paddles
SEES PingConstants, PingEventNames, PingVariables
USES VerticalEdge.
VAR oldLocation: Point

HANDLER "initialize" DOES
  --set boundary and picture interaction
  SetLook(SolidPicture(40,8));
  SetBoundary(SolidPicture(40, 8));
  SetEventResponse(UserMoved Left, "move left");
  SetEventResponse(UserMoved Right, "move right");
  SetInteraction(VerticalEdge, "hit edge");
ENDHANDLER "initialize"

HANDLER "start" DOES
  --Put the paddle at the starting location
  MoveMeTo(ASK "starting paddle location" OF TheGame);
ENDHANDLER "start"

HANDLER "hit edge" WITH (anEdge: VerticalEdge) DOES
  --When a vertical edge of the playing area is hit
  MoveMeTo(oldLocation);
ENDHANDLER "hit edge"

HANDLER "move left" WITH (dist: Integer) DOES
  --move the paddle to the left
  oldLocation : = GetLocation;
  MoveMeBy(0.0 - PaddleMovementDistance * dist @ 0);
ENDHANDLER "move left"

HANDLER "move right" WITH (dist: Integer) DOES
  --move the paddle to the right
  oldLocation : = GetLocation;
  MoveMeBy(PaddleMovementDistance * dist @ 0);
ENDHANDLER "move right"

HANDLER "location" RETURNS Point DOES
  --what is the current location of the paddle?
  RETURN GetLocation
ENDHANDLER "location"

ENDCLASS Paddle

Figure 2.4: GAMBIT is an early example of a high-level language for game development [81]. This example creates a “Paddle” object for use in a Pong-like game.
CHAPTER 2. VIDEO GAME SKETCHING

Scripting languages

Today’s most popular high level languages for game design are the scripting languages built directly into game engines [114]. In the context of game programming, scripting languages are defined as languages that:

- require a parent game engine application to execute
- favor rapid development and simplicity over efficiency
- are implemented with an interpreter rather than a compiler
- communicate well with components embedded in the game engine

Games built on existing engines with scripting languages typically have most of the game mechanics implemented in scripting languages. The purpose of these languages is to provide developers with a more rapid means to produce content for the game, and in some cases, to enable end-user hobbyists to produce their own content. Some popular examples include UnrealScript [47], QuakeC [62], TorqueScript [51], and Renderware Script [38]. Where GAMBIT is useful for generalized development of many types of games, scripting languages are designed to support rapid development of specific game genres supported by the game engine. They provide first-class language support for behaviors built into the engines and hide the details of lower-level functions such as graphics rendering, networking, and audio processing. These languages are most commonly object-oriented and similar in design and syntax to C/C++ (Figure 2.5).

UnrealScript is a popular scripting language based on the Unreal engine [47]. Figure 2.5 shows an example program written in UnrealScript. The language is
designed to create modifications to an existing game written in C with the Unreal Engine. To modify game components, the designer extends a built-in “Mutator” class that provides functions that can be overloaded to inject new properties and behaviors. For example, Figure 2.5 overloads the “ModifyPlayer” function to inject new default properties for the player’s health and jumping ability into the game.

The main disadvantage to scripting languages are the limitations imposed by the game engine. In most cases, game engines are designed for a single type of game. For example, the Unreal Engine [47] is designed to support development of first-person shooters, and UnrealScript (Figure 2.5) includes language-level support for common first-person shooter objects such as weapons and ammunition. However, developing a game from a different genre, for example a car racing game, would be cumbersome.

Scripting languages for Story Telling

ScriptEase is a language designed to support the creation of realistic artificially intelligent non-playable characters [40]. The language was built as a scripting tool for the Neverwinter Nights engine [17]. Neverwinter Nights is a large role-playing game based on Dungeons and Dragons that includes hundreds of different non-playable characters with distinct behaviors. Rather than requiring the programmer to create different behaviors for each character, ScriptEase allows these behaviors to be expressed through patterns based on templates [90, 41]. Characters are given a pattern of behaviors they should follow and exhibit emergent behaviors based on those patterns. Behaviors are created with a programming language and assigned to characters through a simple user interface.

The main advantage of ScriptEase is its ease of use. The language is designed for
class PlayerBoost extends Mutator;

    var int MultiJumpCount, MultiJumpBoost, StartingHealth, MaximumHealth;

Function ModifyPlayer(pawn other)
{
    Local Xpawn x;  //Xpawn = changeable controllable character

    x = Xpawn(other);  //Xpawn provides an implementation for an abstract
    //"pawn" object passed as a constructor argument

    if (x != None)
    {
        x.MaxMultiJump = MultiJumpCount;
        x.MultiJumpBoost = MultiJumpBoost;
        x.Health = StartingHealth;
        x.HealthMax = MaximumHealth;
    }
}

defaultproperties
{
    MultiJumpCount=10
    MultiJumpBoost=50
    StartingHealth=250
    MaximumHealth=250
    GroupName="Jump Boost Mutators"
    FriendlyName="Jump Boost Mutator"
    Description="Changes your initial health and Multi-Jumping ability"
}

Figure 2.5: UnrealScript is a high-level scripting language built into the Unreal game
ingine[47]. This example creates a “Mutator” object that modifies the
player’s health properties and jumping ability. These languages hide low-
level implementation details such as graphics rendering, networking, and
audio rendering so the developer can focus more directly on designing
game content. However the games that can be developed with these
languages are restricted to those which the game engine will support.
entry-level programmers and has even been used as a learning tool in high school education [108]. Students with no programming experience were able to use ScriptEase to create interactive stories that could be experienced as playable adventures in Neverwinter Nights [30]. However, like other high-level scripting languages, ScriptEase only supports the functionality built into the game engine and its usefulness for generalized game development is limited.

**Stage-Based Scripting**

Stage-based scripting tools provide a combination of programming and WYSIWYG scene manipulation tools to create game prototypes. These tools typically include a “stage” that presents a real-time view of the 3D world at a given point of execution within an accompanying script. The user can manipulate state transitions through programming and design particular game states visually on the stage. Stage-based scripting tools are interesting because they provide a means of asynchronous collaboration between programmers and non-programmers, where the programmer can create scripts and the non-programmer can manipulate the scene. Example stage-based scripting tools include Alice [36], Adobe Director [2], and Game Maker [121].

Alice is a stage-based scripting tool [36] which has been studied extensively. Evidence suggests Alice is a particularly effective tool for teaching children basic programming skills [82]. As a result, Alice is currently used in numerous introductory programming courses at all levels of education [31]. A catalog of assets such as pre-made blank 3D environments, pre-built characters, and interactive objects is provided. Each object and character contains a collection of typed behaviors (the ice skater in Figure 2.6 has a “prepare to skate” behavior, for example). Scripts are
Figure 2.6: Alice provides a “stage” where the user can visually build a 3D scene and then a text-based programming environment to script behaviors for objects in the scene [36].

created with a constructive program editor, where the user creates statements by choosing phrases rather than typing text. The scripting language has a simplified object-oriented design, where objects can have simple behaviors and primitive properties. Like high-level language tools, Alice is well suited for game behavior scripting, but hides implementation details, which limits its usefulness to tasks supported by the behavior system.

Adobe Director is a similar tool to Alice, but is built for a more commercial audience [2]. The tool is designed to support the end-to-end development and internet
deployment of games with a small performance footprint, such as those meant to run in a web browser. However the tool has also become a common prototyping tool used in professional game studios [29]. The tool produces content for the Shockwave Player Plugin, which serves as a runtime container on the client [4]. The tool is the 3D companion to Adobe Flash [3]. The tool provides a stage and scripting environment similar to Alice, but augments it with a “timeline”. The timeline provides an interactive visualization of the state space of the game, where games states are represented at “frames” within the timeline. When a behavior is written, the state for a particular object is referenced by the index of the frame. For example, to make an object start at its initial state and progress through each state until it reaches a stopping point, the developer would send a message to the object to “play” from frame 0. The tool also includes a graphical user interface (GUI) for creating simple generic behaviors such as changing color, traveling from one point to another, and changing size. To create a behavior using the GUI, the user selects a start and stop frame, chooses a particular behavior from a list, then specifies behavior-specific parameters. Like Alice, Director is well suited for scripting game behaviors, but provides limited support for more advanced prototypes.

Visual Programming

Visual programming environments offer the game designer with a toolset that allows for the rapid construction of prototypes. Additionally, visual programming tools encourage modular architectures that enable components from successful prototypes to be included in finished games. Today, the best known example of a visual programming environment built specifically for game design is VirTools [42]. The tool is a
stage-based scripting environment that includes a visual programming language (Figure 2.7) in addition to a scripting language. Users manipulate the virtual scene space using a mouse, but construct game mechanics by wiring together circuit-like patches of behaviors.

The main advantage of VirTools is support for collaboration with non-programmers in the development process. Non-programmers are able to create simple game behaviors using the visual language. When a more complex behavior is needed, a programmer can create a module in C++ that can be referenced in the visual language. However the visual language requires extensive tool-specific knowledge to operate and requires developers to follow an architecture that supports integration with the visual language.

**Analysis of Programming Tools**

The most fundamental problem with programming tools is the need for technical expertise when sketching games. Even with visual programming environments such as VirTools, tool-specific programming expertise is required and creates a barrier for entry. Also, programming tools do not support synchronous collaboration on the design of the game. Most programming environments are designed for a single user to operate the tool on a desktop computer. Collaboration only occurs asynchronously through the exchange of developed components.

Additionally, a trade-off exists between expressiveness and ease of development. High-level language, stage-based scripting, and visual language tools offer a simplified development model that includes first-class language support for game components,
Figure 2.7: Virtools provides a stage-based scripting environment for game development, but includes a visual programming environment for connecting behaviors to objects in the 3D world [42].
but the expressiveness of these tools is limited to the behaviors built into the engines the languages are designed to support. Alternatively, high-level languages such as GAMBIT [81] provide support for more generalized game development, but less language-level support for game behaviors is provided.

2.3.3 Scene Design Tools

Where programming tools are designed to create game mechanics, scene design tools support creation of game content. Game content includes any of the assets that are used by the game engine to display the 3D scene. These assets include 3D models, textures, sounds, and terrains. Considerable work has been done to improve the techniques used to create 3D models. For sketching, many of these assets can be created rapidly using the same tools used to create actual game content or leveraged from previous projects. Scene design tools are used to organize that content into meaningful scenes. Example scene design tasks include level design, cut scene animation (non-interactive animations of content), and multi-player map design. These tools can integrate with programming tools to produce playable sketches that roughly resemble finished games.

Desktop Graphical Tools

Today most scene design tools are based on a desktop computer. These tools are typically bundled into the feature set of 3D modeling applications. Four popular modeling tools exist that include scene design functionality: Maya [9], 3DS Max [8], Softimage XSI [10], and Blender [19]. The modeling components of these tools are similar in design to drawing applications, where the user has a “canvas” containing
CHAPTER 2. VIDEO GAME SKETCHING

the 3D scene and a palette of tools used to manipulate the scene. Scene design is supported through a similar interface, and is augmented with a dynamics engine and in some cases scripting languages which apply physical simulation and animation to the models. Nearly all home design, landscaping, and architecture programs include some element of 3D scene design; however their use in the video game industry is limited. Desktop graphical tools have two major drawbacks:

- **A mouse only affords two degrees of freedom** (possibly three if you include a scroll wheel), and must be augmented with a means of switching “modes”. For example, a toggle can be used to switch the mouse between a “rotation” and a “translation” mode.

- **Object transformations must be specified with little context-awareness.** Desktop graphical systems typically do not know how an object will react when placed in a specific context. For example, a picture frame dragged across a wall that is not perpendicular to the designer’s point of view might enter an unnatural situation of floating in mid-air rather than staying flush against the wall.

**Two-Dimensional Sketching**

Two dimensional sketching interfaces have been proposed as a means of simplifying the process of creating 3D scenes. These tools use context awareness and constraints to provide a 2D drawing interface for creating a 3D scene [44]. Figure 2.8 shows Sketch3D [44], which is an example of such a system. For example, the user could create a 3D track for a racing game by drawing a top-down view of the track and rely on the sketching system to interpret a 3D scene from the drawing. The user would
Figure 2.8: Sketch3D is a two-dimensional sketching tool. The system provides a simple sketching interface that the user can use to draw an informal 2D view of a scene. The system interprets a 3D scene from the provided 2D drawing. Image acquired from [44]

use a drag-and-drop interface to place objects such as trees and walls onto a terrain and the tool automatically projects the scene to a 3D environment. The 2D interface improves on 3D modeling environments because it simplifies the “mode switching” required to navigate and manipulate the scene. Considerable work has been done on this technique as an example application for artificial intelligence and computer vision systems, as surveyed by Wang and Grinstein [117]. Also, most computer-aided design (CAD) programs contain some form of sketching interface, although few actually extend that interface into a 3D scene. Two major challenges exist with these systems:

- **Accuracy of the generated 3D scene is poor.** These systems rely on the accurate performance of components such as object recognizers, depth estimators, and content generators that infer what content should be placed in occluded regions of the scene. Each of these problems is challenging and the
subject of ongoing research. This problem is complicated in sketching interfaces because the drawing is not precise.

- **The designer must draw the entire scene before the 3D world is generated.** This is done because objects’ context is crucial to the scene generator’s accurate projection of the 3D scene. For example, the scene generator might detect a parking lot and infer that an unknown object on the parking lot sketch is most likely a car. Additionally, a full scene will contain more information about depth perspective than a half-drawn scene, which helps the depth estimator accurately determine object placement along the designer’s line of sight.

Later work has attempted to generate the 3D world “on the fly” while the user is actively sketching [122]. These systems use 3D direct-manipulation interactions similar to those surveyed by Hand [59] (for example, techniques for pointing at objects in a 3D environment). This enables the designer to change the point of view in the scene, which reduces the need for the system to generate appropriate content in occluded regions of the scene. To correctly generate portions of the scene before it is entirely designed, these systems utilize constraint systems generated from the objects’ semantics and their spatial relationship to each other [24, 52, 103]. Additionally, the user can interactively correct mistakes made by the system.

**Direct Manipulation with Multi-Modal Interfaces**

Advanced multi-modal interfaces have also been used to refine the interface for scene design toolkits. Early work focused on combining 3D input devices such as a spatially tracked mouse with either desktop displays [98] or head-mounted displays [26, 22, 80]. Hinckley et al. provide a survey of issues in the design of these interfaces [60].
3D Palette [15] and HoloSketch [43] combine the advantages of the stereoscopic view of a head-mounted display and the spatial awareness of a desktop interface. These tools use shutter glasses to present a stereoscopic view on a desktop display. An advantage of this approach is support for bi-manual interaction [56]. Users interact with the system by holding the tablet in one hand and making 2D and 3D gestures with the stylus in the other. On screen, the user sees a virtual tablet and stylus, along with the scene they are creating. A 3D menu system is used similarly to the toolbars in Microsoft Paint and can be navigated with either the stylus or through a speech interface afforded by the head mounted microphone. The user manipulates the scene by selecting tools and objects on the palette and placing them into the scene with the pen. 3D Palette extends the metaphor to include a 3D tracked Wacom Tablet [116] that contains the 3D menu system.
Text-Based Declarative Modeling

Textual descriptions have also been used to design 3D scenes [45, 70] (Figure 2.10). With these systems, the scene is generated from a description given from the perspective of an observer who travels through the scene. The declarative design model consists of three main steps: description, generation, and lookup. In the description stage, objects in the scene can be described qualitatively in terms of attributes, functions, methods, and components. For example, in Figure 2.10 the carpet in the first statement is described in terms of width, height, and depth. In the generation stage, these objects are then assembled into a hierarchical structure and spatial properties are applied using locative relations within that hierarchy. An example locative relation is found in the last line of Figure 2.10, where the chairs’ positions are described relative to the position of the table. This method, however, is prone to include inconsistencies in the scene description. Therefore, the lookup step is necessary, where a two step process is used to resolve these inconsistencies: a logical inference is used to check the topological consistency of the scene, and the output of that inference is passed to a global optimization algorithm that provides a solution, which is then presented to the user. For example, the position of the table in Figure 2.10 is described as being “on the carpet”, and the chairs’ positions are described as being to the left and right of the table. The logical inference would infer that the table should be placed in the middle of the carpet to ensure there is ample room around it to place the chairs.
- The carpet is two centimeters high, five meters wide and three meters deep.
- The table is seventy centimeters high and its width and depth are unfixed.
- Both chairs are one meter high and their width and depth are unfixed.
- The carpet is in front of the locator.
- The table and the two chairs are on the carpet.
- Both chairs are respectively on the left and on the right of the table.

Figure 2.10: Declarative modeling provides a text-based means of describing the content in a 3D scene. This example describes a simple dining room set. Figure acquired from [45]

2.4 Conclusions

Today it is difficult for non-programmers to create sketches that are actually playable on a computer. Paper prototypes provide a means of sketching game designs in low-fidelity to identify basic problems, but these prototypes are limited to only games that can be acted out with physical objects. Additionally, paper prototypes give limited understanding of what the completed game will actually look like. However paper prototyping can be done in groups and supports rapid development of throw-away prototypes.

Programming tools provide robust support for development of game mechanic prototypes, but are typically designed for single users and require specialized skills to operate. Tools such as high-level languages and visual programming environments exist that provide domain-specific tools which simplify development of game prototypes, but these tools are limited to the particular game genres they are designed to
support. Additionally, these tools do little to support synchronous collaboration.

Research has proposed numerous approaches to designing 3D scenes, but a need exists for tools that actually make those scenes interactive.

I designed *Raptor* to bring together the advantages of paper prototyping, programming, and scene design toolkits into a single tool. Like paper prototyping, Raptor encourages collaboration among groups of end users around a table. Raptor also supports Wizard-of-Oz prototyping. Like a programming tool, Raptor provides a robust API that can leverage existing game libraries. And like a scene design tool, Raptor enables game content to be organized into rich 3D environments.
Chapter 3

Tabletop Computers

To create a collaborative tool to support video game sketching, I included a tabletop computer in the design of Raptor. A tabletop computer consists primarily of a large touch-screen display that covers the surface of a table, around which multiple users can sit (Figure 3.1) [73]. They differ from desktop computers by providing a user interface that more closely mimics the way people interact with everyday physical objects. For example, a tabletop photo browser would allow users to manipulate a pile of digital photos with their hands using the same gestures used to handle paper photos, such as dragging photos across the surface, rotating photos with two fingers, or tossing from one side of the table to the other [73].

Tabletop computers have been studied extensively in academic research labs. Most work has focused on better understanding fundamental design considerations for tabletop applications. The technology used to implement tabletop computer systems has recently become inexpensive enough to create commercial tabletop computers, and a need exists for examination of potential application domains.

This chapter presents design considerations for tabletop computer applications and
concludes with a discussion on previous work that has applied tabletop computers to video game sketching.

3.1 Creating Tabletop Applications

Significant differences exist between applications developed for tabletop and desktop computers. Tabletops encourage collaboration among groups or collocated users and natural, gesture-based interaction with content on the table. This section presents
design considerations that must be taken into account when designing tabletop applications. These considerations are a subset of a more in-depth discussion of tabletop design recommendations from Scott et al. [101].

### 3.1.1 Designing for Collaboration

A natural advantage of a tabletop computer is support for groups of users, and consequently tabletop applications must be designed with collaboration in mind. Applications must not hinder the fundamental mechanisms people use to communicate with each other when collaborating [46], such as deictic referencing [57], gaze direction [115], and physical gesturing [13].

Additionally, tabletop applications must consider that users will interact with the system from a variety of locations around the table. This can have implications on both the form factor of the hardware [105] and the design of the applications themselves [61]. The system must also account for territoriality among groups of users [102], where users must respect the personal space of other users. Also, collaboration must take place within users’ “distance zones,” which are the areas in which users can comfortably interact with others [58]. The group task may also influence user location around a table [107]. For example, tasks requiring coordinated actions might benefit from closer proximity than those that make frequent use of personal spaces, or tasks requiring frequent conversation might benefit from a face-to-face arrangement [105].

Orientation is also an important consideration for collaboration. When users sit on opposite sides of a table, asymmetric information such as text may not be presented to all users simultaneously. The orientation problem can be solved either
by an automated means of choosing the most appropriate orientation [95] or letting the users manage it themselves.

Tabletop applications must also allow users to interact with the table simultaneously [109]. Most of today’s interface development toolkits are designed to support input from a single source, which is managed with a focus mechanism. For example, on a Microsoft Windows system, the window manager allows only a single window to be active at one time, indicated by moving the window to the foreground and brightening its border and title bar. All user input is then routed to the active window. Providing simultaneous multi-user input is both a hardware and software issue, where the tabletop hardware must detect multi-user input and the software must handle input from multiple widgets at once.

3.1.2 Supporting Physical Objects

Tabletop computers have the unique characteristic of providing a surface upon which physical objects can be easily placed. These objects can be task-relevant objects and non-task related objects (a coffee mug, for example). Tabletop applications that support interaction with these objects will benefit from the years of practice users have working with objects around tables. Some systems use generic objects that provide generalized input [48, 93] while others use task-specific objects [7, 112, 113].

3.1.3 Fluid Movement Between Activities

Tabletop computers also must not impose undue overhead when switching between activities, such as typing and drawing [109]. For example, paint programs require users to frequently switch between tools depending on the shape to be applied to the
drawing. When working with a tabletop computer, people do not make a distinction between interaction modes; rather they move rapidly between interaction modalities without explicitly acknowledging it [109].

Additionally, the form factor of a tabletop computer encourages connections to activities taking place in the local environment. It is therefore desirable for tabletop applications to allow users to easily transition between the tabletop to elsewhere in the environment. For example a user may wish to perform collaborative activities at the table, but move to a local desktop computer to perform individual activities.

### 3.2 Tabletop Computers and Game Sketching

Tabletop computers have been used to support collaboration in 3D scene design toolkits. For example, Rekimoto et al. used 3D scene design as an example application when describing “Cyber Codes” [94]. The system is an augmented reality furniture catalog that allows the user to see what the furniture for sale in the catalog would look like placed in a virtual model of their home. To create a scene, the user places a specially designed magazine on the surface of the table. Each page of the magazine is tagged with a spatially encoded marker. A ceiling mounted camera detects these markers and a pointing device is used to choose items on the page. A virtual model of the piece is then rendered on the table, where a pointing technique is used to move the item to the desired position and orientation. When the user is ready to view the scene on a secondary display, he/she places a small plastic model of a camera on the table at the position and orientation from which they want to view the scene. A view of the environment is then rendered on the secondary display from the vantage point of the camera.
Figure 3.2: Magic Land provides a direct manipulation interface for scene design on the surface of a table. The system enables users to load pre-configured virtual environments and place interactive models within those environments. Additionally, users can create animated 3D models of themselves with a 3DLive! motion capture studio and include those models in the virtual world. Image acquired from [33]
Magic Land extends the tabletop design metaphor to include 3D models of users themselves [33] (Figure 3.2). This system is focused on game mechanic prototypes, where the user can create interactive stories and explore plot lines. This is done with a 3D-live motion capture studio [16] and two interactive tables. The system uses a “box” metaphor to move animations from the capture studio to the tables. To begin, a user inserts a physical box into a slot by the 3D-live recording studio. Then the user enters the studio and records a variety of different actions. Once completed, the user removes the box from the slot and carries it over to the table. Once there, the user can open the box to see their avatar leap out of the box and onto the table, where it will interact with other recorded avatars. The table can also integrate computer generated characters such as dragons, pandas, and tigers, which are placed in the scene by choosing them from a menu in the interactive scene. These computer generated models also have awareness about the avatars. For example, when a human character gets close to a dragon, the dragon will breathe fire on the human. Predefined scenes can also be loaded onto the table through the use of a separate “puzzle table”. The puzzle table has a collection of puzzle pieces on it, each of which represents a preloaded scene. When the user wishes to add a scene to the main table, they drop a puzzle piece into a slot and the scene loads in real time. Qualitative results indicate that the system was useful for non-programmers to create interactive 3D stories [33].

Terrain modeling has also been done on a tabletop computer using a depth-sensitive camera (Figure 3.3) [118]. The system placed the camera and a projector above the table surface. The camera detected the shape of objects placed on the table surface, and the projector displayed a collection of cars driving around the table. A dynamics engine was then used to collide the virtual cars with physical objects. This
CHAPTER 3. TABLETOP COMPUTERS

Figure 3.3: MicroMotoCross is a 3D car racing game that uses physical objects placed on a table surface to rapidly create a 3D terrain. This image is a rendering of the 3D terrain created from a table with ramps made from folder paper placed on the surface. Image acquired from [118].

enabled designers to create 3D terrains by placing simple objects such as folded paper on the table surface.

A natural problem is that tabletop tools only provide support for scenes that can be approximated on a table surface. This restricts the design space to scenes that are largely planar and do not have much occlusion when viewed from above. This is suitable for many games that include common earth-bound simulations (car racing games, for example), but makes the tools inappropriate for games that have a large 3D component, such as a spaceship simulator.
3.3 Conclusions

Design principles for tabletop and desktop applications are very different. Recent work has applied tabletop computing to various tasks in game sketching such as terrain moulding [118] and storytelling [33], but a need exists for a robust tool that enables the creation of complete game sketches that combine multiple aspects of game design. To fill this hole, Raptor applies the tabletop design principles described in this chapter to the domain of game sketching and provides a more complete toolset than previous work.
Chapter 4

Raptor

When designing *Raptor*, my principal goal was to demonstrate how a tabletop computer could support collaborative game sketching. Specifically, *Raptor* demonstrates that people find sketching using gestures and physical props to be easier and more fun than traditional desktop-based tools, that collaboration is supported by the tabletop’s horizontal layout, and that *Raptor*’s natural interface is accessible by non-programmers.

This chapter presents a detailed description of *Raptor*. It begins with a discussion on the design processes *Raptor* is designed to support. The following section then describes the rationale behind *Raptor*’s key design elements. The tabletop user interface is then described in detail, followed by a description of the programming API. The chapter concludes with two example usage scenarios that illustrate how *Raptor* could be used in an industrial setting. The chapter concludes with a discussion on the key limitations of the system’s design.
4.1 Iterative and Participatory Design

Iterative design has long been a key theme in human-computer interaction research [54, 39]. In an iterative design process, prototypes are created, refined, and tested repeatedly throughout the development process. Rapid iteration is particularly important early in the design process, where the system design is not well defined and many ideas could and should be examined [69]. As the system design matures, the time between iterations grows as more effort is shifted toward the implementation of the design, and the fidelity of prototypes increases [54]. Research has shown that a need exists for tools that provide computerized support for sketching new ideas quickly throughout the design process [120]. Research has also shown that “participatory design”, where end users are involved in the design process, leads to better interactive system design [100, 84]. An important aspect of this approach is that users act as fully empowered participants [20, 21]. It is also critical that end users are exposed to the implementation technology throughout the design process [20, 21].

Raptor is a tool designed to support iterative, participatory design throughout the pre-production phase of game development. The physical layout of Raptor encourages intimate communication between team members, and a multi-touch tabletop interface allows multiple users to interact with the system at the same time. Raptor also includes tool transparency, where users create game sketches using simple gestures and physical props rather than pointing and clicking on a desktop computer. Furthermore, Raptor supports an egalitarian design process, where participants can fluidly move between the roles of play tester and designer.

Raptor encourages an iterative design workflow, where sketching sessions on the tabletop computer are followed by programming tasks on a desktop computer. This
approach enables a connection between the programmed implementation of game behavior and the design activities supported by the table. During a design session, developers and end users are invited to collaborate using the tabletop sketching interface. Game behavior is determined by a combination of pre-programmed behaviors built into interactive game components and Wizard-of-Oz prototyping, where designers use their hands to act out game behaviors. Sketching sessions seek to identify key game design ideas, which serve as requirements for subsequent programming tasks. During a programming task, the pre-programmed behaviors are refined and expanded to provide a more robust palette of behaviors available during the next design iteration. As the process moves forward, behaviors built into interactive objects become increasingly sophisticated, and culminate in a distributable playable prototype based on an underlying game engine included with Raptor.

### 4.2 The Design of Raptor

To support sketching in an iterative, participatory design process, Raptor applies design principles from previous tools and techniques. This section presents the design rationale behind the key elements of Raptor and relates them to previous work in game sketching and tabletop application design.

The tabletop sketching interface extends some of the advantages of paper sketching techniques toward digital sketches. For example, Raptor’s sketching interface supports collaboration among groups sitting around a table and requires no programming expertise to operate. Sketches can be created very rapidly, but the elements created in the sketch are throw-away at the end of the session and not intended to be reused in future prototypes or sketches. However, because the sketches are digital,
they provide a look-and-feel more similar to a finished product than a paper prototype. Sketches created with \textit{Raptor} also include pre-programmed behavior, which expands the space of games that can be prototyped with \textit{Raptor} to include games that require real-time interaction. For example, the racing games built in the evaluation presented in Chapter 6 would be difficult to sketch with paper.

4.2.1 Modular Programming Architecture

The programming model used to create game behaviors is similar to the architecture of previous visual programming environments [42]. Behaviors are defined in reusable components. The modular architecture is crucial to the design of \textit{Raptor} because it allows behaviors to carry forward into future sketches, higher fidelity prototypes, and even finished games executable on Windows PCs and the XBox 360 [71]. Behaviors must be created by a programmer using Microsoft XNA Game Studio [79] prior to a design session. However once a behavior is defined, it can be added to a game prototype using a simple gesture-based interface.

4.2.2 Applying Tabletop Application Design Principles

\textit{Raptor} applies the tabletop application design principles presented in Chapter 3. To facilitate users interacting with \textit{Raptor} from multiple locations around the table, \textit{Raptor} presents the virtual game world from a top-down perspective. This displays virtual game objects as though they are physically present on the table surface. Orientation of game objects is determined by their in-game behavior, and no bias is given to any particular side of the table.

\textit{Raptor} also interacts with physical objects placed on the table surface. \textit{Raptor}
responds to two types of physical objects: game controllers and physical props. When a game controller is placed on the table surface, Raptor immediately responds with a gesture-based interface for connecting the controller to the virtual world. When a prop such as a small toy is placed on the table, Raptor immediately places a virtual copy of that object in the game world at the location of the prop, treating the prop as a stamp.

Raptor also provides fluid movement between design activities on the table surface. For example, users can switch between molding the terrain in the game world and mapping input to game behaviors simply by performing different gestures on the table surface. Raptor also connects the table to other displays present in the local environment to support movement to design activities away from the table, but still in the local environment. This design element is crucial to Raptor’s egalitarian design process, where users assume the role of designer while sitting at the table, but can move to individual network displays to assume the role of tester.

4.3 Sketching With a Tabletop Computer

This section describes the gestures used when creating sketches with Raptor. The gestures are performed by individual users working together on a shared view of the same game world. Each gesture is described in detail and illustrated in an accompanying figure.
Figure 4.1: Designers can manipulate the virtual terrain with physical gestures, e.g., using a scooping motion to create a hill.

4.3.1 Creating Virtual Worlds

To begin creating a game sketch, teams design a 3D scene. Designers are first presented with a top-down view of a flat, bare terrain on the table surface. The terrain can be navigated using panning and zooming gestures commonly found in multi-touch map applications, such as the map application found on the iPhone [6]. To zoom in and out, designers use a “pinching” gesture. Zooms can be performed with the finger and thumb of a single hand, or large zooms can be performed bi-manually. To pan, users place a single finger on the table and drag the terrain as if moving a paper map across the table surface.

To contour the terrain, designers use gestures that mimic manipulating sand in a sandbox. To create a hill, designers use a “scooping” gesture (Figure 4.1). Similarly, to create a valley, a “spreading” gesture is used.

To add objects to the scene, designers perform a “stamping” gesture with a physical object, somewhat similarly to Build-It [92]. Stamps are small objects, typically toys, that are used to add similar virtual objects to the scene. Stamps also include
a small sticker that the table sensor uses to identify the stamp and determine its position and orientation. For example, Figure 4.2 shows the addition of a car into a racing game by “stamping” a physical toy car onto the tabletop. To add the car to the scene, the user simply taps the table surface with the object at the desired position and orientation. The virtual car is then added to the scene at the location where the physical car is stamped. To remove objects, a rubber eraser is used in a similar fashion.

To rearrange objects in the scene, users simply touch and drag them with his/her fingers. When a user touches an object, the object is immediately raised well above the terrain and follows underneath the user’s finger as (s)he drags. When the user lifts his/her finger, the object is placed on the terrain beneath the finger’s last location.

4.3.2 Adding Interactivity

Console games are normally played using a game controller, a special-purpose handheld input device providing buttons and joysticks for manipulating play in a virtual world. The responsiveness and intuitiveness of input mechanisms is crucial to the
Raptor allows designers to easily prototype game input via mixed-reality interaction with the tabletop. Figure 4.3 shows how a designer places an Xbox controller input device onto the table. A ring of “pins” surrounds the controller, representing the different input and output channels the controller provides (e.g., the different buttons and joysticks.) As the controller is moved around the tabletop surface, the ring of pins moves with it, creating a truly mixed physical-virtual entity. The pins on the controller can be connected to pins on other objects; the designer attaches the “A” and “B” buttons to the car’s gas pedal and brake pins, allowing these buttons to be used to accelerate and decelerate. To connect two pins, the designer touches the source pin, and then touches the target pin. A line is drawn between them to show the connection. Once a connection is established, the designer can pick up the controller to try out the interaction immediately, and then return it to the tabletop to refine input/output connections.
4.3.3 Controlling Network Displays

To view the game world from an angle other than the top-down view on the table, designers can connect a display via the local area network (Figure 4.5). The viewing position and angle is controlled by placing a small plastic camera on the table surface. To move the camera position, the designer simply positions the camera on the table surface in the desired location and rotates it to the desired angle. Designers can also tap an interactive object with the plastic camera to create a “chase” view. When a camera is in “chase” mode, it will follow closely behind the interactive object as it moves about the game world.

4.3.4 Wizard-of-Oz Prototyping

One of Raptor’s most powerful aspects is its support for a Wizard-of-Oz [65] style of play testing. Rather than having to program new game ideas, designers can act them out collaboratively on the tabletop display with his/her hands. Testers then “play” the game while sitting at a traditional display. As designers manipulate the game world, the tester sees the changes to the game world in real-time. This supports fluid and easy collaboration, allowing designers to rapidly move between design and playing roles, and supporting a very rapid iterative design cycle. However, behaviors sketched using the Wizard-of-Oz technique cannot be reused in future sketches or created by designers working by themselves.

Figure 4.4 shows a tester sitting at a PC playing a game as the designers modify it. Here, the designers are creating a racing game. Testers and designers have different viewpoints on the game. Designers have a top-down, two-dimensional view of the game world (Figure 4.4), while testers see the game in a more traditional 3D form.
Figure 4.4: Designers and testers collaborate using *Raptor*.

(Figure 4.5). For example, the designer may wish to examine a rule that says the player must pass through a gate on the track to move an obstacle. To accomplish this, the designer watches the tester’s car move on the tabletop display. When the car passes through the gate, the designer uses an eraser to remove the obstacle from the scene.

### 4.3.5 Supporting Collaboration

*Raptor* is designed to support synchronous collaboration among groups of co-located users. The system includes support for multiple users to interact with the same virtual world, but it does not provide any explicit interactions for collaborative manipulation.
Figure 4.5: A tester can play a game as it is being created, creating a fluid testing process and supporting Wizard-of-Oz prototyping.

of that world. Because the table supports multi-touch input, multiple users can gesture simultaneously. For example, one user could “scoop” on one part of the table while another “spread” on a different area of the table. This provides a collaborative experience where users are free to follow natural social protocols without having to manage turn-taking with the system.

To prevent users from interfering with each other when gesturing on the table surface, all multi-finger gestures are restricted to a maximum diameter of 12 inches. This avoids situations where two users touching the table in separate locations will be interpreted as a single gesture. The diameter was determined through simple trial and error with the gestures that required the most space on the table: the terrain
molding and input mapping gestures. Because all gestures were intended for use by a single user, the 12 inch diameter typically goes unnoticed.

4.4 The Raptor API

The Raptor API is used to create pre-programmed behaviors used when creating sketches, and can also be used to build stand-alone game prototypes executable on a Windows PC or the XBox 360 [71]. The API is based on a custom game engine I created based on Microsoft XNA [79]. The API consists of a Microsoft .NET [76] programming library and can be programmed with Microsoft XNA Game Studio [79].

4.4.1 The Raptor Engine

I designed the Raptor Engine to provide a simple programming model for creating interactive objects. As such, the engine is designed for simplicity and ease of use rather than providing a robust feature set. The engine is also designed to encourage programmers to enforce game rules in a modular behavior system that is leveraged in the sketching interface. The engine is divided into seven high-level components:

- “Entity” objects represent each object in the game world.
- The “Terrain” stores a simple heightmap representing the ground in the game world.
- The “Scene” is a container class for Entity objects and maintains a single Terrain.
- “Camera” objects maintain a viewpoint from which the scene is observed.
• The “Renderer” draws the Scene on the primary display from the point of view of a Camera.

• “Behavior” objects act on the scene to apply game rules and control the behavior of interactive objects.

• “Controller” objects gather input from the XNA subsystem and apply it to any connected behaviors.

Entity objects contain a unique identifier and geometric information for interactive objects in the game world. The terrain is a simple 2D array of floats with a collection of methods that manipulate the terrain when “scooping” and “spreading” gestures are performed. The Scene keeps track of all the entities in a game world and provides a common interface for setting Entity properties. Camera objects are an extension of the Entity objects and provide only geometric information about the viewpoint from which a display should view the scene. The renderer provides a “Draw” method that encapsulates all the details of drawing a Scene on a display. “Behavior” objects implement the rules that determine the behavior of interactive objects. Controller objects contain an event for each input found on the physical controller including buttons, joystick axes, and triggers. During each update call on the controller, the underlying XNA input framework is polled for the current controller state. Each input is examined for a change in state. If an input has changed, the corresponding event is triggered, which is routed to any connected behaviors through the event system built into the C# language.
4.4.2 Creating Interactive Objects

To create an interactive object such as a character or a car, the designer must create a Behavior (Figure 4.6). A Behavior object implements the IBehavior interface. The IBehavior interface contains the “Update” method called during each pass of the game loop, a reference to the Scene, and a global unique identifier (GUID) used to reference the Entity that holds the position of the interactive object.

To add Pins to the Behavior, a “Pin” attribute is added to a public method that accepts a single float as an argument. The Pin attribute accepts parameters for a path to the icon to be displayed on the tabletop display when connecting the interactive object to a controller. Additionally, a name must be provided to identify the Pin in case the icon cannot be found. Pin values are reported as floats between -1.0 and 1.0. When input is received from a Controller, a call is made to the Pin method connected to the appropriate Controller event and the Pin value is passed as an argument. For example, if a pin is connected to the “A” button on a controller, the property will have a value of -1.0 when the button is pressed and 1.0 when the button is released. Analog inputs such as triggers and joystick axes are represented as a range of values. For example, if a pin is connected to the Y axis of a joystick, when the joystick is neutral the value will be 0.0. When it is halfway to the bottom, the value will be -0.5. When it is moved to its forward-most position, its value will be 1.0.

Figure 4.6 shows a behavior that controls a Car interactive object. The car includes gas, brake, and steering input pins, and forwards that input to a Car simulator I created with a custom physics simulator. When the Behavior is connected to a Controller, input is received as calls to the method marked with the appropriate “Pin” attribute. These methods then store the input value in a member variable. When
the Update method is called, the member variables are polled for the current input state, which is then applied to the Car simulator.

To associate the interactive object with a physical stamp, a registration system is provided. Each physical stamp includes a small sticker placed on the bottom so it comes in contact with the table surface. This sticker is encoded with a two-digit hexadecimal code that identifies the stamp. To associate a stamp with a given Behavior, Raptor includes an XML configuration file that maintains a mapping between the two-digit hexadecimal code on the sticker and the name of the associated class. So when a new interactive object is added to Raptor, a new entry must be made in the XML configuration file.

4.4.3 Using the Raptor Engine

To create distributable game prototypes with the Raptor Engine, programmers must create a new Windows or XBox 360 Game project in Microsoft XNA Game Studio [79] and reference the Raptor Engine assembly, as well as any assemblies containing behaviors to be included in the game. By default, the project includes a “Game” class, which provides the main game loop that calls an “Update” and “Draw” method in each frame. The programmer adds a new Scene to the Game as a member, as well as any desired behaviors. Any necessary Controller objects, Cameras, and Renderers must also be added to the Game class as members. In the “Update” method of the Game class, the “Update” method on each of the Raptor classes must also be explicitly called. Connections between Controllers and Behaviors can be easily established programatically through a C# event handler. The programmer simply registers the desired method marked as a Pin with the desired event published from the Controller
Figure 4.6: Interactive objects are created by implementing the IBehavior interface in the Raptor API. In this example, a car behavior has gas, brake, and steering pins, and uses a car behavior model from a physics engine to provide a realistic driving experience.
class. The Raptor Engine handles polling the underlying input subsystem and reports events only when the state of the controller changes.

The Renderer hides the complexity of drawing a Scene from the programmer. The Renderer contains a “Draw” method that accepts a Camera, Scene, and an XNA “RenderTarget” as parameters. A “RenderTarget” is analogous to a whole or portion of a Window where the graphics subsystem will draw a 3D scene. The Renderer must be called during each call to the “Draw” method of the Game.

4.5 Usage Scenarios

To illustrate how Raptor supports an iterative, collaborative development process, I have created two scenarios. These scenarios synthesize my observations of how people use Raptor to sketch games, particularly illustrating collaborative development and Wizard-of-Oz prototyping. The scenarios also show that Raptor can be used to sketch a diverse range of game styles, including fantasy role playing games and multi-player shooter games, as well as the racing game described in the Chapter 6. All screenshots in this section show the results of enacting these scenarios with Raptor.

4.5.1 Building a Role-Playing Game

A fundamental element of many role-playing games is the game narrative. In this scenario, a team of designers wishes to experiment with simple story ideas. They are creating a medieval fantasy game including knights, dragons, monsters, and other common medieval story elements.

The team decides to work on a scene in a forest. In this particular scene, the
player must collect several hidden items. After all the items have been collected, a large impediment will be removed from the player’s path so (s)he can progress to the next scene.

The team begins by bringing out a box of plastic objects resembling entities in the game. They use a series of “scooping” and “spreading” gestures to contour the terrain and to create a smooth path through the middle. To create the forest, the team uses a small plastic tree in a “stamping” gesture. The team then hides the various items throughout the forest by stamping the objects throughout the scene. Similarly, they place a large boulder at the end of the path to inhibit progression to the next scene.

To add the playable character, a designer stamps a small plastic knight. To make
the character playable, they place an Xbox 360 controller on the table. A ring of “pins” appears around the controller. The designer then taps the character and a ring of “pins” appears around the character, including “walk”, “run”, and “attack” pins. The designer then connects the various character pins to the desired controller pins by tapping on the icons. The team wishes to view the game with a chase camera on a network display, so they tap the character with a small plastic camera. Immediately, a view of the world from the perspective of the character is shown on a nearby screen.

**Wizard-of-Oz Behaviors**

One of the designers then carries the XBox controller to the television and assumes the role of “player”. The remainder of the team stays by the table to act as wizards. As the “player” walks through the environment, (s)he finds the various items hidden in the forest. When the player bumps into each hidden item, the wizards use the eraser to remove the items from the scene, indicating that the player has collected them. Once all of the items have been collected, a designer drags the large boulder from the path so the player can continue to the next scene.

**Creating New Behaviors**

The team decides that the scene would benefit from the inclusion of a simple combat mechanism with some monsters roaming the forest. However the team does not possess a Monster behavior. The team decides that they can go no further without creating a Monster behavior, so they break off the sketching session and move to a programming phase. One of the team members is a software engineer from the game
production group, so he takes the responsibility for extending the knight behavior and creating the Monster behavior.

To create the Monster behavior, he begins by collecting a small plastic toy monster from his desk. To make the toy recognizable to the table, he places a small sticker on the bottom. He then makes note of the two digit code on the sticker. To program the behavior, he creates a new class called “Monster” that implements the IBehavior interface from the Raptor API. He does not wish to make the Monster objects respond to controller input, so he does not add any Pins to the class. In the “Update” method, he adds a simple pathfinding mechanism that causes the Monster to navigate the forest without bumping into trees. He also programs the Monster to charge toward any knight that comes within close proximity, and to respond when attacked by a knight. When the Monster behavior is complete, he goes to the Google 3D Warehouse [53] and downloads a free 3D model of a monster. He adds the 3D model to the class by importing it with the XNA Content Pipeline [79], and adds a line to the Monster behavior class with a path to the model. Finally, he makes a new entry in the XML registry of interactive objects that associates the Monster behavior with the two digit code from the sticker.

Once the monster behavior is created, the team returns to the table and the programmer brings the new Monster stamp with him. The team recreates the forest scene on the tabletop, and can now add several Monsters into the scene.

Conclusion

This scenario shows the ease of collaboratively creating a new scene and animating it through Wizard-of-Oz techniques. The scenario also shows the limitations of the
sketching technique. The sketch does not involve non-player characters with detailed dialogue trees, and to add a new interactive object to the scene, the team was forced to return to their desks to begin a programming phase to add more behaviors to the system. Later, the team was able to reconvene to continue sketching once the new behavior was added. The result of this limitation is an iterative, back-and-forth process where designers create sketches using the tabletop interface, then use the ideas generated in sketching sessions to influence the development of behaviors which can be used in later sketching sessions.
4.5.2 Creating a First-Person Shooter

The next scenario involves sketching elements of a first-person shooter (FPS) game. FPS games, such as the Halo series [25], often place a heavy emphasis on environment design and realistic physical simulation. They commonly are composed of a series of “levels” that consist of an interactive 3D environment and convey a part of the game’s narrative. Additionally, some first-person shooters include a multi-player component, where several players occupy the same specialized level. Multi-player FPS games include “Deathmatch”, where players score points by shooting other players, and “Capture the Flag”, where players attempt to penetrate the opposing team’s base, collect a flag, and return it to their own base. This scenario examines how Raptor can be used to design the environment of a multi-player first-person shooter and explore its gameplay rules.

Creating the Environment

A team of designers building a new World-War-II style first-person shooter gather around the table. They decide they would like to create a realistic urban battlefield experience for multiple players. To begin, they load a rocky, asphalt and concrete terrain onto the table surface and contour it to include several craters, giving the terrain a damaged, battle-worn look. To add interactive content, the designers open a box of plastic toy soldiers, buildings, and various weapons. They begin by stamping several buildings into the scene to create an urban environment. They then stamp weapons into the environment. The team then adds two avatars for the playable characters. Each character has several input pins, including “walk”, “side-step”, “fire”, and “special weapon”. To make the characters playable, the designers place
the controllers on the table and connect them to the avatars. To create a playable view on the network display, the designers touch one avatar with a small plastic camera. Immediately, a designer can assume the role of that “player” by carrying the controller to the network display, and can begin to explore the environment. A team of designers building a new World-War-II style first-person shooter gather around the table. They decide they would like to create a realistic urban battlefield experience for multiple players. To begin, they load a rocky, asphalt and concrete terrain onto the table surface and contour it to include several craters, giving the terrain a damaged, battle-worn look. To add interactive content, the designers open a box of plastic toy soldiers, buildings, and various weapons. They begin by stamping several buildings into the scene to create an urban environment. They then stamp weapons into the environment. The team then adds in two avatars for the playable characters. Each character has several input pins, including “walk”, “side-step”, “fire”, and “special weapon”. To make the characters playable, the designers place the controllers on the table and connect them to the avatars. To create a playable view on the network display, the designers touch one avatar with a small plastic camera. Immediately, a designer can assume the role of that “player” by carrying the controller to the network display, and can begin to explore the environment.

**Wizard-of-Oz Multi-Player Game Rules**

To create a multi-player experience, a second designer connects a controller and display to the other avatar and walks to a network display. The designers wish to prototype a simple “capture the flag” game in the new environment. The game “wizards” at the table place “flags” at opposite ends of the environment and move the
avatars into their starting positions. To prototype picking up the flags, when a wizard sees a character pass over a flag, s(he) erases it from the environment to create the appearance of the flag being picked up. When the game begins, the wizards keep a careful watch on the table to keep track of how many times each player is shot by the other. They decide that a total of three shots constitutes a “kill”. The designers determine that when players are killed, they immediately drop any flags they are carrying and return to their starting position. To enforce this rule, when the wizards count that a player has been shot three times, they use their hands to place the dropped flag at the location of the kill and then drag the avatar back to the starting position. When a player brings the opposing teams’ flag back to its starting position, the player is deemed to have won the game.

**Iterative Collaborative Development**

The team decides that the game design is interesting and they wish to experience a more robust sketch, where the characters respond automatically to the game rules. To do this, they record a play-test of the sketch with a video camera to preserve the rules they developed. This video then serves as a form of requirements specification for the addition of more functionality to the character behaviors. In the character Behavior class, support is added to detect when a bullet strikes the player, and a counter is maintained to determine when the character is killed. Additionally, awareness of the flag Behavior is added so that a flag will travel with the player when he comes within close proximity.

With these rules now programmed directly into the characters, the team is ready to return to the table and start a new round of sketching. This time, the team does
not have to enforce the basic rules of the game through Wizard-of-Oz prototyping and can focus on other aspects of the game such as scene layout or control schemes.

**Building a Distributable Prototype**

After creating the sketch of the game, the team decides the game design is worth pursuing. To share the design with remote team members, they decide to invest in development of a distributable prototype. The team then moves to a programming phase, where they create an executable game based on the Raptor Engine that reuses the behaviors from the sketch. To create a distributable prototype, the team first creates a new XNA game project in Microsoft XNA Game Studio [79] and references the Raptor Engine and the class library that contains the behaviors from the sketch.

In the main class of the new game project, a Scene is added as a member. Because the game is a multi-player first-person shooter, they configure the Game to display a “split screen” view of the scene, where two separate views of the same scene are shared on a single screen. Cameras are added to represent the viewpoints shown on the split-screen.

Several Behaviors are also added to represent the various interactive objects in the scene, including buildings, weapons, and playable characters. In the “Initialize” method of the game the scene is programatically laid out, and the cameras and controllers are connected to the two playable characters. Because the rules of the game were iteratively refined and added to the character Behaviors during the iterative sketching and programming phases, the programmer does not need to re-program these rules into the new prototype. When the scene is laid out, the programmer compiles the game into an XNA game package and shares it with his remote colleagues.
Conclusion

From this scenario, we see that quite complex game rules can be “acted out” using Wizard-of-Oz techniques, and that Raptor can be used to simulate interesting multiplayer gameplay. The tabletop surface is crucial to being able to do such scenarios, as the use of gestures and physical props allows designers to rapidly manipulate the scene. The scenario also shows how the game engine and behavior system used when creating sketches can be reused to produce a redistributable game prototype based on the output of the sketching process.

4.6 Limitations

As will be discussed in the evaluation section, Raptor provides strong support for collaborative game sketching. Designers – including programmers and non-programmers – are indeed able to rapidly demonstrate gameplay. It is easy to fluidly transition between design and testing roles, and same-place collaboration is effectively supported. Raptor nevertheless has limitations.

An obvious limitation is that the tabletop only allows prototyping of games from a top-down view. While I have successfully applied this approach to genres as diverse as racing games, role playing games and first-person shooter games, this limitation precludes games with large amounts of vertical motion or occlusion from above. For example, it would be difficult to create a sketch of Nintendo’s Super Mario Galaxy [85] because the game world consists primarily of movement around a series of three-dimensional spheres rather than across a planar terrain.

Also, I have found that locating a physical object to serve as a stamp can be
difficult to exactly match with a virtual object. For example, the toy car seen is Figure 4.2 is a yellow jeep, but the 3D model is actually a white sports car. I have not observed any noticeable effect of this limitation on the performance of the system, but it is nevertheless a violation of the basic design of the system. Additionally, an obvious extension of the modularized programming model would be to distribute interactive objects over the internet. However, each time a new object would be downloaded, the user would have to pair that object with a physical toy. I have developed a simple workaround for this problem where, if I cannot find a suitable stamp, I use a printed picture of the object instead.

4.6.1 Input Limitations

The mapping of input from controllers to interactive objects can only support one-to-one mappings, where the value from the controller matches the value received by the behavior. If the designer wishes to change the way a behavior processes input, (s)he must alter the code of the underlying Behavior class. An interesting solution to this problem might include creating an input transformation system, where the user can specify a function to be applied to the input value before it is passed to a Pin.

Input mapping is also troublesome when the desired control scheme should switch control from one interactive object to another. For example, when the ball is passed in a soccer simulation, control may follow the ball from one player to another. With Raptor, the only way to sketch this switch is to return the controller to the table during each transition and rewire the control scheme to the new object. Currently the best option for addressing this issue is to bypass the input sketching mechanism entirely when this situation arises and to program response to controller input directly.
into the interactive objects.

4.7 Conclusions

Raptor is designed to support an iterative, collaborative design process for video game sketching. In particular, Raptor demonstrates how a tabletop computer can support collaborative game sketching in the context of a larger process based on iteration and participatory design. The tool provides an egalitarian design process where non-programmers are not disadvantaged during sketching session. The tool also supports fluid movement between the roles of game designer and tester.
Chapter 5

Implementation

Implementing Raptor was a significant challenge. Raptor combines the challenges of implementing a video game, a multi-touch tabletop application, and a distributed system into a single implementation. To create Raptor, I first created the Raptor Engine described in Chapter 4, then built the tabletop system on top of that engine. Also, I utilized three off-the-shelf components to simplify development: Microsoft XNA [79], the Microsoft Surface SDK [73], and Fiia [28].

This chapter describes the current implementation of the sketching interface of Raptor. The chapter begins with a description of the software architecture of the sketching interface. Then I describe in detail the implementation of the tabletop user interface and its underlying components. That is followed by a description of the tabletop computer. A description of the network display implementation follows. The chapter concludes with a discussion on some key limitations of the current implementation of Raptor.
5.1 Architecture

Raptor is constructed with a modularized, extensible architecture. The system consists of three basic components:

- The Scene is the basic artifact being sketched. The Scene is implemented in the Raptor Engine.
- The Scene Editor implements the tabletop sketching user interface.
- The Remote Displayer provides a view of the sketch on a network display.

These components are tied together with Fiia [28]. Fiia is a toolkit that simplifies the development of collaborative augmented reality systems. Fiia enables application developers to express their application at a conceptual level using the Fiia modeling language without worrying with the details of networking. The Fiia.Net [28] runtime will then map the conceptual specification to a runtime implementation.

A conceptual specification in the Fiia Modeling Language [28] is shown in Figure 5.1. The Scene is a Fiia “Store”, and the Scene Editor and Remote Displayer are Fiia “Actors”. The Fiia.NET runtime mediates communication between these components and maintains consistency between synchronized copies of the scene residing on the Scene Editor PC and the Remote Displayer PC.

5.2 The Scene Editor

The scene editor is implemented as a Microsoft XNA [79] game that uses the Raptor Engine. The tool was developed with Microsoft XNA Game Studio [79] and a modular architecture was used to separate implementation concerns into logical components.
Figure 5.1: *Raptor* is based on Fiia, a toolkit for building augmented reality applications. It is composed of three basic components: a Scene built with the *Raptor* Engine, a Scene Editor that implements the tabletop sketching interface, and a Displayer that renders the Scene on a network display.

The core component is an XNA Game called the Editor. From the *Raptor* Engine, the Editor includes a Scene as a member variable, as well as a Camera and Renderer used to display the top-down view of the game world. The Editor has an “Update” method that is called by the XNA subsystem during each frame. The Editor is modularized into several components that extend from a common EditorComponent base class. Each EditorComponent includes a reference to the Editor, which mediates communication between components.

The Editor is separated into the following components:

- The *SurfaceInputManager* encapsulates the input from the tabletop computer and publishes input events that other components can subscribe to. Input events
are sorted into three groups: fingers, blobs, and tags. The component also uses the Microsoft Surface SDK [73] to detect arrive, drag, and release events. The Surface SDK is described in more detail in Section 5.4.

- The `EditorCameraManager` subscribes to drag events from the `SurfaceInputManager` and manages the position of the top-down view of the tabletop display. This component detects “pinch” gestures by determining when two proximous fingers are drawn toward or away from each other. To zoom in/out, the `CameraManager` adjusts the height above the terrain the camera is positioned.

- The `EntityAddComponent` subscribes to tag events from the `SurfaceInputManager` and determines when a stamp has been placed on the table surface. When a new stamp arrives, this component looks up the two-digit hexadecimal code found on the stamp in the mapping between codes and referenced Behavior classes. It then reflectively instantiates the Behavior as a Fiia.NET Actor. This component also detects when an eraser has been placed on the tabletop surface. When an eraser arrives, the component checks to see if an interactive object is beneath the eraser. If there is, the component removes the object from the scene and disposes of it.

- The `EntityMoveComponent` subscribes to touch events in the `SurfaceInputManager` and determines when a user is dragging an interactive object across the table surface. When a touch arrives, this component checks to see if the touch is on an interactive object. If it is, that object will remain directly under the finger as it is dragged around the table surface. When the finger is raised, the object is free to move about the game world once more.
• The InteractionGraphComponent provides the implementation of the input mapping interface. This component subscribes to input events from the SurfaceInputManager. This component determines when a game controller is present on the table surface and draws the ring of pins around the controllers and interactive objects. The component also interprets touches to connect/disconnect controllers and interactive objects. Connections are established as Fiia.NET SubscrConnectors. When new input events arrive from the Controller, Fiia.NET routes these events to the appropriate Behavior Pins.

• The TerrainManager subscribes to blob events from the SurfaceInputManager and performs terrain deformations. This component detects “scooping” and “spreading” gestures. When two blobs are determined to be proximous and moving toward each other, a “scoop” is detected and the terrain between them is raised. When two proximous blobs are moved away from each other, a “spread” is detected and the terrain between them is lowered. To give a realistic feel to the “scooping” and “spreading” gestures, a $\sin$ wave is applied to the raised or lowered terrain, where the $\sin$ wave crests at the midpoint between the blobs and troughs directly under the blobs. This provides a realistic looking “mound” or “hole” of terrain that resembles the behavior of real dirt or sand.

• The NetworkDisplayManager subscribes to tag events from the SurfaceInputManager and detects when a network display controller arrives on the table surface. This component maintains a list of connected displays and associates them with a given toy camera. This component also sends Camera position updates to Network Displays through Fiia.NET. More information about the Network Display implementation is provided in the following section.
5.3 The Remote Displayer

Compared to the SceneEditor, the Remote Displayer implementation is quite simple. Like the SceneEditor, the Remote Displayer is implemented as an XNA Game that uses the Raptor Engine. However, unlike the SceneEditor, the network display has a read-only view of the Scene. Each network display has its own unique plastic camera that controls its viewpoint when placed on the table. When a Remote Displayer first connects to the SceneEditor, the Remote Displayer sends the two-digit hexadecimal code that identifies the plastic camera associated with that display. The SceneEditor then provides a string identifier of a camera that the Remote Displayer can synchronize through Fiia.NET.

5.4 The Tabletop Computer

The SceneEditor uses a commercially available tabletop computer called Microsoft Surface [73], seen in Figure 5.3. The table display surface is 30 inches wide with a 4:3 aspect ratio. The table also includes a wide border around the display that is convenient for placing objects when not in use. The table includes built-in support for object tracking and multi-touch sensing for up to 52 simultaneous contacts. The table surface is composed of a one inch thick acrylic tabletop with a rear-projected embedded display. The display resolution is 1024x768.

The multi-touch sensor is composed of an infrared lamp beneath the table surface and an optical sensor consisting of 5 redundant cameras (Figure 5.3). The light shines infrared light up through the table surface and, when a contact is made on the table surface, the light is reflected back beneath the table where it is detected by the
A robust SDK is provided with Microsoft Surface. The SDK consists of a core component that encapsulates all the basic functionality of the system and provides a set of basic gesture recognition processors. Contacts are reported as a 2D coordinate on the surface, an identifier used to track a single contact across frames, and an orientation. Contacts are also separated into 3 categories: finger, blob, and tag. Fingers are identified by their shape and size, tags are identified as a dot pattern (described in the following subsection), and blobs are the remaining contacts that are not fingers or tags. The SDK core is designed to be integrated with XNA applications and can be polled for the state on the table surface. Additionally, a programmer can
Figure 5.3: Microsoft Surface provides a sturdy acrylic tabletop (1), an optical sensor consisting of an infrared light (2) and an array of cameras (3), and a projector (4). Image acquired from [73].

subscribe to contact events on the table surface through the event system built in to .NET applications. When building *Raptor*, I chose to use the event model.

Additionally, the SDK provides a Windows Presentation Foundation (WPF) [78] component based on the core component. WPF interfaces are expressed in an XML file and includes a robust animation system. Microsoft Expression Blend [75], a what-you-see-is-what-you-get (WYSIWYG) design tool, can be used to build Surface interfaces based on WPF without the need for any programming. However WPF applications have poor 3D rendering performance, so I was unable to use the WPF layer when building *Raptor*. 
Figure 5.4: *Raptor* uses the tag tracking features built into the Surface SDK to track objects on its surface. Image acquired from [73].

### 5.4.1 Object Tracking

Object tracking was performed by placing a small, unique tag called a “Domino” [73] on each object to be tracked (Figure 5.4). Dominos are approximately 3/4 inches square with a black background and array of white dots. For a tag to be detected, it must be placed flat against the table surface. The tags used with *Raptor* are vinyl stickers, although tags can be printed on common printer paper if the black ink used is not reflective in infrared. The white dots are arranged such that the orientation of the tag can be determined, and the tag can be uniquely identified. Tags contain 8 bits of information, enabling up to 256 unique tags to be identified. The 8 bit tag is printed as a two-digit hexadecimal value in the center of the tag.

### 5.5 Limitations

*Raptor* has one known limitation that is a result of the implementation. Currently, for a 3D scene to be displayed accurately on a network display, the Remote Displayer must be connected to the SceneEditor before any edits are made. This is due to a limitation in Fiia.NET [28]. Currently, Fiia.NET does not support the ability to
synchronize the current state of a Store. Rather it only forwards changes to a Store as they occur. Therefore any changes that are made to the scene prior to the connection of the Remote Displayer will not be visible on the network display.

5.6 Conclusions

The modular design of Raptor is instrumental in the success of its implementation. The system makes use of middleware components to interface with subsystems [79, 73] and to manage software distribution across the local network [28]. The system is fully implemented and provides a clear example of how a tabletop computer can support game sketching.
Chapter 6

Evaluation: An End-User Study

I designed an empirical study for the purpose of evaluating Raptor. This chapter discusses that evaluation and the obtained results. The goal of the evaluation was to determine if Raptor could successfully be used by groups of end-users to create game sketches, and whether the tabletop computer provides a significant improvement over a comparable desktop interface.

The evaluation consisted of groups of end-users creating a game sketch with Raptor in both a tabletop and desktop format. In brief, the participants’ task was to create an off-road racing game. Participants would create enough of a sketch to form an opinion on both the tabletop and desktop versions and answer a series of questions designed to determine which system they preferred along a set of criteria. Additionally, users were asked to provide feedback in a discussion format.

The following section describes the systems used in the evaluation. The game content provided to users to build their sketches is discussed in Section 6.2. Section 6.3 describes the procedure followed during each experimental session. Section 6.4 details the participants used in the study. Sections 6.5 and 6.6 describe the statistical
methods used to compare conditions and the results obtained. The chapter concludes with an interpretation of the results in Section 6.7.

6.1 Conditions and Apparatus

The tabletop apparatus was the *Raptor* implementation described in Chapter 4. The desktop apparatus was functionally equivalent to the tabletop version of *Raptor*, but had a simple point-and-click interface in place of the tabletop. To ensure the desktop and tabletop were as similar as possible, the desktop version was created with the same underlying software components and ran on the same computer hardware.

The desktop interface was operated with a single mouse and included 2 windows: a scene and a palette (Figure 6.1). The scene provided the same view of the 3D world as the tabletop. The palette provided a set of tools used to manipulate the
Figure 6.2: The desktop graphical representations of the camera and controllers scene. Users’ would manipulate the scene by choosing a tool from the palette and clicking on the scene. Graphical equivalents were provided for each of the physical tools used in the tabletop condition. For example, in the tabletop condition, the user placed a controller on the table and a ring of “pins” was shown. Then the table’s touch sensitivity was used to connect pins together. In the desktop condition, the user would click on a toggle to show a picture of the controller with the ring of pins around it, and the mouse was used to connect pins together. Graphical representations of the controller and network display are shown in Figure 6.2. The desktop interface was operated with a single mouse and included 2 windows: a scene and a palette (Figure 6.1). The scene provided the same view of the 3D world as the tabletop. The palette provided a set of tools used to manipulate the scene. Users’ would manipulate the scene by choosing a tool from the palette and clicking on the scene. Graphical equivalents were provided for each of the physical tools used in the tabletop condition. For example, in the tabletop condition, the user placed a controller on the table and a
Figure 6.3: The desktop condition provided a collection of easy-to-use tools meant to correlate to functionality from the tabletop condition.

ring of “pins” was shown. Then the table’s touch sensitivity was used to connect pins together. In the desktop condition, the user would click on a toggle to show a picture of the controller with the ring of pins around it, and the mouse was used to connect pins together. Graphical representations of the controller and network display are shown in Figure 6.2.

The tools provided in the desktop condition are shown in Figure 6.3. They included:

- An *eraser* used to remove objects from the scene
- A *hand* used to pan around the scene (the mouse scroll wheel was used to zoom), and also used to move objects
• *Terrain* buttons used to raise and lower portions of the terrain. For example, if the user wanted to create a hill, (s)he would click the “Raise” button and click on the desired location in the scene.

• *Stamps* used to add objects to the scene

• *Controller display toggles* enabled the user to show hide the controller and network display objects.

### 6.2 Provided Content

The tabletop and desktop conditions used the same collection of interactive objects (Figure 6.4). The objects were sufficient to create a simple sketch of a 3D off-road racing game:

- A *Car* provided an interactive vehicle that could be driven on the terrain. The car included three pins: gas, brake, and steering. The car was modeled with a physics simulator as a rectangular body with cylindrical wheels, a suspension system similar to an off-road buggy, and a rear-wheel drive engine. Steering was modeled by turning the front wheels like a real car.

- A *Parking Cone* provided a simple marker used to lay out a desired track on the terrain. The cone had no input pins. The cone was modeled with a physics engine and could collide with the terrain, cars, and other cones.

The objects were placed on a rocky terrain to give the feeling of an off-road environment. In the remote display, a textured sky was also provided to provide a more realistic look and feel.
Figure 6.4: Participants were given an interactive car and a parking cone to build their sketches.

(a) An interactive car with gas, brake, and steering pins
(b) A parking cone with no pins

6.3 Procedure

Participants were placed in groups of three. At the beginning of each session, each participant read and signed a letter of information and consent form, and completed a short demographic questionnaire (Table 6.1). Then they were introduced to the other participants. Each participant then consented to participate with their group members and the trials began.

A within-subjects design was used, where every subject group completed both the tabletop and desktop conditions. To account for ordering effects, the conditions were rotated on a Latin square. For each condition, the groups were given a tutorial on how to operate the interface. They were then asked to demonstrate understanding of each feature and given as much time as needed to become comfortable with the system’s operation. Once each participant indicated they fully understood how to
use the system, the trial would begin.

Trials were untimed. Users were asked to create a sketch of an off-road racing game with the following properties:

- Players should traverse an off-road track as quickly as possible.
- Each lap must contain an element not present in the previous lap. For example, the game designers might introduce new obstacles into a specific area while the player was driving on a different portion of the track. This was done to encourage subjects to use the Wizard-of-Oz technique to sketch an in-game behavior.
- The terrain must contain multiple hills and valleys.

6.4 Participants

For the study I recruited 21 volunteers from the information technology office of a major automotive manufacturing company. No volunteers were rejected. Participants all used desktop computers daily as part of their jobs. The participant group consisted of 14 males and 7 females ranging in age from 21 to 61 (average age of 36). 33% of participants had used a tabletop computer before the study. Also, subjects had varied levels of programming and game development experience. Participants also spent varied hours per week playing video games, ranging from 0 to more than 10.
6.5 Data and Statistical Analysis

Ten Likert items presented a statement and asked the participant to indicate how strongly they agreed along a five point scale ranging from “strongly disagree” to “strongly agree” (Table 6.1). The same set of Likert items was presented for both conditions. The responses to the Likert items were compared with a Kruskal-Wallis test [68]. A Kruskal-Wallis test is a one-way analysis of variance that provides a non-parametric method for testing equality of population medians among groups. I chose the Kruskal-Wallis test over a one-way analysis of variance (ANOVA) because it does not assume a normal population. This is accomplished by replacing the data points with their ranks in their respective group. However the test does assume an identically shaped and scaled distribution for each group. These assumptions are valid since the Likert items used the same continuous scale for each condition. For the purpose of this study, I interpreted a statistically significant result as the probability of the null hypothesis (p) being less than 0.01.

I also measured the effect size for the differences between the two conditions. Effect size was determined through a meta-analysis of the collected data to determine whether the differences were statistically significant and also to report the size of those differences. I used Cohen’s $d$ value [35] to determine effect size. Cohen’s $d$ value is the most appropriate test for measuring effect size when comparing means. It is defined as the difference between the two means divided by the pooled standard deviation. Cohen suggests the following interpreting the $d$ value: small, $d = 0.2$; medium, $d = 0.5$; large $d = 0.8$. 
Table 6.1: Means and Standard Deviations of responses to Likert items. The responses were on a five point scale ranging from “Strongly Disagree” to “Strongly Agree”

<table>
<thead>
<tr>
<th>Question</th>
<th>Tabletop</th>
<th></th>
<th>Desktop</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I was able to build a playable game prototype.</td>
<td>4.52</td>
<td>0.60</td>
<td>3.95</td>
<td>0.59</td>
</tr>
<tr>
<td>The game I designed was fun.</td>
<td>4.33</td>
<td>0.73</td>
<td>3.52</td>
<td>0.75</td>
</tr>
<tr>
<td>I consider my design experience a success.</td>
<td>4.62</td>
<td>0.50</td>
<td>4.10</td>
<td>0.83</td>
</tr>
<tr>
<td>The environment was easy to use.</td>
<td>4.38</td>
<td>0.50</td>
<td>3.90</td>
<td>0.89</td>
</tr>
<tr>
<td>Creating my 3D scene was easy.</td>
<td>3.95</td>
<td>0.86</td>
<td>3.52</td>
<td>0.87</td>
</tr>
<tr>
<td>Adding input to my game was easy.</td>
<td>4.62</td>
<td>0.50</td>
<td>4.19</td>
<td>0.81</td>
</tr>
<tr>
<td>Prototyping game rules was easy.</td>
<td>4.00</td>
<td>0.84</td>
<td>2.71</td>
<td>0.85</td>
</tr>
<tr>
<td>I enjoyed building the game.</td>
<td>4.57</td>
<td>0.51</td>
<td>3.95</td>
<td>0.86</td>
</tr>
<tr>
<td>Having a partner to work with was useful.</td>
<td>3.62</td>
<td>0.67</td>
<td>2.00</td>
<td>0.84</td>
</tr>
<tr>
<td>I collaborated with my partners on the game.</td>
<td>4.43</td>
<td>0.51</td>
<td>2.00</td>
<td>0.84</td>
</tr>
</tbody>
</table>

6.6 Results

For the Likert items, the results show significance for 4 questions. The strongest results were seen in the two Likert items dealing with collaboration with partners, indicating that the tabletop interface does in fact support local, synchronous collaboration better than the comparable desktop interface. Additionally, the results show that participants found it easier to prototype game rules and produce a more fun prototype with the table.

When asked to choose which condition they preferred, participants showed a strong preference for the tabletop condition. The results show that participants felt the table was easier to use, more fun, supported collaboration better, and generally preferred to use it. Interestingly, participants felt neither the tabletop or desktop condition produced a better prototype, which somewhat conflicts with the result from the Likert item comparison indicating the tabletop produced a more fun prototype.
Table 6.2: Comparison of means of responses to Likert items. The responses were on a five point scale ranging from “Strongly Disagree” to “Strongly Agree”

<table>
<thead>
<tr>
<th>Question</th>
<th>Kruskal-Wallis (p)</th>
<th>Cohen’s D</th>
</tr>
</thead>
<tbody>
<tr>
<td>I was able to build a playable game prototype.</td>
<td>0.003</td>
<td>0.96</td>
</tr>
<tr>
<td>The game I designed was fun.</td>
<td>0.001</td>
<td>1.09</td>
</tr>
<tr>
<td>I consider my design experience a success.</td>
<td>0.034</td>
<td>0.79</td>
</tr>
<tr>
<td>The environment was easy to use.</td>
<td>0.067</td>
<td>0.69</td>
</tr>
<tr>
<td>Creating my 3D scene was easy.</td>
<td>0.113</td>
<td>0.49</td>
</tr>
<tr>
<td>Adding input to my game was easy.</td>
<td>0.083</td>
<td>0.65</td>
</tr>
<tr>
<td>Prototyping game rules was easy.</td>
<td>0.001</td>
<td>1.53</td>
</tr>
<tr>
<td>I enjoyed building the game.</td>
<td>0.016</td>
<td>0.90</td>
</tr>
<tr>
<td>Having a partner to work with was useful.</td>
<td>0.001</td>
<td>2.15</td>
</tr>
<tr>
<td>I collaborated with my partners on the game.</td>
<td>0.001</td>
<td>3.61</td>
</tr>
</tbody>
</table>

6.7 Discussion and Conclusions

The results of the empirical study support our hypothesis that the novel physical, gesture-based interface built into Raptor is useful for game sketching. In both cases, users were able to build a playable game prototype. Given that participants were not at all familiar with the interface before participating in the study and had a variety of experience programming computers, we feel that the design of the system proved simple enough to be operated effectively by end-users. Given this result, we feel a particularly interesting area for future work would be in educational environments where Raptor could be used as a creative educational tool.

Additionally, the results clearly indicate a strong preference for the tabletop version of Raptor. While this supports our hypothesis that the tabletop computer would better support collaboration among co-located groups, the results do not provide any indication about other forms of collaboration or, in particular, single-user operation. However, our finding that the tabletop interface was more fun to use supports the
hypothesis that designers would prefer to use Raptor’s physical stamping gestures and touch-based interface even when working alone.

Tabletop vs. Desktop Computers

While our results suggest the tabletop version was an improvement over the desktop version of the Raptor interface, the study is not sufficient to make any claim about tabletop versus desktop computers in general. Still, our study highlights some key differences. The tabletop computer provided a more collaborative experience where multiple users could interact with the tool simultaneously rather than following an explicit turn-taking mechanism as with the desktop computer. One participant commented that with the tabletop, “instead of having observers and one manipulator interacting with the environment, multiple users could stay active and interested at once.” Additionally, participants appeared to enjoy sitting in a circle around the table rather than crowding together at a single desk and facing the same direction towards a monitor. This made for a more sociable experience, and appeared to lead to more conversation.

Noted Shortcomings

Participants regularly noted a frustrating aspect of the tabletop tool, where one designer would interfere with another by manipulating the tabletop’s shared view of the world. For example, a designer would stamp an interactive object in an undesirable place because, just before the stamp touched the table, another designer would pan/zoom the table’s view of the world. Additionally, when designers worked close together on the table surface, the gesture recognition system could malfunction. For
example, two designers dragging interactive objects towards each other could be interpreted as a “zoom out”. These observations illustrate a common problem with tabletop computers, where application designers must carefully manage public and private spaces on the table surface.

Limitations of the Study

While the study indicates the role of the tabletop computer and validates the effectiveness of Raptor as a game sketching tool, but does have limitations. For example, the study does not indicate what specific interactions contribute to its successful design. For example, it is not clear that the “stamping” gesture is more effective than a more conventional alternative such as a software-based palette or menu. Additionally, the study only provides subjective results. A future study might seek to more quantitatively define the effectiveness of Raptor when support collaboration. Also, the study only examines the tabletop design interface and does not at all validate the iterative design process Raptor is designed to support.
Chapter 7

Summary and Conclusion

*Raptor* is a novel tool designed to demonstrate how a tabletop computer can support video game sketching. *Raptor* provides engaging tool transparency, where sketches can be created and manipulated using gestures and physical props rather than pointing and clicking on a desktop computer [5]. *Raptor* also supports engaging collocated collaboration, where the physical layout of a table allows intimate communication, and where the table’s multi-touch input allows more than one person to interact with the sketch at a time. And finally, *Raptor* encourages iterative, participatory design through an egalitarian design process, where people without programming skills are not disadvantaged, and where roles can be fluidly set and changed.

In Chapter 2, I introduced the concept of game sketching, and placed it in the context of the game development process. In Chapter 3, I introduced tabletop computers, and discussed several design considerations for tabletop applications. Chapter 4 described *Raptor* in detail. The chapter began by describing the iterative, collaborative development process *Raptor* is designed to support. The design of *Raptor* is then related to previous systems, and design principles for tabletop applications are
applied. The sketching user interface and programming API are then presented and illustrated through two usage scenarios. Chapter 5 then describes the implementation of Raptor and focuses on the sketching user interface implemented on the tabletop computer. Chapter 6 presents a controlled study I performed that compared the tabletop sketching interface to a comparable desktop interface and found that Raptor is indeed usable for collaborative game sketching, and the tabletop interface is easier and more fun to use than the desktop.

7.1 Summary of Contributions

This thesis made the following contributions to the domains of video game sketching and tabletop computing:

- I presented a novel game sketching tool based on a tabletop computer. By using a tabletop computer, the tool naturally supports collaboration and provides a rapid means of sketching games. The tool illustrates a collection of interactions based on emerging trends in human-computer interaction research that provide a logical mapping between tabletop computing and game sketching.

- A novel, practical application of table computers was presented. Tabletop computers have shown promise in the research lab, but applications for the technology have been thus far lacking.

- Tabletop computers were compared directly to desktop computers. While tabletop computers show promise as an emerging platform, little work has been done to compare them directly to desktop systems.
7.2 Limitations

The most obvious limitation of Raptor is the range of games that can be sketched. Raptor is only useful for sketching games that can be approximated on a tabletop surface. For example, games that include heavy occlusion when viewing the world from above are difficult to create. Additionally, games that require extensive vertical movement, such as 3D space simulations, are awkward to sketch. Also, requiring a physical prop to add content to the world limits the ability to share content over the internet. For example, if a user downloads a new entity from the internet, (s)he must pair that virtual content with a physical prop before it can be used on the table surface.

The tool is designed for simple operation by end users. This limits the functionality of the system. For example, the system design avoids complicated control mechanisms camera viewpoint shown on the tabletop, but the world can only be seen from a top-down view. However an experienced design team might prefer to use a more complicated tool that includes more features, even if that tool requires extensive training.

7.3 Future Work

Future work with Raptor will follow two paths: supporting larger groups with a bigger table and examining the tool when used in longer durations.

Currently the only available tabletop computer that supported all the functionality required by Raptor had an interactive surface with a 33 inch diagonal. While this table was suitable for groups with four or less people, larger groups would find the size
of the table too small. An interesting area for future work might examine what new interactions must be included when the table is scaled to larger groups. In particular, an interesting problem could arise when manipulating content in the middle of a larger table which would be uncomfortable to reach from a seated position. This would require new interactions to be developed that support manipulation of the game world outside of the user’s private area on the table.

Also, the current study only examines the role of the tabletop design toolkit in a controlled setting. A future study might take a more longitudinal approach that gives the tool to a professional game design group and is put to use for longer durations that require iteration. This study could include both the tabletop design toolkit and programming with the \textit{Raptor} API. Also, the study would help better understand what is gained and lost from the design experience when using an end-user tool such as \textit{Raptor}.

### 7.4 Conclusion

Sketching is a powerful technique for rapidly examining design ideas early in the development process of interactive systems. However tool support for sketching is limited, particularly in the domain of video game development. This work addresses two problems with current game sketching tools. First, today’s game sketching tools require domain-specific technical knowledge to operate. This provides a significant barrier of entry for those without technical training. Second, today’s tools are designed primarily for single users. Most tools are based on desktop computer interaction with a keyboard and mouse and do not support multiple concurrent users well. This work addresses these two problems by providing a collaborative user experience
that requires no prior technical skills to sketch video games. The tool is based on
tabletop computer, which provides support for a tool transparency, collaboration,
rapid development, and an egalitarian, iterative design process.
Bibliography


[37] Creative Labs, Inc. Open Audio Library (OpenAL), 2009.


