Test-driven Development in the OCaml Programming Language: An Experience Report

Michael DiBernardo

Department of Computer Science
University of British Columbia

INTRODUCTION

One of the primary goals of software engineering research is to be able to produce robust and easily maintainable software at ever increasing speeds. To this end, programming language designers seek to design and implement programming languages with features that support rapid development, safety, and other desirable software qualities. However, the widespread adoption of a new language is an infrequent occurrence. This is unsurprising, because the work that is required to provide sufficient library support, establish useful processes and conventions, and retrain developers is very likely to outweigh the benefit provided by the new language. It is frequently the case that many successful projects must be written in the new language by enterprising supports of it before larger, more influential organizations will even begin to consider an adoption strategy.

The OCaml programming language was designed to provide developers with a variety of very high-level programming constructs while still being able to produce very efficient executables. Despite over a decade of active development and support, OCaml has seen very little practical use outside of academia. I have done the bulk of my software development in C++, and over time I have grown extremely dissatisfied with the amount of work that is required to produce even simple programs with it. OCaml appeared on paper to be a good replacement for C++, because of its emphasis on portability and efficiency.

As such, I decided to undertake a significant implementation in OCaml, so that I might assess the utility of the language and, if successful, outline a possible adoption strategy for moving to it from a 'mainstream' object-oriented language such as C++. My focus is on the practical: I was primarily concerned with the availability of supporting technologies (editors, build tools, etc.), the maturity of the supporting libraries, and the extent and utility of the available resources (documentation, community, etc.). I was also curious about how popular development methodologies such as test-driven development (TDD) would work with a higher-level language such as OCaml.

This paper describes my experience, identifies several useful supporting technologies, suggests some OCaml program design strategies, and outlines a sample adoption plan for switching from a development process that uses a 'mainstream' language to one that uses OCaml. To my knowledge, there does not exist any work of this kind in the public domain: The entirety of the OCaml literature is focused on language implementation details, and does not address how these language features are useful or how a 'mainstream' programmer can acclimatize to them. The closest work is the book Practical OCaml [16], which received uniformly bad reviews from the OCaml community. There are a number of good tutorials and introductory materials on the web, such as the OCaml tutorial [7] and several free books, the most popular of which is Developing Applications in Objective Caml [13]. However, these materials discuss language features in isolation, and do not provide much advice on how to integrate these features in a nontrivial program. Also, these resources concentrate on the functional aspects of the language, and do not discuss how to marry object-oriented design principles with the functional foundations of the language. It is my hope that this work will provide some insight into these previously unaddressed problems.

The paper is laid out as follows: section 1 discusses the OCaml programming language and identifies features that may potentially contribute to increased programmer productivity; section 2 gives a description of the program that I implemented in OCaml, and section 3 describes the implementation and what I learned from it; and section 4 summarizes my findings and suggests a possible adoption strategy for those who wish to try migrating to OCaml.

1 BACKGROUND

OCaml is often referred to as being a "kitchen sink" programming language because it combines many features from many programming paradigms. The academic *literature* is most concerned with the implementation of these features, and does little to discuss how these features are supposed to increase programmer productivity. Thus, the discussion in this section is largely derived from experience reports and pedagogical books on OCaml, rather than research papers. The purpose of this section is to dissect the features of OCaml and identify how they may each potentially contribute to programmer productivity.

Language features

Static typing and type inference

OCaml is a strongly and statically typed language. Many programmers appreciate static typing because it eliminates an entire class of possible logical errors (type mismatch errors) from occurring in their programs [15]. Unlike many other statically typed languages, however, OCaml automates much of the process of specifying type information by *inferring* the types of expressions at compile-time. This means that OCaml programs tend to be much more succinct than programs written in other statically typed languages, because type declarations and other type-system related statements are not required to be present in the program text. Because compile-time errors can typically be fixed faster than runtime errors, moving the detection of type errors into the compiler could potentially result in shorter overall development time by reducing the time spent identifying and fixing these errors.

Algebraic data types and pattern matching

The OCaml type system supports the usage of algebraic or *variant* data types, in which an instance of a single type may take on a series of different values with different structures. These types may be also be defined recursively. The flexibility of algebraic types allows the programmer to concisely and naturally declare complicated data types in a way that is easy to read and understand.

For example, a type that represents shapes of primary colours could be defined as follows:

```
type primary_colour = Red | Blue | Yellow;;
type shape = Square of primary_colour
           | Triangle of primary_colour
           | Circle of primary_colour;;
```

A popular example of a recursive variant type declaration in OCaml is the binary tree. In this example, we create a binary tree type that holds integers at the leaves:

```
type binary_tree = Node of int
                 | Tree of binary_tree * binary_tree;;
```

An instance of a binary tree can then be specified as a literal in the program, as in the following example:

```
let my_tree = Tree(Node(1), Tree(Node(2), Node(3)));;
```

If one compares the declaration and instantiation of the binary tree type in OCaml to mainstream languages such as C++ or Java, it is apparent that the OCaml style is much more concise, especially when it comes to specifying literal instantiations of the type in the code. This brevity can be especially helpful in test code, especially when using agile methods, where the test code is not only supposed to exercise the program but is supposed to document the usage and intent of the unit being tested.

OCaml also provides *pattern matching* as an elegant mechanism for operating on variant types. Pattern matching allows for the simultaneous deconstruction and naming of components of variant types. This is especially useful for mutating complex structures. For example, suppose we wish to write a function that replaces the left subtree x of a binary tree with a new subtree that has x as its right child and a 0-node as its left child. This function would require at least a few lines to write in a language such as Java or C++. Using the pattern-matching facilities of OCaml, we could specify this function as follows:

```
let wacky_tree_operation t =
  match t with | Tree(x, y) ->
                 Tree(
                  Tree(Node(0), x), Tree(Node(0), y))
               | Node(x) -> Node(x);;
```

Pattern matching allows the programmer to define algorithms in a manner that closely mimics the mathematical or "conceptual" statement of the operation, which potentially reduces the effort required to translate algorithm descriptions into code.

Diverse methods of execution

In the spirit of being a "kitchen sink" language, the implementors of OCaml have provided several ways of executing OCaml programs. The OCaml 'toplevel' allows for the interactive input and evaluation of programs at the command line, a feature which many interpreted languages provide (such as Lisp/Scheme, Python, Ruby, and Perl). The toplevel allows the developer to incrementally test components of her program, and is widely espoused as being an excellent mechanism for detecting errors in design and implementation early, especially by the Lisp community. The OCaml interpreter is also capable of executing OCaml programs as scripts, which allows for the possibility of using OCaml as a scripting language.

The OCaml bytecode and native code compilers are well-known for their ability to produce extremely efficient executables. The performance of OCaml in the Language Shootout (performing worse than only C, and better than C++) is an oft-quoted statistic in the OCaml community, because it demonstrates that it is possible to use a high-level language and still produce an efficient implementation. Many developers use high-level interpreted languages such Python for rapid prototyping, with the intention of implementing the final version in a "production language" such as C++ for purposes of efficiency. The promise of OCaml is that it can be used both for rapid prototyping and for implementation, which obviates the need for costly and error-prone translation efforts.

The fact that OCaml can be used in many different areas of programming – shell scripting, prototyping, and implementation – means that the OCaml developer can do everything in one language and thus become extremely proficient with it, instead of having to maintain a "toolbox" of many different languages for different purposes. Of course, this also implies that the design and implementation of the OCaml language must be good enough to match competing languages in each of these areas.

Integration of object-oriented, functional, and imperative programming paradigms

OCaml provides object-oriented, functional, and imperative programming constructs (e.g. for/while loops and mutable variables), so that the programmer may choose the paradigm which is most appropriate for the task at hand. These features are all implemented with varying levels of maturity, with the functional constructs being the most mature, the object-oriented constructs next, and the imperative features the least mature. Some attempt has been made to add features to each of these paradigms to allow them to interoperate; for example, the object-orientation layer provides a very terse syntax for returning clones of objects with certain instance fields changed, so that immutable objects may be easily implemented for use in functional programs that operate on them.

The consequences of this integration are twofold: The first is that programmers have a great deal of flexibility in choosing how to design a program that will be implemented in OCaml; the second is that programmers who have been trained in one paradigm can use OCaml to gradually become accustomed to the other.

Garbage collection

Like most modern programming-language implementations, OCaml provides a garbage collection facility so that programmers do not have to explicitly manage memory. What makes OCaml unique in this respect is that it provides garbage collection in a runtime environment that also allows for very fast execution.

Native data structures

OCaml provides several data structures that are built directly into the language core instead of being provided as libraries. Lists, arrays, records, and tuples are all 'native' OCaml data types. The benefit of having these types built into the language is that the compiler can detect many type and semantic errors that it would not be able to infer if these types were supplied as a library. This also implies that these types can be used in pattern matching, which allows for terse implementations of algorithms that operate on them. Finally, the syntax of these types allows for concise definitions of literal instantiations of them, which again leads to more readable code that clearly communicates intent.

Libraries

The standard OCaml distribution ships with core libraries that implement a variety of rich data structures and utility frameworks for common programming tasks (I/O, parsing, graphics, etc.) These libraries are written in a functional style, and provide a great deal of functionality in a set of extremely small and well-defined interfaces. The small size of most of the interfaces allows the programmer to keep much of the agglomerate standard library interface "in

her head", so that little time is spent browsing through library documentation to find the desired interface component.

The OCaml Hump [5] is a repository for 3rd party libraries that implement functionality not found in the standard library. For example, there are several projects that are currently targeted at implementing an object-oriented standard library for programmers who feel more comfortable operating on objects as core types. There are also mechanisms that allow OCaml programs to easily interface with existing C libraries and Perl CPAN modules, greatly increasing the number of libraries that the OCaml developer has available to her.

Language extension and implementation tools

Implementing a domain-specific language can be a useful technique for solving problems that require very general solutions. Such an implementation is usually a significant undertaking, one that requires the use of external tools such as lex and yacc.

OCaml provides language implementation features that are built directly into the OCaml language itself. Parsers specified in OCaml produce instances of user-defined variant types as output, so that the impedance mismatch between the output of the generated parser and the data types used to represent that output in the program. The parser generator syntax itself is quite similar to existing tools so that the programmer does not have to familiarize herself with a completely foreign interface. Also, because the parser generator is built into the language, the compiler can catch many errors at compile-time that would be impossible to diagnose in a library implementation.

In instances where the programmer herself requires a particular language feature that is not present in the OCaml language, OCaml provides a syntax extension tool so that she may specify her own syntactic constructs. In practice, however, this feature is rarely used by OCaml developers, unlike the macro system in other popular functional languages such as Lisp.

Sophisticated debugger

While the OCaml design philosophy is to eliminate as many errors at compile-time as possible, it does provide tools to aid in the detection and elimination of runtime errors. The OCaml debugger is a so-called "time-traveling" debugger, in that it allows the programmer to run the program being debugged to the point of failure and then step *backwards* from the breakpoint in order to find where things began to go wrong. This is much more intuitive than attempting to infer a breakpoint that is "early enough" in the execution sequence so that the error can still be observed, and is especially helpful in diagnosing errors that only occur long after the program begins execution. Because the diagnosis and repair of runtime errors can occupy a great deal of a programmer's time, a tool that reduces the length of the process can greatly reduce overall development time.

How OCaml's language features (supposedly) increase productivity

In the discussion of OCaml's language features, we identified specific ways in which those features increased developer productivity. However, there are three fundamental qualities of OCaml that we can extract from this discussion that are the core contributers to increased productivity when developing applications in OCaml.

Brevity

OCaml provides a variety of language features that allow one to write solutions to problems in a very terse and yet natural way, if used correctly. By permitting the concise and localized description of complicated algorithms and constructs, OCaml makes the development and maintenance of programs easier than in more verbose languages, especially for new developers who are recruited to work on a mature project. Of course, if misused, this inherent terseness can obfuscate a program just as easily as it can clarify one.

Performance

We have mentioned that OCaml is capable of competing with popular 'production languages' such as C/C++ in terms of performance. However, performance is not usually a characteristic that is considered to improve programmer productivity.

Many experienced developers count on 'throwing away the first one' when they implement a new system, so that the nuances of the problem can be exposed before solidifying the final design. This practice has encouraged many developers to write application prototypes in a 'prototyping language' that tends to be a high-level interpreted language such as Python. By simultaneously providing high-level language features that contribute to rapid development *and* generating fast executables, OCaml allows the developer to progress from prototype to final implementation without having to reimplement a lot of functionality or deal with functional differences in the prototype and the first production version. This time savings in turn provides more time for developers to work on optimization. This is one reason why many OCaml programmers claim that they can develop programs that perform better than equivalent C++ programs in the same amount of time.

Flexibility

OCaml allows programmers to write programs using a variety of paradigms that can be executed in variety of ways, which theoretically means that a developer could do the vast majority of their program development using only OCaml. This in turn allows the developer to become very proficient at using the language and to build up a large library of reusable components, instead of having to maintain separate libraries for 'single-purpose' languages.

2 PROBLEM STATEMENT

In the previous section, we described reasons why OCaml can potentially increase programmer productivity, and then discussed how, despite this, it still has not been readily adopted by many agencies in industry or academia. The lack of large-scale practical implementations makes it difficult to understand how a development process that uses OCaml would differ from development processes in more mainstream languages. Thus, the purpose of this project was twofold: First, I intended to undertake an implementation of a nontrivial application in OCaml, paying close attention to the available tools, best practices, etc. that one would want to be aware of before integrating OCaml into a production development environment. Second, I hoped to document this experience to serve as an example 'adoption manual' for others interested in OCaml, because existing OCaml projects are generally poorly documented and in particular do not document the mistakes that the authors made before arriving at the final product.

In order to make this guide useful to the 'mainstream' programmer, I decided from the outset to try to use object-oriented design principles and test-driven development, allowing for the possibility that this would not work and that I might potentially have to switch paradigms in the middle of the project. In the event of failure, I would still be able to document this switch to an unfamiliar paradigm in a way that mainstream programmers would understand, because I would be starting from a platform that they are exquisitely familiar with.

A subgoal of this project was, of course, to learn the OCaml programming language for my own personal use. Prior to this project, I had very little exposure to functional programming (despite several abortive attempts to learn Scheme and Lisp), and I was hoping that OCaml would help me bridge the knowledge gap between object-oriented and functional programming paradigms.

The task

Of course, in order to learn about application development in OCaml, I had to pick an application to implement. After considering a few possibilities, I finally settled on the game of Tetris, for a variety of reasons. Firstly, game programming is a hobby of mine, and I am continually surprised by how long it takes to implement even simple games in common languages for the task (C/C++/Java), so I was hoping that OCaml would provide me with a better alternative. Games are an area where rapid development and efficiency are both primary concerns, and so on paper OCaml appears to be a very good tool for the job because it produces very efficient executables while providing very high-level constructs in the language. OCaml also has a variety of mature 3rd party graphics frameworks (GTK, OpenGL bindings) owing to the ease with with it can interface with C libraries, which is not always common in rapid prototyping languages (Perl, for instance). Finally, writing Tetris in a completely unfamiliar language is a significant undertaking, but it is not so wildly ambitious as to make the task impossible to complete in 60-100 hours of work, which was about as much time as I was willing to allocate to this project.

I selected Tetris specifically because it has all the elements of a typical real-time game (event loop, scoring, end conditions, etc.) without requiring excessively complicated support tools (e.g. level editors, character editors). It also helps that Tetris required very little artistic ability, since I am severely lacking in that area. Finally, the fact that Tetris is a real-time game implies that I would have to pay at least some attention to efficiency.

There has been a little bit of work in using OCaml for game development, all of it academic, and essentially all of it done in the functional style. Our very own Robert Bridson developed a 3D first-person shooter in OCaml for a course project at Stanford [12] using the lablgl OpenGL bindings; however, no report accompanies his implementation. Francois Pessaux, a former researcher from INRIA, developed a 3D Doom-like engine in standard ML and published a paper on the experience [14], with an emphasis on demonstrating the adequacy of ML in developing 3D graphics applications; however, this focus is primarily on performance, and only touches productivity issues lightly. Also, the work is quite outdated, as it was performed in 1998 with an archaic version of the standard ML compiler.

3 PROCESS

This section will describe the process that I followed to implement the game of Tetris in OCaml using test-driven development. I have decided to present this discussion as a chronological presentation of the issues that I encountered in the process, with frequent section breaks for reflection, as this will allow readers who are undergoing their own adoption attempt to "'follow along" and to frame the problems that they encounter in the context of the work presented here.

Getting started: The spike

When I first started this project, there were so many aspects of designing applications in OCaml that I was unfamiliar with that I decided to write a very small "toy" program that would allow me to get at least some idea of how OCaml and its related tools operated before starting any serious work on the Tetris prototype. The goals for the spike were to learn basic OCaml syntax and core features of the language, to set up my editor and development environment, to decide on a build process, and to identify potential resources for help in the event that I encountered problems that I could not solve in a reasonable amount of time on my own.

I decided to implement a small, primitive game in which two players alternate selecting from the head or tail of a list of positive integers, each seeking to end up with the largest sum of numbers at the end of the game. This is on the order of a program that one might expect as a first or second assignment in a CS1 course, and so I thought it was appropriate for my level of expertise with OCaml at this point. I considered using test-driven development here but decided against it, thinking that this would be overkill for the scope of the task.

Editor support

The first step I took was to set up my editor for use with OCaml. I had previously attempted to do work in Lisp before starting this project, and Lisp is a language where the choice of editor is coupled tightly with the choice of language; almost all Lisp development is done in Emacs. I have used the Vim editor for many years now, and I found to my surprise that many OCaml developers did the same: I was under the impression that most languages with toplevels used Emacs as the development environment, because of the ease with which command-line tools can be made to interoperate with it. I also looked into using an OCaml IDE, but the only IDEs I could find were Camelia [1] and an Eclipse plugin for OCaml [3], both of which are immature even by the standards of the authors.

In less than ten minutes, I was able to find and install VIM syntax and indentation files for OCaml. There was only one option for each of these, so I did not have to spend any time at all making a choice between varying implementations of these files, which was nice. I contemplated looking for a plugin that would allow me to shuttle highlighted code from VIM into an interactive OCaml toplevel, but these sorts of things are usually poorly implemented in Vim because it was not designed with this purpose in mind, so I decided that I would live with simply copying-and-pasting source files into the toplevel as required. This was the only respect in which the OCaml Emacs mode was superior to the Vim OCaml integration tools, as far as I could tell, and I did not think it would be worth struggling with a new editor just to have a tighter interface with the toplevel.

Build tools

I then went searching for build tools that would be sophisticated enough to handle the requirements for the Tetris prototype without being excessively complicated. I needed the ability to build multiple targets, and perhaps to build supporting libraries if I ended up factoring out code that I thought might be useful in future projects.

I was surprised at the variety of tools that were available for this task. I found a number of standard-issue OCaml Makefiles for use with GNU make that required very little customization and that supplied all of the various switches and options needed to link in some of less-frequently used supporting libraries, although few of these allowed building multiple targets. I also found a couple of more sophisticated options. The first was an OCaml mode for SCONS [10], which is a Python-based language for writing build scripts that allows very terse declarations of library and executable targets and their dependencies. The second was a full-fledged build system written in OCaml called OMake [8], which is similar in spirit to SCONS but uses syntax that is closer to Makefile syntax, and is specifically designed to strongly support C and OCaml projects. The OMake system is developed and maintained by a research group at the California Institute of Technology, and is used to build their own large-scale OCaml projects . Ironically enough,

the OMake documentation goes into far more detail about building C projects than it provides for OCaml projects, and so I was not convinced that I would be able to get things working with OMake in a reasonable amount of time. It was reassuring and somewhat inspiring, however, to see a sophisticated system like OMake implemented in OCaml.

I ultimately decided to use one of the OCaml-specific Makefiles, simply because it worked out of the box without requiring any real work on my part. I intended to switch to OMake later on if the Makefile proved insufficient for my needs.

Issues encountered while writing the spike

Because the program that I was writing was so small, and because the 'default' paradigm used in most OCaml tutorials is functional, I implemented the spike in the functional style (no mutable state, few side effects). One of the first issues I encountered was my unfamiliarity with functional programming; even 'let' statements confused me at first. However, I quickly got the hang of the basic constructs (functions and let-bindings) by copying and pasting small expressions into the toplevel. The toplevel was extremely helpful in understanding how the type inference was working as well, since I was able to get immediate feedback instead of having to invoke the compiler. I was also helped along by the OCaml community on the IRC channel #ocaml throughout this effort: Usually there were two or three people there at all hours who were of varying degrees of helpfulness.

I was somewhat concerned that I wasn't really 'understanding' the functional style, though, because I hardly used closures anywhere, except when applying library functions like List.map. Adding to my confusion was that I was not really sure when to use tuples, lists, arrays, or records; Representing the sequence of integers as a list seemed like a good idea at first, but it made retrieving the tail quite awkward, for example. Tuples especially confused me, because at this point I did not quite appreciate what the use of pattern matching was, and retrieving values from tuples is extremely ugly if one does not use pattern matching. The available tutorials and books did not discuss what data types would be most appropriate to use in different situations, and so I was left feeling hopelessly ignorant at the prospect of not being able to make such a simple decision.

Another difficulty I encountered was in splitting the program across multiple source files. OCaml has some very funny naming conventions when it comes to specifying module names. Implicit modules are declared for each file, but the way that the filename interacts with the assigned module name is unintuitive: the first letter of a file must always be lowercase, but will be interpreted as uppercase in the module name, while subsequent letters in the filename are used unchanged in the module name. Hence, the source file 'oCamlTest.ml' will place functions in the namespace 'OCamlTest'. It took me an embarrassingly long time to figure this out, because I could not tell from the cryptic compiler errors if I was missing a directive of some sort in the source file, or if I was just naming things incorrectly. This was one case where asking questions in #ocaml resulted in very little useful help: Most people referred me to the documentation for OCaml's module system, which is an extremely complicated piece of work, and the documentation does little to identify quick fixes for common problems and use cases.

Finally, while the compiler errors produced for failures in type inference were generally easy to understand, other syntax errors usually produced the unhelpful response "Syntax error" from the OCaml compiler, and the line/character error targeting was often incorrect. It took me quite a bit of practice to get to the point where I could diagnose common error cases. A notorious feature of the OCaml syntax that leads to a lot of confusion is when to end an expression with a single semicolon, a double-semicolon, or no semicolon at all; the OCaml tutorial describes a heuristic that one can use to properly place these syntax elements that requires *5 paragraphs* to explain. At first, I attempted to understand exactly what the difference between the two elements is (the ;; is used to separate statements at the toplevel, while the ; is a sequence point). However, after I became more familiar with the heuristic, I basically stopped thinking about what these elements meant and was able progress without messing it up too often. One unfortunate consequence of the sequence point operator is that the last line in a sequence of operations is not supposed to have one: This can make commenting out the last few lines of a sequence of operations difficult, because you have to remove the semicolon on the new last expression in the sequence. I thus wrote a small Vim function to do this for me, but it does not cover all situations and so I still often had to do this manually. Small time-wasters such as this can cause a lot of time to be wasted in the aggregate.

Reflection and next steps

Writing this spike was useful in helping me set up the basic tools that I would need for development, and in familiarizing myself with some of OCamls quirks as described here. It was especially useful in helping me learn how much code I could write before stopping to verify that what I was doing was sane by running it through the compiler. I was pleased to note that I did not encounter a single logic error while writing this program, which was rather refreshing. However, I still did not have any idea about what stylistic conventions I should be following, and I was completely lost about how to design a program outside of the object-oriented paradigm. Thus, I sought out some style guides on the web, of which there are only a couple of commonly used ones (the most popular is the Cornell CS312 style guide [2]), and I read the two discussions that I could find on reconciling object-oriented and functional design [13, 6]. Unfortunately, both of these discussions were extremely abstract, and did not shed any light at all on what situations would lend themselves more to an object-oriented solution and what situations would be handled more easily with a functional program. It did not help that I did not even really know how to design programs that used a purely functional style. Asking questions of people on #ocaml did not help, because essentially everyone I talked to said that they had never used the object-oriented features of OCaml, as they preferred writing programs in functional style. I found this somewhat frustrating, since the inclusion of object-oriented features in OCaml is often used as one of its selling points: However, it seems that very few OCaml programmers use these features, and their use is much more poorly documented and supported than the functional aspects of the language.

This made it difficult to decide on how to proceed, because I needed to know what style of interface I would be using so that I could write tests to specify it as per the TDD style. Thus, I decided to seek out and study some large OCaml projects, in the hopes that the developers of those projects would have found uses for the object-oriented paradigm and that I would be able to infer good ways to use it in concert with functional programming elements by studying their work. To this end, I downloaded and browsed the MLDonkey [4] codebase, which did contain a few object-oriented interfaces. It appeared to me as if these were largely used to write wrappers around external libraries (e.g. GTK), and to encapsulate very large core components of the system (e.g. the server was represented as an object, but connections were not). However, very few classes were defined in general, and the project lacked documentation. The design document provided with the distribution discussed the low-level protocol design details, but provided no justification for their choice of overall software architecture. None of the other projects that I explored used object-orientation anywhere in their implementation.

I thus decided to proceed by using what I already knew – that is, by using an object-oriented design for the game model. I was very uncertain at this point that this would work out, because I wasn't entirely sure about how OCaml's object-oriented features differed from the object-oriented languages that I was used to (Java and C++); however, I felt that even if using the object-oriented features failed, the discussion of how and why I failed would still be of value to mainstream programmers wanting to switch to OCaml.

Design and Implementation of the Tetris Model

Having decided to use an object-oriented design for the Tetris game model, I sought out literature on the design of Tetris, since I did not want to reinvent the wheel by redesigning a game that has been implemented many times over. I found an excellent guide at the Stanford CS library [11] used in their CS1 stream for a Java programming assignment, and I loosely followed their design principles when writing tests to design my interfaces.

TDD with OUnit and the toplevel

Test-driven development involves the evolution of interfaces by iteratively writing unit tests that describe some component of the interface, implementing the defined components, and then running the tests to ensure that the use cases have been implemented correctly. Thus, my first task was to find a unit-testing framework for OCaml that I could use to write my tests.

The only such framework that I could find is the OUnit package [9]. Unfortunately, this framework is somewhat limited compared to the more full-featured ones that I have used previously (JUnit, and some proprietary unit-testing utilities in private companies that I have worked at). When a test fails, the only notification produced by the test suite is the name of the test that failed: In order to determine the line number of the assertion that failed, you must either run the program through the debugger or provide your own output before each assertion. This often involved commenting out some blocks of code, which interacted badly with the sequence-point placement problem discussed in the previous section. This made the TDD process *much* slower than it would normally be, because I was forced to do some diagnosis each time a test failed just to discover what component of the test was failing. This is, in my opinion, a critical flaw with using OUnit for TDD, because the TDD process relies heavily on getting instant feedback from your tests, and any deficiency in the testing framework that delays this feedback significantly reduces the utility of the process. It is entirely possible that I am unaware of some functionality in the package that allows for more useful output, but the documentation for OUnit is extremely sparse, and I was unable to find any mention of such features. One final issue I had with the framework is that it requires you to assemble the test suites by hand: There is no JUnit-like option that allows you to run all tests in a source file by default without writing any code. This is a minor annoyance, but resulted in several instances of confusion where a forgotten test instantiation left me wondering why my most recently defined tests weren't being run.

Fortunately, once the problem had been diagnosed, the toplevel made it quite easy to explore why the errors were occurring. It wasn't too much work to copy and paste the required program text from Vim into the toplevel, and being able to interactively instantiate the TetrisPiece and TetrisBoard objects that I was designing made it very easy to observe their actual behavior relative to what I expected. For this reason, I never found cause to use the debugger in the development of the model, since most runtime errors were easily diagnosed in the toplevel.

Issues in implementing the design

The simple design that I arrived at using TDD and the Stanford CS Library guide involved two classes; a TetrisPiece class to represent a single piece, and a TetrisBoard class that implemented most of the game logic. The TetrisPiece class represented each piece in its own coordinate space with a list of 2-element lists. Also, the Stanford guide describes an auxiliary structure called a 'skirt' that can be associated with a piece to improve the efficiency of computing where a falling piece should land, which I presumed was unnecessary but seemed like it would be fun to implement, so I did that as well. This was also implemented with a list. Again, I encountered 'data-type confusion' here, as I was unsure about which type would be the most appropriate for this purpose. The difficulty arose from the fact that in the object-oriented paradigm, there are a variety of ways in which data structures that are used to maintain state can be mutated; via pattern matching, or by directly assigning to them. In the functional style, there isn't really any notion of state and pattern-matching becomes the most obvious choice for manipulating data; thus, using the object-oriented paradigm and explicitly maintaining state forced me to make several decisions about representation that I felt poorly equipped to make. Indeed, at this point I was still only vaguely aware that OCaml had a tuple data type, and I really didn't understand how it differed from the native lists, arrays, and records that were my other options. My previous experience with Lisp made me fairly comfortable with lists, though, and so I stuck to those in these early stages.

The TetrisBoard was designed as a state machine, and so its interface was simple: There are several 'getter' methods that allow the view to query the state of the board, and a single 'evolve' method that takes user input as a parameter (e.g. rotate piece clockwise, slide piece left, drop piece) and progresses the state of the game by one 'tick'. Several ticks can elapse before a piece descends by one row. This differs significantly from the board design described in the Stanford guide, which exposed many more mutators to the client than I felt comfortable with.

The Tetris board 'grid' was represented as a 2D array of the variant type 'tetris_grid_cell', which can either be Empty or Full. A full grid cell carries an integer ID that can be used to assign 'classes' to squares; A GUI view would use these classes to assign colors to squares, while, say, a terminal-based interface might use them to assign ASCII characters to squares. The current position of the falling piece on the board was represented with a Point record type for clarity; this would later interact badly with my Tetris piece design, which represented coordinates as 2-element lists. I recognized at this juncture that the decision to represent points as lists was probably a mistake, but the record solution resulted in code that was slightly more verbose than I liked in places, so I wasn't quite satisfied with it either. The 'tuple' type could not be used to maintain mutable state because the elements of tuples cannot be mutated. Thus, I left my design as it was. I was frustrated that my implementation wasn't as elegant as OCaml code that I had seen that solved much more complicated problems, but I felt that pushing forward would perhaps give me enough experience to come back and fix things later, especially with the test suites for support.

By the time I had finished developing the Tetris model, I felt that I had a good idea of how to integrate the functional and object-oriented features of OCaml, using a method that I like to call the "fat object" method. Usually in TDD, the resulting program is composed of many small objects that each have a very narrow purpose. The implementation of complex algorithms is often accomplished via interactions between these objects. In OCaml, I found that many of these interactions could be replaced with either a closure, or a pattern-match that operates on some built-in structure. For example, the piece generator is one instance in the Tetris design where I would have used the Strategy design pattern in C/C++ to allow for 'pluggable' generator behaviours, whereas in OCaml I instead

used a closure to provide this behaviour. In the unit tests, I would configure the TetrisBoard object being tested with a mock generator closure, whereas in the game itself a random generator is used.

Thus, objects in OCaml are best used to provide interfaces for elements of your program that are easiest to thing about as concrete entities. Internally, these objects use OCaml's higher-order functions to maintain state and terse pattern matching features to implement algorithms that would otherwise be implemented via interactions between small objects. Externally, the core application "driver" logic (in this case, for the view) can be implemented as a functional program that operates on these fat objects. In this way, the number of interacting entities in the program is kept quite small, which makes for an easier understanding of the interfaces being used.

Interestingly enough, once I had finished this project I found a couple of OCaml-related newsgroup posts that mentioned that others had also adopted this strategy with success in large (proprietary) projects; however, this general design strategy has not to my knowledge been described in a widely public forum as of yet.

Basic software engineering practices in OCaml

The first version of the Tetris model was written without any concern for following basic software engineering practices, such as separating the interface of the model from the implementation, or following common style conventions. Once I was finished the first iteration, however, I took some time to do both of these things so that I might have some idea of how such things would work in a large OCaml software project.

OCaml separates interfaces from implementations by keeping the interface for a module in an OCaml 'mli' file. Multiple implementations of the interface can be specified in individual 'ml' files. While this sounds simple, OCaml's module system is sufficiently complicated that getting even basic module imports to work can be challenging for the neophyte. The documentation of the module system that I was able to find [6] focused discussion on extremely esoteric uses of it, such as restricting interfaces in multiple 'interface views' and combining restricted interfaces to compose new ones. I could not find any information at all on basic usage of OCaml's module system that gave examples on how to separate the interface of a class from its implementation. The few simple examples that I found were for purely functional module interfaces. Again, questions posed on the IRC channel #ocaml went unanswered, as no one there had used the object-oriented features extensively before. None of the example OCaml projects that I had available to me used the OCaml interface separation mechanisms on object-oriented code, either.

After many hours of trying to make the compiler happy, I sought out an alternate source of help. I eventually posted a message on the OCaml Beginner's group, a Yahoo! group that is specifically designed for OCaml newcomers. Within a few hours, someone replied with a concise example of what I was trying to do. The solution was quite simple: If you copy the class implementation code and paste it into the toplevel, the interpreter will generate a signature for the class. One can than simply copy that signature into the mli file and remove the signatures of the instance fields and the private member functions.

Once I had this working, I was quite pleased with the results. The fact that you can automatically generate the interface from the implementation using the toplevel prevented me from having to write a Vim script to generate template interfaces from boilerplate code, which is my usual strategy for writing C++ header files. These interface files also use OCaml's terse type specification syntax to describe the types of the member function inputs and return values. This terseness allows you to place several related class interface specifications compactly into a single module, which helps with readability. I find it much easier to understand interfaces that textually group related elements together: In Java, classes are typically each stored in their own file and grouped by package, while in C++ related classes are often grouped together in a single header, but the verbosity of the language can cause these interfaces to seem cluttered. For instance, the heavily documented mli file for my Tetris model, which consisted of two classes that had a combined total of 19 member functions and 8 auxiliary datatype declarations, was a total of 120 lines in length. As an experiment, I wrote an equivalent C++ header file, and found that it could not reasonably be written in less than 250 lines at 80 columns per line. Also, in C++ one cannot hide the declarations of instance fields of a class without using workarounds such as the pointer-to-implementation idiom, which has a tendency to cause code bloat. Java's mechanism for separating interface from implementation (writing an interface and then having a class implement that interface) still requires the programmer to explicitly specify the dynamic type of the implementation object, which means that some code must be written to configure an application to use different implementations, even if the choice of implementation can be made at compile time. In OCaml, implementations of interfaces can be easily selected at both compile-time and at runtime.

There was one aspect of OCaml's module system that I found troubling, however. The auxiliary datatypes that were declared in the TetrisModel interface had to be redeclared in the TetrisModel implementation, which resulted in the necessary duplication of 10 lines of code. I will admit that after reading some documentation and talking to a couple of experienced OCaml developers, I still do not understand why this duplication is necessary: Apparently it has something to do with the fact that mli files are never actually compiled. While the compiler is extremely good at catching declarations in implementations that do not match interface specifications, I am still wary of instances of forced code duplication.

I also attempted to impose some sort of 'accepted style' on my code by reviewing available OCaml style guides [2], which are primarily designed and used by computer science instructors: The only place in which many of them can be said to differ is in the usage of pattern matching. There seem to be two opinions about the use of pattern matching in OCaml code: The first is to use it absolutely everywhere you can, which pretty much obviates the need for branching constructs like if/else, while the other is to use conditional statements if the expression that you would otherwise be matching on is a boolean expression. I chose to use the latter style because I felt that it led to a clearer expression of intent.

Unfortunately, all of the style guides I could find for OCaml were staggeringly long, which I suspect is because OCaml is such a large language with a cornucopia of syntax elements that are all capable of being abused in one way or another. Unfortunately, none of the style guides provided guidelines for writing object-oriented code, which further contributed to the feeling that the object-oriented features of OCaml are more of a vestigial trait than a core feature of the language.

Language features that were used in the development of the model

There were three major OCaml language features were used in the development of the model that merit attention here: Functional programming, object-oriented programming, and pattern matching.

I was surprised at how easily I was able to implement the core algorithms (e.g. piece intersection) using the functional programming constructs of OCaml, especially the provided libraries. In general, most programs involve operating on basic data types such as lists, and games especially involve frequent manipulation of data structures such as lists and trees. The functional libraries that are provided with OCaml make these sorts of algorithms easy to implement by providing higher-order operations that can be written con-

cisely. I found 'map' and 'fold_left' to be especially useful in implementing algorithms that involved computing properties of Tetris pieces. As a token example, computing the skirt of a piece requires knowing what the smallest *y* value is in the list of points that represent the body of the piece. This is trivially computed as:

```
let list_ind_extract n l = (List.nth l n) in
let smallest_y = List.fold_left min 0
                 (List.map (list_ind_extract 1) body)
```

I am sure that more experienced OCaml programmers could find an even more elegant way of expressing this.

The basic object-oriented features of OCaml were pleasantly easy to learn and apply, although this may be because my design did not involve any complex relationships between objects (e.g. no inheritance). There were a few quirks that required a bit of adjustment – classes can declare only one constructor, for example – but none of these were major obstacles. I have read complaints from some programmers that the inability to make dynamic casts in OCaml severely cripples their ability to write large object-oriented programs in it, and unfortunately this project did not give me the experience I needed to be able to support or refute those claims. However, I suspect that by combining the 'fat object' design strategy with other OCaml features such as parametric polymorphism, one may be able to still use object-oriented features to one's advantage, albeit in a way that is somewhat different than one would use objects in a 'mainstream' OO language such as Java.

Pattern matching is a feature of OCaml that I still require a lot more practice with. At first the construct confused me, as the simultaneous labeling and deconstruction of types made me wonder what its primary purpose was. Thus, I initially used pattern matching as a high-octane 'switch' statement, until I was able to read more examples of its usage and get a better feeling for how it could be used to manipulate data instead of just being used to make decisions. However, all of my uses of pattern matching seemed distinctly less elegant than the examples that I learned from, and so this is why I say I feel that I need more practice in order to understand when to use it, and how to structure my data so that it can be applied most effectively.

Was I more productive?

Writing the model took me approximately 12 sessions, which occupied roughly 60 hours of time. I expect that this is about two times as long as it would have taken me write a similar implementation in C++ or Java. The 60 hours includes the time I spent setting up my environment, learning about the language and about functional programming in general, and writing support scripts for OCaml development. OCaml enthusiasts frequently claim that switching from an "inferior" language such as C++ wastes no time, arguing that these developers can simultaneously learn OCaml and write a working program just as quickly as they could write the same program in a language that they are familiar with. I cannot say that I experienced this conservation of efficiency myself, but I am actually quite pleased with the amount of time I spent implementing this part of the project. At the outset, I expected things to take much, much longer. I suspect that I will at least be on par with my Java/C++ development time in OCaml after another couple of medium-sized project implementations.

The test library that I wrote to specify the Tetris model is currently composed of 2 suites that contain a total of 25 unit tests. 17 of these tests passed as soon as I implemented their target functionality, 6 failed, and 2 passed but caused failures in other tests. Thus, about 3/4 of the unit tests passed on the first try. The tests that failed were of two general types. The first type of failure involved silly mistakes with array accesses (i.e. fencepost errors), and were trivial to fix. The second type of failure was more interesting, however; these occurred while I was trying to incrementally develop the algorithm to handle piece "landings".

In TDD, core algorithms are usually developed incrementally, just like other components of the software. Unit tests are written for base cases first, and a minimal amount of code is written to make those tests pass. Then, more complicated cases are specified in tests, and the algorithm is "refactored" or extended to handle these cases. This process continues until you feel that you have developed an algorithm that is of sufficient generality to handle all possible inputs. I initially used this method to write the "landing" algorithm, and found that I was having to rewrite significant portions of the algorithm with each iteration. So, I decided to write and implement the general-case algorithm in a single iteration instead. The algorithm that I wrote "on paper" was quite easy to map directly to OCaml syntax, especially because I was able to define all of the required data structures very concisely. All of the tests passed on the first run.

This experience led me to reassess how TDD should be used in an OCaml project. I still think that the process is a useful one. Certainly, the test code provides good documentation for the project, and good test coverage helps to increase confidence in the quality of the product being developed. However, I feel that much more of the design can be done "up-front" in an OCaml project, which allows one to take much larger steps when planning iterations and writing new test cases. This is interesting, because I assumed that the "behaviour" of the methodology would be largely independent of the implementation language being used. I intend to explore this issue further in subsequent OCaml projects, as I don't feel that I can make any more concrete statements about it without more experience with the language.

Design and implementation of the Tetris view

Compared to the development of the Tetris model, the implementation of the view was a much shorter affair. I initially intended to add more features before the project deadline, but time constraints (and the reality of having to complete two other projects this term) prevented me from providing anything more than basic functionality in the interface. In this section, I describe the tools I used to develop the view and its basic design, and I close with another short discussion about productivity.

Tools used to develop the view

The primary decision that had to be made before implementing the view was what graphics framework to use. Considering that OCaml is not a widely used language, I was confronted with a surprising variety of options. The OCaml standard library contains a graphics module that provides very primitive drawing tools that can be used to render simple objects, but the provided functionality was sufficiently sparse that I did not feel comfortable using it. A second possibility was the lablgtk port of GTK to OCaml, which is a full-featured framework for building graphical user interfaces. The framework is cross-platform, although Mac OS/X executables must be run through X11, which is awkward for deployment. Also, the framework is quite complex, and it felt like overkill for what I was building. The third option that I considered was the lablgl framework, which is a set of bindings to OpenGL for OCaml. I have previously implemented games in C++ using OpenGL, and so this felt like the natural choice.

The interface for the lablgl framework is extremely similar to the OpenGL interface in C++, and so the learning curve was minimal. There were a couple of bugs that I had to work around, however, including a major one that prevented me from specifying handlers for both normal keys and 'special' meta keys. I was unable to find any documentation of this bug anywhere, so I don't know if it was

something specific about what I was doing that was causing it, or if it is a genuine flaw in the implementation of the framework. Time constraints prevented me from investigating this further, and I was able to devise a workaround by altering the controls to use only normal characters, which conveniently resulted in a better control layout as well.

Design of the view

Because I was unsure of how stable the lablgl framework was, I decided to throw together the view in a rather haphazard fashion, so that not much time would be wasted in the event that I had to switch frameworks partway through because of instability. Thus, I kept the design very simple. I wrote a series of rendering functions, each of which is responsible for rendering the game in a specific 'mode' (e.g. title screen, normal gameplay, game over screen). The code to switch the view 'modes' and to draw the core user-interface components is all located in the UI module, which is a poor separation of responsibilities; however, by hacking things in this way I was able to complete the view implementation in less than four hours.

Because of time constraints, the refactoring was never done. I plan on eventually having the entire project code-reviewed by some other members of the OCaml community, so a re-engineering of the project will take place before then.

Another discussion about productivity

I decided not to use test-driven development for the UI, because I have not personally found TDD to be very useful for UI testing in the past. I have already mentioned that I wrote a rather coarse implementation of the view in less than four hours, which was far faster than I expected to complete it. The drawing code took about as long to write in OCaml as I would expect it to take in C++, and was of about the same length. This is because OpenGL code is somewhat incompressible, in that it involves rendering polygons as series of points. The ability to easily specify literal lists and iterate over them with List.iter made some of the drawing code somewhat easier to read, however.

The lablgl framework makes heavy use of labeled arguments to functions. While this makes reading code that uses these functions much more readable, and allows for currying functions in any argument order, I found that using labeled arguments led to frequent typos and thus frequent recompilations. Editor features such as word completion tend not to help much here because the argument labels are typically quite short (1-3 characters), and so it is very hard for the editor to 'guess' what word it is that you are trying to complete. This simple detail slowed me down more than I care to think about.

I experienced the biggest disappointment of this project in the process of implementing the view. After hooking the view up to the model, I discovered a bug that would only occur after many minutes of playing, in which entire columns would be inexplicably filled after a piece landed. A cursory initial analysis could not determine what caused the bug or even under what general conditions it would occur. Thus, I decided that this would be the perfect opportunity to use the backstepping features of the debugger.

The OCaml debugger feels a lot like GDB, so it wasn't too difficult to figure out how it works. I set a breakpoint at the line of the program that signaled a 'game over', which was almost always the result when the bug was triggered. I then attempted to step backwards, excited at the prospect of finally being able to use a time-traveling debugger. Unfortunately, the only result was a series of error messages about not being able to find the last savepoint. After spending an hour or so trying to figure out the cause of this error, I wasn't able to discover anything definitive. Asking about the problem in public forums was futile, as very few people use the OCaml debugger in practice. I suspect that the cause of the error was that the lablgl framework is actually implemented as a set of bindings to a C OpenGL library, and I wasn't able to get an answer about how the debugger reacts when introduced to foreign code. The bug ended up being enormously difficult to diagnose without the debugger, and it in fact still crops up occasionally in the current version. I am not sure whether this implies that runtime bugs are inherently difficult to diagnose in OCaml, or if it is merely my inexperience that contributed to this difficulty.

4 Conclusion

Despite the many difficulties that I encountered throughout this project, I am confident that OCaml is sufficiently robust for my personal needs, and that it will likely usurp C++ as my "performance language" of choice. Given the community and library support of the language, I could also see it being used by small groups of developers, such as in a startup environment, although this would likely depend on the maturity of other OCaml components that I have not had the opportunity to test – the concurrency libraries, for instance. However, I think it is unrealistic to assume that OCaml could replace more entrenched languages such as C++ and Java in large organizations for a variety of very obvious reasons, the main one being the extent of the retraining that would be involved in such an enterprise. I am also not convinced that OCaml would really result in a profound increase in productivity on projects with large numbers of developers, as the problems that cause those projects to progress slowly are generally more a result of sociological problems than technological ones.

My experience with test-driven development in OCaml suggests that it still a useful methodology to use when developing OCaml programs; however, I would encourage users of this methodology in OCaml to train themselves to take larger 'steps' and to use fewer iterations, especially when writing heavily algorithmic components of the program. Often times the combination of OCaml's ability to easily translate pseudocode directly to code and the strength of the static checking makes implementing algorithms less error-prone in OCaml than in more common languages – at least, I have found this to be true in my limited experience.

For those 'mainstream' developers who do think that OCaml is worth trying, but are unsure of how to proceed, my experience suggests the following adoption policy: Stick with what you're good at, but keep an open mind. That is, do not worry so much about how to design a functional program, but instead approach the problem in OCaml as you would in an object-oriented language. Use test-driven development to specify interfaces, if you are used to using that methodology. As you develop your program, try to familiarize yourself with the functional aspects of the language by applying them to builtin types when implementing algorithms using the standard library. Eventually, I believe that you will begin to recognize more and more situations where functionality that you are providing via objects can be implemented using some other more lightweight construct (e.g. using a closure instead of an object to act as a Strategy). This will lead you to the "fat object" pattern that I described above. I suspect that continued exposure to OCaml will eventually induce developers to design programs that are written almost entirely in the functional style, if only because these aspects of the language are the most mature and the most widely used. I am personally beginning to find that writing in the functional style leads to far fewer bugs, simply because most of the program data is immutable and thus there is little room for errors involving inconsistent state.

One interesting application of the adoption strategy described here would be in teaching functional programming to students trained in object-oriented languages. Students could write a couple of assignments in the object-oriented style in order to become used to OCaml's syntax, and to develop some experience with higher-

order functions through interactions with the standard library. Later assignments could gradually begin to focus more strongly on functional programming. I know that OCaml is used extensively in Europe (especially France) as an teaching language; However, as far as I am aware, these courses tend to use the functional constructs of OCaml exclusively. I think it would be a worthwhile exercise to leverage OCaml's object-oriented features for pedagogical purposes.

Future work

I have made it quite obvious in this report that I was not able to quite finish the project in the time allotted to me. I do intend to continue work on it, however. I think it would be a worthwhile exercise to rewrite the entire program again in an attempt to produce a more elegant solution, using the experience that I gained in developing the first prototype. I would then like to have this code reviewed by some experienced OCaml developers: A couple of generous souls that I met on #ocaml have expressed the willingness to do so.

Also, I am investigating the possibility of extending OUnit to make it easier to use in test-driven development. It is possible that the functionality I desire is actually provided by the library, in which case I will simply contribute documentation to the project. Finally, on an unrelated note, I have looked into resuming development of the BioCaml project, which was started by a couple of developers in Japan but discontinued in 2004. I find that working in OCaml is entertaining enough that I would like to devote some of my spare time to contributing to these public OCaml projects. It is my hope that readers of this report will be sufficiently motivated by my minor successes (and sufficiently undeterred by my many failures) to attempt an OCaml project of their own, and to develop a similar appreciation for programming in this quirky but beautiful language.

REFERENCES

[1] Camelia: An ide for ocaml. `http://camelia.sourceforge.net/`.

[2] The cornell cs312 ml style guide. `http://www.cs.cornell.edu/courses/cs312/2002fa/handouts/style.htm`.

[3] The eclipse open development platform. `http://www.eclipse.org/`.

[4] The mldonkey project page. `http://mldonkey.sourceforge.net`.

[5] The ocaml hump. `http://caml.inria.fr/cgi-bin/hump.cgi`.

[6] The ocaml manual. `http://caml.inria.fr/pub/docs/manual-ocaml`.

[7] The ocaml tutorial. `http://www.ocaml-tutorial.org`.

[8] The omake build system. `http://omake.metaprl.org`.

[9] Ounit, a unit test framework for ocaml. `http://www.xs4all.nl/~mmzeeman/ocaml/`.

[10] The scons build system. `http://www.scons.org`.

[11] The stanford cs library tetris design guide. `http://cslibrary.stanford.edu/112/`.

[12] Robert Bridson. Spaceman spiff in escape from zorg. `http://www.cs.ubc.ca/~rbridson/personal/spiff/`.

[13] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications with Objective Caml*. O'Reilly, 2000.

[14] Francois Lessaux. Ocamldoom: Ml for 3d action games. In *Workshop ML*, 1998.

[15] Kenneth Louden. *Programming Languages: Principles and Practice, Second Edition.* Thomson Brooks/Cole, 2003.

[16] Joshua Smith. *Practical OCaml*. APress, 2005.