ALGORITHM ANALYSIS FOR RETURNING ADULT STUDENTS*

Kelvin Sung
Computing and Software Systems
University of Washington, Bothell
Bothell, WA 98011
ksung@u.washington.edu

Peter Shirley
School of Computing
University of Utah
Salt Lake City, UT 84112
shirley@cs.utah.edu

ABSTRACT

Returning adult students typically have expectations of gaining both near-term applicable skill sets and long-term foundational concepts from their courses. Most Algorithm Analysis classes are designed for traditional students and do not have teaching practical skill set as one of their major goals. This paper describes an attempt to meet the expectations of both applicable skills and foundational concepts in an Algorithm Analysis class. The class emphasizes implementation and experimentation while covering concepts supporting students' future self-learning. We also present our approach to support the teaching of this highly demanding course to students with busy life schedules.

Categories and Subject Descriptors
K.3.2 [**Computer and Information Science Education**]: Computer Science Education, Curriculum.

Keywords
Adult Students, Non-traditional Students, Algorithm Analysis.

1. INTRODUCTION

Recognizing the importance of education in the current information-driven economy, record numbers of working adults are returning to pursue college degrees [1,2]. These students are focused, motivated, and have demanding life schedules. They

typically have somewhat rusty mathematical skills. Because of adult students' maturity, work experience, and pragmatic needs, they demand near-term applicability of skills in addition to foundational concepts from their courses [3].

The study of algorithms is the cornerstone of Computer Science (CS) [4]. It is essential that CS students have the ability to select and apply algorithms appropriate to particular situations [5]. Many existing Algorithm Analysis courses (e.g. [6, 7]) follow the classical textbooks (e.g. [8, 9]) which emphasize on studying the foundational mathematical concepts (e.g. [10]) and abstract problem solving (e.g. [11]). These courses are designed specifically for traditional university students. It is expected that the foundational concepts will mature as the students gain experience in life and in the CS field. Near-term applicability of the knowledge covered is typically not a specific goal of these courses.

Significant mismatch in students' expectations and course contents can result in disengaged and discontented students which in turn can impede learning [12]. Courses designed for adult students should draw from the students' life experience and strengths in real-world problem solving. In alignment with these observations, we have recently re-designed the Algorithm Analysis course at the University of Washington, Bothell to incorporate considerable system implementations and run-time analysis of software programs. The hands-on components of our course are designed such that implementations are tightly coupled with detailed run-time complexity analysis. In this way students gain practical skills in software development while at the same time apply and reinforce the concepts learned in the lectures on real implementations.

Section 2 of this paper presents the guiding principals for our redesign effort. Section 3 describes the details of the redesigned course concentrating on the efforts and experiences in integrating the practical components. Section 4 discusses our experience in teaching/working with adult students and some of our efforts in supporting the students. Section 5 summarizes the paper with a discussion of what are and are not goals of this course.

2. BACKGROUND

A primary mission of the University of Washington, Bothell (UWB) is to serve *time and space*[1] bound adult students [13]. The Computing and Software Systems (CSS) degree program [14] at UWB is designed with the understanding that successful adult degree programs are not simple reorganizations of traditional undergraduate classes [15]. We have a continuous and concerted effort to re-examine pedagogical and technical approaches to teaching computer software design and development.

The current version of our Algorithm Analysis course was completely re-designed in early 2000 [16]. After careful articulation of faculty pedagogical philosophy and considerations based on student feedback, the re-design effort was governed by three underlying principals:

[1] Students with busy life schedules, who must work full/part time, and cannot afford to relocate to on-campus-housing during the course of their University education.

1. Practicality of Hands-on Exercises - It is important to demonstrate to students that the knowledge learned is not merely *on-paper-mathematical-analysis*. The concepts learned in the lectures should be reinforced in two ways: programming assignments and analysis of given software systems.

Programming Assignments: design exercise for students to develop software implementing real-world solutions studied in lectures, either based on well understood or newly derived algorithms. Students must demonstrate the feasibility of their systems with real-world (i.e. very large) data sets. In addition, based on the mathematical tools learned in lectures, students must analyze the run-time of their own implementations and criticize/discuss any inefficiency.

Analysis of Given Software Systems: design exercise for students to analyze/study the efficiency of some given software systems. Students must derive the run-time complexity of the given software systems, and articulate/predict run-times for any given data sizes. In addition, based on the concepts learned in lectures, students must discuss the accuracy/reliability of their predictions. These exercises are similar in spirit to Sander's "*Teaching Empirical Analysis of Algorithms*" [17]. However, instead of organizing the entire course based on developing formal empirical experimentation, we advocate a balance between synthesis and analysis based on the tools and concepts learned in lectures.

2. Relevancy of Examples - It is important for students to appreciate that in addition to analyzing simple and familiar algorithms, the mathematical tools learned can be used to evaluate practical algorithms in areas that are relevant in their daily life. In this way, straightforward practical examples would be preferred over powerful abstraction of general classes of problems.

Goodrich and Tamassia described how the topics in algorithm classes can be organized and presented based on popular and important applications on the Internet [18]. Here we are advocating that the specific applications used to exemplify each topic area should be straightforward and practical.

3. Foundation For Future Self-Learning - It is important that students who have completed the class have learned sufficient fundamentals to continue gaining knowledge in the area. It is reassuring that, based on student feedback, even the most *practical-minded* students agree that the coverage of sufficient theory is important for their future career development. Theory coverage must be thorough and more importantly, the distinctions of concepts from the above heavy implementation and experimentation efforts must be cleared.

These principals serve as guidelines for designing the syllabus of our course. They do not affect the topics covered or the actual methods for teaching the lectures. For example, specific algorithmic topics can still be taught with existing powerful teaching tools (e.g. [19, 20]), or with fascinating/creative approaches (e.g. [21, 22]). The above guidelines lead to emphasis on practical hands-on implementation and examples for each topic covered.

3. THE ALGORITHM ANALYSIS CLASS

The organization of the topics in our course follows that of standard algorithm textbooks[2]. The schedule of our 10 week Quarter is divided into the coverage of three general topic areas: **Analysis Tools, Applying the Analysis Tools,** and **Algorithm Design**. Our presentation of each of these topic areas is influenced by the above guidelines and involves significant implementation and hands-on experimentation. In the rest of this section, our approach in covering these topics is briefly discussed with details on our strategies in designing hands-on exercise for students to practice and reinforce the concepts involved.

Analysis Tools. Following the standard textbooks (e.g. [9]), we cover the traditional algorithm analysis framework where our discussions centered on *applying* the tools. We explicitly avoid *deriving* the theories behind the tools. For example, when teaching *Master's Theorem*, we concentrate on conditions upon which the theorem can be applied and avoid the details of the proof entirely. Instead, the time is spent on extra examples and assignments that allow students practice using the tools.

The programming assignment for this stage of the course would be developing software that process large data sets based on familiar data structures and algorithms. As part of the assignment, students are required to hand-in design documents where they must apply the mathematical tools learned in lectures and show the detailed analysis of the run-time complexity of their implementations. Since the assignment is based on *familiar data structures and algorithms*, students can be confident in their implementation and concentrate on practicing the analysis tools. The *processing large data set* requirement leads to relatively longer turn-around time (minutes to 10s of minutes) during debugging. This slower response time enforces students to follow good coding practices and careful attentions to subtleties during implementations. One example assignment would be implementing a simple memory management system with garbage collection based on linked lists build on a one-dimensional integer array. Students must demonstrate the feasibility of their implementations by managing hundreds of millions of memory units, while handling millions of allocation requests. When presenting the analysis of the run-time complexity, students would become acutely aware of the importance of pseudo code abstraction. This is especially the case for the array based linked list implementation, where the algorithms are straightforward and yet the implementations are often complicated and tedious.

Interesting surprises often arise during the final testing phase of the programming assignments. For example, there are numerous occasions where students' systems would operate perfectly with moderate input data size (tens of thousands of transactions), and yet "hang" with real large input data set. One common problem is $O(n)$ implementations of $O(1)$ operations. For example, for linked list insertion, in an attempt to preserve the order and avoid the extra memory for a tail pointer, students would traverse the entire linked list for each insertion, effectively rendered the building of a n-integer linked list an $O(n^2)$ operation. Implementing straightforward algorithms with careful run-time

[2] We use [9] as our textbook, with extra handout for examples.

complexity analysis coupled with large input data set testing help students become aware of subtle and yet important efficiency issues.

Application of Analysis Tools. In addition to illustrating the usefulness of the mathematical tools in analyzing familiar algorithms, it is desirable to further demonstrate/practice the effectiveness of the tool-set on new/useful/relevant algorithms. As discussed in [21], new/extra example algorithms for practicing the analysis tools often involve the introduction of new application areas which can be time consuming. However, the extra time can be justified if there are a rich variety of algorithms involved, and if additional hands-on practicing experience can be gained. For example, we dedicate lecture time to learn a new application area, and to develop the new algorithms involved. In programming assignments, students would implement the algorithms. At this point with experience from the previous assignment, students would have some competencies with the newly acquired mathematical tools. They would further practice using the tools in the design documentations associated with the programming assignments by presenting the details of the run-time analysis of their implementations. The coding of a new algorithm and the associated detailed run-time analysis help students appreciate the subtle distinction between high-level algorithm/solution development and the actual tedious/detail implementation efforts.

In our class, we use 2-dmimensional spatial searches as our example application area. The reasons for our choice are: this is a relatively straightforward application area to introduce; there are a healthy variety of different algorithms involved including iterative, recursive, adaptive, etc.; there are many practical applications that students can relate to (e.g. selecting a city on an interactive map). To maximize the opportunities for hands-on experience, we have designed two programming assignments around this topic. In the first assignment, students must implement a system that first accepts large number of input points in a 2D space. Their system must then allow a user to interactively select subsets from these points based on specifying 2D ranges. This assignment implements uniform spatial subdivision where most algorithms are iterative in nature. Once again, students must demonstrate the feasibility of their implementation by processing large data sets (e.g. tens of millions). With careful analysis and testing, students can expect/experience first-hand how rapidly the performance of their systems would degenerate with unfavorable input data distribution (highly clustered point sets). This leads to the introduction of adaptive spatial subdivision where most algorithms are recursive in nature. To help motivate students, the second programming assignment is disguised as a 2D interactive "space invader game". The aim of the game is to shoot cannons to destroy alien objects in a 2D space. A functional game with well defined API is provided to the students. In this provided system, collision detection is based on linear searches and the performance degrades rapidly as the number of cannon/alien object increases. Students are required to replace the linear search by an adaptive search strategy. Since the involved algorithms are recursive, students practice deriving and solving recurrence relations when analyzing the run-time complexity in their design documentation.

An **exercise on software system analysis** is assigned after the above topics are covered. This exercise is modeled after physical science experiments. Two software systems implementing different sorting algorithms are given to students where, through

experimentation with different input data sizes, they must: derive mathematical models describing the run-time behavior of the systems; validate their models by predicting and verifying run-times; criticize their models by discussing the accuracy and usefulness of their results. To emphasize the importance of their results, the assignment includes questions where students must make informed decisions or otherwise pay the price of waiting for the results (sometimes for a very long time). For example: "*according to your run-time models for each of the sorting programs, please predict how much time it will take each program to sort 1 million entries. For each program, if the predicted run-time is less than 5 minutes then please run the program and compare the actual run-time to your prediction and comment on the difference. If your prediction is more than 5 minutes, then please predict how many entries the sorting program can process in 5 minutes. Please compare your prediction to the actual run-time and comment on the difference*." Of course, with today's machines, *O(nlgn)* algorithms will return in seconds, while $O(n^2)$ algorithms will take many hundreds of minutes to process 1 million entries.

Algorithm Design. Following the standard algorithm textbooks (e.g. [9]) we cover the traditional algorithm design strategies: divide-and-conquer, greedy algorithms, dynamic programming, etc. We introduce these topics with straightforward and pragmatic examples. For example, we introduce greedy strategy by developing solutions for the supermarket change machine (minimum coins). We then continue to discuss the conditions on the currency denomination that the algorithm assumes (different coin values differ by at least twice the amount), and move on to develop solutions when this assumption is violated. For example, minimum number of coins for $12 with currency denomination of {$1, $6, $7}: greedy algorithm will return an answer of 6 coins with {$7, $1, $1, $1, $1, $1}; while we know the correct answer should be 2 coins with {$6, $6}. This of course, is a simple example of dynamic programming. We continue to discuss these topics using 0-1 and fractional knapsack problems as examples. The solutions to the more complicated problems (e.g. the matrix chain multiplication problem) are discussed only after students have developed some appreciations for the topic areas.

With the above syllabus, the intended student learning outcome of this class is similar to that as specified for CS210 on Page 205 of *Computing Curricula 2001* [5]. Students who completed our class should: understand the mathematical framework for analyzing complexity of algorithms; be able to select and apply algorithms appropriate to a particular situation. However, due to the heavy emphasis on implementation and the relatively short 10-week Quarter, our goals in the last topic area (algorithm design) are more modest. Instead of the ability to select and apply algorithm design strategies in formulating new solutions for complex problems, our goal is for students to understand/appreciate/evaluate existing solutions formulated based on the different algorithm design strategies. On the other hand, students that completed our algorithm class would have considerably more hands-on experience with implementing algorithms and with analyzing the run-time of physical programs.

4. MEETING THE NEEDS OF STUDENTS

Because of adult students' high motivation and serious learning attitude, teaching and working with them have been extremely rewarding. It is refreshing to work with students who care more about what they are learning than what grades they are getting.

However, teaching adult students can also be challenging in many ways. For example, when a student said that there is "*a personal emergency*," she might mean one of the kids is sick; or when another student said "*I am too busy to work on the assignment*," she actually meant that there is an important deadline at work and she is working 20 extra hours and she actually sincerely *does not have time*. Extra and often unconventional efforts are required to support these students well.

At the institutional level, a significant portion of our classes are scheduled in the evening to avoid interfering with students' work schedule. To minimize students' commute to campus, we have reorganized lecture meetings from five 50-minute sessions to two 2-hour-5-minute sessions. In addition, much of the campus administration and student affairs support office hours are scheduled in the evening.

Individually, in our classes it is important to:

- Predict students' questions and provide answers/hints on-line before the questions are asked. Most students honestly *do not have time* to examine assignments closely until weekends. It is important that the answers to simple/frequently-asked questions are available without any delays.
- Provide off-hour access. Evening or late-night office hour appointments (9pm) are common practices. This also includes answering emails during the weekends.
- Schedule assignment due times strategically. Middle of the week deadlines usually mean students must take vacation days to finish their homework. Early in the week deadlines usually mean large numbers of email correspondence for the faculty member during the weekend before the deadline.
- Clearly and explicitly explain the difference between implementation and foundational concepts. This is especially important for classes with heavy emphasis on hands-on experiments. Inevitably, we hear comments like, "*I know this algorithm in Java*" or "*I know how to analyze that in C++*." To avoid/remedy this situation, different programming assignments in our class are in different programming languages (either C++ or Java).
- Adapt to students' skills and be flexible. For example, in the earlier offerings of the course, we often observe correct description and derivation of recurrence relations for given algorithms but, because of the rusty mathematical skills, students often become entangled during the mechanical algebraic simplifications. This observation leads to shifting of lecture time to fewer examples with much more detailed illustration of simplification procedures. In the beginning we allow a "cheat-sheet" for the exams and we soon realize most students were copying trivial algebraic identities. To help students focus their valuable time on learning the concepts, we adapted by allowing open book/note exams.

5. DISCUSSIONS

Ultimately, our aim is to educate competent software professionals. In our Algorithm Analysis class, we want students to walk away with a set of ready to use tools for discussing/evaluating/implementing algorithms. More importantly, we want them to have the ability to understand and learn more tools when desired. Our goal is to achieve

a delicate balance between near-term applicable skill sets and foundational concepts. Comparing to most existing Algorithm Analysis classes, a conscious decision has been made to trade off theoretical foundation for practical hands-on experimentation. As a direct consequence, we expect knowledge from this class to have direct/positive/immediate impact on how students develop/evaluate software systems.

6. ACKNOWLEDGMENTS

Thanks to all CSS443 students for working so hard with the first author on constantly providing feedback, and to Charles Jackels for his encouragements and support.

REFERENCES

[1] "Serving Adult Learners in Higher Education: Principles of Effectiveness," Executive Summary, Chicago: Council for Adult and Experiential Learning, 2000 (pdf document available at http://www.cael.org).

[2] Tonkin, H. "Continuing education's time of promise," *Continuing Higher Education Review*, No. 62, PP. 40-56, 1998.

[3] Barbara Boucher Owens, Gene Bailey, Ted Mims, Shell Heller, and Laurie White, "The Non-Traditional Student in Computing: Characteristics, Needs and Experiences," *27th ACM SIGCSE Technical Symposium on Computer Science Education*, 1996, PP. 372-373.

[4] Anany Levitin, "Do We Teach the Right Algorithm Design Techniques?", *Proceedings of the 30th ACM SIGCSE Technical Symposium on Computer Science Education*, 2000, PP 179-183.

[5] "Computing Curricula 2001 Computer Science," Final Report, The Joint Task Force on Computing Curricula, IEEE Computer Society and ACM, Final Report, December 2001.

[6] CS560: Algorithms and Their Analysis, Department of Computer Science, San Diego State University, http://www.cs.sdsu.edu/, September 2003.

[7] CS157: Design and Analysis of Algorithms, Department of Computer Science, Brown University, http://www.cs.browh.edu/courses/cs157, September 2003.

[8] Sara Baase, and Allen Van Gelder, *Computer Algorithms Introduction to Design & Analysis*, Third Edition, Addison Wesley, 2000.

[9] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, *Introduction to Algorithms*, Second Edition, McGraw Hill, 2001.

[10] Anany Levitin, "Design and Analysis of Algorithms Reconsidered," *Proceedings of the 31st ACM SIGCSE Technical Symposium on Computer Science Education*, 2000, PP 16-20.

[11] Daniel Liang, "Teaching Dynamic Programming Techniques Using Permutation Graphics," *Proceedings of the 26th ACM SIGCSE Technical Symposium on Computer Science Education*, 1995, PP 56-60.

[12] Magolda, M. B. B. (Ed.). Teaching to promote intellectual and personal maturity: Incorporating students' worldviews and identities into the learning process. *New Directions for Teaching and Learning*, No. 82. San Francisco: Jossey-Bass, 2000.

[13] Home Page, University of Washington, Bothell, http://bothell.washington.edu, September 2003.

[14] Home Page, Computing and Software Systems, University of Washington, Bothell, http://bothell.washington.edu/CSS, September 2003.

[15] "Adult Degree Programs: Quality Issues, Problem Areas, and Action Steps," Council for Adult and Experiential Learning and the American Council on Education, March 1993 (pdf document available at http://www.cael.org).

[16] CSS433: Advanced Programming Methodologies. Computing and Software Systems, University of Washington, Bothell. http://courses.washington.edu/css443, September 2003.

[17] Ian Sanders, "Teaching Empirical Analysis of Algorithms," *Proceedings of the 33rd ACM SIGCSE Technical Symposium on Computer Science Education*, 2002, PP 321-325.

[18] Michael Goodrich, and Roberto Tamassia, "Teaching Internet Algorithm," *Proceedings of the 32nd ACM SIGCSE Technical Symposium on Computer Science Education*, 2001, PP 129-133.

[19] Michael Goodrich, and Roberto Tamassia, "Teaching the Analysis of Algorithm with Visual Proofs," *Proceedings of the 29th ACM SIGCSE Technical Symposium on Computer Science Education*, 1998, PP 207-211.

[20] Arturo Concepcion, Lawrence Cummins, Ernest Moran, and Man Do, "Algorithma 98: An Algorithm Animation Project," *Proceedings of the 30th ACM SIGCSE Technical Symposium on Computer Science Education*, 1999, PP 301-305.

[21] Darrah Chavey, "Songs and the Analysis of Algorithms," *Proceedings of the 27th ACM SIGCSE Technical Symposium on Computer Science Education*, 1996, PP 4-8.

[22] Patrick Heck, "Dynamic Programming for Pennies a Day," *Proceedings of the 25th ACM SIGCSE Technical Symposium on Computer Science Education*, 1994, PP 213-217.

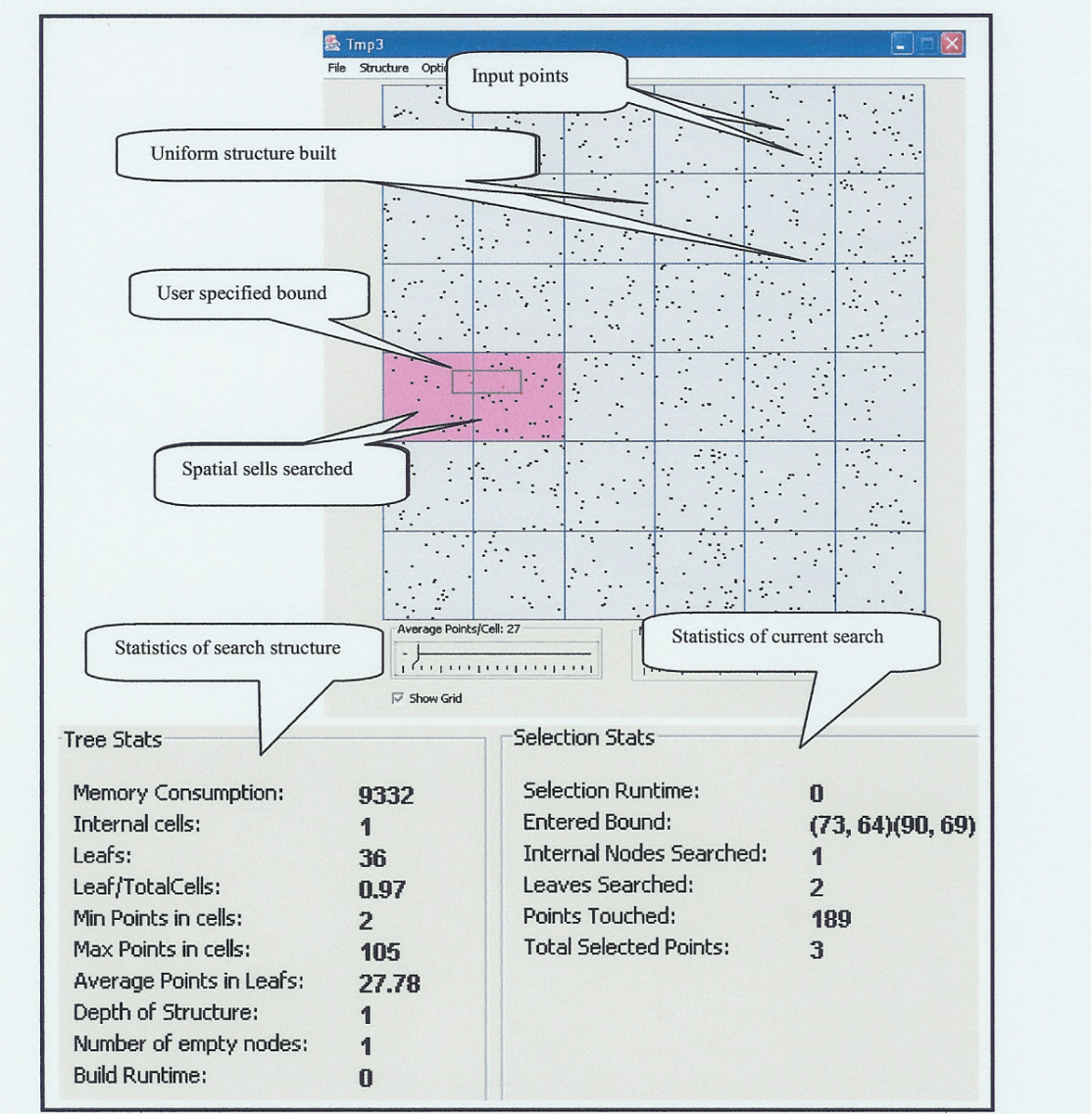

Figure 1. Uniform Spatial Search (Graphical User Interface Version)

APPENDIX: EXAMPLE ASSIGNMENTS

Figure 1 shows the screen shot of an example programming assignment based on uniform spatial search. This assignment requires students implement their system to support both Graphical User (GUI) and Console Interfaces. GUI is important during debugging phase where students can examine the graphical output of their algorithms, while console interface is more practical to support very large data set (10's of million of search range with 10's of millions of points). Notice this assignment is implemented in Java and Swing GUI library.

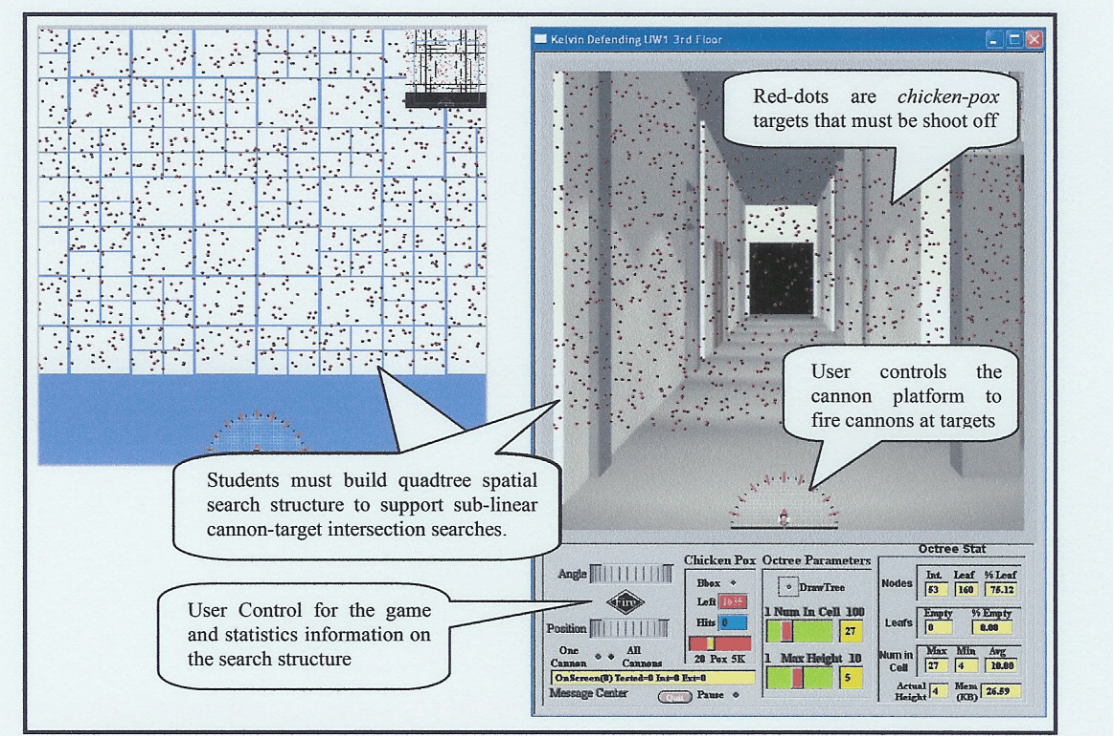

Figure 2. The Space (Chicken-Pox) Invader Game on Non-Uniform Spatial Subdivision

Figure 2 shows the space invader (chicken-pox) game on Non-Uniform Spatial Subdivision. As described, in this assignment a functional game with well defined API is provided where collision detection is based on linear searches. Students must implement a non-uniform search structure to support sub-linear intersection searches. Simple API calls are provided for students to draw their tree structure. This assignment is in C++. The graphics and GUI support are opaque to students.