Be a Computer Scientist for a Week The McGill "Game Programming Guru" Summer Camp

Alexandre Denault, Jörg Kienzle, and Joseph Vybihal
McGill University, adenau@cs.mcgill.ca, joerg@cs.mcgill.ca, jvybihal@cs.mcgill.ca

Abstract - **Motivating high school students to consider Computer Science as their future field of study at the university level is a challenging endeavor. This paper describes the McGill Computer Science Summer Camp targeted at high school students from grade 10 to 11 (ages 15 to 17). We first motivate our choice of using computer games as the main camp theme, and then present the teaching methodology used throughout the camp. A day-by-day breakdown of the camp is provided, as to better illustrate the distribution of the material throughout the week and the evaluation methods used to track the progress of the students. We also present the game environment we developed in which the students exercise their problem solving skills during the lab sessions. We conclude by illustrating the positive effect of the camp, using a combination of code analysis and evaluation questionnaire filled out by the students and their parents.**

Index Terms - About four, alphabetical order, key words or phrases, separated by commas (for suggestions: Camera-ready, FIE format, Preparation of papers, Two-column format).

I. Introduction

In general, high school students do not get exposed to the broad variety of specialized research areas that Computer Science offers and that are available to students after they complete the first two years of undergraduate classes. Often, Computer Science is mistaken to be focused solely on programming, which puts our field into a completely wrong light. As a result, Computer Science programs at the university level are often overlooked, or confused with other more programming-oriented degrees. It also happens that high school students who have not on their own developed an interest in computers do not choose the appropriate optional courses that allow them later on to pursue a major degree in Computer Science or Software Engineering. With the idea of changing that situation and attracting bright students towards science and Computer Science in particular, the School of Computer Science at McGill University began to organize, starting Summer 2005, a Computer Science Summer Camp targeted at high school students from grade 10 to 11 (ages 15 to 17). In the camp, the students take on the role of the computer scientist and are presented with several problem solving challenges. Along the way, they are introduced to various Computer Science fields, such as algorithms, graphics, physics, simulation and artificial intelligence.

II. Computer Game Background

Many young people are fascinated by computer games. This often translates into a desire to develop their own games. It is only natural to exploit this enthusiasm, to motivate them and increase their interest in Computer Science.

Games and Computer Science

Creating a successful modern video game requires in-depth knowledge of many areas in Computer Science, especially if the goal is to create an immersive virtual environment, where players forget their current environment and become completely focused on the game. Computer Graphics are an essential component of any video game, given its role in communicating the game to the player. However, the fields of physics, numerical approximation and simulation play an equally important role, as they are used to describe the behavior of objects in a virtual world. In addition, the proper implementation of challenging computer-controlled opponents can only be done with a proper understanding of artificial intelligence. Furthermore, it is also necessary to consider the importance of multiplayer games, where various fields, such as distributed systems, concurrency, networking and fault tolerance, are critical.

Games not only push all these areas of Computer Science to the extreme, but also bring together artists and technically skilled people, allowing everyone to express their creativity.

Games and Teaching

The concept of organizing a summer camp to promote Computer Science is not new [1]-[2]. This is not surprising, given that the positives effects of such camps on kids have been proven [3]. University of Alberta's Summer Camp is particularly interesting, since it shares our strategy of teaching Computer Science using game development. Their camp focuses greatly on content generation using visual tools, such as Neverwinter Nights [4], and allows their participants to create virtual worlds. This is very different from our camp, where the focus is placed on problem solving using textual programming languages.

Vrjie University recently finished developing VU-Life 2 [5], a game designed to promote Computer Science and their university based on the Half-life 2 SDK. The game allows students to visit the faculty of Computer Science at the university. After playing the game, students are encouraged to create their own variation of the game by using the Half-life game development tools.

Some Universities also use games within their Computer Science courses. Rudy Rucker, of San Jose State University, teaches software engineering using games as context for implementation [6]. Joe Warren, of Rice University, teaches a class where students are required to work as a team to complete a large-scale game project [7]. These classes, however, use up an entire semester, and are designed for undergraduate students that have already taken beginner programming classes.

III. TEACHING METHODOLOGY

The Summer Camp is a one week event. The last day of the week is reserved for lab work and the final competition. Each other day is assigned a topic, and starts with a keynote presentation introducing that topic. This presentation is given by an industry invited speaker, who demonstrates a real-life application of the day's topic.

The keynote is then followed by a 90 minutes in-class lecture (with a 5 minutes break) presented by a university professor or lecturer. These lectures elaborate on the day's topic, focusing on the knowledge that the students require for the afternoon's lab session and the A.I. competition.

To successfully create an A.I. to pilot a ship in *Spaceracer*, students need to learn how to program in Java, how to design a decision tree and devise an optimal solution to navigate a spaceship through a field full of asteroids. To give a broader overview of Computer Science, the students are also introduced to 3D computer graphics, automated content generation and simulation. Given the large quantity of material that must be taught, we opted for a traditional type of lecture. In our experience, interactive lectures given in the computer labs have a tendency to be slower.

The afternoons are used as lab sessions, in which the students are split into "research" groups of 2 or 3 people, and focus on solving a series of progressively more difficult game-programming-related exercises in the context of our *Spaceracer* environment. Once they successfully complete their exercises, students are encouraged to start working on the code for the team-competition that is held on the final day.

Language Subset

Given that the camp only lasts one week, it is unrealistic to try and teach the students object-oriented programming. Thus, our game programming framework is designed so that students are only required to understand a small subset of features found in a typical programming language:

- Variables and how to use them,
- Standard types (integers, floating-point numbers, Booleans etc.),
- Calling functions and using their return values,
- Boolean logic (*true, false, and, or*),
- IF statements,
- Iteration and looping.

It was greatly debated if we should ask students to write new functions. However, in the end, it was decided that there was little benefits in teaching them how to write methods and time would be better spent focusing on the above points.

IV. SPACERACER

In order to run a successful summer camp, we needed to develop a computer game that would keep the high school students motivated during the whole week. We wanted a game with a competitive aspect, as to keep the high school students motivated. However, we decided early that the game should be as non violent as possible, thus eliminating any game design that would involve players directly or indirectly attacking each other.

Thus, we created *Spaceracer*, a game where players must navigate a space ship along a horizontal race track as fast as possible, while avoiding deadly asteroids and comets. Students must write a simple A.I. pilot[1] that safely navigates the spaceship through the race, avoiding all obstacles.

Working on such a game exposes the students to various Computer Science areas, such as graphics, artificial intelligence, physics and simulation. It was therefore possible to design all of the lab sessions around the *Spaceracer* central theme. One advantage of using a racing game is that a race only lasts a few minutes. That way, the students don't loose too much time during testing. In addition, the game rules and physical laws can be designed to be fairly simple. That way, the students do not have any conceptual problems understanding the game / physics, and hence are able to concentrate on the essence of the problem at hand and on the implementation challenge.

The Race

In *Spaceracer*, the race track is a horizontal area of variable length. The top and the bottom of the track are guarded by walls. The spaceship's initial position is on the start line, completely on the left of the track. We used a simple Cartesian coordinate system to encode the position of objects. The origin is placed at the center of the start line. After a short countdown, the race begins. The pilot of a ship can accelerate or decelerate, move up or down. The ship cannot, however, move backwards. Moving up and down slightly slows down the forward movement. In addition, each ship has a shield that, when activated, protects the ship from any damage for five seconds. However, once the shield deactivates, it can not be re-used until it is fully recharged by the shield generator, which takes approximately 30 seconds.

[1] Note that the A.I. code the students implement is basically a decision tree. By no means are the students expected to implement a learning algorithm.

The Race

In order to keep the students motivated throughout the week, we announced a competition to be held on the last day of the camp. The ultimate goal for the students is therefore not only to build a A.I. pilot that can successfully navigate the race, but their A.I. must do so faster, and hopefully in a smarter way, than the A.I. of the other students. At the end of the camp, the student's A.I. pilots compete on tracks of varying difficulty. The team having written the A.I. obtaining the best overall score wins.

Implementation of the Spaceracer Platform

We decided to implement *Spaceracer* in Java [8] for multiple reasons. First and most importantly, we wanted a simple language were students would not have to deal with complex issues such as memory management and pointers. In addition, when errors do occur, either at runtime or compile time, Java's error message is quite explicit, facilitating debugging activities. Secondly, we wanted to give the students an introduction to a state-of-the-art object-oriented programming language. A third advantage is that Java is multi-platform, thus making it easier for the students to bring the game home should we decide to distribute it.

Object-Oriented API

We wanted to keep the API for the students as simple as possible. To this aim, we modularized all the state and behavior that the students needed to access and modify the game state in 6 classes, namely `Asteroid`, `Comet`, `ShieldRecharge`, `SpaceshipControl`, `Spaceship`, and `Radar` (see Figure 1).

The API itself is designed to be easy for the students to learn. This was achieved through the use of good software engineering practices, such as encapsulating each component of the game in its own object. For example, a ship can detect the presence of asteroids using its radar. Thus, the functionalities of the radar are encapsulated in the simple to use Radar object. In our experience, high-school students have no problem understanding the concepts of objects and thus can learn to use the API for *Spaceracer* during the first day. Most of the methods provided to the students have no side-effects on the state of the game. This prevents the students from "breaking" the game, even if they experiment with different method calls.

V. Target Audience of the Camp

At the beginning of the Fall session (September), the School of Computer Science of McGill University sends out invitation letters to all high schools in the Montreal area. We ask them to identify several students that are strong in Math, Science, and other computer-related skills. Given the large amount of material covered and the short length of the camp, we believe that a student with a weak Math and Science background will encounter difficulties during the camp. In addition, students should be creative and able to work well with others.

On average, we received 30 applications each year, all of which were accepted to participate in the camp. The students that participate have usually little or no programming experience. They do, however, have some experience with the concepts of variables and Boolean logic. Thus most students can easily learn the basics of programming in one day. Experience shows us that the students quickly understand that finding an approach to solving a problem and coming up with an algorithm is a lot harder than programming itself.

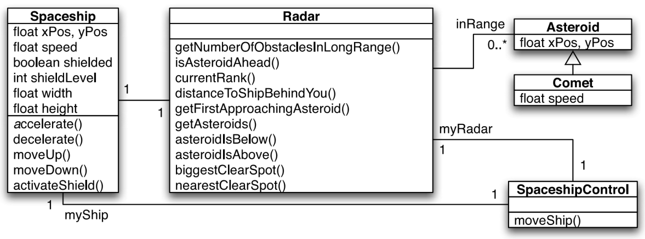


FIGURE 1
Spaceracer API

VI. Course Material

Day 1: Game Programming

The first day is dedicated to teaching the students the programming language subset they will need during the week. The day starts with a keynote that introduces the challenges in game development. Students learn about the different kind of people needed to create a game and the different challenges they face. The keynote is followed by a lecture that introduces the students to computers, programming languages and compilers, focusing mostly on the subset presented in section 3. The examples used during the lecture are all inspired by the *Spaceracer* theme, which exposes them already to the context in which they are going to do their practical exercises.

Exercises

The exercises for the first day mainly center on teaching the students to move the ship. The first exercise requires them to write a simple key handler that captures key presses on the keyboard, and, depending on the key pressed, calls the appropriate method to move the spaceship. The code for capturing key input is already provided. Successfully completing this exercise indicates an understanding of method calls and if statements.

The second exercise requires students to write their first A.I. pilot for their ship. There is no obstacle in this race, so the code is very trivial. However, successful completion of this race demonstrates an understanding of the game's main loop.

The third exercise requires them to avoid their first asteroid. This asteroid is placed directly in front of the ship and can only be avoided by moving the ship either up or down to avoid it. Students must thus learn to use the radar to detect the presences of asteroids and move on the Y axis.

The remaining exercises of the day all feature races with asteroids placed in different configurations. These configurations are all designed to test if the students can come up with more elaborate logic for their AI pilot. For example, the race shown in Figure 2 is impossible to complete if asteroids are always avoided by moving up.

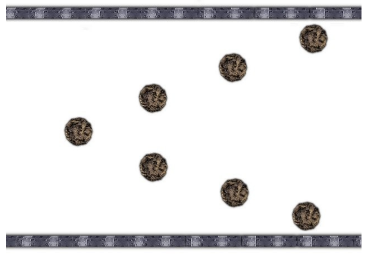


FIGURE 2
ASTEROIDS IN V CONFIGURATION

Day 2: Computer Graphics

The theme of the second day is computer graphics. The keynote introduces the use of computer graphics in various domains, including the movie industry, medical imagery and video games. The lecture presented after the keynote focuses more on the basic concepts of 3D modeling, i.e., polygons, surfaces, textures, lighting, cameras. In addition, students are introduced to a simple open-source 3D modeling software called Wings3D [9]. In order to improve the students' understanding of the modeling process, we handed out play-doh to each student, asking them to form a sphere or cube, and then try to model a spaceship by deforming the initial body.

In the afternoon lab sessions, the students get some hands-on experience with Wings3D. The first exercises focus mostly on deforming primitive shapes to create complex ones. Students can then start the special project of the day, which is to model their own spaceship. The ship they model can then be used in the final *Spaceracer* competition to represent their team. This increases group cohesion and motivates students to build a better A.I., given that it is "their" ship that participates in the race.

III. Day 3: Artificial Intelligence

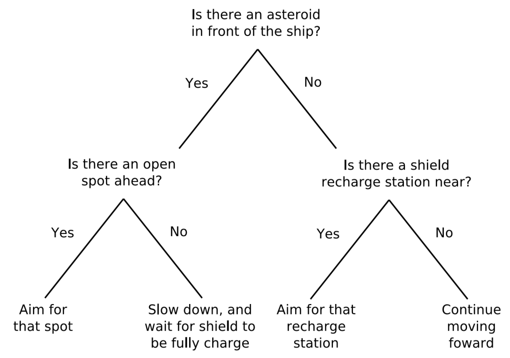
The theme of day 3 is artificial intelligence. The keynote speaker introduces the importance of artificial intelligence in games, especially in games where opponents and allies of the player are controlled by the computer. The presentation usually uses a specific game as a case study.

The day's lecture gives a brief overview of various A.I. techniques, outlining the difference between a scripted A.I. and a learning A.I. The lecture then concentrates on simple decision making and path finding. A key point of this lecture is the importance of decision trees, especially when building an A.I. for *Spaceracer*, as shown in Figure 3. Students are also introduced to the advanced functionalities of the radar, which scans the area ahead of the ship and finds safe ranges void of asteroids.

Exercises

The first exercise of the afternoon is designed to get students comfortable with the new radar functions. The race is composed of walls of asteroids with a small hole in each wall. It is impossible to complete this race using the greedy approach (always avoid the nearest asteroid) used during the first day. Instead, students must learn to find a safe spot and fly the ship towards it. This represents an important shift in logic for their A.I.; instead of avoiding targets, they must aim for a specific one. Successful completion of this race indicates that they understand the new radar functions and were able to aim for specific targets.

The remaining exercises of the day focus on improving their A.I. by choosing better spots to aim for. For example, ships should avoid spots if they are too small for them. Furthermore, if a ship is moving up to reach a specific spot, it should check for asteroids directly above it.


Is there an asteroid in front of the ship?
Yes
No
Is there an open spot ahead?
Is there a shield recharge station near?
Yes
No
Yes
No
Aim for that spot
Slow down, and wait for shield to be fully charge
Aim for that recharge station
Continue moving foward
FIGURE 3
NAVIGATION DECISION TREE

Day 4: Game Physics and Content Generation

The activities of the fourth day are centered on physical simulations and automated content generation. In 2006, half of the lecture was dedicated to teaching the physics behind racing games. The talk described the various laws of physics that can be found in games and how they are often approximated. The following year, the talk was changed to content generation. Different methods of automated content generation were discussed, such as maze generation and terrain generation. The second half of the lecture was pretty similar in both years. It covered some more advanced aspects of *Spaceracer*, including Comets (2006), Shield Recharges (2007) and dealing with impossible situations.

Exercises

The radar provided to the students has a limited range. As such, it is possible to make a decision that seems optimal at the moment, only to discover that the path is blocked later on. In addition, some of the races are randomly generated. Thus, a race could be generated where no safe path exist to reach the finish line.

To deal with these situations, all the ships are equipped with shields. These shields only last a limited amount of time and take 30 seconds to recharge. When faced with an impossible situation, the only solution is to traverse the obstacle using the shield.

The exercises of the day allow the students to write code that deals with this situation. The races have multiple wall barriers, impossible to cross without shields. However, given the proximity of these barriers, students must slow down their ship and allow the shields to recharge before crossing each barrier.

Day 5: Competition

No keynotes or lectures are planned on the fifth day. Students are given the entire morning to tweak their artificial intelligence in preparation for the afternoon's competition. After working a week on their A.I. pilot, students are curious to know how well their A.I. compares with the ones of the other students.

Although the afternoon's competition varies from one year to another, the format remains the same: all A.I. participate in a series of different races and are ranked according to their performance.

VII. Evaluation

Student Programming Skills

To evaluate the progression of the students throughout the week, statistics on the A.I. code they wrote at the end of each day that involved programming, i.e. day 1, day 3, day 4 and day 5, were gathered. For each team, the number of lines of code, the number of references to `myShip`, the number of references to `myRadar`, and the number of if statements was counted, both in 2006 and 2007 (see Table I and II).

TABLE I
Student Code Complexity 2006

Average number of . . . per team	Day 1	Day 3	Day 4	Day 5
If conditions	6.14	9.43	15	18.29
Reference to `myShip` object	10	17.14	24.29	25.29
Reference to `myRadar` object	5.43	9.14	12.86	12.71
Number of brackets { }	6.86	11.29	20.14	22.43
Number of lines of code	41.29	59.71	105.57	107.71

TABLE II
Student Code Complexity 2007

Average number of . . . per team	Day 1	Day 3	Day 4	Day 5
If conditions	3.91	5.18	14.73	22.82
Reference to `myShip` object	9.82	22	34.82	36.18
Reference to `myRadar` object	6.64	8.64	11.64	12.18
Number of brackets { }	6.09	15.73	25	26.45
Number of lines of code	55.73	103.18	144.55	150.64

The statistics revealed some interesting facts. There is steady growth in the code complexity in the first four days, but that progression decreases in the last day, when the final version is due. A detailed analysis of Day 4 code and Day 5 code revealed that students used the final day to restructure their code for the final competition. In general, the code produced at the end of the fifth day is a lot simpler and contains much more commented code than at the end of the fourth day. For example, in 2006, at the end of the fifth day, the A.I. developed by the students averaged 107.7 lines of code. However, the average decreases to 82.6 if we ignore code that was commented-out.

This evolution in the code size can easily be explained. During the first four days, when courses are given in the morning, the students spend the afternoon adding new ideas to their code. Thus each new day results in new features and improved behavior for their A.I. pilot. Only on the final day they take a step back and try to integrate all the different ideas into a single cohesive block. It is important to note that as the student experiment, they comment out experimental code rather than deleting it. This explains the large presence of comments in the student's code, especially on the final day.

Furthermore, the high increase in the number of if statements on the fifth day (especially in 2007) can be explained by the improvements students made to their decision trees. Although this material was covered both on the third and fourth day, students didn't really master the material until the fifth day. This demonstrates the need to improve the teaching methodology on this subject for the following years, as the sharp increase should have occurred much earlier during the week.

Piloting Strategies

In 2006, two features were found in all the auto pilots: they activated the shield when a collision was imminent, and they reduced their speed when a certain number of incoming asteroids was detected by the radar. However, the only omnipresent feature found in all the A.I. in 2007 was the use of shields when a collision was imminent. This can be easily explained by the fact that the lecturer that first presented the idea of slowing down when numerous asteroids are detected, was not available to teach in 2007. This illustrates well the influence the lecturers have on the student's solutions and the importance of properly preparing in class examples and sample races.

Winning Strategies

The ship that won the competition in 2006, AI33, featured a relatively simple auto-pilot. When faced with an obstacle, it searched the map for the nearest clear spot and directed the ship towards that spot at full speed. This aggressive racing behavior helped it win during the easy races. However, the greedy strategy did not always perform well in more difficult races.

Firebird, the second place winner in 2006, featured a smarter algorithm that performed very well during the harder races. When faced with an obstacle, this auto-pilot would search for the nearest open spot. Unlike all the other A.I.s, it would also check the size of the spot. If the spot was too small, it would instead aim for the biggest open spot. It should be noted that this feature was much more common in 2007, as many of the races featured small spots.

The most impressive student A.I. seen up to date was Asian Invasion, the winner of the 2007 competition. This A.I. featured a very complicated, but detailed decision tree that allowed it to have the proper reactions to many different situations. The decision tree not only controlled movement on the Y-axis, but was used to calculate the maximum safe speed at which the ship could travel. Asian Invasion is also the only team we have seen so far that successfully used the distance equation ($\sqrt{x^2 + y^2}$) to determine if an asteroid was too close. Most teams tested the X and Y axis separately.

Student Feedback

In 2007, a questionnaire was distributed to the students both at the beginning and at the end of the camp. One key question required students to describe what they think Computer Science is. In the questionnaire distributed before the camp, common themes were: "Doing stuff with computers", "Programming" and "Studying what computers can do". The same question was asked to the students at the end of the camp. This time, the common themes were: "Understanding what you can do with computers", "Science that deals with computers", "Using computers to solve problems" and "Programming".

The three first themes indicate that students understood the lesson we were trying to teach. However, the presence of the fourth theme might indicate that we still need to reduce the emphasis of programming in the course content.

Another important result of the 2007 questionnaire is the number of students interested in studies in Computer Science. Before the camp, 50% of the students indicated a desired to pursue a career in Computer Science. Surprisingly, that number did not change in the questionnaire after the camp. However, an important result is that 20 out of 22 students expressed that the camp had positively improved their view of Computer Science. The impact of this number can be better understood through the feedback of one of our 2006 student which said: "It has definitively impacted me, but I'm still going to pursue Mathematics. But it made me think about how I could work with engineers using my math."

VIII. Conclusion

In this paper we described the idea and organization of the McGill Computer Science "Game Programming Guru" Summer Camp, organized in Summer 2006 and 2007 with the goal of attracting bright students towards science and Computer Science in particular. The camp was targeted at high school students from grade 10 to 11 in order to awaken their interest as early as possible. This allows them to choose the appropriate optional courses in their final high school years that allow them later on to pursue a major degree in Computer Science or Software Engineering.

We showed in the paper the teaching methodology used to introduce Computer Science to high school students, and at the same time how to use a computer game theme to keep the students motivated throughout the week.

Based on the code evaluation of each team's game code and the answers obtained through a student feedback questionnaire we conclude that the camp was a big success. The responses showed an increase in the understanding, the interest and the appreciation of the field of Computer Science.

In the end, to know if we actually achieved our concrete goal, i.e., attracting more students towards Computer Science or Software Engineering studies at the School of Computer Science at McGill University, we will have to wait until September 2008, when the first graduates from the Summer Camp will start their university education.

References

[1] University of Saskatchewan 2006 Summer Camp, http://www.csss.usask.ca/2005/index.php?c=summercamp, September 2006.

[2] Purdue University 2006 Summer Camp, http://www.cs.purdue.edu/, September 2007.

[3] J. P. Walsh, G. Crombie, J. Flanagan, and V. Hall, Positive Effects of Science and Technology Summer Camps on the Attitudes of Young Canadians: Initial Quantitative Evidence. Poster presented at the 62nd Annual Convention of the Canadian Psychological Association, June 21-23, 2001.

[4] Bioware. Neverwinter Nights. http://nwn.bioware.com/, 2007.

[5] A. Eliens and S. V. Bhikharie, Game @ VU - Developing a Masterclass for High-school Students using the Half-Lide 2 SDK. In Game-On-NA 2006 - 2nd International North American Conference on Intelligent Games and Simulation, pages 49 – 53. Eurosis, September 2006.

[6] R. Rucker, Software Engineering and Computer Games. Addison Wesley, 2003.

[7] S. Schaefer and J. Warren, Teaching Computer Game Design and Construction. Computer-Aided Design, 36(14), December 2004.

[8] J. Gosling, B. Joy, and G. L. Steele, The Java Language Specification. The Java Series. Addison Wesley, Reading, MA, USA, 1996.

[9] Open Source Community. Wings3D, http://www.wings3d.com/, September 2007.

Author Information

Alexandre Denault, PhD Student, School of Computer Science, McGill University, adenau@cs.mcgill.ca

Jörg Kienzle, Professor, School of Computer Science, McGill University, joerg@cs.mcgill.ca

Joseph Vybihal, Lecturer, School of Computer Science, McGill University, jvybihal@cs.mcgill.ca