Chapter 7
Using Graphics to Inspire Failing Students

David Collins

Abstract The chapter summarises recent challenges faced by teachers of first year undergraduate programming and their causes. It proceeds to describe a pragmatic means of addressing such problems through the provision of parallel second programming module provision. The approach provides a means of motivating weaker students, introducing remedial prerequisite knowledge whilst avoiding the sacrifice of employability and other perceived goals of early *high-flyers* in the subject.

Keywords First programming languages · Motivating computer science students The processing language · Failure in programming · Learning edge momentum theory · Novice programmers

7.1 Introduction

A range of factors have made the teaching of computer programming in UK universities ever more problematic in recent decades. General grade inflation at A Level (UK university entrance qualifications) has meant that the skill and experience level of entrants is difficult to predict. Changes made to both the nature and assessment of such school qualifications have led to greater selectivity within subjects making it difficult to rely upon adequate coverage of the individual topics within a discipline—for example, trigonometry or statistics within mathematics. The expansion of the university system and the concomitant competition between institutions has paradoxically often led to the direct or indirect (through the university application clearing system) reduction in entry qualifications. The same competition has led to league tables which prioritise student satisfaction and 'added value' which has resulted in strong pressures to improve retention and grant higher award levels.

Whilst the above factors have been of influence, computer science courses continue to experience problems with widespread first-year retention difficulties and low pass rates. These ultimately manifest as low numbers of upper class degree awards

D. Collins (✉)
University of Keele, Keele, UK
e-mail: d.j.collins@keele.ac.uk

J. Carter et al. (eds.), *Higher Education Computer Science*,
https://doi.org/10.1007/978-3-319-98590-9_7

(the second worst subject across UK universities) and make CS the worst subject for students either leaving with no award or with a lower award than they had originally targeted (Woodfield 2014).

In attempts to deal with these problems, many universities have experimented with different programming languages and paradigms for their first-year courses. Although the jury is probably still out regarding the success of most such endeavours, it is worthy of note that a recent comprehensive survey of first year programming language adoption in the UK revealed that such experimentation (alternatives to Java and C) is more common in lower tariff (lower entry qualification requirement) universities where the need to consider alternatives is likely to be more acute (Murphy et al. 2017). In my institution, faced with very poor first year performance levels and consequent retention problems, we also sought a remedy. In common with similarly ranked universities, at least a third of our students were not significantly benefitting from their first-year programming experience based upon assessment of their end of year performance. Some studies suggest that the figure could be even higher, given a failure of first year university assessments to accurately measure student programming knowledge and ability (Ford and Venema 2010).

Naturally, we regarded this as a serious problem and engaged in an exhaustive review incorporating internal focus group interviews and extensive analysis of student performance statistics. The overall conclusion reached was that we were not attracting the calibre of students that would benefit from the reasonably mainstream pattern of programming skill development that was embedded in our CS curriculum. That much was evident, but there were considerable differences of opinion regarding an appropriate solution to the problem. The main obstacles were a certain degree of inertia, a desire to meet expectations of more able students and the ultimate goal of maximising employability skills.

Streaming students based upon their initial aptitude for programming was a possible route forward. However, this presented several challenges: could we identify such students?; could we resource parallel streams?; could we guarantee the efficacy of an alternative approach for failing students? What we were sure of was that by the end of the first programming module (Let's call it CS1), we could identify the failing group and what the future might hold for them without some form of intervention.

Through the focus group discussions, we identified three major barriers to learning present in the failing section of the cohort: (i) Expectations and motivation, (ii) Prior knowledge and experience and (iii) The nature of the programming learning process. I will consider each of these in turn.

(i) Expectations and Motivation
For many students, the CS1 course was not what they were expecting to encounter in their CS degree. Their knowledge of the subject was heavily influenced by their experiences in school and the portrayal of the subject in popular culture. At the time, school experiences were extremely varied but most likely to be focussed on IT solution packages such as spreadsheets and 'databases'. The adoption of the new computing curriculum within schools has changed this somewhat, but the delivery of the curriculum appears to be

extremely varied at present. In popular culture, computing could be an exciting subject capable of addressing (or causing) almost any societal problem. Hacking, social networks, artificial intelligence, robotics and gaming were common themes expressed by students. In contrast, CS1 exercises tended to be mathematically focussed and rather dry. Many studies, for example (Jenkins 2001), suggest that only a minority of CS students are *intrinsically* motivated by the subject of computer programming, and it is difficult to see how the use of such exemplars would be likely to improve upon this situation.

(ii) Prior Knowledge and Experience
There was a time when most universities could rely upon a common level of skills and experience in their UG cohort. For example, a GCE O Level qualification in mathematics guaranteed a fundamental knowledge of trigonometry and calculus—but attained to varying levels. This was no longer the case and many of the exemplars used in CS1 contained concepts and terminology which were not universally shared by the cohort. The provision of a discrete mathematics module had been abandoned some time earlier when it became apparent that failing students on the CS1 module also tended to fail this compensatory module.

(iii) The nature of the Programming Learning Process
There are now a wide range of theories to account for the unusual distribution of performance produced by year one computer programming modules. Of these, the author finds the *learning edge momentum* theory to be the most compelling (Robins 2010). In this theory, success in acquiring one concept makes learning other closely linked concepts easier, and failure makes it harder. The tightly integrated nature of the concepts comprising a programming language, drives students towards extreme outcomes. Coupled with this, early failure to master concepts is de-motivational and further exacerbates the problem. Failing students typically disengage early in a programming module reporting that they were unable to grasp the introductory concepts. Failing early often means failing completely.

We needed a prompt and pragmatic solution to the above problems. The essential approach was to agree an alternative to CS2 (Programming II—Data Structures and Algorithms) for the failing section of the CS1 cohort that would attempt to both meet the broad learning objectives of CS2 and address the three barriers to learning described above. CS2 at that stage occurred in the second semester of the first year of studies. Institutional constraints required that such a module (an alternative to CS2), would also be available to some *successful* CS1 students and that completion of the new module might allow access to CS3 (Advanced Programming) in year two. CS1, CS2 and CS3 were all taught using the Java programming language. I trust that the reader appreciates the low probability of a satisfactory outcome!

The choice of language for the new module was astoundingly straightforward. We needed to build upon previous experience in Java and allow for possible progression to subsequent Java based modules (CS3). The only language that met these criteria and also offered some other pedagogic possibilities was the Processing language

developed at MIT by Chris Rea and Daniel Shiffman (Reas et al. 2007). Processing is a graphics oriented language designed for non-science students to create visual art. The language comes with a simple development environment (subsequently much improved but no more complex) and a large set of example programs (termed *sketches* in the Processing terminology). The language is essentially implemented as a Java library which entails that it could be imported into a conventional Java program within a conventional Java IDE. This afforded the possibility of returning to 'mainstream' Java toward the end of the new module without requiring any major conceptual leaps. By now, I am sure that many readers are familiar with the Processing language and environment so subsequent discussion will relate to how it was used to address our specific problems. An excellent exposition of the possibilities of the language is given in Daniel Shiffman's book (Shiffman 2016).

7.2 The Learning Objectives of Our CS1 and CS2 Modules

Before progressing further, I present the aims and learning objectives of our CS1 and CS2 programming modules:

CS1
Aim: To introduce computer programming concepts using a generic (non-context specific) computer language and to develop problem-solving skills in the framework of computer programming.
And Objectives:

- Demonstrate an understanding of the basic concepts of computer programming.
- Evaluate the suitability of computer language data and control structures to achieve basic problem-solving.
- Demonstrate an understanding of the basic software engineering principles.
- Show practical experience of those basic concepts.

CS2
Aim: To develop new programming skills as part of an exploration of several important data structures and algorithms used in Computer Science.
And Objectives:

- write a program that demonstrates important features of computer programming using an object-oriented programming language;
- describe, explain and evaluate the principles and operation of several data structures that are widely used in computer science;
- use a programming language to operate, test and evaluate one or more of the widely used computer science data structures;
- select class, data and control structures for program-based problem-solving.

In common with many other universities these aims and objectives are quite broad and allow implementation using a wide range of languages and, to a lesser extent, paradigms (object-oriented has a specific mention). The nature of *Processing* is such

that it affords much the same opportunities as Java but the reduction in verbosity and bureaucracy coupled with graphics and animation support allows more creative problems to be addressed and expressed early in student's encounter with the language.

The challenge became one of designing a curriculum keeping the afore-mentioned three barriers to learning in mind. We sought to (i) attempt to provide a level playing field for students in terms of prior knowledge: (ii) motivate students by selection of exemplars that would stimulate interest: (iii) ensure that students mastered concepts incrementally, thus preventing early disengagement and failure. In the pages that follow we describe some of the main aspects of the module using tables to summarise how we addressed the identified barriers. The module progresses using exemplar programs, generally incomplete, as the context in which topics are taught and explored by students. During the practical sessions associated with each topic, completion of the associated individual exercises is confirmed and recorded by post-graduate demonstrators.

7.3 The Module

7.3.1 Drawing and Graphic Transformations

In order to introduce the *Processing* language and provide some revision of basic programming principles, the first topic provides an introduction to the procedural production of graphic images using primitive drawing shapes. Students are also introduced to graphical transforms and coordinate systems. Exercises include the reproduction of example images using repetitions of translations, rotations and scaling.

Additional knowledge and skills	CS programming concepts	Motivational material
Coordinate systems	Control structures	Students are asked to produce multiple flower and plant shapes with different morphologies and at different scales
Graphical transformations	Functions and decomposition	
Colour representation	Variables and scope	
Alpha transparency	Constants	
	Formal and actual arguments	

7.3.2 Animated Clock

The second major exemplar involves the animation of a real-time analog clock. Students are initially provided with a program demonstrating radial motion which

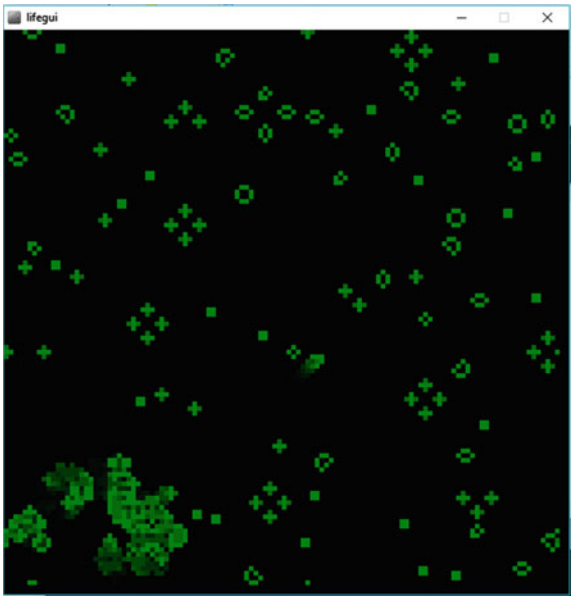
also interactively depicts Pythagoras's theorem, trigonometric relationships and the expression of angles in degrees and in radians (the majority of our students being ignorant of the latter upon entry to the module). Some basic initial exercises are provided which are intended to give students with the opportunity to explore these concepts. As a final exercise, students are required to EFFICIENTLY simulate the action of an analog clock.

Additional knowledge and Skills	CS programming concepts	Motivational material
Trigonometry	Library function calls	The production of a functional program with visual design selected by the student
Frame-based animation	Continuous versus discrete events	
Time zones and time representations		

7.3.3 The Game of Life

For this example, we provide students with a simple but visually captivating version of Conway's game of life (Gardner 1970). This provides a stimulating introduction to the concepts behind finite state machines. Exercises involve modification of survival and reproduction rules, changes to the grid size and resolution and modification of the 'neighbourhood' definition. The exercise provides the opportunity to master the use of 2/3 dimensional arrays and reasonably complex selection constructs. Students are expected to modify the code to permit different rules and neighbourhood definitions to be 'plugged in'.

Additional knowledge and skills	CS programming concepts	Motivational material
State machines	2/3 Dimensional arrays	Students are generally fascinated by the complex emergent behaviours that derive from simple rules
Turing, Von-Neumann and computing history	Cellular automata	
Modular arithmetic	Selection control constructs	
	Procedural abstraction	
	Tracing, reading and debugging code	



Conway's game of life implemented in processing with fade-out effects for dying cells

7.3.4 Card Games

Students are provided with graphics for presentation of playing cards and are asked to design algorithms for dealing and shuffling. We present some sorting algorithms and Durstenfeld's algorithm for shuffling (Durstenfeld 1964). This provides an opportunity to introduce the topic of complexity and we discuss the performance of shuffling algorithms in this context. The card decks provide an excellent medium for exploring stacks and queues and associated ADTs.

Additional knowledge and skills	CS programming concepts	Motivational material
Numeric distributions	Sorting and shuffling algorithms	Card games require a range of algorithms and data types. The relevance and purpose is immediately apparent to students. Students are invited to devise their own shuffling algorithms initially which provides a natural vehicle for exploring complexity
Statistical concepts	Algorithmic and computational complexity	
Randomness	Arraylists	
	Random number implementations	
	Abstract data types, stacks and queues	

7.3.5 Sprite Based Animation

Animation in Processing is essentially frame-based. In this section of the module we introduce the use of sprite sequences on scrolling backgrounds. Sprites may be controlled via the keyboard or with the use of a gaming input device.

Additional knowledge and skills	CS programming concepts	Motivational material
Seamless textures	File handling	Weaker students in particular are delighted to be able to produce a program that approaches the quality of a published (but retro) game
Easing in animation	Object arraylists	
Frame buffering	Asset management	
Multiple animation timelines		

7.3.6 Lunar Lander

Students are provided with a background lunar image and a transparent GIF representing the landing craft. They are also provided with an example program depicting objects falling under the influence of gravity that uses a skeleton implementation

of a Vector class. They are then expected to complete the Lunar Lander Program using the keyboard to deliver 2 dimensional thrust to counter gravity and gracefully land the craft at a randomly selected location. This requires implementation of new methods for the Vector class.

Additional knowledge and skills	CS programming concepts	Motivational material
Forces and Newtonian physics	Classes and objects	A playable game with realistic control
Vectors	Keyboard polling/state detection	
Discretization	A vector ADT and its implementation	

7.3.7 Image Processing

Students receive basic lectures on Image formats, manipulation of pixel information and the construction of filters using convolution matrices.

They are provided with a scene of crime image in which a poorly illuminated suspect is seen next to a car with a number plate which is indiscernible. The final task requires the construction and application of filters (incrementally) that will clean-up the image to the extent that the suspect is identifiable and the number plate readable.

Additional knowledge and skills	CS programming concepts	Motivational material
Matrices	Convolution filters	The real-world challenge present in the exercise task is highly motivational to students and encourages them to master a topic that often produces despair when presented in a more traditional image processing context
Image representation and compression schemes	Matrix manipulation	
Bitmap versus scalar graphics		
Normalisation		

7.3.8 Collision Detection

Students are presented with a complete program (based upon the game of Asteroids) that visually represents coarse and fine collision detection and their affect upon maximum animation frame-rates. Students are required to assess the relative performance of coarse and fine approaches and derive a strategy for optimising the compromise between accuracy and performance.

Additional knowledge and skills	CS programming concepts	Motivational material
Further trigonometry	Collision detection algorithms	Students tend to be familiar with this problem and are inspired to find strategies that produce optimal game-play
	Heuristic lgorithms	

7.3.9 Boids

This is by far the most complex programming example provided to students—an object-oriented version of Craig Reynold's (Reynolds 1987) Boid simulation written in the *Processing* language. Lectures introduce the concepts involved and describe the overall design, the main classes and broad implementation details.

The exercises require students to read and follow the logic of this relatively complex piece of code. For example, they are instructed to alter the weighting applied to the three vectors that determine the speed and acceleration of Boids at each discrete decision cycle (which corresponds to the frame-rate). In order to provide more realistic behaviour, they are encouraged to experiment with allocation of weights using a Gaussian distribution. They are expected to modify the definition of neighbourhoods and, as a final exercise, to consider means of improving performance of the algorithm to provide real-time flocking utilising more sophisticated Boid animations. Specifically, they are asked to consider a means of avoiding the need to check the position of each Boid in each cycle in order to determine whether it might be considered a neighbour for purposes of calculation of the required alignment vector.

Additional knowledge and skills	CS programming concepts	Motivational material
Numeric distributions	OO design	Students are fascinated by Boid behaviour and in particular with the complexities of behaviour that can result from the application of such simple rules
	Complexity	
	Code optimisation	
	Algorithmic optimisation	

7.3.10 Recursion

Recursion can be a difficult subject for students and benefits greatly from a graphical treatment. Students are provided with a range of examples including binary trees, Sierpinski triangles, Koch snowflakes and Mandelbrot sets. A range of exercises provide experience in using linear, binary and tail recursion. The final (non-trivial) exercise requires that students add realism to the construction of a binary tree.

Additional knowledge and skills	CS programming concepts	Motivational material
Fractals	Recursion and recursion types	Recursion is explored through graphics—binary trees, Sierpinski triangles and fractals. Students are highly motivated by the task of introducing realism into the construction of a graphical binary tree
Complex numbers	Recursion performance	
	Binary trees	
	Binary search trees	

7.3.11 The Relationship Between Processing and Java (and Python, Javascript and Android)

The final topic of the module provides students with the opportunity to use *Processing* as a Java library within the NetBeans IDE. Lectures describe the language implementation and the manner in which it can be incorporated in a conventional

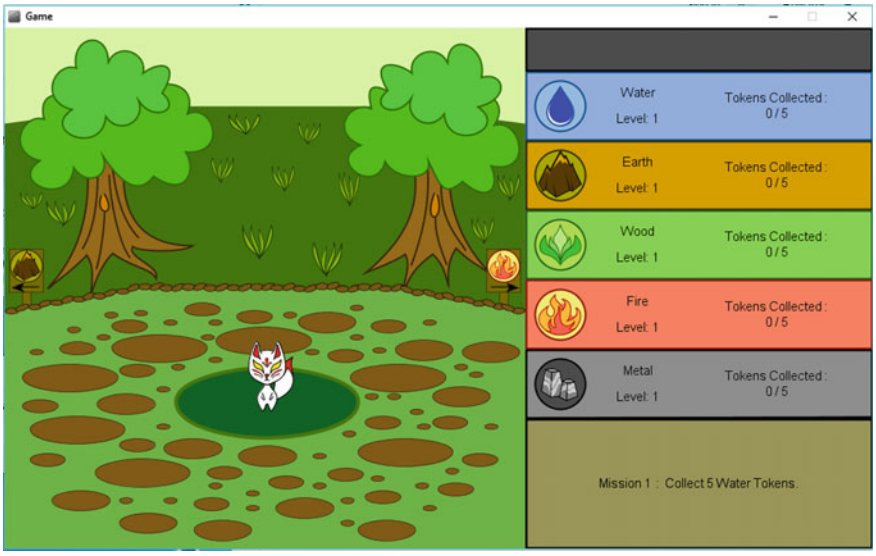
Java application. Processing implementations for Python, Javascript and Android are also described and demonstrated.

Additional knowledge and skills	CS programming concepts	Motivational material
Comparative programming languages	Programming frameworks	At this stage many students have discovered the relationship between java and processing. This topic formalises that understanding
	APIs	
	Wrapper classes	

The above list of exemplar-led topics is not exhaustive and we do change the examples used with each delivery of the module. The module has now been running for 8 years and we recognise that some of the examples are becoming stale and would benefit from new materials. We are also aware that the materials may have a gender bias that we would like to address without falling prey to obvious stereotypes.

7.4 Module Assessment

The module is assessed by examination and coursework. During the fourth week of the module students are required to submit a proposal for their assignment work (with the benefit of a list of future topics that will be covered). We point them at the more complex Processing exemplars provided with the environment and at the OpenProcessing.org website (www.openprocessing.org) for inspiration. The latter provides a virtual gallery through which students may showcase their work to the world. Each student is briefly interviewed regarding their choice of subject and is only allowed to proceed if the lecturer considers the project to be suitably challenging and capable of expressing the module's learning objectives. The progress of each student on their assignment work is monitored throughout the remaining practical sessions.



A game produced by a first year student

7.5 Discussion

There are many variables that determine the success and failure of a module. The introduction of this module was not a controlled experiment and it would not be appropriate to make comparisons or draw conclusions as if this were the case. However, we can broadly point to some of the benefits and problems.

Retention problems were dramatically reduced and weaker students were undoubtedly inspired by the new approach. Much of the student work produced in coursework assignments far exceeded our expectations and continues to do so to the present day. Stronger students had sufficient new materials as to be challenged by the approach but there were some criticisms such as the absence of access modifiers and the simplification of type casting methods in the Processing language. In practice, some of the more capable students possibly felt patronised by the introduction of what they perceived to be a simpler programming environment. Indeed, many were quick to import the *Processing* core into a Net Beans or Eclipse Java environment.

Processing has a pre-processor that wraps the processing code inside a Java class and allows access to functions without having to declare and instantiate the classes to which they belong. Combined with rich graphical features this allows the student to focus on algorithmic content. In reality, the language *is* a slightly more accessible form of Java coupled with a simple IDE and a feature rich Graphical API. Changes to the intrinsic Java are generally akin to being able to introduce **println()** rather than explain **System.out.println()**. The latter requires suspension of full understanding

in weaker students and partially explains why the language creates a *learning-edge* problem with failure to understand one concept *at a specific time* leading to lack of confidence and unpreparedness to tackle subsequent related concepts.

The module has undoubtedly led many failing students from CS1 to develop an *intrinsic* motivation to further develop their programming skills. Students exhibit pride in their work and a desire to demonstrate to, and seek the opinions of their peers. The current module lecturer has persuaded a major European company to sponsor an annual award for the best student assignment work. The module's introduction was a pragmatic solution to a problem and saved us from having to meddle with the now long standing and conventional approach to teaching programming with the Java language that is embodied in our CS1-CS2-CS3 programming strand. Since that time, many UK universities have adopted *Processing* as a first programming language. Interestingly, Murphy et al's recent survey of language use concluded:

> The results in this first UK survey indicate a dominance of Java at a time when universities are still generally teaching students who are new to programming (and computer science), despite the fact that Python is perceived, by the same respondents, to be both easier to teach as well as to learn. (Murphy et al. 2017)

Of course, the likely explanation for the above is that the respondents have goals other than '*teachability'* in mind when selecting a first language — employability would undoubtedly feature highly.

From my perspective as a teacher of programming, the module undoubtedly extended my 'shelf life'. Instead of trying to avoid the weary gaze of yawning students as they tested stack implementations full of meaningless characters and numbers, I was able to witness genuine excitement in students as they retrieved cards from a discard pile or added visual leaf nodes to a binary tree. Early on, it was apparent that some students from CS1 had not really understood two-dimensional arrays. Conway's life provided both motivation and visual feedback that allowed weaker students to both perceive their relevance and master the topic. Students with little mathematical or scientific background picked up an understanding of trigonometry, Newton's laws, parabolic curves and frequency distributions pretty much as a side-effect of wanting to make a game more playable or realistic. Above all, students became far more enthusiastic and attended lectures and practical sessions with relish rather than a sense of guilt or obligation.

Even better was the fact that no compromises were being made in the teaching process. I was still teaching Java to attain the same learning objectives in the context of Computer Science and Software Engineering as were adopted by our CS2 module. Effectively I was provided with an additional teaching tool with which I could create teaching materials that were stimulating yet understandable. The overheads involved in using Processing rather than pure Java were negligible and ultimately led students to understand how they might design an effective application framework. I am indebted to the Processing team for their generous contribution to computing pedagogy.

References

Durstenfeld R (1964) Algorithm 235: random permutation. Commun ACM 7(7):420

Ford M, Venema S (2010) Assessing the success of an introductory programming course. J Info Technol Educ 9:133–145

Gardner M (1970) Mathematical games—the fantastic combinations of John Conway's new solitaire game "life". Sci Am 223:120–123. ISBN 0-89454-001-7

Jenkins T (2001) The motivation of students of programming

Murphy E, Crick T Davenport JH (2017) An analysis of introductory programming courses at UK Universities. Art Sci Eng Program 1(2), Article 18

Reas C, Fry B, Maeda J (2007) Processing: a programming handbook for visual designers and artists (1st edn). The MIT Press, p 736. ISBN 0-262-18262-9

Reynolds CW (1987) Flocks, herds, and schools: a distributed behavioral model. Comput Graphics 21(4):25–34

Robins A (2010) Learning edge momentum: a new account of outcomes in CS1. Comput Sci Educ 20(1):37–71. https://doi.org/10.1080/08993401003612167

Shiffman D (2016) Learning processing—a beginner's guide to programming images, animation, and interaction. Morgan Kaufmann. ISBN 978-0-12-394443-6

Woodfield R (2014) Undergraduate retention and attainment across the disciplines [Internet].Higher Education Academy, New York. Available from www.heacademy.ac.uk/node/10293. Accessed 17 June 2018