THE TURTLE SYSTEM: AN EASY WAY INTO TEXT-BASED PROGRAMMING AND COMPUTER SCIENCE

The Turtle System, with teaching resources and tools for setting and marking coursework, is available free thanks to a new project at Oxford University co-funded by the Department for Education. Peter Millican, Professor of Philosophy at Hertford College, explains the principles behind the system he has developed.

PROBLEMS CROSSING THE POST-SCRATCH GAP

Visual drag-and-drop programming systems – notably MIT's *Scratch* – have proved extremely popular at primary level. Prior programming with *Scratch* should make text-based programming easier, by consolidating some essential concepts (e.g. variables and loops). But experience suggests that crossing the gap to textual coding is still a major challenge for both teacher and pupils, involving at least four difficult problems:

- Choice of language.
- Complications setting up and starting to program.
- Coping with syntax errors.
- Difficulty understanding non-visual program effects.

Programming language debates have gone on for decades: should you start with a language designed for teaching (like *BASIC* or *Pascal*), or with a commercially popular language (like *Java* or *Python*)? Industrial-strength systems can force novices to confront complexities of language far too soon, like the standard entry point for any Java program (which could almost be *designed* to intimidate beginning teachers and pupils):
public static void main(String[ ] args){

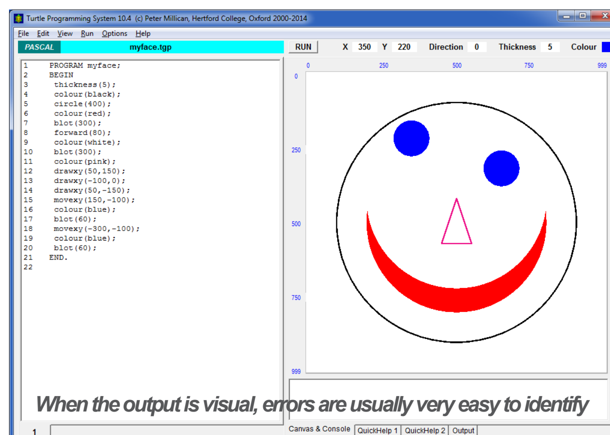
Moreover, such systems are set up more for textual than graphical output, so structures like loops and conditionals standardly get introduced using numerical examples. But for most people, these are far less easy to understand than pictures, especially when errors occur (see image right).

Text-based programming is notoriously difficult, both to teach and learn. But when I was teaching programming to novices at Leeds University in the 1990s, I found that by far the biggest problem was *syntax errors*. These were hugely time-consuming, as helpers rushed from student to student trying to identify the (mostly trivial) mistakes. Practical sessions often seemed like an obstacle course, "success" being defined more by stubborn endurance in the face of irritating and confusing obstructions than by any delight of creative achievement. No doubt this will be a familiar scenario for many readers!

The Turtle System aims to solve this and the other three big problems listed at the left. The last of them – deriving from our natural preference for visual effects – has a well-known solution: Seymour Papert's wonderful idea of *Turtle Graphics*. Papert's original version used his own language *Logo*, specially designed for the purpose, but add-on *Turtle* units have been created for many other languages, and are very widely used in teaching.

This suggests the idea of a simple, integrated environment designed to make Turtle Graphics as easy as possible, *using a general programming language of the user's choice*. Keeping things simple requires a "barebones" version of the relevant language, combining standard "core" features (e.g. constants, variables, arrays, subroutines, conditional and looping structures, operators and bracketing) with special features designed to make graphics and interaction straightforward. The core features allow standard programming techniques to be taught perfectly well. Built-in commands for graphics (e.g. **circle**, **colour**, **forward**, **print**), canvas control (e.g. **fill**, **pause**, **pixcol**, **update**), and simple access to keyboard and mouse events, make it very easy for beginners to get started and *have fun!*

Simplifying the language brings another great benefit, by *enabling error messages to be more precisely targeted*, because restricting users to a small number of core structures makes their mistakes easier to identify and correct. The current version of *Turtle Pascal*, for example, has around 150 syntax error messages, designed to point out *exactly* where the problem lies, and giving a precise hint for correction. After introducing my earlier version at Leeds, I found students were able to fix the vast majority of their syntax errors without needing any further help at all.



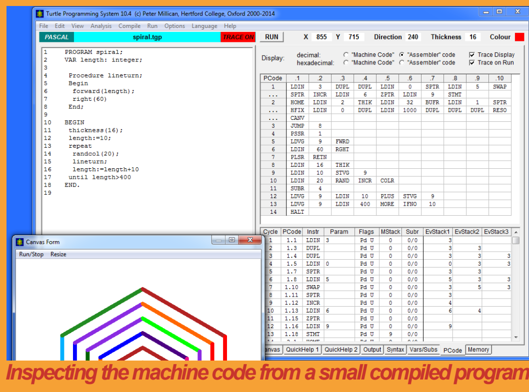
When the output is visual, errors are usually very easy to identify

The barebones approach also makes it feasible to offer a choice of languages (e.g. *BASIC*, *Java*, *Pascal* and *Python*), enabling pupils to compare the same algorithm in different languages, and ap-

LINKS TO COMPUTER SCIENCE: THE TURTLE MACHINE

In my first *Turtle* system (early 2000), programs were *interpreted*, i.e. read and executed line-by-line, but it proved very difficult to give well-targeted error messages within "nested" program structures (e.g. an "if" inside a "for" loop inside a "repeat" loop). I therefore wrote a *compiler*, so that the user's program would be translated into a form of *machine code* (or "PCode", short for "portable code"), whose instructions are executed when the program runs. Compiling involves a complete syntax analysis, helping to solve the error message problem. This also made it easy to provide information for teachers (e.g. about the structures and commands used), so we could check students' programs against specified requirements and mark coursework very quickly – a feature that will probably be appreciated by schoolteachers!

Compilation also opened a new possibility, of using the system to teach *Computer Science* concepts in a novel way. The compiled PCode – essentially a sequence of numbers – runs on a virtual (i.e. software-simulated) "Turtle Machine". These codes include simple instructions for moving, turning, drawing circles etc., and others for internal logic and memory operations. The latter (e.g. handling variables and subroutines) are quite sophisticated, providing plenty of potential for extension work (including finding ways to "hack" the Turtle Machine without any risk). But the basic drawing operations are very straightforward, and can be made open to view and even "traced" as they run, so *all pupils can learn about machine code by seeing how their own programs are analysed and executed in real time and with real effects*.



Inspecting the machine code from a small compiled program

preciate how all are translated into the same underlying "Turtle Machine" code (see box above). This code too is designed to be simple and understandable, making a virtual Turtle Machine easily portable to different devices. Running their own apps and games on the web and smartphones is exciting for pupils, and helps reinforce the vital lesson that algorithms can be understood quite independently of specific languages or hardware.

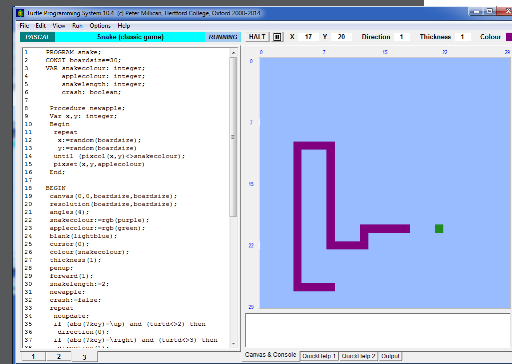
The language choice problem raised by the "Post-Scratch Gap" thus turns out to have a natural solution: teach pupils *explicitly* about the variety of computer languages, using barebones versions that make the comparison easy to understand. Pupils are greatly empowered by realising that programming skills are so easily transferable, and moving forward is far less intimidating when *problem style*, *programming language*, and *working environment* are changed one by one.

What pupils most need to learn is computational thinking and solving problems using algorithms. How those algorithms may then be expressed in some particular language is a secondary matter, because most computer languages are fundamentally very similar (far more than natural languages like English, French, German etc). Once pupils have learned how to express a simple algorithm within one syntax (e.g. *BASIC* or *Pascal*), it should be fairly easy for them to pick up another (e.g. *Java* or *Python*) within a day or two; moreover encountering this variety is, in itself, a valuable learning experience.

In the next issue of **SWITCHEDON**, I shall illustrate *The Turtle System*'s use in practical teaching, but in the meantime, it is freely available with plenty of teaching resources for introducing the new National Curriculum from www.turtle.ox.ac.uk. Please do take a look.

EDUCATIONAL BENEFITS OF BAREBONES SIMPLICITY

A huge amount can be done with a barebones language that has the core features listed, despite its relative simplicity. Illustrative programs provided with *The Turtle System* include a traditional "Snake" arcade game (65 lines), a "Paint" application (102 lines), implementations of famous cellular automata including the "Game of Life" (49 lines), and an infallible noughts-and-crosses program that uses the AI technique of recursive "minimax" analysis down the search tree (116 lines). Though a simple system, it can thus be used to explore algorithms that Computer Science undergraduates would study in their upper years at university.



A growing snake in the traditional arcade game

Learning on a barebones language is also entirely compatible with moving on to an industrial-strength version in due course (e.g. when starting A-level, university, or employment). Programming in *Turtle Pascal* or *Turtle Python* might not be quite the same as programming in *Delphi Pascal* or *Python*, but the basic algorithmic syntax and logic remain identical.

It is much easier getting into a full-strength system after the *Turtle* experience, crossing one hurdle at a time rather than having to learn about *both* computational thinking *and* a highly complex environment at the same time. If you have any questions, please contact me at peter.millican@hertford.ox.ac.uk.