Teaching Digital Technologies in Secondary

James Curran (james@groklearning.com)
Associate Professor, School of IT, University of Sydney
Director, National Computer Science School
CEO and Co-founder, Grok Learning
Today’s program

9:30  Masterclass begins

11:00  Morning tea

13:00  Lunch

12:30  Afternoon tea

15:30  Masterclass ends
Today’s program?

- Programming (everyone)
- Embedded systems (less than everyone)
- Integration (some)
- Unpacking language (more than some)
- Assessment (more than more than some)
- Teacher PD (less than more than some)
- Compulsory..? Culture change
- ICT versus DT versus CS versus CT
Programming in Digital Technologies

- What you need to cover
- Platforms and languages
- Grok Learning / NCSS Challenge
Some good reasons to teach $x$

- $x$ is a different way of thinking or making
- $x$ tells us something about the world
- $x$ is challenging, engaging and creative
- $x$ is fundamental for understanding other disciplines
- $x$ is a skill that takes years to master, so we should expose kids to it early
Some bad reasons to teach $x$

- industry wants $x$
- there are jobs in $x$ now
- the money is good in $x$
- $x$ gives you a high ATAR
- $x$ is in the exam
- ...
What you need to cover
Aspects of programming languages

- visual or text programming
- object-oriented programming
- teaching (specialised) or real-world (general purpose)
Visual programming

- mandatory Year 3-6, maybe Year 7/8
- drag and drop elements to create code
- avoids syntax (and some semantic) errors

- unplugged: flowcharts, decision trees
- teaching: Scratch, Alice, Blockly, Lego NXT
- real-world: Labview
Text programming

- mandatory by Year 8, maybe Year 6-7
- write code by hand
- more efficient and succinct than visual

- unplugged: pseudocode
- teaching: Actionscript, *Basic, Python, ...
- real-world: Actionscript, C++, C#, Java, Javascript, Python, Visual Basic
Object-oriented (OO) programming

- mandatory by Year 10, maybe Year 8-9
- data and code organised as objects
  - objects model the real world
  - support classes, inheritance, and polymorphism
- unplugged: the world is all objects
- teaching: C#, Java, Python, VB(ish)
- real-world: C++, C#, Java, Javascript(ish), Python
- Scratch has objects, but isn’t OO
Specialised versus general purpose

- **some languages are domain specific:**
  - teaching (Scratch, Alice, Blockly, ABC, Blue)
  - game development (Game Maker, UnrealScript)
  - mathematics (Mathematica, MATLAB)
  - databases (SQL) and spreadsheets (Excel formula)
  - often lack common language features

- **most languages are general purpose:**
  - solve problems in a wide range of domains
  - solve specific domain problems less elegantly
Specialised versus general purpose

- Language may be general, but platform not migrate specialised → general purpose
  - Javascript (now used on the backend as node.js)
  - Scratch (used for control on Raspberry Pi)
- General purpose → specialised libraries
  - Python: Django and others to hide SQL
  - Python: numpy to replace MATLAB
- Digital Technologies mandates general purpose by Year 8 to provide students with a practical programming tool
Implementation

4. Implement simple digital solutions as **visual programs** with algorithms involving **branching** (decisions) and **user input**

6. Implement digital solutions as simple visual programs involving branching, **iteration (repetition)**, and user input
Implementation

8. Implement and modify programs with user interfaces involving branching, iteration and functions in a general-purpose programming language.

10. Implement modular programs, applying selected algorithms and data structures including using an object-oriented programming language.
## Implementation

<table>
<thead>
<tr>
<th>Concepts</th>
<th>Programming</th>
<th>Test and debug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can do physical programming (e.g. Bee Bot)</td>
<td>Branching (decisions) and user input</td>
<td></td>
</tr>
<tr>
<td>Visual programming</td>
<td>Iteration (repetition)</td>
<td></td>
</tr>
<tr>
<td>General purpose text programming</td>
<td>User interfaces and functions</td>
<td>In algorithms content descriptor</td>
</tr>
<tr>
<td>Object-oriented programming</td>
<td>Modularity, algorithms and data structures</td>
<td>In algorithms content descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Specification and Algorithms

2. Follow, describe and represent a sequence of steps and decisions (algorithms) needed to solve simple problems.

4. Define simple problems, and describe and follow a sequence of steps and decisions (algorithms) needed to solve them.

6. Define problems in terms of data and functional requirements drawing on previously solved problems.

6. Design a user interface for a digital system.

6. Design, modify and follow simple algorithms involving sequences of steps, branching, and iteration.
Define and decompose real-world problems taking into account functional requirements and economic, environmental, social, technical and usability constraints.

Design the user experience of a digital system, generating, evaluating and communicating alternative designs.

Design algorithms represented diagrammatically and in English, and trace algorithms to predict output for a given input and to identify errors.
**Specification and Algorithms**

1. Define and **decompose real-world problems** precisely, taking into account **functional and non-functional requirements** and including **interviewing stakeholders** to identify needs.

2. Design the user experience of a digital system by **evaluating** alternative designs **against criteria** including **functionality, accessibility, usability, and aesthetics**.

3. Design algorithms represented diagrammatically and in structured English and validate algorithms and programs through tracing and test cases.
## Specification and Algorithms

<table>
<thead>
<tr>
<th>Describe problems</th>
<th>Follow/design algorithms</th>
<th>Design user interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Follow, describe and represent a sequence of steps and decisions</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Define simple problems</td>
<td>Design a user interface</td>
</tr>
<tr>
<td>6</td>
<td>Define problems using data and functional requirements</td>
<td>Design a user interface</td>
</tr>
<tr>
<td>8</td>
<td>Decompose real-world problems; Consider economic, environmental,</td>
<td>Represent algorithms using diagrams and English; trace</td>
</tr>
<tr>
<td></td>
<td>social, technical and usability constraints</td>
<td>algorithms</td>
</tr>
<tr>
<td>10</td>
<td>Interviewing stakeholders to identify needs</td>
<td>Validate algorithms and programs through tracing and test</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cases</td>
</tr>
</tbody>
</table>

- Decompose real-world problems; Consider economic, environmental, social, technical and usability constraints
- Define problems using data and functional requirements
- Define simple problems
- Describe problems
- Describe problems using data and functional requirements
- Decompose real-world problems; Consider economic, environmental, social, technical and usability constraints
- Interviewing stakeholders to identify needs
- Validate algorithms and programs through tracing and test cases
- Evaluate designs against criteria: functionality, accessibility, usability, and aesthetics
- Interviewing stakeholders to identify needs
- Define problems using data and functional requirements
- Define simple problems
- Follow/design algorithms
- Design user interfaces
- Decompose real-world problems; Consider economic, environmental, social, technical and usability constraints
- Interviewing stakeholders to identify needs
- Validate algorithms and programs through tracing and test cases
- Evaluate designs against criteria: functionality, accessibility, usability, and aesthetics
Platforms and languages
Bee bot!
Cambodian Children's Trust
Lightbot (HoC activity)
lighbot.com/hocflash2014.html
You can use the P1 command inside PROC1 to make a loop!
Code.org and Hour of Code
code.org/learn
Scratch
scratch.mit.edu
Scratch

- Animate your name
- Design a holiday card
- Create a pong game
Grok Learning
https://groklearning.com
Recognise and explore patterns in data and represent data as pictures, symbols and diagrams

Recognise different types of data and explore how the same data can be represented in different ways

Examine how whole numbers are used to represent all data in digital systems

Investigate how digital systems represent text, image and audio data in binary

Analyse simple compression of data and how content data are separated from presentation
<table>
<thead>
<tr>
<th>Representation</th>
<th>Types of data</th>
<th>Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Represent data as pictures, symbols and diagrams</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The same data can be represented in different ways</td>
<td>Different types of data</td>
<td></td>
</tr>
<tr>
<td>How whole numbers are used to represent all data</td>
<td>All (simple) data: <em>types should be more complex</em></td>
<td></td>
</tr>
<tr>
<td>Represent data in binary</td>
<td>Text, image and audio</td>
<td></td>
</tr>
<tr>
<td>Content vs. presentation: <em>documents are represented</em></td>
<td>All data: <em>structured data</em></td>
<td>Simple compression of data</td>
</tr>
</tbody>
</table>
**Collection and interpretation**

2. Collect, explore and sort data, and use digital systems to present the data creatively.

4. Collect, access and present different types of data using simple software to create information and solve problems.

6. Acquire, store and validate different types of data, and use a range of software to interpret and visualise data to create information.
Collection and interpretation

8 Acquire data from a range of sources and evaluate authenticity, accuracy and timeliness.

8 Analyse and visualise data using a range of software to create information, and use structured data to model objects or events.

10 Develop techniques for acquiring, storing and validating quantitative/qualitative data from a range of sources, considering privacy and security requirements.

10 Analyse and visualise data to create information and address complex problems, and model processes, entities and their relationships using structured data.
<table>
<thead>
<tr>
<th>Collect</th>
<th>Organise / create</th>
<th>Visualise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collect and explore data</td>
<td>Sort data</td>
<td>Present the data</td>
</tr>
<tr>
<td>Collect and access different types of data</td>
<td>Create information and solve problems</td>
<td></td>
</tr>
<tr>
<td>Acquire, store and validate different types of data</td>
<td>Interpret data to create information</td>
<td>Visualise data to create information</td>
</tr>
<tr>
<td>Evaluate authenticity, accuracy and timeliness</td>
<td>Use structured data to model objects or events</td>
<td>Visualise data using a range of software</td>
</tr>
<tr>
<td>Validating quantitative and qualitative data; considering privacy and</td>
<td>Model processes, entities and their relationships using</td>
<td>Visualise data to create information and address complex problems</td>
</tr>
<tr>
<td>security</td>
<td>structured data</td>
<td></td>
</tr>
</tbody>
</table>

2  
4  
6  
8  
10
Follow, describe and represent a sequence of steps and decisions (algorithms) needed to solve simple problems.

Define simple problems, and describe and follow a sequence of steps and decisions (algorithms) needed to solve them.

Define problems in terms of data and functional requirements drawing on previously solved problems.

Design a user interface for a digital system.

Design, modify and follow simple algorithms involving sequences of steps, branching, and iteration.
Define and decompose real-world problems taking into account functional requirements and economic, environmental, social, technical and usability constraints.

Design the user experience of a digital system, generating, evaluating and communicating alternative designs.

Design algorithms represented diagrammatically and in English, and trace algorithms to predict output for a given input and to identify errors.
 Specification and Algorithms

1. Define and decompose real-world problems precisely, taking into account functional and non-functional requirements and including interviewing stakeholders to identify needs.

2. Design the user experience of a digital system by evaluating alternative designs against criteria including functionality, accessibility, usability, and aesthetics.

3. Design algorithms represented diagrammatically and in structured English and validate algorithms and programs through tracing and test cases.
<table>
<thead>
<tr>
<th>Describe problems</th>
<th>Follow/design algorithms</th>
<th>Design user interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Follow, describe and represent a sequence of steps and decisions</td>
<td></td>
</tr>
<tr>
<td>Define simple problems</td>
<td>Describe/follow a sequence of steps and decisions (algorithms) needed to solve problems</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Design/modify simple algorithms (also) involving iteration</td>
<td>Design a user interface</td>
</tr>
<tr>
<td>Define problems using data and functional requirements</td>
<td>Represent algorithms using diagrams and English; trace algorithms</td>
<td>Generate and evaluate alternative designs</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decompose real-world problems; Consider economic, environmental, social, technical and usability constraints</td>
<td>Validate algorithms and programs through tracing and test cases</td>
<td>Evaluate designs against criteria: functionality, accessibility, usability, and aesthetics</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interviewing stakeholders to identify needs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementation

4. Implement simple digital solutions as **visual programs** with algorithms involving **branching** (decisions) and **user input**.

6. Implement digital solutions as simple visual programs involving branching, **iteration (repetition)**, and user input.
Implementation

8. Implement and **modify programs** with user interfaces involving branching, iteration and **functions** in a general-purpose programming language.

10. Implement **modular programs**, applying selected algorithms and **data structures** including using an object-oriented programming language.
# Implementation

<table>
<thead>
<tr>
<th>Concepts</th>
<th>Programming</th>
<th>Test and debug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branching (decisions) and user input</td>
<td>Visual programming</td>
<td></td>
</tr>
<tr>
<td>Iteration (repetition)</td>
<td>Visual programming</td>
<td></td>
</tr>
<tr>
<td>User interfaces and functions</td>
<td>General purpose text programming</td>
<td>In algorithms content descriptor</td>
</tr>
<tr>
<td>Modularity, algorithms and data structures</td>
<td>Object-oriented programming</td>
<td>In algorithms content descriptor</td>
</tr>
</tbody>
</table>
4 Embedded systems
What is an embedded system?

○ a dedicated computer that controls or monitors a larger mechanical and/or electrical systems, often in real-time.
○ from tiny (watch) to huge (power station)
○ the brain of most intelligent devices:
  ● toys, watch, phone, alarm, radio, TV, ...
  ● air con, washing machine, security, CCTV, lifts, ...
  ● cars, trains, traffic lights, automatic tolls, ...
  ● point of sale, barcode scanner, ATM, ...
  ● wifi router, ADSL modem, network switch, ...
A car is a network of embedded systems

- airbags
- air conditioning
- anti-lock braking (ABS)
- automatic windows, mirrors, ... seats
- cruise control
- collision detection and cameras
- engine control system(s)
- entertainment system
- display

...
Embedded systems (used to be?) low-power

- embedded devices often as small, low powered, robust, and cheap as possible
- minimum computer needed for the task

- the distinction is harder to make now e.g. landline -> mobile -> smart phones
- hardware constraints led to lower-level programming languages: assembler, C/C++, (some) Java
Teaching embedded systems

- *depends a lot on the device*
- *same algorithms / programming concepts*
- *lower-level of abstraction, more:*
  - binary and bit manipulation
  - direct access to hardware: interrupts, I/O pins, ...
  - simpler data structures (e.g. arrays)
- *hardware constraints visible:*
  - running out of RAM
  - 16-bit integers overflowing
- *digital (and analog) electronics and circuits*
Advantages (depending on device)

- cheap enough for kids to keep / take home
- real-world interaction and applications
  - projects with a purpose
  - integrate Science / Health and Physical Education
- engage physical and visual learners
- the “brain” of every maker(space) project
  - integrate with D&T / STEM projects
- ROBOTS!
- electronics is fun
- great for showing off
Disadvantages (depending on device)

- some devices are fairly expensive
- kids break components – learning!
  (need to consider parts as consumables)
- hard to tell if program or circuit is wrong
- the real world is hard to program!
- electronics isn’t in the (crowded) curriculum
- integration: D&T part can overtake DT part
- need to watch that concepts are covered
- C/C++ are not very beginner friendly
Most popular devices in schools

- Arduino – low-power
- Lego Mindstorms
- Raspberry Pi – high-power
- Intel Galileo / Edison / ...
Arduino/Genuino – low-power devices

- **Uno (most popular) ~ $37**
  - 2kB RAM, 1kB EEPROM, 32kB Flash ROM
  - 16-bit 16 MHz CPU
  - 14 digital I/O pins (6 for PWM) + 6 analog pins

- **101 ~ $52**
  - 24k RAM, 196 kB Flash ROM
  - 32-bit 32 MHz CPU
  - 14 digital I/O pins (4 for PWM) + 6 analog pins
  - Bluetooth low-energy

- **Mini / Mini Pro ~ $15**
  - similar to Uno in power, no USB or headers
Arduino Uno
Arduino Pro Mini
○ Arduino practicalities

○ programming is in basic C/C++
  ● main loop is hidden in setup/loop functions

○ program on Windows/Mac OS X/Linux

○ also Android devices:
  ● ArduinoCommander or ArduinoDroid

○ only one LED on the board

○ lots of “shields” to provide other hardware
  ● ethernet / bluetooth / wifi / sensors / ...

○ also wire onto a breadboard via headers
  ● for LEDs and other sensors
Esplorer board ~ $81
Raspberry Pi – high-power devices

- Released several models:
  - Versions 1-3 (released in last few weeks)
  - Model A and B (B/B+ is more powerful)

- Closer to a “full” computer
  - 32/64 bit CPU, 700MHz (B) -> 1.2GHz (3B)
  - 512MB RAM (B) -> 1GB (2,3B), MicroSD Flash
  - 40 general purpose I/O pins
  - 4 USB ports + ethernet
  - GPU + HDMI output (up to quite high resolution)
  - Version 3 adds Wifi and Bluetooth LE
  - Camera and display headers
  - Audio/composite display output
Raspberry Pi 2 Model B ~ $55
Raspberry Pi 3 Model B (just released) ~$62

- **Dimensions**: 85.6mm x 56mm x 21mm
- **40 Pin Extended GPIO**
- **Broadcom BCM2837 64bit Quad Core CPU** at 1.2GHz, 1GB RAM
- **On Board Bluetooth 4.1 Wi-Fi**
- **MicroSD Card Slot**
- **DSI Display Port**
- **Micro USB Power Input. Upgraded switched power source that can handle up to 2.5 Amps**
- **4 x USB 2 Ports**
- **10/100 LAN Port**
- **3.5mm 4-pole Composite Video and Audio Output Jack**
- **CSI Camera Port**
- **Full Size HDMI Video Output**
- Raspberry Pi is the computer
  - just plug in power, HDMI monitor/TV, USB keyboard/mouse
  - programming in many languages
    - Python is preferred / Ruby / C/C++
  - only one LED on the board
  - lots of “hats” to provide other hardware
    - the sense hat (aka astropi) is a really good one
  - also wire onto a breadboard via headers
    - for LEDs and other sensors
Raspberry Pi practicalities

- runs Linux, but also Windows 10 IoT Core
  - Linux is still not quite as beginner friendly
- have to boot up/down to modify circuit
- loose/broken circuit can lock up device
- friendlier programming options:
  - Python with APIs for GPIO and hats
  - Scratch with APIs for GPIO
Questions?

Find me at:
@drjamescurran / @groklearning
james@groklearning.com