International Journal of Computer Science Education in Schools, May 2017, Vol. 1, No. 2
ISSN 2513-8359

Learning basic programming concepts with Game Maker

Claire Johnson
claire.johnson70@virginmedia.com

DOI: 10.21585/ijcses.v1i2.5

Abstract

Game Maker is widely used in UK secondary schools, yet under-researched in that context. This paper presents the findings of a qualitative case study that explores how authoring computer games using *Game Maker* can support the learning of basic programming concepts in a mainstream UK secondary setting. The research draws on the learning theory of constructionism, which asserts the importance of pupils using computers as 'building material' to create digital artefacts (Papert, 1980; Harel and Papert, 1991), and considers the extent to which a constructionist approach is suitable for introducing basic programming concepts within a contemporary, game authoring context. The research was conducted in a high achieving comprehensive school in South East England. Twenty-two pupils (12 boys; 10 girls; 13-14 years old) completed a unit of work in computer game authoring over an eight-week (16 x 50 minute lessons) period. In planning and developing their games, they worked in self-selected pairs, apart from two pupils (one boy and one girl) who worked alone, by choice. Nine of the ten pairs were the same gender. Data were collected in planning documents, journals and the games pupils made, in recordings of their working conversations, and in pair, group and artefact-based interviews. Findings indicate that as well as learning some basic programming concepts, pupils enjoyed the constructionist-designed activity, demonstrated positive attitudes to their work, and felt a sense of achievement in creating a complex artefact that had personal and cultural significance for them. However, the findings also suggest that the constructionist approach adopted in the research did not effectively support the learning of programming concepts for all pupils. This research arises out of a perceived need to develop accessible, extended units of work to implement aspects of the Computing curriculum in England. It suggests that using *Game Maker* may offer a viable entry, and identifies the programming concepts and practices which pupils encountered, the difficulties they experienced, and the errors they made when authoring computer games. It also offers recommendations to increase the readiness with which students engage with key programming concepts and practices when using this visual programming software. In so doing it makes a practical contribution to the field of qualitative research in secondary computing education.

Keywords: Game Maker, visual programming, making computer games, Key Stage 3 computing

1. Introduction

A well-designed computing curriculum will introduce computer programming across a variety of contexts. In addition to learning textual programming languages, such as Python, to complete short 'closed' tasks (popular projects at lower secondary level may include creating chatbots, quizzes, calculators and drawing tasks (e.g. PG Online, 2013; Roffey, 2013a, 2013b)), pupils may benefit from using visual languages to undertake more 'open-ended' programming projects. This paper argues that extended projects such as making computer games are an important element of the computing curriculum because they give pupils the opportunity to engage with the design process, emphasising the importance of planning and testing as key programming practices, as well as developing pupils' computational thinking skills and building their resilience as learners.

1.1 Purpose and objective of the study

The purpose of the research was to explore the introduction of a new unit of work delivered as part of the curriculum in school, in which pupils created a computer game using *Game Maker* software (YoYo Games, 2007), a visual programming tool. Pupils were in 8th Grade, known as Year 9 in England. The unit of work followed was an implementation of a constructionist learning activity, characterised by its collaborative work pattern, extended time frame and personally and culturally meaningful outcomes (Kafai and Resnick, 1996). Pupils worked in pairs over an 8 week, 16 hour period to plan and make their games. In their pairs, they worked collaboratively, in so far as they pursued a single goal, negotiating and sharing their conceptions of the task and how to tackle its elements, co-constructing their understandings through interactions with each other and the software. At other times, they worked

cooperatively within their pairs, pursuing separate tasks, or with other members of the class, viewing each other's work in progress, sharing their knowledge and showing others how to solve problems or achieve particular effects.

The research considers whether the game authoring activity supports the learning of basic programming concepts in a mainstream secondary setting and seeks to answer the following questions:

1. What programming concepts and practices do pupils encounter when authoring computer games using *Game Maker*?

2. What difficulties do pupils experience and what errors do they make when programming computer games using *Game Maker*?

3. To what extent is a constructionist approach suitable for this kind of work?

2. Related work

Game Maker is widely used in UK secondary schools, but under-researched in that context (Johnson, 2014). Although a growing body of literature internationally refers to *Game Maker*, most studies provide little detail of the programming concepts learned when using this visual programming tool, and are rarely situated in the mainstream secondary phase.

Research conducted in the United States reports how *Game Maker* has been used at tertiary level to introduce computing concepts associated with game implementation (Chamillard, 2006; Dalal et al., 2009). In this context, *Game Maker's* graphical interface was found to be useful for introducing programming concepts before transitioning to a textual language and resulted in improved student performance in programming assessments (Hernandez et al., 2010; Dalal et al., 2012). Other US research describes how *Game Maker* was used in a summer camp for pupils in Years 6-12 (n=18) (Guimaraes and Murray, 2008) and highlights the importance of allowing students to practice reading and modifying code in sample games before they engage in code creation themselves, noting that students are usually given the task of creating programs before they have learned how to read and understand them.

US researchers have also investigated how *Game Maker* can be used to address learning objectives in other subjects, as well as supporting the learning of computer science concepts (Doran et al., 2012). This study describes the evolution of a 10 week out-of-school game authoring programme and makes the following recommendations: i) give pupils time to plan and write the pseudocode for their program segments before they implement their games; ii) include 'guided errors' to increase pupils' debugging abilities (pupils responded best when they were encouraged to make mistakes rather than avoid them) iii) clarify the sorts of games pupils can realistically create and include more structure, targeted lessons and more development time.

At primary level, Baytak and Land's work investigates how authoring games with *Game Maker* can enhance the learning of science (Baytak et al., 2008; Baytak and Land, 2010; Baytak et al., 2011). This case study research follows Year 5 pupils (n=10) who make games to teach younger pupils about nutrition. Findings show that the activity was engaging for pupils and allowed them to represent their knowledge about nutrition in concrete and personally meaningful ways (Baytak and Land, 2010). However, there were challenges, notably with implementing game designs with limited programming skills. While the research observes that pupils used increasing numbers of actions in their games as the project progressed, there is no reference to learning about programming beyond this. *Game Maker* also features in research surrounding the development of a 'computational thinking' curriculum (Jenson and Droumeva, 2015) at primary level. The instructional framework followed focusses on variables, functions, mathematical operations and conditionals. Results of pre- and post-tests show that CS knowledge improved over the 15-hour intervention, but the authors emphasise the need to provide a structured and scaffolded curriculum that includes direct instruction of computational concepts in addition to self-directed learning.

Seaborn et al., (2012) describe the development of a game construction curriculum to replace a traditional secondary computer science class. High school students (n=12) were taught elementary programming using *Game Maker* over a six month semester and worked in groups of 3 to create 3 computer games, alternating their roles as artist, designer and programmer for each. In addition to collecting students' overall impressions, they evaluated students' technical competency and self-efficacy at the start and end of the semester. Their findings show that the curriculum had a positive, statistically significant effect on CS concept comprehension, but no detail is given about these concepts. Other *Game Maker* research does not investigate the learning in programming that is achieved when pupils create computer games. Rather, the focus is on how the software has been used to enhance particular aspects of learning,

such as creativity (Eow et al., 2010), digital literacy and multi-literacies (Sanford and Madill, 2007; Beavis and O'Mara, 2010; Beavis et al., 2012; O'Mara and Richards, 2012), multimedia design (Beavis et al., 2012), game design (Redfied and Uhlig, 2012) and how the program has been used as a motivation for learning in other subjects (Fluck and Meijers, 2006; Baytak et al., 2008), or to enhance collaborative working practices and promote social constructivist learning environments (Madill and Sanford, 2009). There are few studies which focus on *Game Maker* and how it is used to introduce programming concepts in the UK secondary curriculum (Hayes and Games, 2008; Daly, 2009) and few studies of whether authoring computer games increases young people's understanding of computer science concepts (Denner et al., 2012; Seaborn et al., 2012), or what kind of knowledge students learn from creating games using visual programming languages (Koh et al., 2010). Moreover, there are few studies which look at the learning of computing concepts through game authoring within a classroom setting (Wilson et al., 2012).

This paper then, addresses a gap in the literature relating to the use of *Game Maker* in the lower secondary school/middle school IT/Computing curriculum.

3. Methodology

3.1 Research design

The purpose of the research was to gain an understanding of what pupils learned and what difficulties they encountered when making a computer game using *Game Maker* - it is therefore a qualitative enquiry. Case study was selected as the research method since it allowed the study of an evolving situation, namely, the introduction of a unit of work in game authoring with a group of Year 9 pupils. A single case design was chosen on the basis that the class selected is a 'typical' case of a wider population of Year 9 pupils. Lessons learned from typical cases are assumed to be informative about the experiences of the average [child/class] (Yin, 2009: 48).

While the case study method is criticised for lacking reliability, validity and generalisability, these are not the chief concern of qualitative research (Merriam, 1998); rather, the focus is on understanding the particular case (Evers and Wu, 2007: 201). To strengthen the reliability of the method in the face of such criticisms, Yin recommends the development of a case study database (Yin, 2009: 45) to store data and procedures followed, so that the research could be replicated. For the current study, a database of pupil voice recordings and interview data, transcripts, interview schedules, and the coding system used at the analysis stage was created and stored in *NVivo 8* (QSR International, 2008). Additionally, documented research procedures, data collection guidelines and a lesson sequence were produced, which serve to strengthen the reliability of the research.

A framework for the analysis of programming concepts evidenced in the games authored was constructed with reference to documents defining generic computer programming concepts appropriate for the students within this age group (e.g. OCR, 2011; Seehorn et al., 2011; CAS, 2012; Edexcel, 2012; NAACE, 2012; Saeli et al., 2012) and is presented in Table 1.

Table 1: Concepts used to frame the analysis of programming constructs

Programming concepts	**Definition**
Program interaction	Input/output, event driven. Events are used as input data.
Functions (actions)	Actions are used to create outputs in the game.
Sequence	Events and actions are sequenced in a sensible order.
Conditional statements	*Test/check* actions are used to test conditions.
Loops	The *step/alarm* event or *repeat* action is used to create a loop.
Variables	Variables (e.g. score, lives) are used to store data in the game.

Logical operators	Logical operators (AND, OR, NOT) are used.
Boolean logic	Boolean logic (true, false) is used.
Relational operators	Relational operators (=, <, >) are used in expressions.
Mathematical operators	Mathematical operators (+, /, *, -) are used in expressions.
Coordinates	Coordinates are used to specify screen position (x, y) of objects.
Angles	Angles are used to specify direction of movement of objects.
Negative number	Negative number is used (e.g. to define speed, position, score).
Randomness	Randomness is used (e.g. to define position or number).
Relative/absolute value	Relative/absolute value is applied to define score or position.

3.2 Sample

The research was conducted with one mixed ability year 9 (8th Grade) class (n=22; 12 boys, 10 girls; 13-14 years old) at a comprehensive school in South East England. Ten of the pupils in the group achieved 'above average' values in their average Cognitive Abilities Test (CAT) scores, which suggests that the group was of above average ability with respect to national profiles.

The sample was achieved by selecting an 'accessible', 'ordinary', 'typical' case (Creswell, 2007). Purposive sampling was achieved within the case in terms of which pupils were selected as members of the interview groups, and for the paired interviews. Three boys and three girls were selected for each group interview, and of these, two were selected from each of the higher, average and lower ability ranges. For the paired interviews, four boys and three girls were selected to represent a similar ability spread. Seventeen of the twenty-two pupils in the class were interviewed either as part of a pair or a group. Pupil voice recordings, authored games and documents were not sampled; all units produced were included in the analysis.

Pupils worked in self-selected pairs over an 8-week, 16-hour period to plan and make their games. Nine of the ten pairs were the same gender; two pupils worked alone, by choice. The lesson planning broadly followed the system development life cycle and was structured to provide a frame and focus for each lesson, a mix of teacher-led, independent and pair work, a range of video, print and computer-based resources; an integration of written, oral and computer-based activities included playing sample games and following a structured set of video tutorials (Jones and Wilson, 2008) to learn to use the software, before planning and making an original game. Homework was set once a week and asked pupils to write about their work in progress and to describe any problems or difficulties they experienced.

3.3 Data collection and analysis

Within the case study, several methods of data collection were selected to strengthen the internal validity of the data: data were drawn from recordings of pupils' working conversations, pair and group interviews, and by scrutinising the planning and design documents they produced and the computer games they created.

Data set

i) Ten transcripts of digital voice recordings of pupil pairs' working conversations (4 hours, 28 minutes).

ii) Two transcripts of semi-structured interviews with two focus groups of 6 pupils (3 boys; 3 girls) talking about their game authoring experience (2 x 43 minutes).

iii) Three transcripts of artefact-based paired interviews, in which pupils' games were loaded and used as the focus (1 x 39 minutes, 1 x 33 minutes, 1 x 53 minutes).

iv) Twelve authored games.

v) Eighty-five pupil documents. Pupils documented their reflections on aspects of their work in an ongoing written journal and completed planning documents (storyboard, game design document, game interactions).

vii) Observation notes recorded throughout the fieldwork.

Transcripts of pupils' voice recordings, interviews and written documents were coded for references to programming concepts and areas of difficulty encountered. Games were play-tested and an initial analysis identified components of each game and categorised actions and events used. Programming constructs evidenced (see Table 1) were recorded for each pair. A written log of the program code was annotated, to identify which elements functioned as intended and what errors were made.

4. Findings

Although *Game Maker* was designed to enable users to create computer games without the need to learn a 'difficult' textual programming language, the young people in this study found some aspects challenging. Analysis of the data shows that, in general, programming errors most frequently occurred due to a lack of precise, logical thinking and a lack of testing/checking. Pupils were not used to thinking algorithmically, decomposing problems, or reading and evaluating their code because these practices had not yet been embedded in the research school.

4.1 Program design

Before they began to make their games, pupils were asked to plan the game interactions by listing objects and specifying the events and actions for them. Some pupils did not complete this task effectively because they were unaccustomed to decomposing programs into their constituent parts, and were not practised in applying precise, logical thinking when planning the interactions in their game. They were also reluctant to spend time on planning tasks because they wanted to begin making their games in the software.

Pupils' initial plans were characterised by incompleteness (not all objects in the game were listed, not all events or actions were visualised or described). Pupils sometimes conflated events and actions, did not break down object behaviour into separate events, or assigned multiple actions to one event, instead of to separate, distinct events. This 'merging' of separate processes is found to be a common source of error in novice programmers (Spohrer and Soloway, 1989).

Later in the planning process, pupils began to separate events and actions, and introduced a wider range of inputs into their plans (for example, they included non-user inputs such as conditional statements, as well as user-controlled inputs, such as a key press). This suggests that they were beginning to 'think computationally', and that an understanding of the need for decomposition and precision in program design was beginning to emerge.

4.2 The language of programming

In their initial planning documents, most pupils did not appropriate the language of *Game Maker*, or the terms they had come across in the tutorials they followed, which made their plans less supportive to them in the implementation phase. Some pupils misinterpreted the context specific meaning of words like 'event', 'action' and 'room'. For example, they understood the word 'event' to mean 'something which happens' in the narrative of the game, rather than an input. This misunderstanding of natural language terms in programming contexts is identified in the literature as a common source of error in novice programmers (see du Boulay, 1986; Pea, 1986). However, 4/12 pairs used correct terminology in their plans; these pupils also produced the most complete games.

Although not all programming terms are made explicit in *Game Maker*, making a game introduced pupils to some programming concepts (objects, events, variables) and also words to describe states, behaviours and interactions (solid, visible) and aspects of game design (collision, sprite, room, challenge, goal). In the interview transcript, more than half of pupils (12/22) made references to enjoying using this domain-specific visual language and becoming increasingly fluent in it. New words gave them access to new concepts and pupils began to use these words as their understanding of computational concepts emerged, such as one pair who confidently discussed their use of variables.

However, not all pupils found this 'new language' easy to embrace. For some the specialised language was a barrier and they avoided using actions whose referents they did not understand, and did not make use of error message text or action definition text to further their learning.

4.3 Learning programming concepts

Using *Game Maker* introduced pupils to several basic programming concepts and gave them some understanding of the precision and detail required in constructing game programs. Figure 1 illustrates how *Game Maker's* visual environment represents some of these programming concepts.

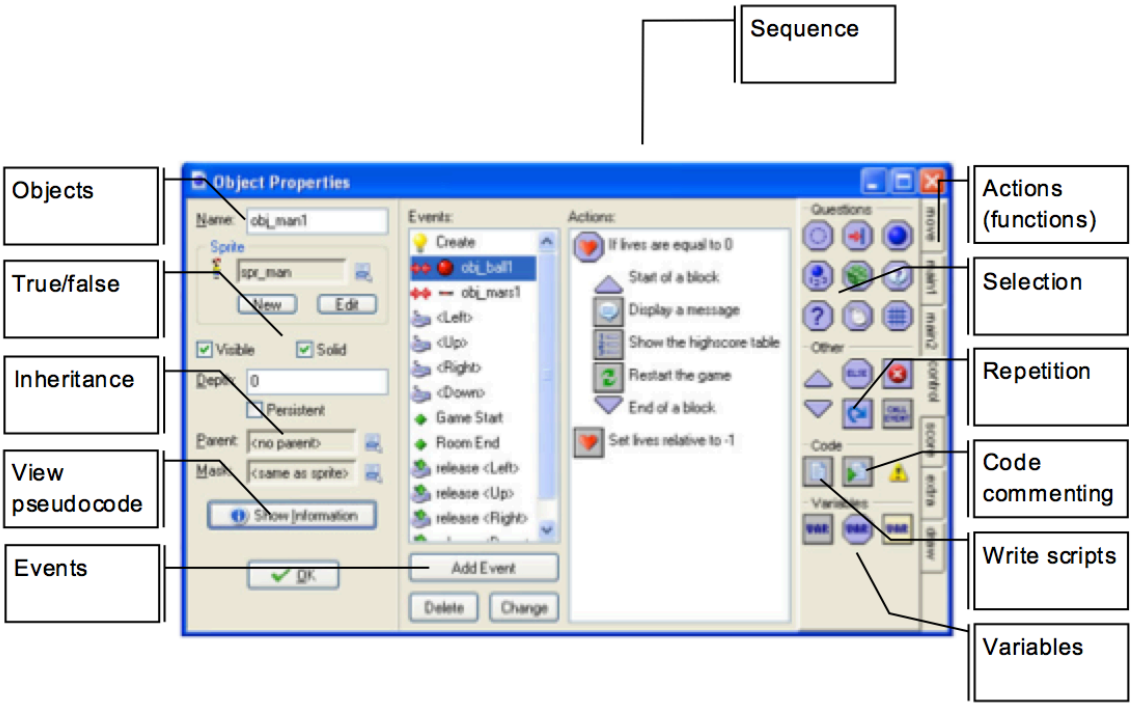

Figure 1: Programming constructs in Game Maker

Table 2: Programming concepts evidenced in authored games

Programming concept	No. of games	Comment
Program interaction (input/output, event driven)	12	All games contained events as triggers for game action (range = 5-84; mode = 11-20).
Functions (actions)	12	All games contained functions (actions) (range = 5-170; mode = 11-30).
Sequence	12	All games involved sequencing actions.
Variables	12	All games included at least one variable (speed, score, lives, health, position x/y, gravity).
Boolean logic (true, false)	9	True/false values were used in nine games to loop sound or to set objects as solid.
Coordinates	9	Coordinates were used to define object location in nine games.
Relative/absolute value	9	These values were used in nine games to add or subtract values from score, health or lives variables; to set speed and specify position.

Negative number	8	Negative number was used to refer to direction, position or to set the value for variables (e.g. score, lives, depth, speed) in eight games.
Conditional statements	6	Half of all games included at least one conditional statement.
Loops	6	Five games included a *step* event as a looping structure. In one game the *alarm* event was also used to repeat an action.
Relational operators (<, >, =)	6	Relational operators were used in expressions in six games.
Randomness	5	Randomness was used to define object position or creation of an object in five games.
Angles	4	Angles were used to define direction of movement in four games.
Logical operators (AND, OR, NOT)	1	The logical operator 'NOT' was used in one game.
Mathematical operators (+, -, /, *)	1	Mathematical operators were used in expressions in one game.

The following sections elaborate on the findings associated with the programming concepts in Table 2, as used in the games pupils created.

Events

In *Game Maker*, all program interaction is achieved by selecting events (user inputs such as a key press, or non-user inputs, such as a collision between two objects). In learning to use these events, pupils were introduced to the idea of event-driven programming and to the key patterns of interaction in a game program (see Figure 3).

Figure 3: The event selector

Whilst pupils were used to the idea that the keyboard and mouse are input devices, in making a game program they learned that inputs can also be controlled by non-user events, such as collisions, conditions and other game states. This expanded understanding of inputs was important learning. Pupils found it easier to understand those events, which are user-activated (i.e. *keyboard/mouse* events), than those which are not (i.e. *step* and *alarm* events). Problems occurred with the use of events because pupils sometimes confused *events* with *actions*, chose the wrong event, duplicated events in more than one object, or used conflicting events.

The average number of *different* event types used in each game was 5, although the average total number of events used was 23. The most frequently used event was the *create* event (see Figure 4), commonly used to set an object in motion when the game is run, or to set variables (such as score or lives) for it. The *mouse* event was used correctly in 9/12 games, usually to select a menu button, or to navigate between screens.

Keyboard events were used in 7/12 of the games, typically to control the movement of the player character using the arrow keys. Some pupils who used *keyboard* events had less success in controlling the stop/start movement of their player characters, since they did not implement an event to control the *stopping* of movement.

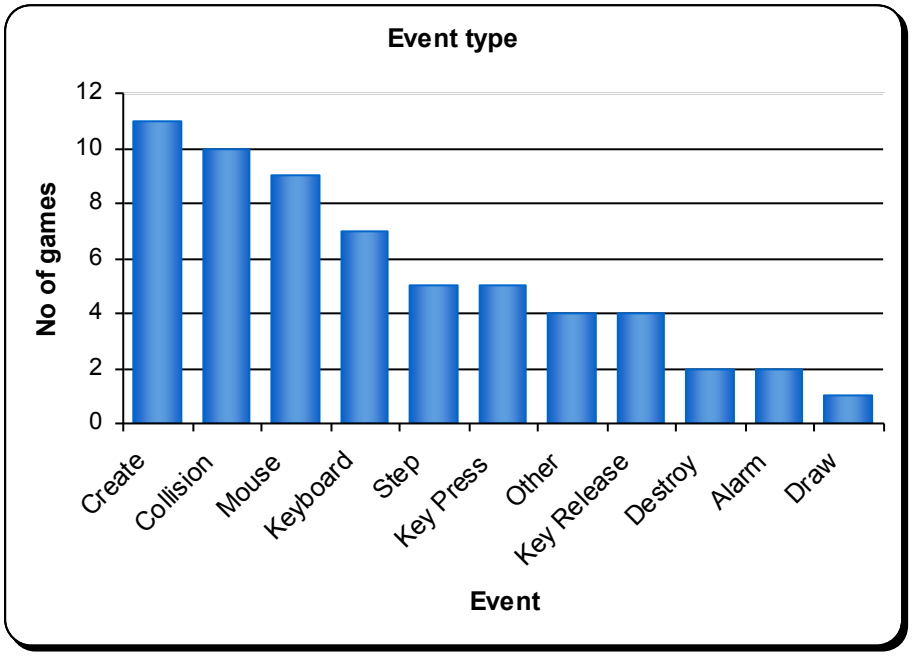

Figure 4: Type of events used in the games

Key press and *key release* events were used in 5/12 games to control the movement of an object. The *key press* event was also used to create an instance of an object when the space bar was pressed, to give the appearance that a missile had been fired, for example. The *collision* event was used in 10/12 games as a mechanism to make objects disappear, to collect items or gain points and to decrease lives or score.

The *alarm* event was used in two out of twelve games to make things happen from time to time, without user input - for example, to set an interval between bullets firing. The *step* event was used in 5/12 games, most often to check values relating to object position and then to perform a particular action. For example, one pair used the *step* event to make objects on a scrolling background reappear at random positions on the screen when they had disappeared from view:

Obj_rocks
Step event:
If y is larger than room_height
  Jump to position (random(room_width), -120)

The *step* event was also used to repeat an action, such as destroying instances of objects once they have disappeared from view or creating objects intermittently, for example:

Obj_enemy
Step event:
If y is larger than room_height + 32

Destroy the instance
With a chance of 1 out of 180 perform the next action
Create instance of object obj_s_enemybullet at relative position (0, 16).

Such use of the *alarm* and *step* events introduced pupils to the programming concept of repetition, and illustrated alternative mechanisms for controlling this pattern. Pupils learned that within the game loop, certain events occur continuously or repeat if certain conditions are met or game states are reached.

The *other* event was used in 4/12 games. Use of this event introduced the idea that game inputs are not only achieved by user input but also by game states (i.e. when there are no more lives, when a level is completed, when an animation ends).

The *draw* event was used in one game to display the score, health and lives graphics on the screen. However, the use of this event was not intuitive. Pupils did not understand that these items are displayed on screen by assigning a *draw event/draw life images* action to an invisible 'controller' object, or that a separate object was needed to display them.

The correct use of these events suggests that pupils understood the idea of simple, event-driven programming involving the concrete use of the mouse or keyboard, or the collision between two objects, as inputs. They also learned that outputs can be controlled by non-user inputs and game states, however, such events are more abstract and were used less frequently.

Objects

Using *Game Maker* introduced pupils to the concept of object-oriented programming - they learned that a game is made up of objects, which are programmed entities. However, while they found it straightforward to view the player character and other game resources as objects, some found it more difficult to understand that interface controls, such as 'start' or 'exit' buttons, were also programmable objects or that invisible objects can be created to manage other game resources, such as governing the display of score, health and lives graphics.

In particular they learned that for the user to be able to interact with objects, they had to be created as separate entities. This was not immediately obvious to some:

> "*I didn't realise you had to have rooms for the game to be made and have all the sprites and objects and have them all separately. Lots of different parts of it, that you have to build up layers to the game*." (Pupil A)

Some pupils did not initially understand that the visual appearance of the game is separate from its underlying functionality. They learned that in *Game Maker*, objects, rather than sprites (the visual appearance assigned to objects) hold programmed behaviour.

> *"There are some things that aren't really sort of logical in the first place, but you can understand them after a while … like having a sprite and then an object. I dunno, the sprites don't seem to do much on their own."* (Pupil B)

This idea that the visual appearance of a computer game is separate from the underlying program behaviour was important learning for pupils. In making a game, they began to develop an understanding of what goes on 'behind the scenes' of the technologies they use at home:

> *"Yeah, 'cos when you play a game you just take it for granted, really, as something that just … works. I didn't even know you could make a game. I've never had any experience of that ever*." (Pupil C)

Actions

Specifying the actions, which objects should perform, is the central programming task of creating a game in *Game Maker*. In using actions, pupils learned to construct their game program in individual steps and gained practice in

sequencing instructions. Errors sometimes occurred when pupils duplicated or used conflicting or incomplete actions, or had difficulty in setting the correct parameters or arguments to achieve required behaviours.

The most commonly used actions were those which define object movement. Other commonly used actions were related to object destruction or progression between levels. *Test* or *set score* and *lives* actions were also widely used.

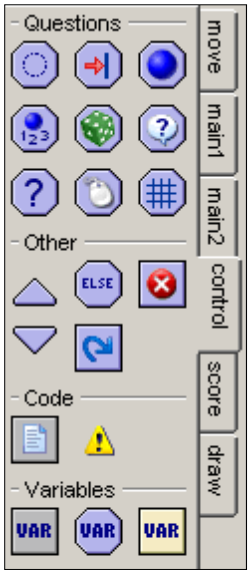

Figure 5: The control actions

More abstract actions such as *test expression* and *set alarm* were used less frequently.

Most of the actions used were pre-programmed (see Figure 5). However, the *execute script* and *execute code* actions can be used to introduce pupils to writing functions themselves using *Game Maker's* textual programming language, GML. In this study, only one pupil used the *execute script* action, sourcing a script from the *Game Maker Community* forum (Overmars, 2003). In fact, there was little support for using these actions in the teaching sequence followed or the commercial resources provided (e.g. Giles et al., 2008; Jones and Wilson, 2008; Reeves, 2008; Waller, 2009); more recent resources similarly do not feature the use of scripts in their core content (see PG Online, 2014) and this is an area for development in resources and sequences of lesson plans which make use of the software.

Parameters and arguments

In *Game Maker*, once an action is selected, parameters or arguments need to be set for it. Some pupils found this challenging and much of their working conversation was concerned with what values to use:

> *"When you drag [an action] across it comes up with an option about all the different settings that you can add to it and that's what's hard, because you've got to work out which settings it needs."* (Pupil D)

> *"I used to [wonder why] computer games used to take so long to come out, and now I know it's 'cos ... every little bit in there needs to have, like, loads of complicated things just to do that."* (Pupil E)

The idea that behaviours, such as speed and direction, have to be defined for an object in order for it to move was also new. In *Game Maker* these behaviours are defined as properties of an object, and involve pupils making decisions and having to think logically about the effects of those decisions:

> *" What I mean is, when you drag [an action] over you've got to actually properly say what you want it to do … you drag the [action] across that you want, but it's actually putting the text into that box to say 'Actually, I want it to do this'…"* (Pupil D)

Setting the parameters for actions introduced more abstract concepts, such as whether a value is relative or absolute, for example. The concept of relative value was most often encountered when pupils wanted to program a score mechanic for their games. They learned to set the score relative to its current value, rather than to an absolute value and this was new thinking for some. In nine of the twelve games, relative values were applied to a variable, to add or subtract from the score, to subtract lives or to decrease health. Relative value was also used to specify object position in five games and in setting the speed in one game, where it was used in error.

Sequence

Creating a game in *Game Maker* involves selecting events and actions for an object and putting them in a logical order, since they are executed sequentially from the top, downwards. Pupils learned about the importance of sequence when, for example, errors in the sequence in which events or actions were ordered meant that the game did not function as intended and when the sequence in which rooms were ordered in the resources tree affected which room was displayed first when the game was run. In this respect, using *Game Maker* supports the development of algorithmic thinking; pupils learn to define specific instructions for carrying out a process, in a visual and/or textual format (see Figure 2).

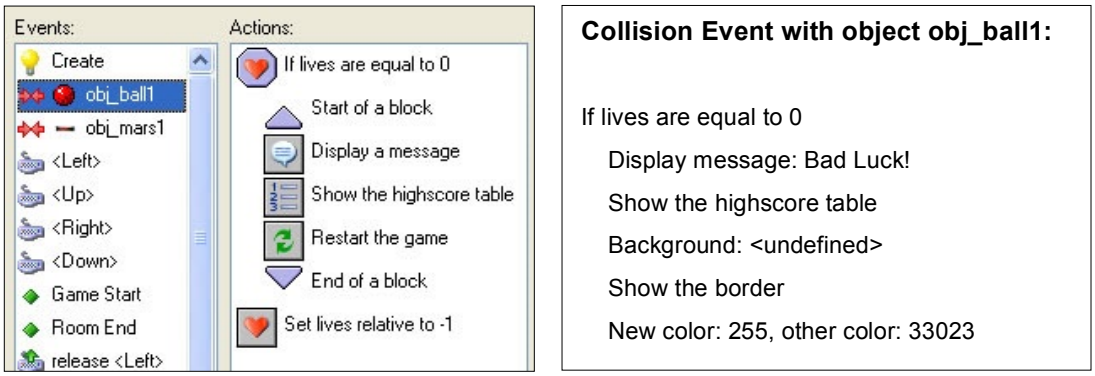

Figure 2: Game Maker's visual and textual information

Variables

Pupils learned that the use of variables is important in computer games, since the player's score or health, an object's speed, direction and position, for example, have to be defined and stored in the game for it to give meaningful game play. In *Game Maker*, several commonly used local variables (x, y, speed, direction, gravity) and global variables (score, lives, health) are inbuilt - they do not have to be declared as is normally the case in textual programming languages. However, although this makes their use straightforward, it 'hides' the underlying concept. All pupils in the study used at least one inbuilt variable but some may have done so without understanding. Most pupils (20/22) did not use the term 'variable' to refer to these features. Variables were not always set or tested correctly and only one pair attempted to create a variable after following a tutorial. These findings suggest that the concept of variables and the role they play in game programs needs to be explicitly taught when using *Game Maker*.

Conditional statements

Conditional statements are achieved in *Game Maker* by selecting one of the *test* or *check* actions that test or check a game state and then trigger one or more actions if the condition is evaluated as true (see Figure 6). While 6/12 games included at least one conditional statement, some pupils found it difficult to implement this construct, suggesting that aspects of games which make use of conditions need to be clearly modelled if they are to be successfully used by all.

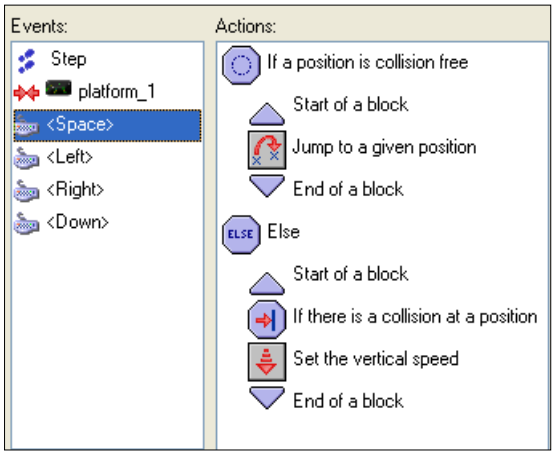

Figure 6: A conditional statement

The most common conditional statement used in the games was the *test variable* action, which was used in 4/12 games; t*est lives* and *test score* actions were also used in five games. The *test variable* action was most often used to check the position of an object on the screen to see if it had passed beyond the boundary of the room, in which case, it would reappear at another location, to remain visible:

Obj_tree
Step event:
If y > room_height
  Move to position (random(room_width), -65).

Loops

Another key concept which pupils encountered is that some processes within a program need to be repeated. This is achieved using a 'loop' construct. *Game Maker* operates a continuous loop during game execution and by using the *step* event, pupils can specify what actions they want to occur in each step of the loop (Chamillard, 2006). Six pairs in this study used the *step event* for this purpose. For example, one pair used the *step* event to make an object reappear after it had disappeared from view. In this case, the *step* event checks the position of the object every second and relocates it every time it disappears beyond the visible area of the game room:

Obj_snowboarder
Step event:
If y > than room_height
  Move to position (random(room_width), -65)

Use of a conditional statement also allows code to be executed repeatedly based on a Boolean condition (true or false) (Kuruvada et al., 2010a). Six pairs used this method to achieve a loop construct.

Using actions such as these taught pupils how many factors have to be considered when creating a game program and the importance of precise, logical thinking in setting arguments and parameters. It also developed their understanding of the structural patterns used in programs, such as conditional statements, loops, and variables.

4.4 Use of mathematical concepts

As well as learning about the programming concepts described above, pupils learned that some mathematical concepts are important in game programs and that these are often used in setting the parameters, arguments and expressions of an action.

Coordinates

The pupils in this study had some prior knowledge of coordinates from their learning in other subjects, but developing a computer game where spatial boundaries are mapped and object position is specified using coordinates was a new application of that knowledge. Pupils learned to conceptualise the room, rather than the screen, as the game space. They learned that an object's position is defined by *x* and *y* coordinates and how to use these values to prevent objects from disappearing from view or to make objects reappear, once they had travelled off screen.

Coordinates were used in 9/12 games to define object position, to indicate the screen location of health or lives graphics, or to move an object to a particular position, as in the following examples:

Obj_player character
Collision event with obj_tree:
Set the number of lives relative to -1
Move to position (320, 48)

Obj_controller
Draw event:
Draw the lives at (16, 420) with sprite spr_life
At position (180, 440) draw the value of score with caption

Angles

Another mathematical concept that pupils met in a new context was the use of angles to specify an object's direction of movement. Pupils most often set this value by selecting directional arrows in the *move in directions* action. At other times, angles needed to be specified. Angles were used to define direction in 4 games, such as in the following example where an angle of 270 is used to determine a downwards movement:

Obj_player character
Step event:
If relative position (0, 1) is collision free for only solid objects
  Set the gravity to 0.5 in direction 270

Negative number

Negative number was used in 8/12 games for several purposes: in three games to refer to direction of movement, where a negative value equates to a move down (-*y*) or to the left (-*x*); in three games to define object position and in five games to decrease the value for score, health or lives variables. Negative number was also used to specify the depth and the vertical speed of an object. Using negative number and understanding its effects for these various purposes was a new way of thinking for most pupils.

Randomness

Pupils had met the concept of randomness in their mathematics lessons, but in making a game they learned that this quality could be usefully applied to enhance game play. The idea that random behaviour can be programmed was novel to pupils. Three pairs used random values to control the reappearance of objects on screen once they had disappeared from view. They learned to set the *x* coordinate of the object so that it reappeared at random positions across the room width:

> *"The jump to given position function is set to x = random (room_width) and y = -50 ... [so] that the cars appear in random positions above the screen. This eliminates the look of repetition that games can sometimes have."* (Pupil B)

Randomness was used to define object position in five games, for example, when an object was set to *jump to a random position*, following a *collision* event. One pair used the *test chance* action to randomise the creation of an object, so that this could not be predicted by the player.

Boolean logic

Pupils also learned that Boolean logic is used to define certain object properties, using true/false values, for example, to set an object as 'solid' or 'persistent' or to specify whether a sound should loop or not. In all the games 'true' values were used to define an object as visible. In two games this property was set to 'false' to make an object invisible. The idea that such properties have to be specified was a new way of thinking for pupils and strengthened their understanding of the precision and detail required in constructing computer game programs.

Boolean logic is also implied in the use of conditionals where a condition is evaluated as either 'true' or 'false'. This binary construct is a common feature of multiple computing processes and becoming aware of its various applications developed pupils' ability to think computationally.

Relational and mathematical operators

Relational operators (<, >, =) were used in 6/12 games, often to test the value of a score, lives or x/y coordinates. Mathematical operators (+, -, /, *) were used in one game to test the coordinates of an object and to set the speed of an object.

4.5 Program organisation

In creating a game, pupils were not only introduced to programming concepts, but also to practices relating to program organisation.

Naming conventions

In *Game Maker*, prefixes such as spr_, obj_, back_ are used to name and identify different types of game components. The resources used in the study introduced pupils to these naming conventions and 7/12 pairs used them effectively most of the time. However, some pupils did not initially understand the need for correctly naming their game components or realise that to do so is useful in terms of managing game assets, referring to objects in the game program and for reading or checking program code.

Code commenting

Although code commenting was not covered in the lesson sequence, some pupils learned about the practice by viewing sample game code. One pair added a *comment* action to their game to remind themselves what the code meant:

Obj_player character
Step event:
COMMENT: Check whether in the air
If relative position (0,1) is collision free for Only solid objects
  Set the gravity to 0.5 in direction 270
Else
  Set the gravity to 0 in direction 270
COMMENT: Limit the vertical [sic] *speed*
If vspeed is larger than 12
  Set variable vspeed to 12

Another pair reused code that contained comments to clarify it:

```
// The direction the sprite faces (down, left, up, right)
direction_faced = "down";

// The current action (none, walk, run)
action = "none";
```

However, pupils would have benefitted from greater focus on this aspect of programming. Adding comments encourages pupils to read and check their code more closely and gives them useful practice in documenting their understanding of the programs they create, an important part of learning to program (CAS, 2012).

4.6 Testing and debugging

As they created their games, pupils were continually testing them to see if the events and actions they applied to objects produced the desired outcomes. Some pupils checked their code and identified obvious errors, but others did not. Observation notes record that generally, pupils were not systematic when trying to correct errors and achieved much of their debugging by trial and error. More emphasis needed to be placed on reading *Game Maker's* textual object information to eliminate obvious errors.

Pupils also needed more support to read and understand *Game Maker's* error messages, which identify the reason for the error, the object where the error occurred, the event where the error occurred and the number of the action which caused the error, as shown in the following example:

> FATAL ERROR in
> action number 1
> of Mouse Event for Left Button
> for object instructions_obj:
>
> COMPILATION ERROR in code action
> Error in code at line 2:
> Move Patrick around using the arrow direction buttons on the keyboard at position 2: Assignment operator expected.

From observation of pupils, they found such error messages discouraging and avoided reading them; only one pair ran their game in debug mode to identify errors. While constructionist learning theory asserts that pupils need to be given the freedom to get things wrong (Papert, 1999), since programming is a continual process of debugging, the data reported here suggest that the planning of lessons needs to focus more closely on showing young people how to approach errors as a source of information, rather than as a problem.

5. Discussion

Based on the observations above, this section makes recommendations for how to improve teaching sequences, which use *Game Maker* to introduce basic programming concepts. It also considers the extent to which constructionist approaches are suitable for this kind of work. The findings of this study suggest that most pupils appeared to need support with being specific and precise at the planning stage, in listing the objects, events and actions required in their games and in using the correct terminology to refer to them; in short, more emphasis needs to be placed on program design, so that pupils effectively plan the game interactions, before they begin to implement their game. This finding is supported by other studies, which also acknowledge children's reluctance to engage in or make use of planning work and their preference for focusing on aspects, which give immediate feedback and satisfaction, such as graphics and animation (see Howland et al., 2013).

Whilst *Game Maker* provides a concrete, visual representation of programming constructs, the findings reported above suggest that some additional theoretical input is necessary to ensure that the underlying concepts have been understood. This can be achieved by encouraging pupils to read the textual information that corresponds to the graphical code they produce (see Figure 2) and to annotate the programming constructs they use. In so doing, pupils practise using programming terms and interpreting a pseudocode equivalent to the visual action icons they select. This encourages them to develop and check the logic of their games and takes them one step closer to expressing code in a textual format.

To support the development of their own games, whilst drawing attention to key programming concepts, teaching sequences need to incorporate a range of scaffolded activities - for example: provide code walkthroughs for common game mechanics and the programming constructs required to achieve these; introduce code reading and code writing tasks, where pupils work with partially completed programs to extend functionality or correct errors; show pupils how to read error messages and/or run their games in debug mode. This would ensure that pupils' preferences for practical work are met at the same time as providing targeted support for making their games. While such approaches have been successfully used in academic studies related to the use of *Game Maker* (e.g. Guimaraes and Murray, 2008; Hernandez et al., 2010), they are rarely recommended in the educational resources available for the software, which show pupils how to make a game, but do not draw out the underlying programming concepts. To remedy this, project briefs need to specify the programming concepts that pupils should use in their games. For

example: a score must be set to introduce the use of variables; a score must be tested to illustrate the use of a conditional statement; an action must be repeated to show the application of a loop in a game program, and so on.

The findings of this study also underline the importance of using correct terminology to refer to programming concepts when using visual languages such as *Game Maker* and suggest that the 'language of programming' needs to be made more explicit in teaching sequences, especially where those terms are hidden by the software. For example, *Game Maker's step events* or *alarm events* hide the program iterations/loops which they generate; *test/check* actions hide that they are conditional statements; common variables are set by default for all objects - but the word 'variable' is not used to refer to them. Such key words need to be brought into use early on. Pupils should be encouraged to use technical terms in their design documents and throughout and teachers need to articulate the programming knowledge that pupils have acquired by drawing attention to the language of *Game Maker's* event selector (see Figure 3) and action icons, particularly the core programming constructs of conditions, variables and loops (see Figure 5). To do so gives pupils an insight into some of the building blocks of computer programs, and demystifies the language used. As pupils begin to use the vocabulary and language of programming, so they begin to think computationally (Grover, 2011) and realise that use of precise language is important for learning to program.

The lesson sequence used in this study was structured following constructionist principles and the data reported here suggest that while making a game in *Game Maker* using a 'learning by doing' approach can introduce pupils to basic programming concepts with some success, certain concepts, such as conditionals, loops, and variables need more direct instruction if they are to be understood and applied effectively by all pupils.

The need for direct instruction is significant. The theory of constructionism suggests that 'learning by doing' and exploratory learning are valid ways of working. However the findings in this study suggest that such approaches may not be appropriate for learning some programming concepts and this idea is supported in several studies which also observe that some programming concepts need to be formally introduced if they are to be used effectively (see Kelleher and Pausch, 2007; Maloney et al., 2008; Kuruvada et al., 2010b; Schelhowe, 2010; Denner et al., 2012). While some studies support the idea that 'bricolage' is a valid way to learn programming concepts *for some learners* (McDougall and Boyle, 2004; Stiller, 2009), others suggest that exploratory learning does not lead all pupils to an understanding of the structure and operation of a programming language or lead them to develop skills such as problem decomposition, planning or systematic testing and debugging; it can also lead to inefficient or frustrating programming experiences (Kurland et al., 1987). Furthermore, the constructionist approach used in this study appeared not to maximize the learning of core programming concepts *for all pupils*. This finding gives support to research which makes a similar claim for other programming environments (see Ben-Ari, 2001; Beynon and Roe, 2004; Beynon and Harfield, 2010; Meerbaum-Salant et al., 2011); it appears that constructionist approaches may not be well suited to the early stages of learning to program (Guzdial, 2009).

In terms of whether making computer games is a suitable context for introducing basic programming concepts, this research found great variation in the extent to which pupils used programming constructs when making their games, with some pupils using very few - and this finding is echoed in other studies (e.g. Bruckman et al., 2000; Maloney et al., 2008; Denner et al., 2012). Other studies also conclude that the games produced only illustrated an understanding of the *targeted* computer science concepts (Chamillard, 2006; Carbonaro et al., 2010).

Research surrounding the use of other visual programming languages to teach basic programming concepts makes similar claims (e.g. Lavonen et al., 2003; Meerbaum-Salant et al., 2011; Denner et al., 2012). In these studies, concepts were only learned when students were explicitly taught the concepts while they created projects that used the concepts (Meerbaum-Salant et al., 2011: 168). Other studies found that while some concepts may be learned without instruction, others need a formal introduction if they are to be used effectively (Maloney et al., 2008; Schelhowe, 2010), since, in creating a computer game, pupils learn basic programming concepts without necessarily being aware that they are using those concepts (Kuruvada et al., 2010a; Good, 2011). In particular, it seems that computer game authoring does not deliver the more complex concepts well without additional teacher input (Denner et al., 2012).

6. Limitations

Despite the contributions made by the research, it also has its limitations:

- The research was conducted with one pilot group (n=23) and one main study group (n=22) in a high-achieving school in an affluent area of South East England. Its findings may not be replicable in different settings.

- Although the group was mixed ability, 10/22 pupils achieved 'above average' values in their average CAT scores; 7 pupils achieved a CAT score of 120 or higher in one or more CAT measures, which suggests that the group was of above average ability. Its findings may not be replicable in different populations.

- The study represents one implementation of a teaching sequence for computer game authoring, using *Game Maker*. It is acknowledged that the particular set of lessons, the game authoring software, and resources made available to the pupils in this study will have delimited their learning of programming concepts. Its findings may not be replicable using other software.

- The small scale of the study limits the reliability and the validity of the findings in so far as additional findings may emerge in larger populations. Its findings are best evaluated as *one amongst other* case studies of game authoring projects, which investigate different tools and settings (see for example, Kafai, 1996; Lavonen et al., 2003; Willett, 2007; Robertson and Howells, 2008; Zorn, 2008; Games, 2010; Hernandez et al., 2010; Baytak and Land, 2011b; Kafai and Peppler, 2012; Macklin and Sharp, 2012; Minnigerode and Reynolds, 2013).

While these are limitations they do not negate the insights into the pedagogy of computer game authoring gained by conducting this research. The local, small-scale, particular features of the present study hold value, since "phenomena are … present in the smallest particulars of practices and institutions" (Maclure, 2006: 230) and can make a useful contribution to the field, or prompt further research of a larger scale.

7. Conclusion

The findings reported here suggest that the level of programming knowledge pupils acquired in creating their computer games using *Game Maker* is, in Pea and Kurland's terms, Level ii - code generator (Pea and Kurland, 1984). At this level, pupils can write simple programs following examples, read and understand someone else's program and detect and correct some errors. There is less evidence of program planning or understanding of how to make programs more efficient.

Whilst their research implies that this level of programming knowledge is not sufficient, the observations recorded in this paper suggest that making a computer game with *Game Maker* introduced pupils to some basic programming concepts and developed their ability to think computationally, engaged them in an iterative design process and gave them some exposure to program planning and testing within an extended project. Goals for programming education need to be realistic and achievable for all abilities in the limited time available at Key Stage 3 and bearing in mind the fact that many teachers need further training to feel confident in delivering this aspect of the Computing programme of study (Nesta, 2014; CAS, 2015). This paper has outlined some areas of difficulty for pupils and offers several recommendations for how to increase the likelihood that key programming concepts and practices are successfully encountered and understood when creating a computer game using *Game Maker*.

References

Baytak, a. & Land, S. 2010. 'A case study of educational game design by kids and for kids.' *Procedia - Social and Behavioral Sciences*, 2 (2), 5242-5246.

Baytak, A., Land, S. & Smith, B. 2011. 'Children as educational computer game designers: an exploratory study.' *The Turkish Online Journal of Educational Technology,* 10 (4), 84-92.

Baytak, A., Land, S., Smith, B. & Park, S. 2008. 'An exploratory study of kids as educational game designers.' In: Simonson, M. (ed.) *Proceedings of the 31st Annual Convention of the Association for Educational Communications and Technology.* Orlando, USA, 6-9 November. Bloomington, USA: AECT Publications. 39-47.

Beavis, C. & O'mara, J. 2010. 'Computer games - pushing at the boundaries of literacy.' *Australian Journal of Language and Literacy*, 33 (1), 65-76.

Beavis, C., O'mara, J. & Mcneice, L. (eds.) 2012. *Digital games: literacy in action*. Kent Town, Australia: Wakefield Press.

Ben-Ari, M. 2001. 'Constructivism in computer science education.' *Journal of Computers in Mathematics and Science Teaching,* 20 (1), 45-73.

Beynon, M. & Harfield, A. 2010. 'Constructionism through construal by computer.' *Paper presented at Constructionism 2010*. Paris, France, 16-21 August.

Beynon, M. & Roe, C. 2004. 'Computer support for constructionism in context.' In: LOOI, C., SUTINEN, E., Sampson, D., Aedo, I., Uden, L. & Kahkonen, E. (eds.) *Proceedings of the International Conference on Advanced Learning Technologies*. Joensuu, Finland, 30 August-1 September. Los Alamitos, USA: IEEE. 216-220.

Bruckman, A., Edwards, E., Elliott, J., Jensen, C. 2000. 'Uneven achievement in a constructionist learning environment.' In: Fishman, B. & O'connor-Divelbiss, S. (eds.) *Proceedings of the 4th International Conference of the Learning Sciences*. Ann Arbor, USA, 14-17 June. Mahwah, USA: Lawrence Erlbaum Associates. 157-163.

Carbonaro, M., Szafron, D., Cutumisu, M. & Schaeffer, J. 2010. 'Computer-game construction: a gender-neutral attractor to Computing Science.' *Computers & Education*, 55 (3), 1098-1111.

CAS, 2012. *Computer Science: a curriculum for schools*. CAS.

CAS, 2015. Closing the gap to achieve a world class computing teaching workforce. BCS. [Online]. Available: http://academy.bcs.org/sites/academy.bcs.org/files/Computing%20At%20School%20-%20Closing%20the%20Gap%202015.pdf [Accessed 14/06/16].

Chamillard, A. 2006. 'Introductory game creation: no programming required.' In: Baldwin, D., Tymann, P., Haller, S. & Russell, I. (eds.) *Proceedings of the SIGSCE Technical Symposium on Computer Science Education*. Houston, USA, 1-5 March. New York, USA: ACM. 515-519.

Dalal, N., Dalal, P., Kak, S., Antonenko, P. & Stansberry, S. 2009. 'Rapid digital game creation for broadening participation in computing and fostering crucial thinking skills.' *International Journal of Social and Humanistic Computing,* 1 (2), 123-137.

Dalal, N., Kak, S. & Sohoni, S. 2012. 'Rapid digital game creation for learning object-oriented concepts.' In: Cohen, E. & Boyd, E. (eds.) *Proceedings of Informing Science & IT Education Conference*. Montreal, Canada, 22-27 June. Santa Rosa, USA: Informing Science Institute. 237-247.

Daly, T. 2009. 'Using introductory programming tools to teach programming concepts: a literature review.' *The Journal for Computing Teachers*. Autumn issue. 1-6. ISTE.

Denner, J., Werner, L. & Ortiz, E. 2012. 'Computer games created by middle school girls: can they be used to measure understanding of computer science concepts?' *Computers & Education*, 58 (1), 240–249.

DFE 2013. *The National Curriculum in England: Computing - programmes of study - Key Stages 3 and 4*. DfE.

Doran, K., Boyce, A., Finkelstein, S. & Barnes, T. 2012. 'Outreach for improved student performance: a game design and development curriculum.' *Proceedings of the 17th Annual Conference on Innovation and Technology in Computer Science Education*. Haifa, Israel, 3-5 July. New York, USA: ACM. 209-214.

Du Boulay, B. 1986. 'Some difficulties of learning to program.' *Journal of Educational Computing Research*, 2 (1), 57-73.

Edexcel 2012. *GCSE Computer Science specification*. Edexcel.

Eow, Y., Wan Ali, W., Mahmud, R. & Baki, R. 2010. 'Computer games development and the appreciative learning approach in enhancing students' creative perception.' *Computers & Education*, 54 (1), 146-161.

Fluck, A. & Meijers, M. 2006. 'Game making for students and teachers from isolated areas' [Online]. Available: http://www.une.edu.au/simerr/pages/projects/79gamemaking.php. [Accessed 05/08/13].

Giles, J., Beard, S. & Street, S. 2008. *ICT 4 life*. Harlow: Pearson Education Ltd.

Good, J. 2011. 'Learners at the wheel: novice programming environments come of age.' *International Journal of People-Oriented Programming*, 1 (1), 1-24.

Grover, S. 2011. 'Robotics and engineering for middle and high school students to develop computational thinking.' *Paper presented at the Annual Meeting of the American Educational Research Association,* New Orleans, USA, 7-11 April.

Guimaraes, M. & Murray, M. 2008. 'An exploratory overview of teaching computer game development.' *Journal of Computing Sciences in Colleges*, 24 (1), 144-149.

Guzdial, M. 2009. 'Question everything: how we teach intro CS is wrong.' *Computing Education Blog* [Online]. Available: http://computinged.wordpress.com/2009/10/02/question-everything-how-we-teach-intro-cs-is-wrong/ [Accessed 07/0716].

Harel, I. & Papert, S. (eds.) 1991. *Constructionism: research reports and essays 1985-1990*. Norwood, USA: Ablex.

Hayes, E. & Games, I. 2008. 'Making computer games and design thinking.' *Games and Culture*, 3 (3-4), 309-332.

Hernandez, C., Silva, L., Segura, R., Schimiguel, J., Ledon, M., Bezerra, L. & Silveira, I. 2010. 'Teaching programming principles through a game engine.' *CLEI Electronic Journal*, 13 (2), 1-8.

Howland, K., Good, J., & Du Boulay, B. 2013. 'Narrative Threads: a tool to support young people in creating their own narrative-based computer games.' In: Pan, Z., Cheok, A., Muller, W., Iurgel, I., Petta, P., Urban, B. (eds.) *Transactions on edutainment X: lecture notes on computer science* Vol. 7775. Heidelberg, Germany: Springer. 122-145.

Jenson, J. & Droumeva, M. 2015. 'Making games with Game Maker: A computational thinking curriculum case study.' In: Munkvold, R. & Kolås, L. (eds.) *Proceedings of the 9th European Conference on Games Based Learning.* Steinkjer*,* Norway, 8-9 October. Reading, UK: Academic Conferences and Publishing Limited. 260-268.

Johnson, C. 2014. 'I liked it but it made you think too much': a case study of computer game authoring in the Key Stage 3 ICT curriculum. PhD thesis, University of East Anglia, UK. [Onine]. Available: https://ueaeprints.uea.ac.uk/53381/1/2014JohnsonCPhD.pdf [Accessed 12/07/16].

Jones, D. & Wilson, D. 2008. *Pixel8 Game Maker tutorials*. teach-ict.com Ltd.

Kafai, Y. & Resnick, M. 1996. 'Introduction.' In: Kafai, Y. & Resnick, M. (eds.) *Constructionism in practice: designing, thinking and learning in a digital world.* Mahwah, USA: Lawrence Erlbaum Associates.

Kelleher, C. & Pausch, R. 2007. 'Using storytelling to motivate programming.' *Communications of the ACM*, 50 (7), 58-64.

Koh, K., Basawapatna, A., Bennett, V. & Repenning, A. 2010. 'Towards the automatic recognition of computational thinking for adaptive visual language learning.' In: Hundhausen, C., Pietriga, E., Díaz, P. & Rosson, M. (eds.) *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. Madrid, Spain, 21-25 September. Los Alamitos, USA: IEEE. 59-66.

Kurland, M., Clement, C., Mawby, R. & Pea, R. 1987. 'Mapping the cognitive demands of learning to program.' In: Perkins, D., Lochead, J. & Bishop, J. (eds.) *Thinking: progress in research and teaching.* Hillsdale, USA: Lawrence Erlbaum. 333-358.

Kuruvada, P., Asamoah, A., Dalal, N. & Kak, S. 2010a. 'Learning computational thinking from rapid digital game creation.' *Proceedings of the 2nd Annual Conference on Theoretical and Applied Computer Science*. Stillwater, USA, 5 November. Stillwater: Oklahoma State University. 31-36.

Kuruvada, P., Asamoah, A., Dalal, N. & Kak, S. 2010b. 'The use of rapid game creation to learn computational thinking.' [Online]. Available: http://arxiv.org/ftp/arxiv/papers/1011/1011.4093.pdf [Accessed 14/06/16].

Lavonen, J., Meisalo, V., Lattu, M. & Sutinen, E. 2003. 'Concretising the programming task: a case study in a secondary school.' *Computers & Education*, 40 (2), 115-135.

Madill, L. & Sanford, K. 2009. 'Video-game creation as a learning experience for teachers and students.' In: Ferdig, R. (ed.) *Handbook of research on effective electronic gaming in education.* Hershey, USA: Information Science Reference. 1257-1272.

Maloney, J., Peppler, K., Kafai, Y., Resnick, M. & Rusk, N. 2008. 'Programming by choice: urban youth learning programming with Scratch.' In: Dougherty, J., Rodger, S., Fitzgerald, S. & Guzdial, M. (eds.) *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. Portland, USA,12-15 March. New York, USA: ACM. 367-371.

Mcdougall, A. & Boyle, M. 2004. 'Student strategies for learning computer programming: implications for pedagogy in informatics.' *Education and Information Technologies*, 9 (2), 109-116.

Meerbaum-Salant, O., Armoni, M. & Ben-Ari, M. 2011. 'Habits Of Programming In Scratch.' In: Rößling, G., Naps, T. & Spannagel, C. (eds.) *Proceedings of the Conference on Innovation and Technology in Computer Science Education*. Darmstadt, Germany, 27-29 June. New York, USA: ACM. 168-172.

NAACE 2012. *Draft Naace curriculum framework: Information and Communication Technology (ICT) Key Stage 3*. NAACE.

NESTA 2014. 'How can teachers prepare for the new computing curriculum?' [Online]. Available: http://www.nesta.org.uk/blog/how-can-teachers-prepare-new-computing-curriculum [Accessed 29/07/14].

OCR 2011. *GCSE in Computing specification.* 2nd edition. OCR.

O'mara, J. & Richards, J. 2012. 'A blank slate: using Game Maker to create computer games.' In: BEAVIS, C., O'mara, J. & Mcneice, L. (eds.) *Digital games: literacy in action*. Kent Town, Australia: Wakefield Press. 57-64.

Overmars, M. 2003. *Game Maker Community forum* [Online]. Available: http://gmc.yoyogames.com/ [Accessed 14/07/16].

Papert, S. 1980. *Mindstorms - children, computers, and powerful ideas*. New York, USA: Basic Books.

Papert, S. 1999. 'Eight big ideas behind the Constructionist Learning Lab.' [Online]. Available: http://stager.org/articles/8bigideas.pdf [Accessed 16/01/15].

Pea, R. 1986. 'Language-independent conceptual "bugs" in novice programming.' *Journal of Educational Computing Research,* 2 (1) 25-36.

Pea, R. & Kurland, M. 1984. 'On the cognitive effects of learning computer programming.' *New Ideas in Psychology*, 2 (2), 137-168.

Pg Online, 2013. *Introduction to Python*. PG Online Ltd.

Pg Online, 2014. *Programming with Game Maker*. PG Online Ltd.

Redfield, C. & Uhlig, P. 2012. 'Game development class in 6 weeks.' In: RESTA, P. (ed.) *Proceedings of Society for Information Technology & Teacher Education International Conference.* Austin, Texas, USA, 5-9 March. Chesapeake, VA: Association for the Advancement of Computing in Education. 2612-2617

Reeves, B. 2008. *ICT Interact for KS3: pupil's book 3*. London: Hodder Education.

Roffey, C. 2013a. *Python Basics*. Cambridge: Cambridge University Press.

Roffey, C. 2013b. *Python: Next Steps*. Cambridge: Cambridge University Press.

Saeli, M., Perrenet, J., Jochems, W. & Zwaneveld, B. 2012. 'Programming: teachers and pedagogical content knowledge in the Netherlands.' *Informatics in Education*, 11 (1), 81-114.

Sanford, K. & Madill, L. 2007. 'Recognising new literacies: teachers and students negotiating the creation of video games in school.' *Proceedings of the Digital Games Research Association Conference*. Tokyo, Japan, 24-28 September. DiGRA. 583-589.

Schelhowe, H. 2010. 'Using construction kits: just learning how to program a computer - or is there more educational benefit?' *Paper presented at the Digital Media and Learning Conference,* La Jolla, USA, 18-20 February.

Seaborn, K., Seif El-Nasr, M., Milam, D. & Yung, D. 2012. 'Programming, PWNed: Using digital game development to enhance learners' competency and self-efficacy in a high school Computing Science course.' In: Smith King, L., Musicant, D., Camp, T. & Tymann, P. (eds.) *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, Raleigh, North Carolina, USA February 29 - March 3. New York, USA: ACM. 93-98.

Seehorn, D., Carey, S., Fuschetto, B., Lee, I., Moix, D., O'grady-Cunniff, D., Boucher Owens, B., Stephenson, C. & Verno, A. 2011. *CSTA K-12 Computer Science standards*. ACM.

Spohrer, J. & Soloway, E. 1989. 'Novice mistakes: are the folk wisdoms correct?' In: Soloway, E. & Spohrer, J. (eds.) *Studying the novice programmer*. Hillsdale, USA: Lawrence Erlbaum Associates. 401-416.

Stiller, E. 2009. 'Teaching programming using bricolage.' *Journal of Computing Sciences in Colleges*, 24 (6), 35-42.

Waller, D. 2009. *Basic projects: Game Maker*. Oxford: Payne-Gallway.

Wilson, A., Hainey, T. & Connolly, T. 2012. 'Evaluation of computer games developed by primary school children to gauge understanding of programming concepts.' In FELICIA, P. (ed.) *Proceedings of the 6th European Conference on Games Based Learning*. Cork, Ireland, 4-5 October. Reading: Academic Publishing International Ltd. 549-558.

Yoyo Games 2007. *Game Maker 7.* [Computer program]. YoYo Games Ltd.