Teaching Programming by Gamification

Tomasz Zajac

A study into effectiveness of video games assisting in computer programming education, based on the use of a proof-of-concept game demo.

School of Computer Science

University of Lincoln

2020

Introduction

While Computer Science nowadays spans a wide range of topics and disciplines, programming is still an important component of it, especially since it enables the use of computing devices in solving almost any type of problem.

It is therefore vital that there are sufficient amounts of programming talent present in the industry to ensure the continued development and innovation of code that has to power the current and future computer software products and more. More importantly, an industry like this requires that those new generations of developers maintain a certain level of skill and knowledge. Failing that has an effect on society, not just on an individual's employability. Developers who lack good programming skills will end up releasing poor quality software. The direct consequence of that might vary in severity. Consumer-grade products, like apps and video games, are often released with numerous glitches of varying significance and exposure, but those can only have minor negative impact in the real world – typically it is a feeling of frustration for the user. However, it can also be an error inside a much more impactful system, e.g. one that controls machinery (e.g. aircraft, missiles) or deals with sensitive or personal data.

This is not to say that software glitches are only introduced by inexperienced developers, but it is very likely that this lack of experience would sooner or later become a hindrance. It is therefore crucial that formal education provided on this subject is as effective as possible. While the content delivered on Computer Science courses is usually appropriate, the format in which it is presented might fail to maintain student engagement (Maxim et al., 2019), which in turn might lead to amotivation and limit them from performing as well as they could otherwise.

Many schools already employ active learning as an approach. It is common to see courses being formed from both lectures as well as practical workshops and using assignments as an assessment method instead of purely relying on exams, where possible. While this is definitely an improvement, this project aims to go a step further and propose a learning method that feels more friendly or even natural to the student.

Almost 43% of Computer Science students in the UK were aged 20 and under on enrolment in academic year 2018/19 (HESA, 2020). At the same time, 85% of British people aged under 35s would frequently play games (MCV, 2019). If the goal is to make the learning experience of programming courses more accessible, then one way to do that would be by delivering within a context that can keep the audience engaged. Given those two sets of demographic information, it is easy to see that gaming should be considered as a format that can achieve just that.

This project therefore spans developing a proof-of-concept solution that aims to replicate programming activities and their effects inside a virtual game world, and assesses its effectiveness and impact on the target audience. The game's design takes into consideration existing systems with similar goals, such as educational software, learning apps and commercial video games. During the design and development stages of the artefact's lifecycle, emphasis is put on ensuring that it maintains the right balance of being both enjoyable as a game as well as providing an opportunity to improve the user's skills and knowledge that can be applied in the real world. However, it needs to be noted that this project does not seek to replace the existing curriculum in its entirety; it merely aims to aid in improving its effectiveness. This does not mean that the project is any less important because of that. In fact, this should be seen as a positive, an advantage, since it means the software artefact is not relied upon as a full source of learning material. This allows it to be developed primarily as a game with educational aspects, not a formal education product, which in turn could mean that students will find it easier to engage with as they know they can expect some joy out of it.

Background & Literature Review

As social trends change, so do the people. Recently, it's been common to see students who lack motivation to put effort into their learning (Ford and Roby, 2013, 101) (Montes-Leon et al., 2019). Answering the question of why that is the case is beyond the scope of this project, but as it pertains to its rationale, some reasons need to be considered nonetheless. The three main reasons include (Ford and Roby, 2013, 103-104):

- Lack of confidence
    - Many students lack self-confidence and doubt their ability to "maintain the effort required to complete their academic studies successfully" (Ford and Roby, 2013, 104)
    - Students are not necessarily required to have prior programming knowledge or experience in order to enrol on Computer Science courses. Confidence issues can therefore be largely detrimental to their ability to familiarise themselves with the subject and perform well, because now they face peer pressure of other students who may have enrolled with significantly more experience, thus resulting in potential stress.
- Extrinsic behaviour
    - While Computer Science students are usually already interested in learning programming prior to starting the course, and only improve that interest throughout the course (Zainal et al., 2011, 281), their motivation to do so is largely extrinsic. This means that their motivation comes from valuing the tasks by rewards they bring, rather than seeing the task itself as an enjoyable activity (ie. intrinsic motivation) (Ford and Roby, 2013, 102).
    - One potential issue with this, essentially for students who have not tried programming prior to enrolment, is that over time they may start to see it as a chore rather than a valuable skill, which might affect their performance.
- Activity characteristics
    - This links with the previous point regarding lack of natural enjoyment in programming courses, but from a different perspective. Students may experience amotivation from tasks that are "uninteresting, dull, boring, routine, tedious, arduous or irrelevant" (Ford and Roby, 2013, 104). Large portions of programming education nowadays are still primarily delivered via lectures, which typically exhibit "low or non-existent" amounts of student engagement (Maxim et al., 2019). Students will therefore often dismiss the subject as unimportant until they are faced with an exam or assignment, at which point it is likely they will only study it to pass and never get to experience it in a way that lets them develop their skills, again leading to worse performance at this task during their career.

Those issues indicate that the current generation of Computer Science students may not be benefiting from their programming courses fully. With the majority of them being young adults (HESA, 2020) familiar with gaming as their relaxation method, hobby or interest (MCV, 2019), Gamification could be seen as one approach to boost their motivation (or, perhaps, prevent demotivation beforehand). Indeed, it has been identified as an alternative learning approach applicable to programming courses in existing publications (Skalka and Drlík, 2018) (Maxim et al., 2019). Gamification was even applied and evaluated within a real context of programming education (Montes-Leon et al., 2019). However, the existing attempts – while successful – primarily focused on the use of "Serious Games" – ie. "games developed for purposes other than entertainment, such as

training, health, advertising, political, business and even military matters" (Montes-Leon, 2019). This project aims to explore this area further, by introducing a software artefact that is essentially the inverse of a Serious Game, ie. one that prioritises enjoyment as a means of maintaining player engagement. The theory behind this approach is that, given the evidence of amotivation within students, perhaps for some users the sole thought of interacting with an educational product may be off-putting, which means they might never engage with it. This prevents them from regaining motivation which is likely to be more of a hindrance than their current programming skills, given the latter is an acquired talent.

As such, the approach used in design of the artefact is to attract the player's interest in it as a video game first, then introduce educational elements to it. More specifically, the key objective is to create a setting for the game in a way that disguises it to look akin to a commercial title which they would normally approach naturally, in a usual manner taken when playing a video game to relax or as a hobby. That way, the student would associate the product with positive thoughts normally associated with their leisure time, rather than as something they must do in order to improve their knowledge, which could potentially make the experience feel stressful and overall unenjoyable.

In order to do that, several commercial videogame titles with programming or problem solving elements have been examined in the process of designing the software artefact. Some inspiration was taken from Valve Software's *Portal*, which is a 2007 3D first-person-perspective game that mixes elements of puzzle solving with action elements. The game is set in a fictional scientific research laboratory called Aperture Science, in which the player is tasked with solving puzzles that would let them traverse each level safely. Most of this requires the use of a *Portal Gun*, which allows the player to deploy a portal from one place to another. For example, early on in the game the player needs to move to a section of the level separated by a pool of deadly acid (Figure 1). The distance is too long to just jump, and at this stage the *Portal Gun* can only deploy one end of the portal (there are two ends: Blue and Orange). The level already has one end deployed (marked 2), to which the player simply needs to teleport by placing their end of the portal next to the start area (marked 1). Once they reached portal 2, they need to now deploy the portal on the wall where the sliding platform stops (marked 3), and travel there by going back in the orange portal. Assuming the player has stepped through the portal when the sliding platform was there, it would then take them to the end of the level (marked 4). This is by no means a puzzle relevant or even directly translatable to programming, but it shows ways of introducing constraints in a manner that can force the player to break the problem into individual parts and solve it. This links to the decomposition aspect of computational thinking (BBC, 2020).

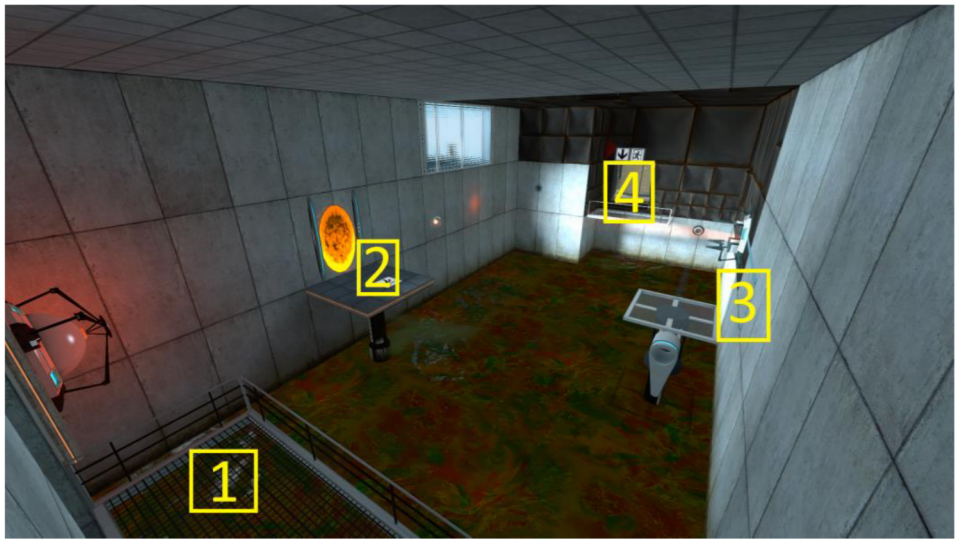

Figure 1: "Test Chamber 08" level from *Portal* (Valve, 2011). Labels were added by author for this report.

Another existing game to be considered was 2011's *Minecraft* by Mojang. It is a "sandbox" game that simulates a procedural world in which the player is free to do virtually anything. Its default "survival" mode sets some challenges for the player, such as finding food to stay alive, building shelter to hide from monsters, crafting and upgrading tools and weapons, etc. This main part of the game is irrelevant to the scope of this project, but the game does incorporate one feature that would become inspiration for part of this software artefact. Redstone is the name of an energy source in the game's world. Dust, blocks and torches of Redstone can be used to build mechanisms in the game utilising triggerable objects. Redstone Dust effectively carries power generated by a block, such as a Redstone Block, Torch, an activated lever, or by an action, such as the press of a button. If a triggerable object is connected to a path of Redstone Dust, it will trigger once it receives power from this path. The construction of such mechanisms is relatively simple; since the game's world is divided into cubic blocks, a piece of Redstone Dust needs to be placed adjacent to the object we want to trigger with it, then also connect the trigger to that Redstone path. The most basic use of this would be to create a door that opens by pressing a button placed some distance away (Figure 2). However, the game allows for more complexity when using Redstone. Basic flow control/decision making can be implemented by using a component called a Redstone Comparator, which takes 3 input signals – two on the sides, one on the rear – then checks if the rear signal is stronger than the side inputs. If it is, then it outputs a positive signal via the output torch at its front. As such, this can be used to implement some basic computer concepts, e.g. logic gates (Figure 3) and utilise them to build more complex mechanisms.

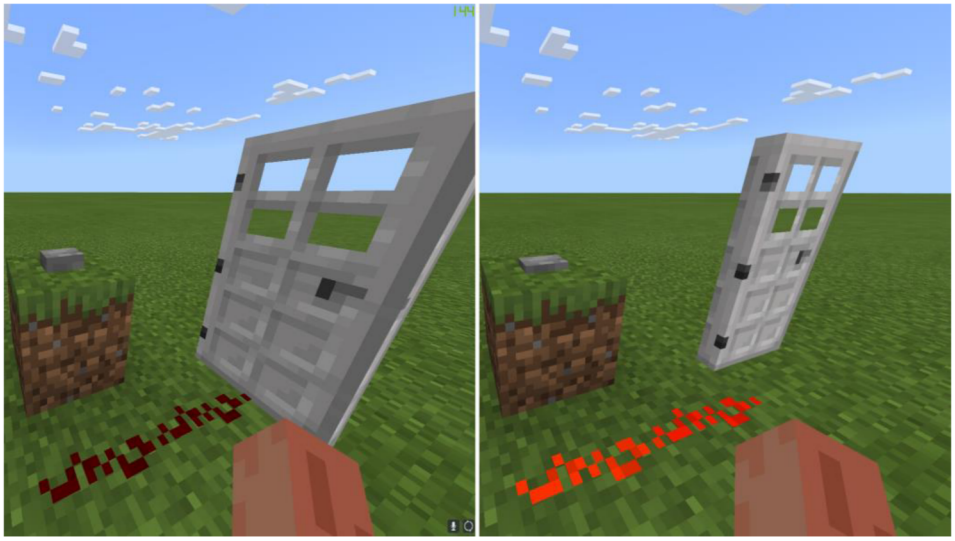

Figure 2: A door opens after pressing a button connected with Redstone in *Minecraft.*

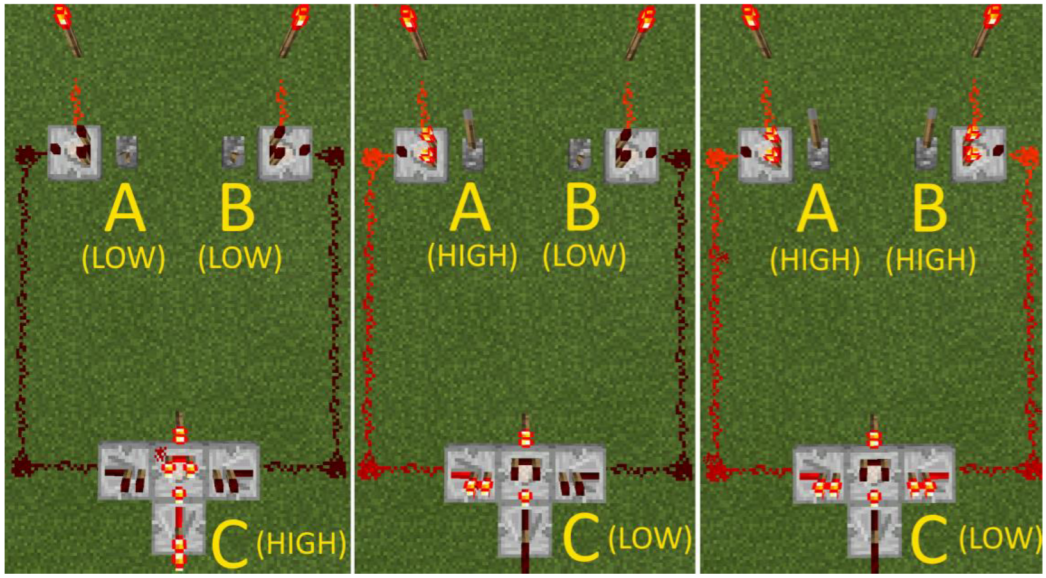

Figure 3: A logical NOR gate built using Redstone in *Minecraft*. Two inputs (A and B) are simulated using levers, and output C is positive if both inputs are negative. It is a negation of the OR operator (Sangosanya et al., 2005).

It is true that logic gates are not all too applicable to most programming work currently, but *Minecraft* proves that it is possible to simulate computer-related concepts within a casual video game. Combined with an abundance of other features present in the game – which is likely what's responsible for its massive popularity – this ability to build mechanisms in-game evolved into engineering communities and even entire courses (CAML Academy, 2019).

Zachtronics's *Shenzhen I/O* was also reviewed. It is a programming/puzzle game focused around building computer circuits and writing low-level code in an assembly language to build logic for devices manufactured by a fictional electronics company. The game has a significant layer of storyline, but for purposes of this project it will go ignored as it's not relevant to any of the project's objectives. The game features a code editor in which the player can add microcontrollers to the device's mainboard, then write code for those individual microcontrollers. Data input and output can be achieved by connecting the correct input/output nodes to the correct registers on the

microcontroller (Figure 4). The programming language used was designed specifically for the game and is part of an entire architecture available to the user, which is also why the game features a full-fledged developer manual document. Given full knowledge of the architecture and the language, it is in theory possible that a player would be able to build successful programs for each level with only one attempt. However, the game features some essential debugging tools, which means a trial-and-error method is a perfectly feasible way of progressing through the game's levels. Those tools include a signal viewer for each output (Figure 5), the ability to set breakpoints (Figure 6) and to step through code (Figure 7) in order to see what effects a line of code can cause on the signal.

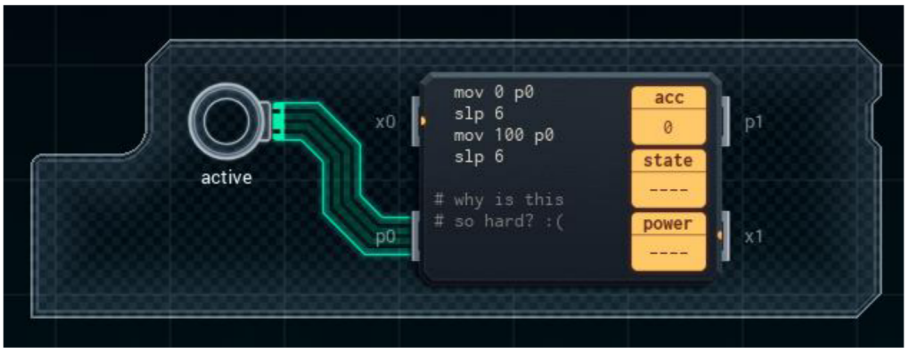

Figure 4: Part of the "Fake Surveillance Camera" level from *Shenzhen I/O*. **active** is an output node that receives an alternating signal every 6 ticks from the microcontroller.

Figure 5: The signal viewer. The bright orange line marks the actual signal, while the dark orange line is the expected signal. Red lines mark where the mismatch occurs. The small orange cursor (top-right of this image) marks the current point in the program's execution.

Figure 6: The breakpoint debugging feature. When the program execution reaches this point, it will stop to allow the player to examine the expected and actual values as well as what line of code is responsible for the state at this point.

Figure 7: Program controls to allow running the code either step-by-step or as a simulation. Breakpoints will pause the execution of code regardless.

Shenzhen I/O is likely to be an overwhelming title for a beginner programmer, mainly because the player is forced to write code in a language that has long been removed from mainstream use. Modern software code is usually wrapped in "high-level" languages (e.g. C, Java, Python), meaning that they are easier to understand for humans and don't expose the low-level computer architecture-specific instructions. There are some disciplines in which low-level coding knowledge is useful (if not required), but this is definitely not the case for mainstream jobs like application development, and assumes previous programming knowledge which is easier to obtain by learning a high-level language first. However, despite this, *Shenzhen I/O* serves as a good example of creating a believable setting in which the player is made responsible for building electronic devices for a company, which is likely the main source of motivation. It also fulfils the need for immediate feedback and assessment by providing the player with immediate and automated feedback and assessment, which has been identified as success criteria for this kind of system in several publications (Skalka and Drlík, 2018) (Oblinger, 2003, 11) (Oblinger, 2004, 8) (Khaleel et al., 2017, 4) (Crocco et al., 2016, 405). *Shenzhen I/O* does this by displaying a histogram comparing the player's solution's efficiency as compared to other players' implementations (Figure 8).

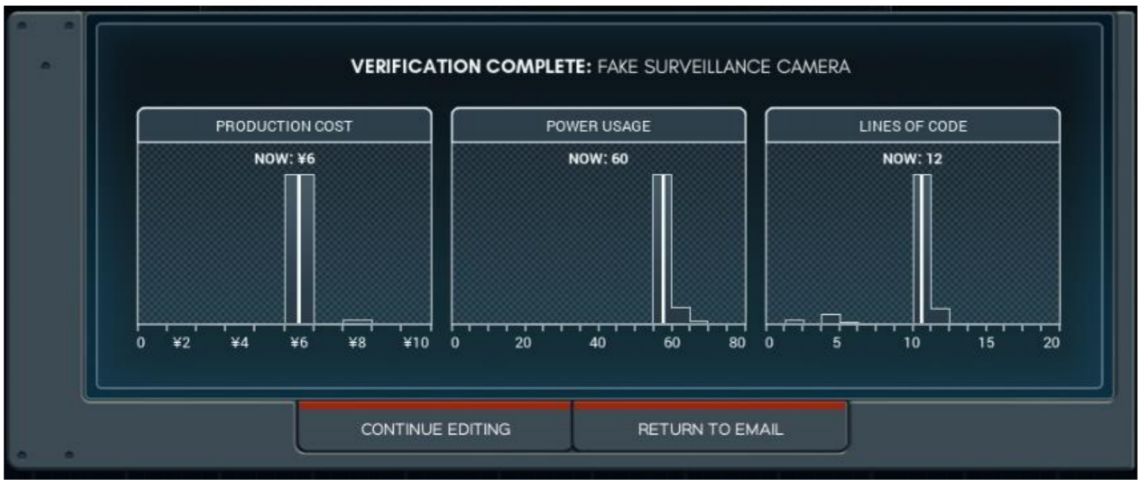

Figure 8: Histograms displayed by *Shenzhen I/O* for three different factors of the implemented solution for the current level. This acts as means of motivation for the player to continue improving their understanding of the architecture and encourage optimisations.

The way it incorporates debugging tools into the in-game code editor, allowing for experimentation, is also a component worth considering in design of this project's software artefact.

Google's *Blockly* visual code editor library was examined when reviewing potential designs for the editor interface in the game. *Blockly* (Figure 9) is very much similar to *MIT Scratch* (Figure 10), but has the added bonus of being able to generate equivalent code in JavaScript, Python, PHP and more languages. That way, the user is able to build programs with an interface that is easier to understand (for beginners or non-programmers) and observe how this changes the source code that they can then simply copy and compile or run. The main block-based visual editor designs in both *Blockly* and *Scratch* might not be the approach taken for the final software artefact in this project, but the code generation in *Blockly* is certainly a beneficial feature, especially for those with interest in learning to eventually write code (as opposed to use visual editors), which opens up more opportunities given it's the primary method for programming in today's world.

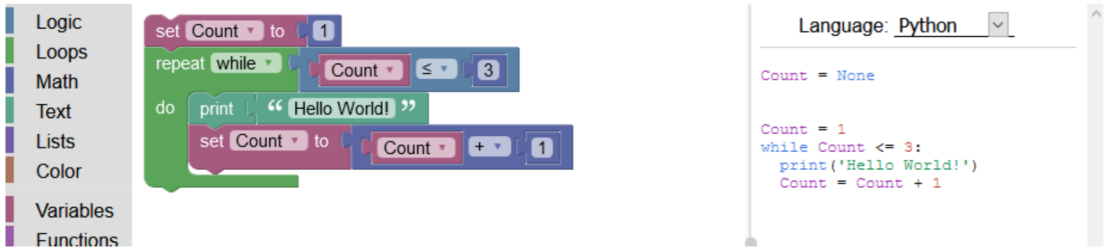

Figure 9: Google's *Blockly* demo. Notice the source code on the right being generated from the code blocks.

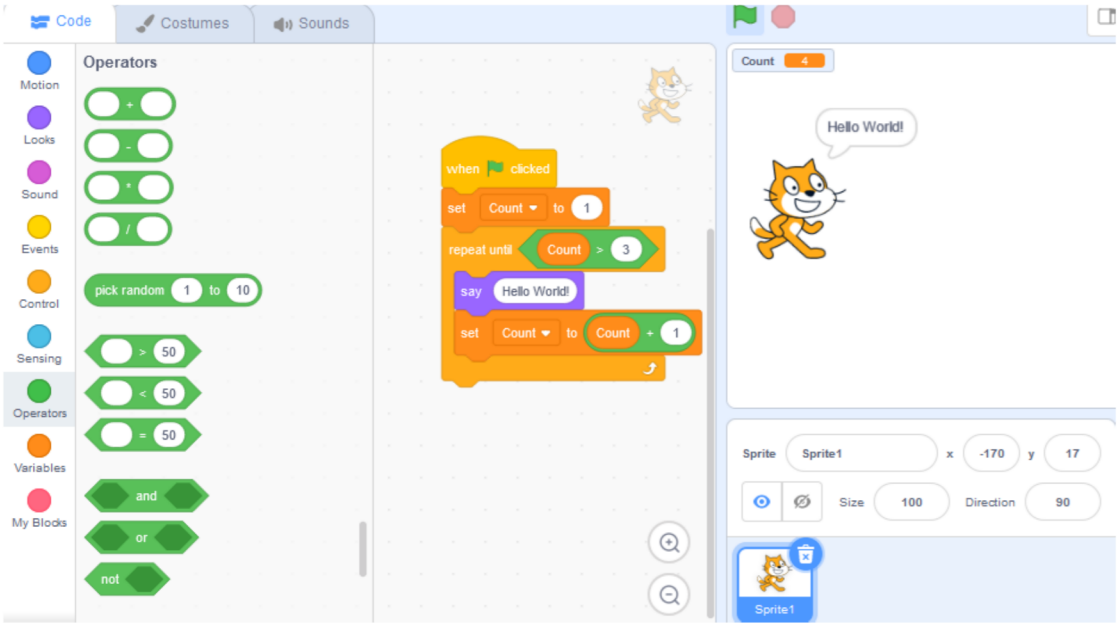

Figure 10: *MIT Scratch* running the same program as above. In *Scratch*, programs are written for each Sprite (graphical object) in the scene.

When exploring technology to build the software artefact with, a popular game suite (featuring both game runtime libraries and editor tools) called Unreal Engine 4 was considered as an option. While the project ultimately used Unity3D instead (another game creation suite), one component of Unreal Engine was found to be of potential inspiration for the design of the game's visual code editor. UE4 features a "Blueprint" scripting system as an alternative to writing scripts as C++ code. The main difference is that while C++ is an entire programming language of its own, which requires some prior programming knowledge to even make use of within a game development context, Blueprint is a visual scripting language that can be used by non-programmers as it's effectively a flow diagram (Figure 11) with access to almost all features available from C++ code.

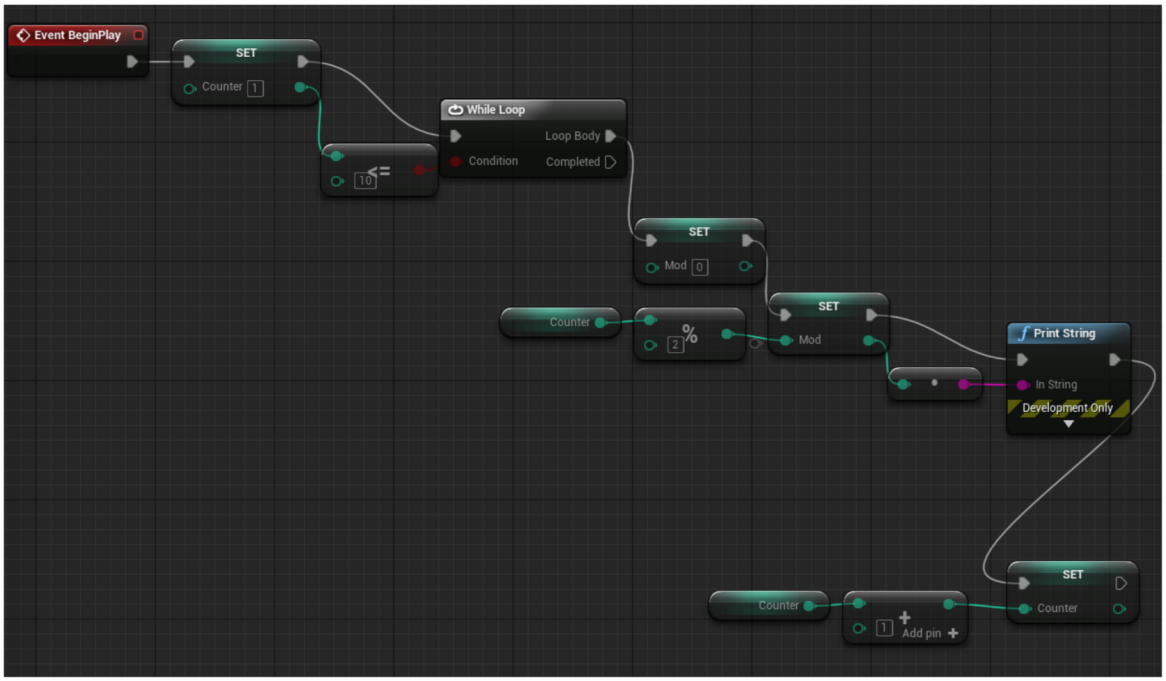

Figure 11: An Unreal Engine 4 Blueprint that prints remainder of division by 2 for every number from 1 to 10.

The Blueprint visual editor gives an idea of how a visual code editor should be designed and implemented in this software artefact, especially considering it is a tool of choice for many developers working with Unreal Engine 4 (Bendell, 2018), which makes it a good example of ensuring practicality as a coding tool and accessibility for a non-programmer audience. The way logical blocks, such as the While Loop pictured in the example (Figure 11), are implemented is worth revisiting when designing this software artefact, as visualising code flow control might be a difficult task outside of block-based editors like *Scratch* and *Blockly*, yet the Blueprint editor achieves that in a quite intuitive way with each command being connected with a "Next" node, except for logical blocks which also have a "Body" node pointing to the first command inside that block (Figure 12).

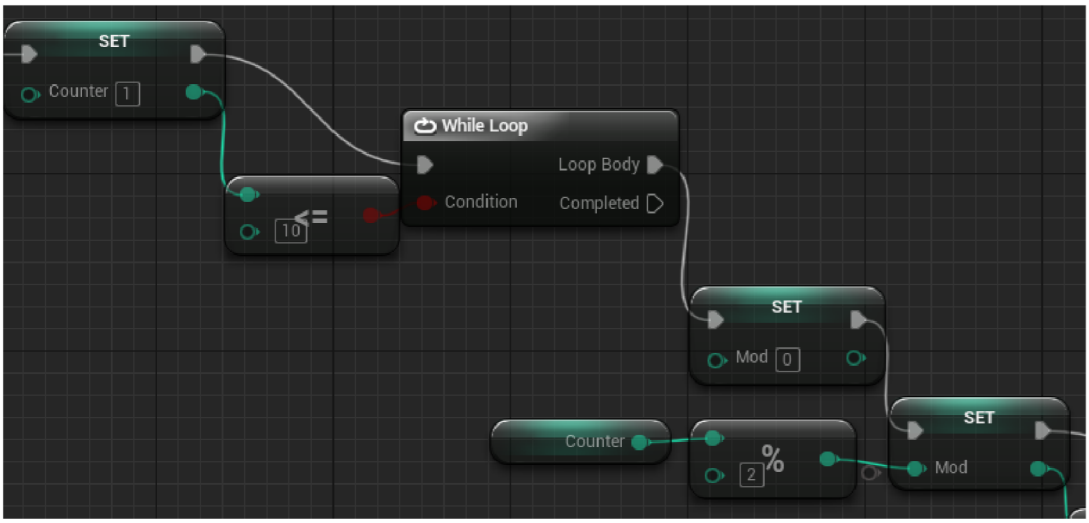

Figure 12: A While Loop command in the Blueprint editor. The "Loop Body" node points to the command to execute at the start of each iteration of the loop. The "Completed" node is the "Next" node, simply pointing to the next command to execute once the loop is done. In this case, it doesn't point anywhere – the program ends after the loop finishes.

The first objective in the project plan was to gather educational requirements; one source used for this stage was the Programming Fundamentals module taught to the first year Computer Science students at University of Lincoln during the academic year 2019/20. The first 6 weeks of the module, which span variables, data types, input/output, program flow, logical statements, loops and functions were identified as topics that are both essential to understanding programming as well as feasible for implementation in the game. Levels would be designed with these in mind to allow students to practice applying them in order to solve puzzles.

Sebastian Lague, an independent game developer, made a video for his "Coding Adventure" series in which he creates a game where coding is used as a gameplay mechanic to control robots, override turrets to target enemies, etc. (Sebastian Lague, 2019) This was a significant piece of inspiration for the software artefact implemented in this project, however it was mainly for the ideology of the game, as opposed to the exact level design etc. In specific terms, the idea of mixing manual skills like movement and timing (typically associated with video games) with cognitive effort (in form of writing code to manipulate objects in the level to the player's advantage) appears to be a design choice that hasn't been tried enough in existing solutions for programming gamification, and thus the game developed in this project would attempt exploring that.

Some research also went into what format the game should be made in; ie. what kind of scene and rendering techniques it should be based on (2D/3D, first-person/third-person/isometric/side/top-down/fixed view). It has been found that the best fit in this case would be a 3D first-person game, meaning that the world is rendered in 3 dimensions and the player sees it from the controlled character's point of view; they can move in any direction, turn around, look up and down etc. This decision was taken in order to make the in-game world feel believable and thus easier for the player to engage with; it was found that players enjoy those environments because they look, behave and feel "real" (Johnson, 2008).

When researching ideas for in-game puzzles, FizzBuzz was found to be a good exercise requiring a combination of some key programming concepts. It is based on the children's game of the same name, and revolves around the idea of saying numbers in a sequence, but with special cases for multiples of numbers 3 and 5, where the latter is replaced with the word "Fizz" and the former with "Buzz", respectively, instead of the number. In case of being a multiple of both 3 and 5, the number is replaced with "FizzBuzz" (Ghory, 2007) (Tom Scott, 2017). This is a game that maps to programming concepts with relative simplicity; in its basic form, it needs loops, flow control, arithmetic, logical statements and an output mechanism. As such, this is likely to be featured as a puzzle in the software artefact.

Methodology

Project Management

As the project was largely led by a single student, there was not a significant amount of formal project management being done, as designing and developing the software artefact was more important and had constantly changing objectives due to user requirements evolving with each prototype test. Reasonable efforts were taken to follow the schedule as per the project plan outlined in the Work-in-Progress document from December 2019 (Figure 13). This plan included completion of five main objectives:

- Gathering educational requirements
    - During this stage, requirements of the system with regards to educational value were identified by consulting existing courses, books and other training resources

available to students or the wider public, as well as the authors of that material (tutors, lecturers etc.). The knowledge gained at this stage would be used in objectives 3 and 4 when creating in-game puzzles for the software artefact.

- Gathering usability requirements
    - User experience needs of the system were gathered from evaluating popular existing systems related to the area of programming and its learning process. This would aid in establishing a clear direction in terms of the artefact's design and development, especially the aspect of presentation (user interfaces, controls, etc.)
- Designing the artefact
    - This stage focused on reviewing the findings from requirements gathering, and using that to create sketches, mockups and playable prototypes of individual components before they are assembled together into a full game. Although this objective was started first, it effectively ran parallel to objective 4 due to the iterative nature of the task, in order to ensure that each component is implemented properly when integrated with the rest of the game's systems.
- Implementing (developing the artefact)
    - With individual parts of the game already working to some extent, work began on connecting them in order to create a demo level. As mentioned before, because integrating separate components of a game often brings out previously unnoticed issues with their individual implementations (either in the sense of usability or technical issues), this objective was completed alongside objective 3 once most issues related to the design have been addressed and the demo level was in a playable state.
- Evaluation
    - Having completed development on the original version of the software, people have been invited to partake in a system validation study. The tests commenced, but only one person was able to take part in it in accordance to the original test plan before the UK Government issued stay-at-home orders due to the Coronavirus outbreak.
    - Originally, any questions with regards to controls and overall help with playing the game would be addressed in person by the researcher; however, due to the lockdown, testing needed to be moved to use online methods. On-screen hints had to be added to the game before the study could resume. This took additional time and was not something that the original plan predicted. It also potentially lowered the number of respondents in the study.

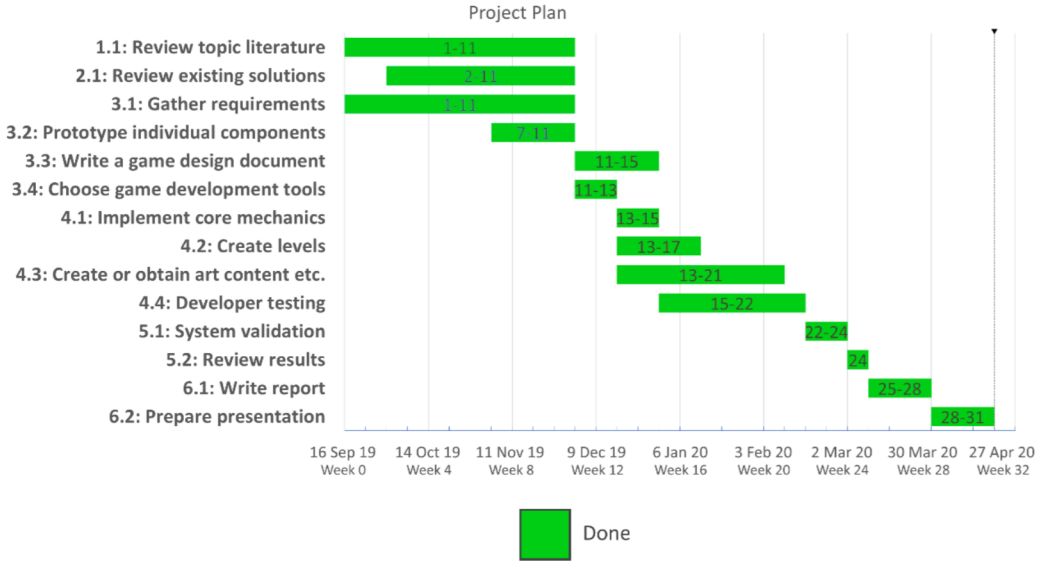

Figure 13: The Gantt chart showing the project tasks and their intended timescales over 31 academic weeks that make up the project duration. The current week (week 31) is marked with the dotted cursor line.

Admittedly, while all objectives have been completed, there were slight delays in reaching them. This was mainly the case with objective 4, which is where the development phase of the game started. The design for the game was gradually crafted by prototyping single elements of the game and evaluating their relevancy, usability and quality of implementation. If an element was deemed a positive addition to the game's design, then it would be marked as part of the final product. Otherwise, it would be reworked a number of times in order to try and find possible modifications that would improve it. This was essentially an elimination process of game mechanics, world objects and level design. While this iterative approach proved to be time-consuming (which was the cause of the delays), it was ultimately needed in order to ensure that all elements are useful to meeting the game's objectives and that their implementation is of sufficient quality, which means it likely saved some time in development of the final version of the product.

As mentioned before, the testing stage of the project's lifecycle (namely objective 5) suffered delays too due to the Coronavirus outbreak, as the UK Government responded with introducing nationwide lockdown mid-March. Because of this, additional development on the software artefact had to be done. With all delays combined, this testing stage effectively began 7 weeks after its original date and had to be performed remotely with use of online tools.

Throughout most of the project's duration, regular meetings were hold with the project supervisor. This was mainly to seek opinion and feedback on the constantly evolving design aspects of the artefact, as well as advice with regards to the research methodology to apply in objective 5.

Software development

The project's software artefact was primarily developed with an Agile approach. A key outcome of this was that changes to the design could be made often, and some elements would be completely replaced.

For example, initially the only way for the user to perform arithmetic operations within the game's visual code editor was to use an "arithmetic chain" (Figure 14). In other words, this meant breaking down a mathematical expression into single-operator arithmetic expressions, placed as commands in the editor, connected in one sequence, then the result of that would be transferred to the first

"set variable" or "function call" command that would occur after the arithmetic. This proved to be a very cumbersome design, so it was entirely scrapped and replaced with proper support for arithmetic expressions as text typed by the user. Those expressions would then be parsed and evaluated into a numerical result, which eventually was a more intuitive implementation of this feature.

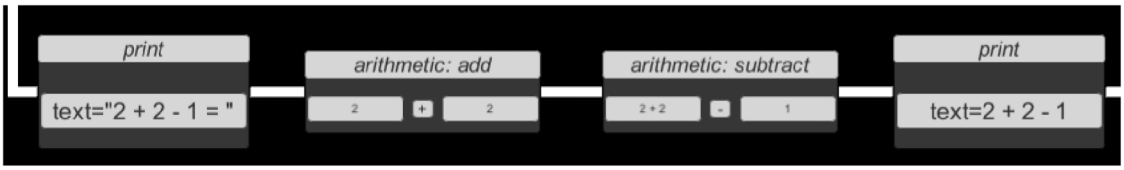

Figure 14: The initial implementation of arithmetic operations in the editor, following the "chain" concept. One arithmetic command is built of three parameters: the left-hand value, the operator, and the right-hand value. To build expressions with more than two values, they need to be connected from left to right. The next arithmetic command would then feature the expression so far as its left-hand value (as shown in the subtract node). While this approach was technically less error-prone, it was very unintuitive to users; as such, the final version of the game removed the arithmetic commands completely, and allows for expressions to be typed as right-hand values for variables or as parameters directly.

While this was a contributing factor to some of the delays in the development phase, Agile was an approach that would allow such changes (Beck et al., 2001), and they were needed in order for the project's objectives to be met. It also encouraged regular iterations and feedback on the software, the latter of which (in form of supervisor meetings and showcasing early prototypes to members of the target audience) helped significantly in identifying areas and components that need improvement or change.

Toolsets and Machine Environments

For developing the software, it was necessary to choose a game engine to develop it with. A game engine is a set of code libraries and tools that allow for creation of interactive applications with 2D/3D graphics, which are mostly computer games. Developing a custom game engine for this project was deemed unfeasible due to the amount of work required for the underlying technologies rather than game features that the user would interact with. As such, two popular solutions have been examined: Unreal Engine 4 and Unity3D. After spending some time with UE4, it was found that it would take too much time learning to develop with it, which would have ultimately impacted the progress on creating the game. Considering the scope of this project, there were no deciding factors between the two solutions; both engines had all capabilities required to implement the design. This, combined with the author's past experience with Unity3D, meant that it would be the tool of choice for building the artefact. UE4 is an adequate choice for many commercial games of larger complexity; in this case, the learning process was taking an unreasonable amount of time, and as such was not a viable choice. Additionally, it was later found during development that Unity3D's scripting framework actually allowed for easier, modular design of in-game systems code. In addition, while the initial release of this software would be designed for desktop computers, Unity can support a variety of different devices including mobile systems and virtual reality headsets, which means the project could reach wider audiences or offer richer experiences with the use of VR technologies.

A version control system called Git was used to maintain the codebase. Git allows separating changes made to program source code into "commits", which can effectively be used as restore points when a change introduces errors or to simply view when specific features were added. In context of this project, this was crucial to integrity of the work as game projects are complex in their nature and as such, having all those versions backed up on a remote host (in this case, GitHub) took away a lot of work solving potential problems when errors are introduced in code or if it is

lost/corrupted. In addition, Git allows for “branches” – separate tracks of code changes – which meant that code for experimental features could be worked on away from the main, working version. The final release of the software amounted to 151 commits made from 4 November, 2019 to 7 April, 2020 (Figure 15).

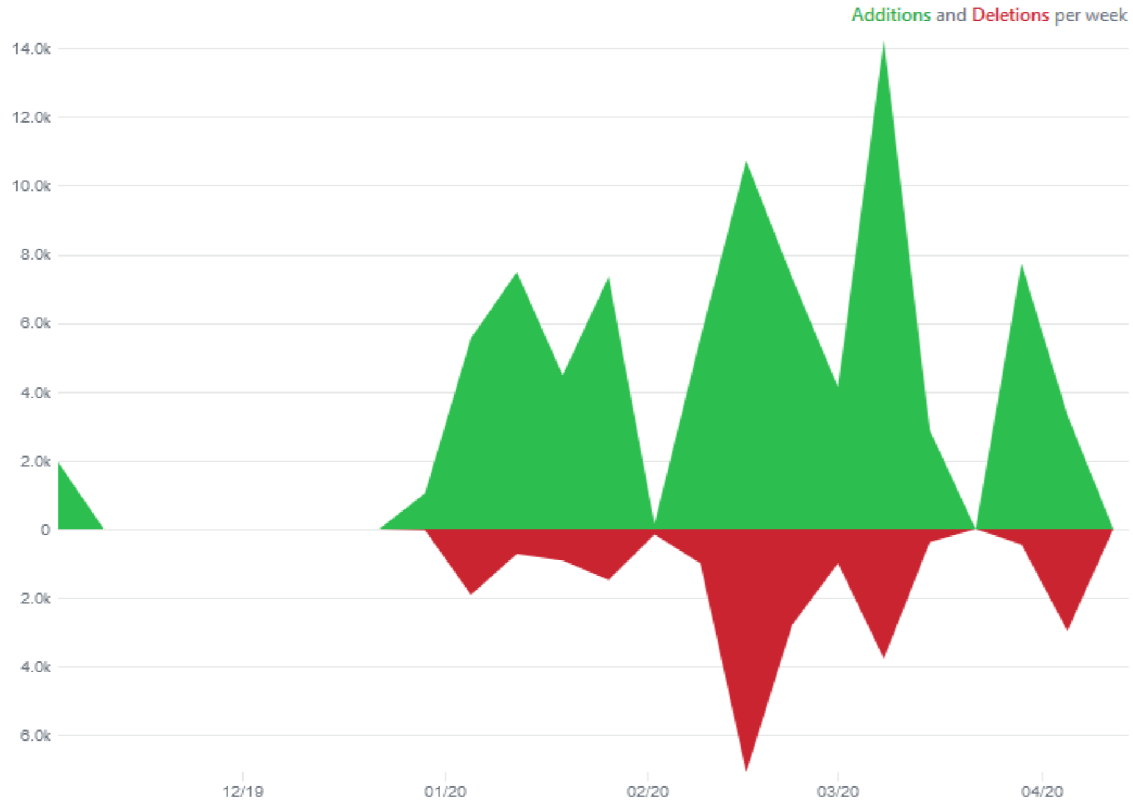

Figure 15: GitHub Code Frequency chart. Changes to the source code are shown as Additions and Changes.

With regards to project management tools being used, there was not a significant amount of it. A Trello board was used (Figure 16), but only to note down some ideas that would appear during development so that they’re not lost. If those ideas referred to full features of the software, and they ended up being developed, then their progress would be tracked via checkboxes. Due to the structure of the software artefact, most game features required three criteria to be met in order to be considered complete: Execution, Serialization, and UI (the reason for this is explained in the development section).

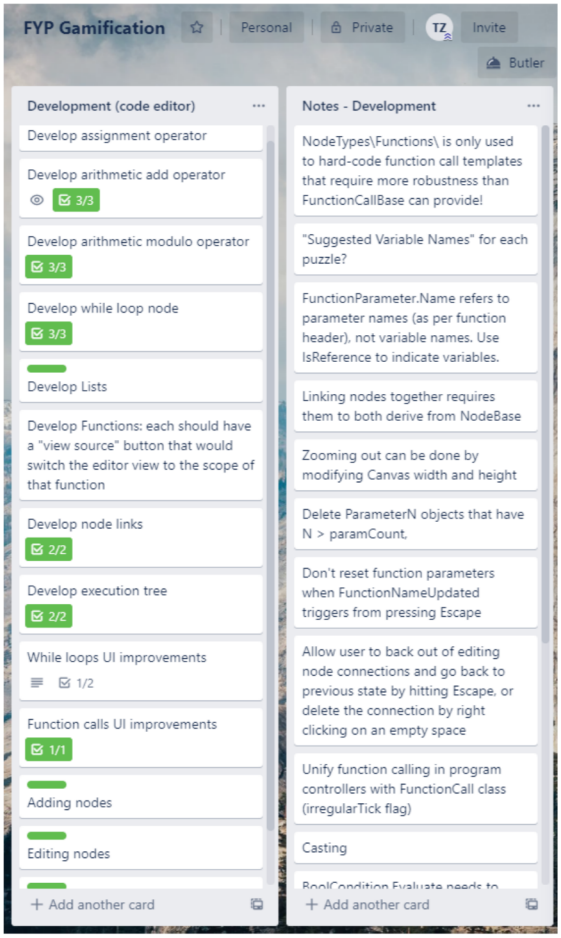

Figure 16: Trello board used throughout the project's duration. The cards were mostly related to the development of the software artefact and were oftentimes chaotic as they would be noted down quickly while working on an entirely different component.

Research methods

The study involved participants playing a demo level of the game in which they had to solve puzzles with the use of programming via a visual code editor interface. Each puzzle would facilitate a path towards the end of the level; it was a linear flow in order to identify any potential problems with the difficulty level. After spending some time game (either completing the level or abandoning the session early, if the player was unable to finish the level) they would complete an online survey (Appendix 1). The questions asked were:

- the participant's prior programming knowledge
  - nominal data based on the participant's current application of programming (none/hobby/student/professional)
- gaming habits
  - interval data of hours spent playing games weekly
  - nominal data of game preferences (genres, types etc.)
- learning style
  - nominal data determined from mode of answers given to a multiple choice question
- the participant's experience with the level, opinion on the puzzles as well as the game
  - nominal data on whether they were able to finish the level

    - ordinal data regarding each game component's effectiveness using a 5-point Likert scale
    - optional comments/suggestions

Design, Development & Evaluation

Software development

Requirements gathering

Most sources for requirements gathering have been identified in the Background & Literature Review section. With regards to the educational value offered by the game, the specific topics it aims to reinforce are:

- Variables
    - A crucial part of programming that allows introducing various pieces of data that can then be used or modified.
- Data types
    - Essential knowledge about different variable types for different kinds of data (e.g. numerical, textual, logical).
    - Not all programming languages enforce explicit declarations of data types, but knowledge of them is still required to understand what operations can be performed on which data
- Input/Output
    - Input refers to retrieving data from the user or another source to use it, in order to use it in the program, e.g. by asking the user to type some text in, or reading a file from disk.
    - Output refers to presenting data by the program, e.g. by printing text to the screen, or saving it to a file.
- Program flow
    - Making decisions on what actions the program should perform
    - Essential to building programs that need to adapt to the current state given some rules.
    - Includes if-else statements, switches, etc.
- Logical statements
    - Ties in with program flow and loops
    - Includes logical algebra, comparison operators, etc.
- Loops
    - Repeating a piece of code if a condition is met
    - Allows writing repeating/iterative operations with clean, intuitive code
    - Has several variants (e.g. while, do-while, for, foreach), but they serve as simplification; most, if not all, loops can be written as while-loops.
- Functions
    - Wrapping multiple operations as one addressable function
    - Allows for code reuse and abstracting unnecessary detail

This list of topics is taken from the "Programming Fundamentals" module delivered to first-year Computer Science students at the University of Lincoln. They also line-up with the Novice/Advanced Beginner levels of programming expertise (Winslow, 1996, 19).

To help facilitate these educational goals, the game would generate code from the user's visual program representation, using the Python programming language syntax (in a similar fashion to Google's *Blockly*).

With regards to the game-specific aspects of the software, the requirements were based on the systems (games and editors) identified in the Background & Literature Review section. As such, the following key design principles were identified:

- 3D graphics
    - A three-dimensional world is easier for the player to immerse in, as it gives an impression of being real (Johnson, 2008).
    - Given that the game would involve manipulating objects via programming, seeing them in a more believable environment could motivate them to engage with the game. A realistic-looking world could highlight the impact of the player's actions, which might translate to increased motivation in learning programming outside of the game.
- Common first-person controls
    - Gamers aged 18-25 prefer playing FPS (First-Person-Shooter) games (Limelight, 2019). This age group fits in with students enrolling on university courses (HESA, 2020).
    - As such, it makes sense to set the game in a viewing perspective and navigation/control scheme they are familiar with.
- Increasing difficulty
    - While the first puzzle(s) would be simple enough for even a non-programmer to make sense of and solve without too much trouble, the difficulty level should be raised as the player progresses through the game.
    - This would keep the player interested in finishing the game; it already relies on repetitive designs being primarily a puzzle game, but changing difficulty could mitigate that feeling.
- Mixing level navigation with puzzle solving
    - As seen with *Portal*, a puzzle game can require both thinking as well as mechanical skills to allow for more creative level designs.
    - Gamers nowadays mainly play games focused on action (MCV, 2019) (Limelight, 2019). As such, this would ensure that audiences with preference of movement-based games (platformers, shooters, etc.) are not bored by just puzzle solving.
- No single solution
    - While the level might be designed in a linear fashion (in terms of the physical layout, checkpoints etc.), each puzzle should accept any version of the program, as long as it allows the user to progress.
    - For example: a program for raising platforms could either be written by placing the right number of function calls and setting the parameters by hand for each of them, or it could utilise loops. Similarly, a program for a FizzBuzz puzzle could either be made the traditional way of comparing the result of each number's modulo 3 and modulo 5, or by storing the outcome of each number in a list and iterating a print function over it.
    - It is important to note that the solution itself should not be checked – the actions performed by the program determine whether it solves the puzzle or not.

    - This worked in *Shenzhen I/O*, where all players are given the same programs to write, but can write them in different ways of varying efficiency. All of them are still able to finish the level successfully.
- No failure
    - This ties in with the previous point: a wrong solution should not mean a level failure. The user just needs to figure out how to correct it to reach the solution, either through analysis or experimentation.

Design

Before beginning work on creating the software artefact, it was necessary to create a design; games are more complex projects, and as such creating fully functioning pieces without knowing if they will remain in the final version of the software would be a significant waste of time and effort.

Primarily, a lot of design work was done through producing sketches and mock-ups (Figures 17, 18, 19). Those are not all too appealing to look at, but they are a quick way of visualising ideas and design thoughts about specific features (or even entire level designs), which enables early observations on their relevance and effectiveness when implemented in the final game.

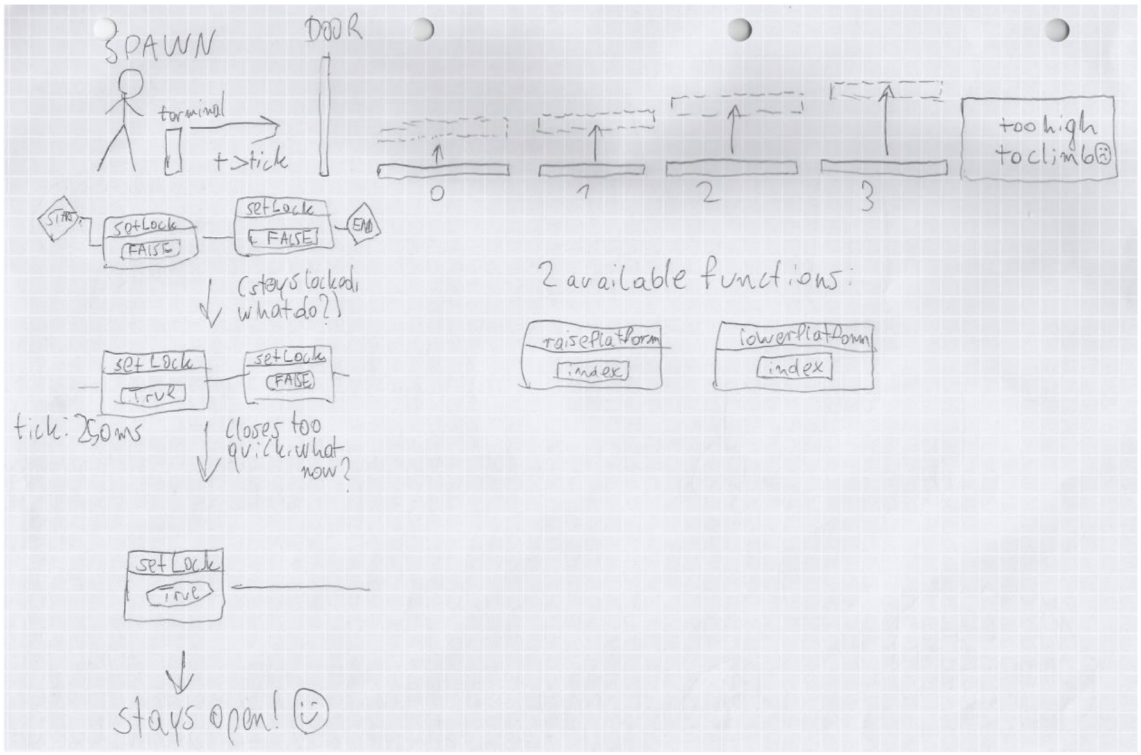

Figure 17: Early level design sketch. All contents of this particular sketch have been implemented in the final version of the game.

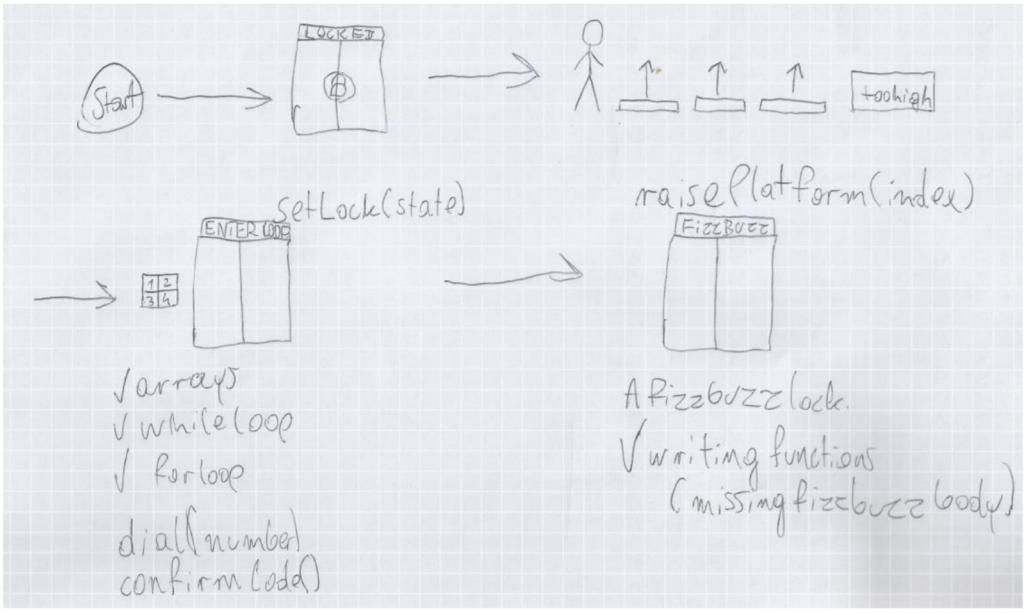

Figure 18: Another version of the level sketch, including two different additional variants of door puzzles: code dial and FizzBuzz. Those variants had their own conditions under which the door unlocked. Only the basic door (operated by the setLock function) and the FizzBuzz variant made it to the final version; the code dial variant was replaced with a robot navigation puzzle as its unlock condition, and the FizzBuzz door was slightly modified since the code editor does not support user-defined functions.

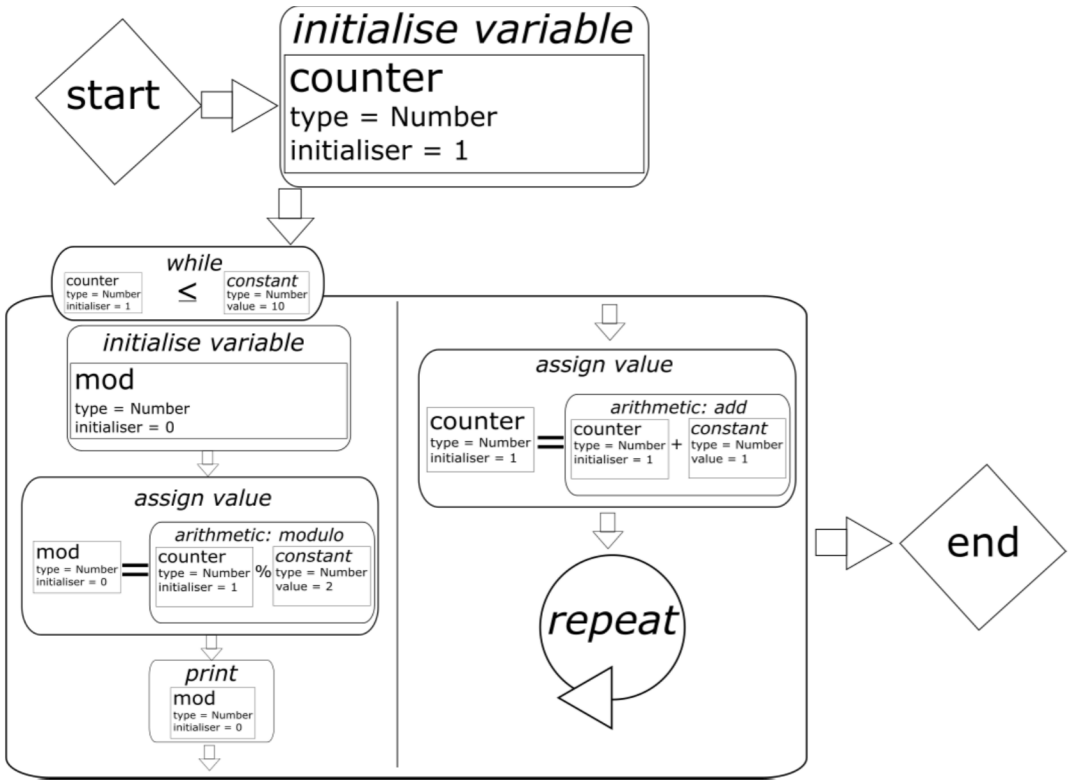

Figure 19: A mock-up of the in-game visual code editor. It is meant to be similar to a flow diagram. The example program is based on Figure 11.

With that said, many elements of the game still had to be prototyped in software form in order to better assess their feasibility in the final product, but they were implemented in lower complexity at first (Figures 20, 21, 22, 23, 24)

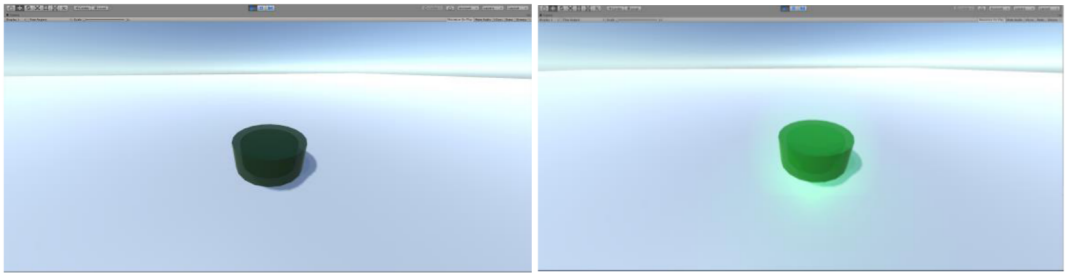

Figure 20: Testing basic first-person controls and interaction with other objects. This scene shows a simple switch that can be toggled when the player is within its radius. This feature was then inherited by the Computer Terminal objects in the final game, which can only be interacted with when the player is near it.

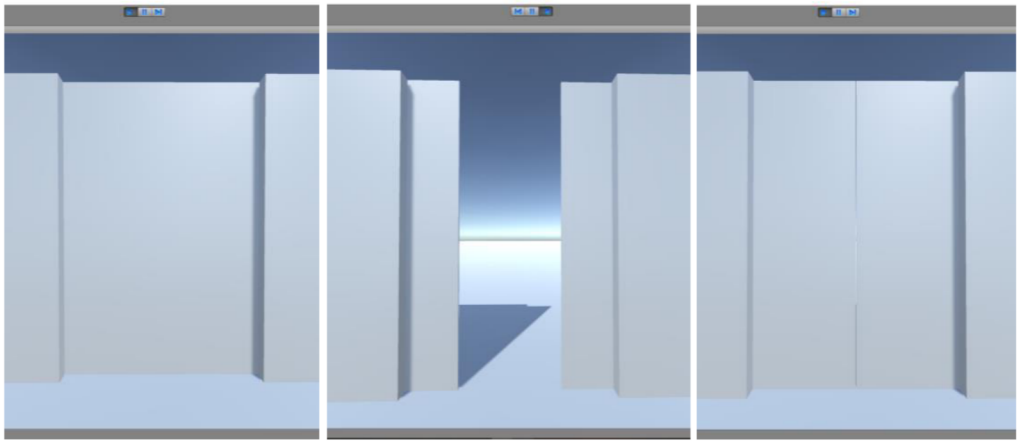

Figure 21: Prototyping door animation via code; this was needed to create puzzles that activate doors. At this stage, the door simply opened when the player pressed a button while within its interaction radius, like with the switch example. That said, the code created for this was easy to adapt later in development and was integrated with the terminals in the level that would control the doors via user programs.

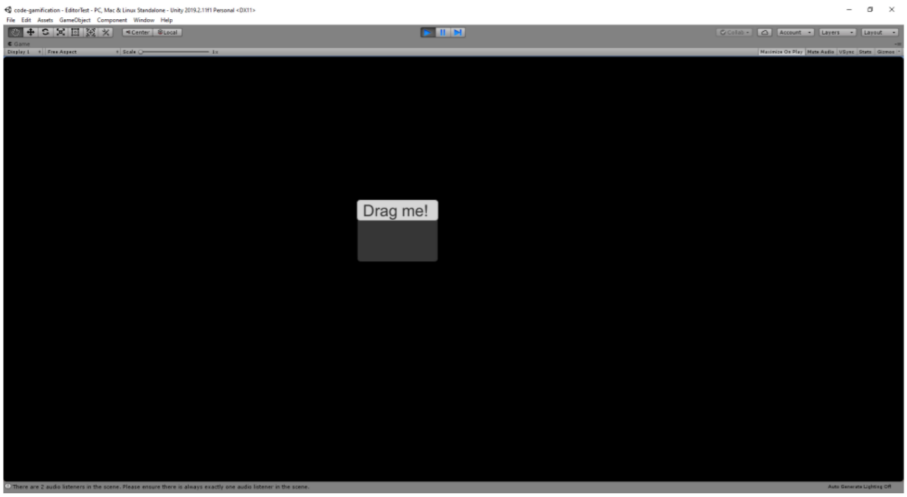

Figure 22: Prototyping mouse controls for the code editor. This also shows the design for command nodes which remained largely the same throughout development.

```
from codegen import model

testSymbols = []
testActions = []
# counter = 1
testSymbols.append(model.LevelSymbol("counter", "Number", model.LevelSymbolInitialiser(1)))
# while counter <= 10
testSymbols.append(model.LevelSymbol("loop1", "WhileLoop", model.LevelWhileLoopSymbol(testSymbols[0], "<=", 10)))
# mod = counter % 2
testSymbols.append(model.LevelSymbol("mod", "Number", model.LevelSymbolArythmetic(testSymbols[0], "%", 2), testSymbols[1].refStr))
# print(mod)
testActions.append(model.LevelAction(model.LevelActionPrint(testSymbols[2]), testSymbols[1].refStr))

lineOrder = model.ProgramLine(testSymbols[0])
lineOrder.next = model.ProgramLine(testSymbols[1])
lineOrder.next.next = model.ProgramLine(testSymbols[2])
lineOrder.next.next.next = model.ProgramLine(testActions[0])

testLevel = model.LevelDiagram(symbols=testSymbols, actions=testActions, lines=lineOrder)
print(testLevel.lines)
```

Figure 23: Early tests of the "Level Diagram" concept used to allow generating Python code from a visual program. Although this example shows a hard-coded intermediate representation of a program, the point of it was to test generating Python code from an in-memory object representation of the program; this can then be automated to take user programs from the visual editor as input and generate this structure on runtime.

```
>>> exec(open("testProgram.py").read())
counter = 1
while counter <= 10:
        mod = counter % 2
        print(mod)
>>>
```

Figure 24: The generated code based on the example program structure set up in Figure 22.

Building/coding

The first part of the game's codebase to be implemented were the controls. It features two classes responsible for this: ControlBase, an abstract class which sets up some shared values for any kind of player controller, and FPPControl, which derives from ControlBase and uses it to implement basic first-person player controls that allow moving forward/back, left/right, and turning around using the mouse. This is considered boilerplate to most Unity3D projects, or indeed to most game projects in general.

Next, the base in-game entities had to be implemented. The game has four main entity types in its codebase:

1. Interactable
   - Objects that the user can approach and activate by pressing a button
2. Unlockable
   - Derived from Interactable, it's an abstract base class for things like doors that can be unlocked (UnlockableDoor). It is up to each child implementation of this class to decide how the locking/unlocking process works.
3. Programmable
   - An entity type given to any object that can be affected by the user's program. Each Programmable references a Computer Terminal (ProgramController) that controls it.

- A Programmable can also be given an index number if the Terminal needs to reference more than one Programmable (e.g. the Raising Platforms puzzle which features 3 platforms that the user program can raise/lower)

4. ProgramController
   - The base implementation of this class provides code for running computer simulation (see: Execution Model) as well as core features and functions of the simulated programming language (print, sleep).
   - There is a Computer Terminal for each puzzle in the game's level. Each Terminal has its own ProgramController. Each kind of puzzle has its own implementation of that class; this is because specific puzzles require different functions in user programs, so each variant of the ProgramController provides its own functions for the player to use in their solutions, and it also controls its associated Programmables.
   - Each ProgramController typically also needs to have an EditorProgram object, which holds the code editor UI for that puzzle's user program (see: Code Editor).

Execution Model

The game needs to simulate a computer that can run any program the user builds using the visual editor. As such, a basic execution model had to be created. It is contained in the ProgramController class.

The main code controlling each terminal's program execution is contained within the ProgramController.Update, ProgramController.ExecuteFrame and ProgramController.ExecuteNode methods. It can be summarised as follows:

1. Arrive at a command node
2. Handle the command node
3. Wait the tick time
4. Move to the next command node (or first node of an if-statement/loop, if the condition is met).
5. Repeat until reaches end of program

For each node then, this is how the handling process operates:

1. Check node type
   - There are predefined command node types:
     - Start/End nodes (indicating beginning and end of the program)
     - AssignValue (setting a variable to a value)
     - FunctionCallBase (allows calling one of the associated ProgramController's functions from within the user program)
     - LogicalBlock and ElseBlock (if-else statements)
     - WhileLoop
     - Continue (skip to next iteration of loop)
     - Break (terminate loop)
     - AllocateArray (creates a list)
2. If the node is recognised, then the ProgramController simulates specific computer behaviour:
   - If the node is not a function call, then handle it as a core feature. For example, if the node type is AllocateArray, that means the user program is attempting to create a list. The ProgramController would handle this example by taking the list name and

size (n), then creating n variables in its symbol table, each under the name of *listName[i]* where $0 \leq i \leq n$.

- If the node is a function call, then it parses the provided parameters and checks if the function call is part of the base function set (print, sleep).
  - If it is a base function, then it is executed in the base ProgramController implementation.
  - If it is not a base function, then the ProgramController sets the handover flag in this node's ExecutionStatus. The handover flag determines whether or not the puzzle-specific child implementation of ProgramController.ExecuteNode needs to be called. Given that only child implementations contain functions that can manipulate in-game objects, handover is an extremely common case. When the child implementation is called, the puzzle-specific functions can be invoked.

This is a significantly simplified simulation model of a computer device, but for the purposes of the game it achieves its goal; the command nodes are meant to refer to core libraries in a programming language, the tick time simulates a CPU clock, and memory is made available via variables and lists stored inside a (rather rudimentary) symbol table.

Serialization

To fulfil the game's educational goals, a design decision was taken to generate Python code equivalent to the user program built using a visual editor. As such, each command node type in the game had to produce its Python code. To achieve this, each node type implements an interface (IProgramNode) which contains one method Serialize (returning a string with a piece of code for this command). As an example, the FunctionCallBase node returns a string in the format of *functionName(parameter1, parameter2, parameter3, … parameterN)* (Figure 25).

```
public override string Serialize()
{
    return $"{GetFunctionName()}({GetParameterListString()})";
}
```

Figure 25: FunctionCallBase implementation of the Serialize method. GetFunctionName provides the function name, GetParameterListString provides a comma-separated list of parameters passed to the function.

If-else statements and loops fall into another subcategory called CodeBlocks. Their serialization process is slightly different in that it's split into two methods: SerializeBlockHeader and SerializeBlockBody. The former contains the header line (ie. the logical condition controlling the if statement or loop) and the latter contains serialized lines inside the block (ie. all commands inside the if statement or loop). The two are then combined in the Serialize method where the block body is split into separate lines and each is prefixed with the correct indentation (Figure 26).

```
public override string Serialize()
{
    string header = SerializeBlockHeader();
    string[] bodyLines = SerializeBlockBody().Split('\n');
    string fullCode = $"{(string.IsNullOrWhiteSpace(header) ? "" : GetLineTabs() + header + "\n")}";

    string bodyLineTabs = GetBodyLineTabs();

    foreach(string line in bodyLines)
    {
        fullCode += $"{bodyLineTabs}{line}\n";
    }

    return fullCode.Substring(0, fullCode.Length-1);
}
```

Figure 26: The CodeBlock.Serialize implementation. GetLineTabs provides a string consisting of the correct number of tabulation characters to indent the header. GetBodyLineTabs provides a string effectively equal to that of GetLineTabs + 1 extra tabulation to indent each line in the block body.

Code Editor

The user interface for the visual code editor in the game is built using Unity's Canvas UI system, with some elements like text input handled by the older IMGUI system (this was done because of issues with adapting the Canvas system to work with a constantly changing UI layout). It allows for placing command nodes that make up different parts of the program, then connecting them in the right order to set the program flow (Figure 27).

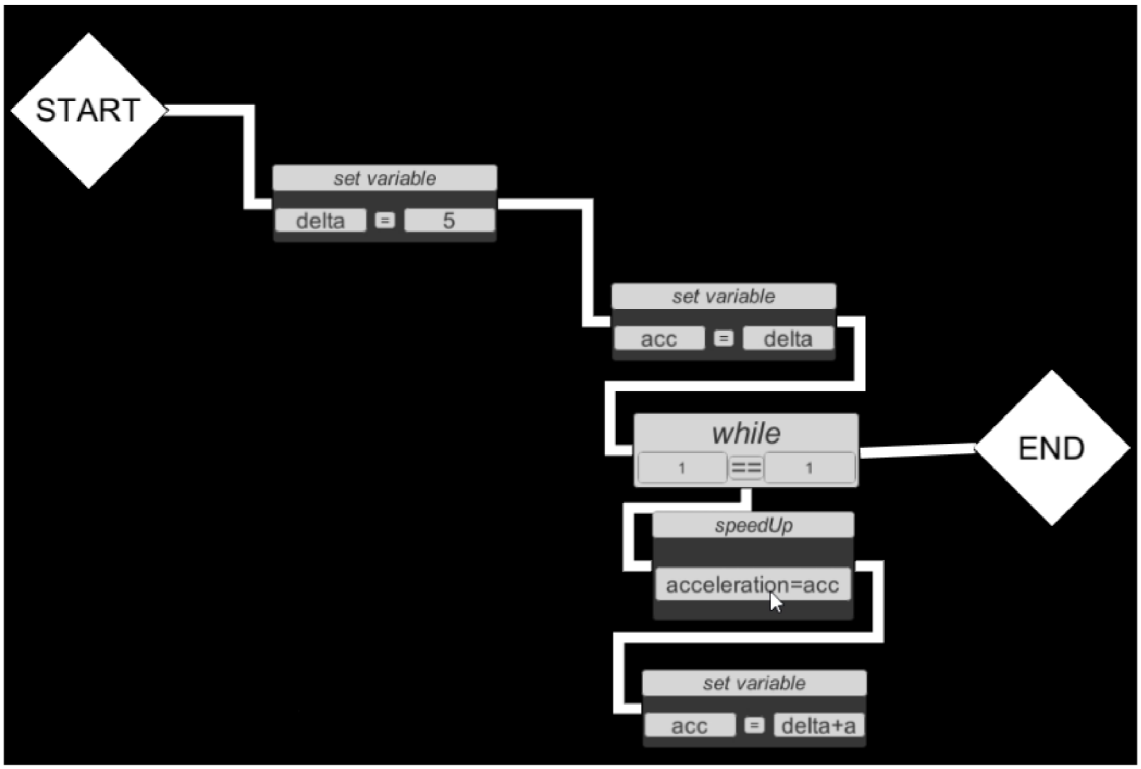

Figure 27: Example program for one of the puzzles in the game, built using the code editor. The program flow is as follows: first, the variable **delta** is set to 5, then **acc** is set to **delta**. Next, the program arrives at a while loop. If the loop condition is met, then it moves to the **speedUp** function call node, which is the first node in the loop body. If it is not met (either at first or because it's now terminated), then the program moves to the END node. The while loop used here is actually an infinite loop; it has a condition that will always be met because it needs to repeat that code forever.

The program flow was meant to resemble that of Unreal Engine 4's Blueprint system (Figure 11). The game emulates the Blueprint editor's "Completed" and "Loop Body" connectors (Figure 12) but in a simpler way by having two unlabelled connection points on a CodeBlock node (Figure 28). Internally it refers to them as "NextNode" and "FirstBodyNode". Connecting nodes is done by clicking the right mouse button on the desired connector area on the first node, then right-clicking the node to execute after.

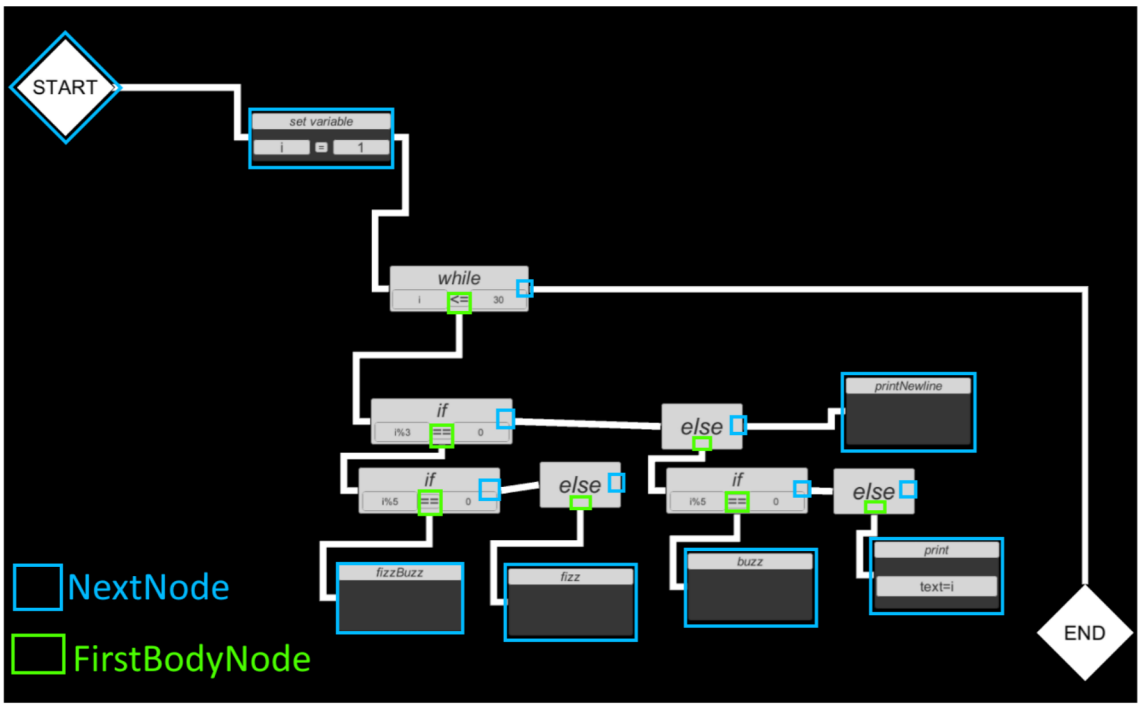

Figure 28: An example of how both connector areas are placed on different types of command nodes in a program. Notice that nodes that don't change the direction of the program flow (ie. anything other than an if-else statement or a loop) can only have the NextNode connector, and thus it reacts to right-clicking anywhere on the entire surface of the node.

When an editable field (such as a condition left-hand/right-hand value or logical operator, or a function call parameter) is double-clicked with the left mouse button, Unity's IMGUI (immediate mode GUI) is activated to enable user text input (Figure 29).

Figure 29: Editing the right-hand value being assigned to the variable **acc** using IMGUI.

The aforementioned FunctionCallBase class has many child implementations in the game's code because it allows the user to edit parameters (as such, things like setting a variable are actually derived from a function call internally). It also has the ability to resize itself in the editor UI according to how many parameters are passed, which was used for the list initialisation node, which allows the user to initialise all elements of the list if they set the list size as a number literal (ie. raw numerical value, not a symbol) as opposed to having to set each element individually with "set variable" command nodes (Figure 30).

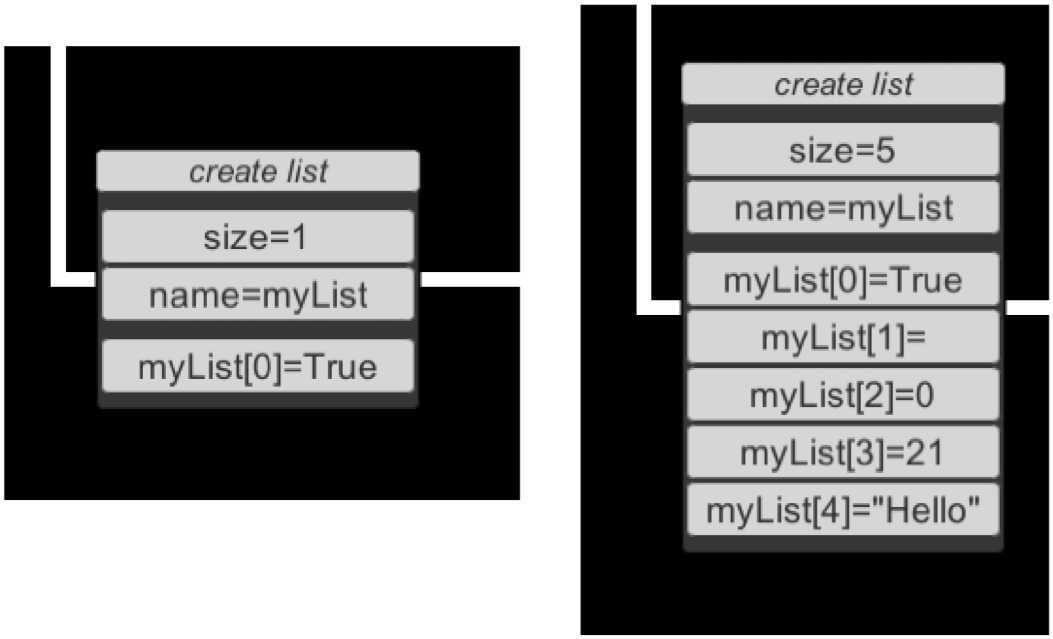

Figure 30: The "create list" command node can take a number literal as the list size, which allows the user to initialise all elements to their desired values in one node. This is enabled by the dynamic sizing code in the FunctionCallBase class.

The lines connecting the nodes are implemented using Unity's LineRenderer component, which is used for rendering lines in 3D space relative to the camera. For this reason, multiple camera objects have to be used in the game, with each editor having its own camera to render the user interface. All editor code is contained in the EditorProgram and EditorDraggableNode classes.

Puzzle types

For purposes of the game's demo version, three main puzzle types have been implemented (including some subtypes). Each has its own child implementation of the ProgramController class:

1. Door
    a. **Basic lock**: can be easily unlocked by setting its lock state to False. Provides a setLock(*state*) function.
    b. **FizzBuzz**: Requires producing the output of a FizzBuzz game (from 1 to some specified number) in order to unlock the door. The output can be done using only the base print(*text*) function, or the provided short-hand functions fizz(), buzz() and fizzbuzz(), each printing the correct word.
    c. **Car (Robot)**: Requires navigating a robot on a grid to a destination point, with obstacles in the way.
2. Platform
    a. Controls a set of platforms that can be raised or lowered using the raisePlatform(*index*) or lowerPlatform(*index*) functions. Each call raises/lowers the specified platform by one step.
3. Fan
    a. Controls the speed of a fan that the player can use to launch themselves to normally unreachable spots in the level. Provides speedUp(*acceleration*) and speedDown(*deceleration*) functions.

Demo level

The game's demo level is meant to showcase the various features of the game and its ability to visualise the effects of computer code via an interactive 3D world. It is structured to be progressed through in a linear manner, where one puzzle needs to be solved before the player can move on to the next one (Figure 31).

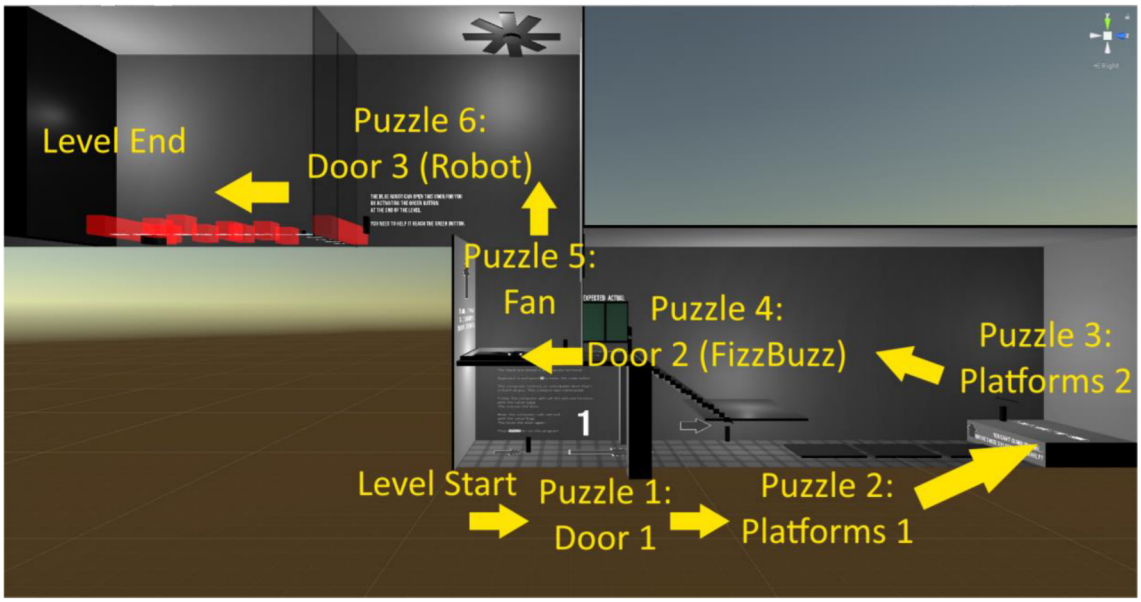

Figure 31: Side view of the demo level, with each area annotated.

There is not much in terms of artistic work done in this level; most meshes are relatively basic. The game makes heavy use of Unity's RenderTexture feature, where 2D images can be generated on runtime, and used as textures on in-game surfaces. This allowed for things like simulating a computer terminal display, or showing the currently executed line of code (Figure 32).

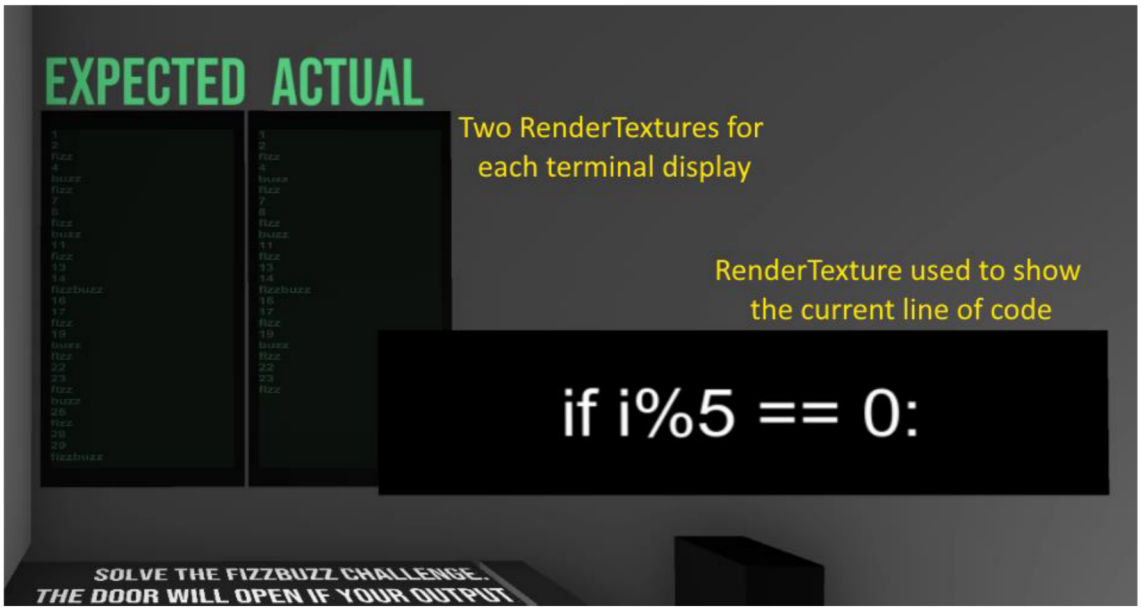

Figure 32: Example use of RenderTextures in the game's demo level

Each puzzle relies on a Computer Terminal object. This allows the player to build a program for the puzzle and to run it (Figure 33).

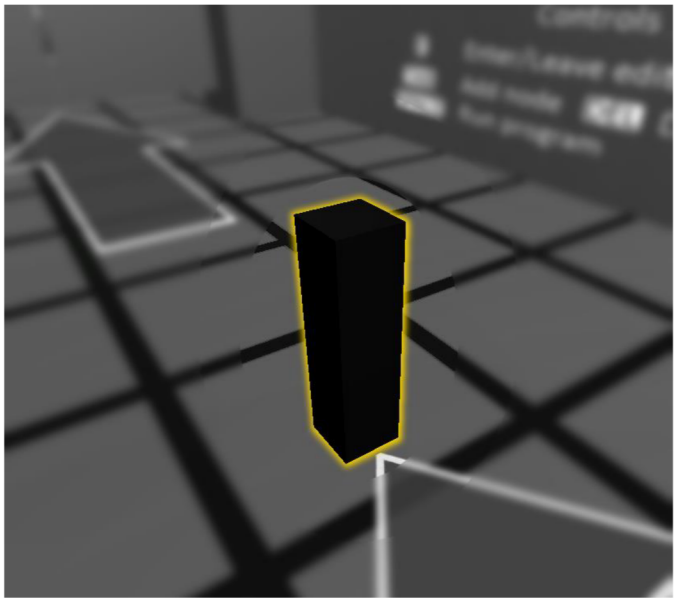

Figure 33: An example Computer Terminal in the game. The player has to approach the black tower to interact with it.

The level was intended to increase in complexity/difficulty with each puzzle completed in order to make the player's progression feel worthwhile.

In order to view/edit the level in the Unity editor, its asset file is located in *Assets/Scenes/Levels/Tutorial*.

Testing

Given the complex nature of games as software, testing was a challenge when developing this artefact. Each feature of the game was implemented in a separate testing environment first (*Assets/Scenes/SampleScene* for world testing and *Assets/Scenes/EditorTest* for editor UI testing).

Once the separate components have been integrated into the demo level, the tests were mainly black-box playtests carried out by the developer. If any issues were identified during the regular playtest of the level – in which the goal was to successfully reach the end of the level with new solutions to the puzzles, where possible – then tracking them down in source code was relatively simple due to the code being split into scripts, as well as the fact the game logs a lot of information during runtime, which can be used to identify culprits.

Research

Participant recruitment

As this project focuses on improving programming skills, which is most crucial in early stages of higher education on Computer Science courses, the aim was to recruit mainly first-year students from the University of Lincoln's School of Computer Science (SoCS). In practice, advertising the study could include distributing information leaflets around the SoCS building or visiting first-year workshops to invite participants (this would only be done with the appropriate permission from academic staff). However, as already mentioned, recruitment for the study coincidentally commenced around the time the UK has entered lockdown due to the Coronavirus outbreak. As such, the recruitment methods were changed to adapt to the new situation; a Web page with all

necessary information was set up (www.tomzajac.co.uk/gamification), with both the Participant Information Sheet (Appendix 2) and the Consent Form (Appendix 3) made available in electronic form.

Ultimately, the recruitment was carried out online; invitations were sent over Twitter (Tomek Zajac, 2020) as well as the University of Lincoln's Computer Science Society's Discord server (Discord being an online chat software). This resulted in 1 in-person and 7 online respondents to the study (total 8).

Ethics

Before starting the study, the project was granted an EA2 (Humans) ethical approval upon submitting the form (Appendix 4). The Participant Information Sheet (Appendix 2) and Consent Form (Appendix 3) were consulted with the project supervisor before beginning recruitment.

With the study being online-based and fully anonymous, the online participants were not asked to submit the Consent Form. This decision was taken out of concern for their security (sending signatures over the Web/email still has a small risk of the signature being stolen, etc.). One participant was able to take part in the study before the UK introduced social distancing measures; they were asked to sign the consent form.

Study design

The study was a system validation study, seeking the users' opinion on using the software and its potential. The main hypothesis was that novice programmers might find the concepts daunting, especially when asked to write real, runnable code in a text editor, but might be motivated/inspired to learn it if they're able to see it used in a more friendly environment (ie. a computer game).

As such, the participants were asked some background questions regarding their use of programming in their lives, as well as what their gaming habits and preferences are. The main part of the survey then focused on how effective the found each puzzle to be and what their opinion on the editor and the game overall was (Appendix 1).

Procedure

Each participant would first visit the Web page with essential information about the study, located at www.tomzajac.co.uk/gamification. This Web page explained the purpose of the study, the procedure and provided the researcher's contact information. From there they could also download the game.

Each participant would play the game, then complete the online survey on Qualtrics (survey software). Finishing the demo level successfully was not a requirement to continuing onto the survey; this would be indicated in one of the questions.

Participants were also given a user manual for the game (Appendix 5) and were also made aware that they can contact the researcher/developer if they need help with the user interface in the game or if they encounter any issues, which most of them made use of.

Results

The online participants' responses were exported from Qualtrics into an Excel spreadsheet where they were combined with the response from the paper-based survey given to the one in-person participant (Appendix 6).

For all nominal and ordinal data, a mode was identified. For ordinal data, also, a weighted average was calculated. The weights are based on the Likert scale used in each question (Table 1), ie. on a 5-point Likert scale, the "Extremely" response is weighted at 100%, then each step lower is worth 25%

less. Equally, on a 7-point Likert scale, the positive “Extremely” response is weighted at 100%, then each step in the negative direction is worth 16.666% less. The interval value from the scale for that average value is also displayed.

5-point	Extremely	Very		Moderately	Slightly		Not at all
	100	75%		50%	25%		0%
7-point	Extremely +	Moderately +	Slightly +	Neither	Slightly -	Moderately -	Extremely -
	100%	83.333%	66.666%	50%	33.333%	16.666%	0%

Table 1: Likert scale weights used in computing the weighted average.

The weighted average is mainly used in questions regarding the game’s effectiveness; this will allow to measure each component’s success value, as negative opinions will lower that score (ie. they’re more impactful/important), which will indicate need for improvements if that value is too low.

The respondents had nearly equally different backgrounds with regards to programming. 3 said they’ve had no programming experience, 2 said it’s their hobby and 3 said they use it in a professional capacity.

It was found that the respondents on average described their understanding of programming basics as “moderately well”, even though the single most popular option was “extremely well” (4 respondents).

The most picked learning styles were Visual (7 respondents) and Kinesthetic (4 respondents). 3 of the participants also said they associate with an Auditory learning style, while no one chose Tactile.

When it comes to gaming habits, most people (4) said they spend over 20 hours weekly on playing video games. Their preferred genres varied and as such it’s not possible to identify a dominant one, but first-person shooters, MMORPG and RTS games were each listed twice.

6 out of 8 respondents were able to finish the level successfully.

Most people (4) said the game’s presentation was “very helpful” to understanding effects of their code. On average, it was found to be moderately helpful (59%). The reasons for it being helpful included the ability to represent results of abstract thinking in a visualised environment (respondent 2), the ability to see the currently executed code (respondent 3) and how the code blocks interact with each other (respondent 5). On the other side, some also suggested a “better step by step presentation” (respondent 3) and “the fact there was no reset to the environment made it frustrating to resolve mistakes” (respondent 5).

With regards to navigating the code editor, most people (5) said it was “moderately easy”. On average it was neither easy nor difficult (62%). Some people described it as “intuitive” and mentioned that they had on-screen instructions made available to them, but also reported that the interface was “buggy” (respondent 3), had “connection glitches” (respondent 4) and editing variables was “a pain” (respondent 5).

3 people said the generated Python code view was “very helpful”, and on average participants found it to be moderately helpful (59%). Some said it was helpful because it helped them to “learn the flowchart’s inner logic” or vice-versa “learn to convert the visual into ‘tight’ programming logic” (respondent 2). Some also suggested that it would be more beneficial if it could also be edited, as they preferred coding in text rather than via a visual interface.

4 out of the 6 puzzles were found to be “extremely easy” for most, except for the FizzBuzz and Fan Launch puzzles, each identified as “extremely difficult” and “slightly easy” respectively (Table 2).

	Mode	Average Score	Average Value
1. Door 1	Extremely easy	67%	Slightly easy
2. Raising Platforms 1	Extremely easy	73%	Slightly easy
3. Raising Platforms 2	Extremely easy	58%	Neither easy nor difficult
4. Door 2: FizzBuzz	Extremely difficult	21%	Moderately difficult
5. Fan Launch	Slightly easy	46%	Slightly difficult
6. Door 3: Robot	Extremely easy	52%	Neither easy nor difficult

Table 2: Participants’ observed difficulty of each puzzle.

The difficulty of the puzzles was potentially affected by errors in the software (some respondents have reported bugs/glitches with some puzzles), inconvenience caused by the editor interface controls, and lack of a tutorial/introduction.

4 respondents said they think this system could be “very effective” in supporting computer programming education. The average score was 69% (moderately effective) and all responses were on the positive side of the 7-point Likert scale, indicating that their overall view of the game is also positive.

Analysis

The results partially confirm the hypothesis laid out before carrying out the study. While only a third of participants with no prior programming knowledge managed to finish the level successfully, the overall opinion was that the solution was helpful to their understanding of code in some way (especially considering that no one found it to be unhelpful in the end).

The study was largely focused on gathering subjective data; everyone’s experiences would be difficult based on their learning style, familiarity with gaming and prior programming knowledge. As observed in the results, people with more programming knowledge (particularly those using it professionally) did not necessarily find the system easier than others did, as they already apply a more complex mental model, suited more to traditional coding as text, which the in-game editor did not facilitate.

At the same time, the study also highlights flaws in the developer testing process, as some bugs/glitches in the software have impacted some participants’ experience with the game.

An obvious problem was an unbalanced difficulty level, with a more complex puzzle (FizzBuzz) being introduced too abruptly and without a proper tutorial that would explain how to implement it or what commands should be used for implementing it.

Project Conclusion

Attempts at Gamification in the area of programming education should be continued, and more varied solutions should be explored. In today’s world, new students are oftentimes young people who have grown up gaming, and find it to be essentially their second nature. Using that entertainment technology to assist in education is therefore a praiseworthy idea, but existing solutions so far are very strict about the fact that they are purely educational products. People have

grown to enjoy video games because they act as an "escape" from the real world. The experiences they have in those games can still impact their lives (or at least feelings, knowledge, opinions, etc.) but the difference is those educational games might just not seem enjoyable for them. This is the very reason why the software artefact in this project was inspired by puzzle games (or games containing puzzle or coding-related elements) with some popularity (which happen to be mainly commercial products – *Minecraft*, *Shenzhen I/O*, *Portal*) because they are known to have accumulated their success due to player engagement. One could say this attempt was even more focused on disguising its roots in teaching programming by presenting itself as an FPP puzzle game. For this reason, the actual learning value from the game might be limited, but this was known from the very start – this project aims to merely assist in that education, not fully replace the main learning material in form of lectures, workshops etc. It is the author's stance that using increasing student motivation to use those main channels is a more effective use of Gamification than trying to make the product a full resource in and of itself.

With limited data due to complicated circumstances at the time of writing, it is not possible to accurately and fully assess the effectiveness of the proposed system. The data available suggests that the system could find an audience that would benefit from using it. The artefact could benefit from more variety in the level. Early plans included concepts such as programmable vehicles for fetching mission items, programmable weapons to use different ammunition and aiming patterns to fight different enemy types, etc. The ambitions of the game had to be lowered, however, due to development trouble. The processes used in developer testing were also not reliable enough, which translated to issues being reported in the study results. Had those issues been avoided/addressed, though, the results could be more helpful to the research question.

References

Literature

1. Maxim, B. R., Decker, A., Yackley, J. J. (2019) Student Engagement in Active Learning Software Engineering Courses. In: *2019 IEEE Frontiers in Education Conference (FIE)*, Cincinnati, USA, 16-19 October. IEEE, 1-5. Available from https://proxy.library.lincoln.ac.uk/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.9028644&site=eds-live&scope=site [accessed 10 April 2020].
2. HESA (2020) *What do HE students study?*. HESA. Available from https://www.hesa.ac.uk/data-and-analysis/students/what-study [accessed 10 April 2020].
3. Blake, V. (2019) *New research shows 'record numbers' of UK people aged 55+ play video games*, MCV. Available from https://www.mcvuk.com/business-news/new-research-shows-record-numbers-of-uk-people-aged-55-play-video-games/ [accessed 10 April 2020].
4. Ford, V.B., Roby, D.E. (2013) Why Do High School Students Lack Motivation in the Classroom?. *Global Education Journal*, 2013(2) 101-113. Available from https://proxy.library.lincoln.ac.uk/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edo&AN=93246675&site=eds-live&scope=site [accessed 10 April 2020]
5. Zainal, N. F. A., Shahrani, S., Yatim, N. F. M., Rahman, R. A., Rahmat, M., Latih, R. (2012) Students' Perception and Motivation Towards Programming. In: *Universiti Kebangsaan Malaysia Teaching and Learning Congress 2011,* Selangor, Malaysia, 17-20 December. Procedia – Social and Behavioral Sciences, 277-286. Available from https://proxy.library.lincoln.ac.uk/login?url=https://search.ebscohost.com/login.aspx?direct

=true&db=edselp&AN=S187704281203724X&site=eds-live&scope=site [accessed 10 April 2020].

6. Skalka, J. and Drlík, M. (2018) Priscilla – Proposal of System Architecture for Programming Learning and Teaching Environment. In: *2018 IEEE 12th International Conference on Application of Information and Communication Technologies (AICT)*, Almaty, Kazakhstan 17-19 October. IEE, 1-6. Available from https://ieeexplore-ieee-org.proxy.library.lincoln.ac.uk/document/8746921 [accessed 13 April 2020].
7. Montes-Leon, H., Hijon-Neira, R., Perez-Marin, D. and Leon, S.R.M. (2019) Improving Programming Learning on High School Students through Educative Apps. In: *2019 International Symposium on Computers in Education (SIIE)*, Tomar, Portugal, 1-6 November. IEEE, 1-6. Available from https://proxy.library.lincoln.ac.uk/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.8970112&site=eds-live&scope=site [accessed 13 April 2020].
8. BBC (2020) *What is computational thinking? - Introduction to computational thinking - KS3 Computer Science Revision - BBC Bitesize*. BBC. Available from https://www.bbc.co.uk/bitesize/guides/zp92mp3/revision/1 [accessed 14 April 2020].
9. Sangosanya, W., Belton, D., Bigwood, R. (2005) *Basic Logic Gates*. Surrey: University of Surrey. Available from http://www.ee.surrey.ac.uk/Projects/CAL/digital-logic/gatesfunc/index.html [accessed 14 April 2020].
10. CAML Academy (2019) *Engineering with Minecraft Redstone*. CAML Academy. Available from https://www.camlacademy.com/courses/redstone-engineering/ [accessed 14 April 2020].
11. Bendell, C. (2018) *Results for Blueprint Usage Survey* [Reddit post]. Sent to r/unrealengine, 09 May. Available from https://www.reddit.com/r/unrealengine/comments/8i82vh/results_for_blueprints_usage_survey [accessed 15 April 2020].
12. Khaleel, F. L., Ashaari, N. S., Wook, T. S. M. T., Ismail, A. (2017) Gamification-based learning framework for a programming course. In: *2017 6th International Conference on Electrical Engineering and Informatics (ICEEI)*, Kedah, Malaysia, 25-27 November. IEEE, 1-6. Available from https://proxy.library.lincoln.ac.uk/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.8312377&site=eds-live&scope=site [accessed 16 April 2020].
13. Crocco, F., Offenholley, K., Hernandez, C. (2016) A Proof-of-Concept Study of Game-Based Learning in Higher Education. *Simulation & Gaming*, 47(4), 403-422. Available from https://proxy.library.lincoln.ac.uk/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=psyh&AN=2016-33896-002&site=eds-live&scope=site [accessed 16 April 2020].
14. Oblinger, D. G. (2004) The Next Generation of Educational Engagement. *Journal of Interactive Media in Education*, 2004(1), 1-18. Available from https://proxy.library.lincoln.ac.uk/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsdoj&AN=edsdoj.b5a38a06edee4c5bb00548ca0d4c81c5&site=eds-live&scope=site [accessed 16 April 2020].
15. Oblinger, D. G. (2003) Improving education for the 'next generation'. *Community College Week*, 16(5), 3 & 11. Available from https://proxy.library.lincoln.ac.uk/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsgao&AN=edsgcl.110227902&site=eds-live [accessed 16 April 2020].
16. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D. (2001) *Principles behind the Agile Manifesto*. Agile

Manifesto. Available from https://agilemanifesto.org/principles.html [accessed 17 April 2020].
17. Sebastian Lague (2019) *Coding Adventure: Coding a Coding Game* [video]. Available from https://www.youtube.com/watch?v=dY6jR52fFWo [accessed 17 April 2020].
18. Johnson, S. (2008) Analysis: The Quandary Of 2D Vs. 3D. *Gamasutra*, 21 November. Available from https://www.gamasutra.com/view/news/112124/Analysis_The_Quandary_Of_2D_Vs_3D.php [accessed 19 April 2020].
19. Limelight (2019) *Market Research: The State of Online Gaming – 2019*. Limelight. Available from https://www.limelight.com/resources/white-paper/state-of-online-gaming-2019 [accessed 19 April 2020].
20. Winslow, L. E. (1996) Programming pedagogy – a psychological overview. *ACM SIGCSE Bulletin*, 28(3) 17-25. Available from https://proxy.library.lincoln.ac.uk/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsbl&AN=RN013567803&site=eds-live&scope=site [accessed 19 April 2020].
21. Ghory, I. (2007) *Using FizzBuzz to Find Developers who Grok Coding*. Imran On Tech. Available from https://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/ [accessed 19 April 2020].
22. Tom Scott (2017) *FizzBuzz: One Simple Interview Question* [video]. Available from https://www.youtube.com/watch?v=QPZ0pIK_wsc [accessed 19 April 2020].
23. Tomek Zajac (2020) *BSc Dissertation Project study invitation* [Twitter]. 10 April. Available from https://twitter.com/_Tomeztos/status/1248676678542462985?s=20 [accessed 22 April 2020].

Figures

1. Valve Corporation (2011) *Portal Test Chamber 08*, Portal Wiki. Available from https://theportalwiki.com/wiki/File:Portal_Test_Chamber_08.png [accessed 13 April 2020].