School Of Computer Science and Statistics

Pytch Junior: Identifying Design Features to Further Ease the Burden of Transitioning from Blocks to Text Using Pytch.

Ellen Whelan

Supervisor: Glenn Strong
August 2022

A Dissertation submitted in partial fulfilment of the requirements for the degree of Masters in Computer Science (MCS)
Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Ellen Whelan
5th August 2022
Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Ellen Whelan
5th August 2022
Pytch Junior: Identifying Design Features to Further Ease the Burden of Transitioning from Blocks to Text Using Pytch.

Ellen Whelan

University of Dublin, Trinity College 2022

Supervisor: Glenn Strong

Abstract
The inclusion of Computer Science in Irish secondary school curriculums from 2018 is indicative of the rising importance of coding education in Ireland. Our UK and US counterparts, having previously introduced coding education in their schools, convey the international interest in the field and provide an insight into why ongoing research in the field is so prolific. The largest gulf in coding education exists when novice programmers make the jump from block-based programming to text-based, setting the stage for so much of this research to be focused on the challenges of such a transition and approaches to overcoming them. Pytch, is part of this research, and one approach to bridging the gap from blocks to text. An intermediary environment for novice programmers to be eased into Python, Pytch supports transitioning to text-based coding, but has scope for additional support.

This work seeks to identify key areas in Pytch that offer room for additional support, to ease the burden of transitioning for novice programmers. It investigates other transitional approaches, to determine what design features, if any, can be incorporated into a new environment to sit between Scratch and Pytch in a programmer’s educational timeline. This work proposes a novel, research-informed design for this environment, known as ‘Pytch Junior’ and provides both a theoretical approach for implementing the design and an evaluation framework for gauging how successful the design is in promoting conceptual understanding of programming, while supporting novice programmers in transition.
Acknowledgements

First and foremost, I would like to extend my thanks to my supervisor Glenn Strong. For all the advice and support he has offered me over the past ten months or so, as well as the continued encouragement and guidance above and beyond the call of an academic supervisor.

Also, to Ben North for generously offering his assistance and time, to help in the technical side of the project.

To my tutor Diana Wilson, my most sincere gratitude. Diana's help and guidance throughout all five of my years in Trinity has been invaluable to me. Her unwavering support, enthusiasm and her willingness to lend a helping hand are things I deeply admire, and traits I hope to take forward in my own life.

Finally, to my mum and sister, for everything they've done for me over the years. This would simply not have been possible without their love and support.

Ellen Whelan

University of Dublin
August 2022
Contents

Abstract ................................................................................................................................................... 4
Acknowledgements ................................................................................................................................. 5
1 Introduction ......................................................................................................................................... 9
  1.1 The Research Question ........................................................................................................... 9
  1.2 Challenges to Be Overcome .................................................................................................... 9
  1.3 Research Methodology ......................................................................................................... 10
  1.4 Paper Outline ........................................................................................................................ 11
2 Background and Literature Review .................................................................................................. 12
  2.1 The Relevance of Coding Education ....................................................................................... 12
    2.1.1 Coding Education in Irish Schools ....................................................................................... 12
    2.1.2 Why Introduce Computer Science in Irish Schools? ........................................................... 13
    2.1.3 Why Use Specific Tools? ..................................................................................................... 14
  2.2 Block-Based Programming ......................................................................................................... 16
    2.2.1 Why Start Off with Blocks? ................................................................................................. 16
    2.2.2 Benefits of Block-Based Programming in Teaching Programming Concepts ..................... 16
    2.2.3 Evaluation of Block-Based Programming Languages ......................................................... 17
  2.3 Block-Based Programming to Text-Based Programming ........................................................... 19
    2.3.1 Motivation for Transitioning from Block-Based to Text-Based Programming ................. 19
    2.3.2 Transitional Difficulties ....................................................................................................... 20
    2.3.3 Approaches to The Transition .............................................................................................. 22
  2.4 The Wonderful World of Text Editors ........................................................................................ 25
    2.4.1 A Brief History of Text Editing ........................................................................................... 25
    2.4.2 What's Out There Now? ....................................................................................................... 25
  2.5 Pytch ........................................................................................................................................... 27
    2.5.1 What is Pytch? ..................................................................................................................... 27
    2.5.2 How Does Pytch Ease the Transition from Blocks to Text? ................................................ 27
    2.5.3 Where Can We Add More Support? .................................................................................... 28
    2.5.4 Introducing Pytch Junior ...................................................................................................... 28
3 Design ............................................................................................................................................... 30
  3.1 Stage 1: Conception of Pytch Junior and Identifying Solid Design Goals ................................. 30
    3.1.1 Conception and the First sketch ........................................................................................... 31
    3.1.2 Research and Identifying Priority Features ........................................................................ 32
  3.2 Stage 2: Preliminary Environment Sketches ............................................................................... 33
  3.3 Stage 3: A Comprehensive Environment Design Justified by Research ................................. 41
    3.3.1 Code Organisation by Sprite ............................................................................................... 41
    3.3.2 Reducing Typing Requirements ........................................................................................... 47
    3.3.3 Visual Cues .......................................................................................................................... 51
List of Figures

Figure 1: Diagram showing the iterative process of Design-Based Research. ......................................... 10
Figure 2: A Pytch class named Banana with some basic functions. .......................................................... 31
Figure 3: The same Banana class organised into a new sprite-organised structure. ............................. 31
Figure 4: Three primary zones in Pytch Junior's UI. .................................................................................. 33
Figure 5: First sketch on how to structure the coding and assets area of the Pytch Junior interface. ....... 34
Figure 6: The third sketch, showing the assets hidden in a tab beside text output and errors. ............... 35
Figure 7: Sketch moving assets back to the coding area and introducing organisation by sprite. ............ 36
Figure 8: Swapping the tab orientation in the coding area. .................................................................... 37
Figure 9: The first sketch of the internal coding space within a sprite tab. .......................................... 38
Figure 10: The new coding window with the variables box presented first to the user. ........................... 39
Figure 11: The full layout to support code organisation by sprite. ........................................................... 40
Figure 12: The full interface mock-up for Pytch Junior demonstrating a typical programmers view for a 'Hello, World!' program. ......................................................................................... 41
Figure 13: Sprite tabs in Pytch Junior. ....................................................................................................... 43
Figure 14: Class tabs in VS Code. ........................................................................................................... 43
Figure 15: The pop-up window triggered by the 'Add Sprite' button. ..................................................... 43
Figure 16: UI mock-up showing the costumes page for the 'elephant' sprite. ....................................... 45
Figure 17: UI mock-up of the process to upload a new costume. .............................................................. 46
Figure 18: Mock-up showing the stage tab, differentiated by colour. ................................................... 47
Figure 19: Process of adding a new event handler function, using a dropdown menu to select the event. ................................................................................................................................. 48
Figure 20: The process of adding a new variable to the 'elephant' sprite. .............................................. 50
Figure 21: A reminder of the Pytch Junior UI design. ............................................................................... 52
Figure 22: A reminder of the Pytch Junior UI design - including the stage. .......................................... 52
Figure 23: A view of the structure of a user's program. .......................................................................... 57
1 Introduction

1.1 The Research Question

Transitioning from block-based to text-based programming is a widely accepted pathway for novice programmers to grow their skills but it is not without its challenges. Such challenges are explored by a number of researchers, and various solutions have been proposed in a variety of papers. Kölling and Brown, in their paper proposing ‘Frame Based Editing’, list these challenges as motivation for their idea, including difficulties with syntax, readability, mechanical typing, managing layout and structure, grouping, and writing expressions [36]. Frame based editing is one of the many proposed solutions out there, to attempt to ease these transitional issues, Pytch is another.

Pytch allows users to create more authentic, flexible programs that are both larger and more complex than programs possible in Scratch, by utilising text-based coding with visual output similar to Scratch. However, in Pytch, feedback from students and teachers tell us that the transitional issues of managing layout and structure, syntax, mechanical typing, and readability remain. These issues make it difficult for novice programmers to continue programming when they make the switch to Pytch and discourage them from continuing their coding education.

The question this work seeks to answer is if there are design features that can be used in a novel way to add support in key areas of Pytch, while maintaining authenticity and flexibility, to produce a new intermediary environment.

The proposed solution, outlined in this work, is an intermediate version of Pytch for novice programmers to engage with before they move up to Pytch - we’re calling it Pytch Junior for the time being. The aim of Pytch Junior is to maintain the authenticity and flexibility introduced by text-based coding, but also to mitigate the loss of clean, code structure, visual cues, and text organisation. By reintroducing code organisation by ‘sprite’ and reducing the amount of physical typing for users (down to just the body of event handlers), the goal is to allow users to focus on the structure of their programs, and the understanding of the high-level concepts they’re using. Syntax errors should be reduced, to alleviate frustration, and code management is simplified, allowing users to know where and how to make changes to their code more easily when the need arises. This code organisation by sprite, as well as helping the user understand program structure, will reduce the volume of text presented to a user at once which has shown to be overwhelming and confusing. Memory aids can also offer an additional layer of support to remind users of the capabilities in Pytch Junior.

To address the research question, this work sets out to identify key areas in Pytch that provide scope for additional support to meet a variety of transitional challenges that users reported they still struggle with in Pytch. Adding support in these areas form the design criteria for this work, and motivate the investigation into what features, if any, could be included in a design to address these areas, with the end goal of supporting a user’s understanding of programming concepts, structures and behaviours.

1.2 Challenges to Be Overcome

With any educational programming environment, the goal is simple - aid in the teaching of programming concepts and computational thinking. But like any educational tool, the learner has to be eased into it. As Pytch Junior is a transitional environment for programmers, there should be a low barrier for entry. Programmers should be supported where possible, to foster confidence and allow them to progress with their coding education. But these supports must strike a delicate balance between, hiding complexity and misleading the novice programmer.

In their panel on the future directions of block-based programming, Brown et al. discussed the importance of modality in coding education. The choices developers in this field make in what representations used to present programming concepts affects how novice programmers understand programming practices [49]. This means care must be taken to ensure the representation chosen is one
that doesn’t mislead the learner in the underlying concepts or practices, while also giving the learner a chance to understand the representation and how they can use it.

1.3 Research Methodology

The research methodology informing this research piece is Design-Based Research, or DBR. This methodology is popular in educational research, and involves developing solutions to problems, testing how they work, and then adapting and re-testing to collect more data, all with the end goal of generating new frameworks for “conceptualised learning, instruction, design processes, and educational reform” [52]. The Design-Based Research collective writes that DBR can help create and extend knowledge about developing, enacting, and sustaining innovative learning environments [53]. This project began as an idea about extending the learning environment, Pytch, in some novel way, to support a user’s conceptual understanding of programming, so DBR was deemed a suitable methodology to follow. DBR grounds itself in the needs, constraints and interactions of local practice, and can help bridge the gap between theoretical research and educational practice, to produce effective learning in educational settings. As a learning tool, Pytch seeks to be effective specifically in the coding educational setting, and by employing DBR to inform the research methods used in this project, it fosters confidence that it will eventually yield a valid result.

Design-Based Research involves a series of phases that this project followed. Figure 1.1 shows a diagram of these iterative phases and how they feed back to refine the solution produced by the DBR method [54].

![Diagram showing the iterative process of Design-Based Research.](image)

Herrington et al. introduced these phases in their paper in DBR for doctoral students [54].

Phase 1 involves the addressing of complex, practical problems by researchers and practitioners in collaboration. This project used feedback from previous Pytch testing sessions with students and teachers alike, as the consultation with practitioners to define the problem statement and research question. Literature review into the background of coding education was also conducted in this phase, and this literature review was returned to at each stage of the iterative design process, building on this initial background, with research into block-based programming and its benefits, transitional challenges and approaches, and the existing state of Pytch to inform the design solution to the research question.

Phase 2 involves developing solutions informed by existing design principles and technological innovations. This is an iterative process of designing a solution, and then returning to the literature for further research, with a more specific focus. This research then informs the next design stage in refining the solution. This iterative literature review process, working in tandem with an iterative design process was followed to satisfy one the Design-Based Research collective’s criteria for good DBR, that the goals of designing learning environments and developing theories of learning are intertwined.
Herrington also notes that DBR protocols demand “intensive and long-term collaboration involving researchers and practitioners”. Completing all the phases of DBR, and the entire iterative process to produce a fully-fledged implemented solution, is beyond the timeframe of this dissertation. Instead phase 1 and 2 are the focus of the research work for this project. However, a framework for the implementation and evaluation involved in phase 3 and 4 is also included.

Iterative cycles of testing and refinement of solutions in practice, comprise phase 3. The framework for implementation in section 4 of this paper suggests a first implementation of the design required in this phase. The framework included in section 5 describes user trials including participants, data collection and analysis and how these trials may inform second and further iterations of implementation. Phase four of DBR is reflecting on these iterations to produce design principles and enhance the final implementation. The data collected by the hypothetical user trials of section 5, combined with further innovations described in the future work section could serve this phase in the future.

1.4 Paper Outline
This report begins in section 2 by establishing the background overview of the project. It begins by exploring the relevance of coding education as the overall motivation for research in the area, before outlining the available tools in block-based programming, and the motivation and obstacles for transitioning to text-based programming. The varying approaches to this transition are identified, and briefly described, before Pytch is analysed in its current form through the lens of the transitional challenges discovered, and in comparison, to the other approaches documented. From this analysis, key areas of Pytch are identified as suitable for increased learning support to target transitional challenges.

Identifying these areas of potential, is what led to the conception of Pytch Junior and the eventual design outlined in this paper. Papert outlines the criteria for a programming language, that they should have ‘low floors’ and ‘high ceilings’ [47]. This design wants to lower Pytch’s floor even more, while retaining its high ceilings. Pytch Junior involves a major UI restructuring to reintroduce code organisation by sprite to boost readability and manageability, reduced typing requirements, visual cues, and memory aids. It involves taking the research outlined in this section and carefully applying it to the initial idea to produce a thorough, informed design via an iterative process, with considerations for how this can be achieved technically. It also includes a means of evaluating the success of the design and this project's body of work.

Section 3 provides the detailed description of the iterative stages of designing this new Pytch, in tandem with the iterative stages of research. It includes design decisions taken, the justifications for them and any limitations they may introduce. The features described in this section are selected to add support in key areas to target specific transitional challenges unearthed in the research phase. This section also contains sketches and diagrams that describe the design process from conception to completion.

The implementation considerations for this design are contained in section 4 including technologies involved, technical specifications and justifications that can be expected for a practical implementation of the design.

The evaluation frameworks for this design are in section 5, and section 6 concludes the report, outlining the successes and failures of the project in answering the research question posed, limitations of the solution and potential future work.
2 Background and Literature Review

This section attempts to provide a thorough review of literature in the area of coding education, specifically focusing on the issues arising from block-based programming, the transition to text-based programming and the tools already available or in development to handle this transition. Once the background has been established, the basis of this project, the hybrid environment Pytch [1], will be introduced and reviewed, before briefly mentioning the changes proposed in this work.

As outlined in section 1.3, this literature review happened in two main stages as per the principles of Design-Based Research (DBR). The first stage of establishing the relevance of coding education was completed as part of phase 1. The second stage was an iterative process happening in tandem with the iterative stages of design as per phase 2 of DBR. At the completion of each design iteration, the literature review was revisited, to identify design principles and technological innovations in block-based programming, transitional approaches and Pytch itself to inform the changes in the next iteration of design.

The first literature review iteration explored the relevance of coding education as motivation for further research on Pytch, the second explored literature to find the key areas of Pytch that have scope for additional support. The third analysed block-based environments and their benefits that lead to transitional challenges when moving to text, approaches to transitional challenges and their successes with a focus on code organisation by sprite, and how other technological innovations may inform how to achieve it. The fourth literature review iteration that took place before the final design stage, reviewed block-based and transitional approaches once more with a focus on identifying innovations in visual cues, memory aids and reducing user typing requirements that may lend support to the design for Pytch Junior.

By following a DBR approach, the literature review becomes an iterative process intertwined with the design process, and thus both are part of the technical content and execution of this project. For the sake of readability, this paper keeps the two processes in separate sections. Additionally, two stages of literature review are included in one section here to afford the reader the full background information necessary to understand the design decisions taken.

2.1 The Relevance of Coding Education

Coding education is becoming increasingly relevant and widespread in modern times. Organisations like CoderDojo, who provide coding classes free of charge reported in 2017 that 160,000 young people had been ‘positively impacted’ by their organisation since its inception in 2011. The organisation's reach across 1800 ‘Dojos’ in 92 countries demonstrates the ubiquity, and relevance of coding education and the global demand for classes [2]. This subsection explores this demand through the lens of coding education in schools, the motivation behind its introduction to curriculums and why specific tools are used both in the school and extracurricular environments.

2.1.1 Coding Education in Irish Schools

Ireland began offering Computer Science (CS) at Leaving Certificate level in September 2018, with the first exam due to be sat in June 2020 (although the COVID-19 pandemic disrupted these plans). Initially forty secondary schools were included in this pilot program of sorts, and were to sit an exam worth 70% of their final grade, with an in-school project worth the remaining 30% [3]. The subject has proven popular with the 40 initial schools, already increasing to 140, and approximately 500 students taking the adapted online exam (due to the pandemic) in November 2021[4].

The Leaving Certificate curriculum contains three main learning strands: practices and principles, core concepts, and computer science in practice. Through these learning strands, students are expected to gain skills in critical and creative thinking, information processing, effective communication, adaptivity and problem solving both as an individual and part of a team [5]. Notably, the concept of classes is not included on the Leaving Certificate curriculum. This becomes relevant later, in the design chapter. By
studying for their written exam, and completing their project, students learn to read and modify computer programs, as well as design webpages and systems. Section A and B of the exam contain short and long questions on a variety of topics, ranging from Boolean logic, to Computer Science history, while section C tests students on their programming skills.

While the introduction of CS to the leaving certificate curriculum, and the introduction of coding and computational thinking in primary schools (included in the maths curriculum), have been welcomed by many, it is worth nothing that Ireland is a little behind the times. Especially when one considers Dublin’s ranking inside the top 20 ‘Top Tech cities’, a list curated by Savill’s World research [6], ahead of both Hong Kong and Melbourne. Our counterparts in the UK began teaching Computer science at A-level in 2015. They also became the first G7 country to introduce mandatory CS education in primary schools in 2014, for all students aged 5 to 16 [11]. Across the pond in the United States, Computer Science classes have been offered in some High Schools since the 1960s. So why have we finally caught up with our friends in the UK and US? The below section explores some of the reasons why Leaving Certificate students now have the opportunity to dip their toe into the world of CS.

2.1.2 Why Introduce Computer Science in Irish Schools?

The first motivation for the new Leaving Certificate subject was a matter of government policy. Richard Bruton, who was Minister for Education and Skills at the time, announced the new subject as part of the government's plan to put Ireland top of the list in terms of education and training by 2026 [3]. Bruton himself used the term ‘digital revolution’ when announcing his department's plans, which speaks to the shift in Ireland’s industry away from traditional to technology, and the associated demand for a workforce skilled in CS and programming. Ireland’s low corporation tax rate has resulted in large tech multinational companies setting up large bases and European headquarters in Dublin, and in turn looking for ‘tech talent’ in unprecedented numbers. The shortage of tech talent has long been a concern of both government and industry [7]. While these tech companies find more, and more ways to squeeze more perks into their office spaces, in the form of ping pong tables, interactive floor displays, and free yoga classes, the reality is Ireland is not producing enough Computer Science graduates. One way to tackle this is to introduce the subject in schools, with the hope that more Irish students will go on to enrol in CS courses at third level.

This shortage at home and abroad is only set to worsen, as we move towards Industry 4.0, a concept coined by the German government in 2011. This is the next revolution in industry and experts say it will involve predictive maintenance using IoT and Machine learning in cyber physical systems [8]. This move towards smart factories means an increase in cloud computing, augmented reality, machine learning, Internet of Things, cybersecurity and more. In a nutshell, it means yet another increase in technology-based jobs and a reduction in more traditional employment.

While in Ireland the primary motivation for having CS in schools is currently to produce a more relevantly skilled, hireable workforce to satiate the needs of modern Ireland and our resident tech giants, there are a variety of other arguments for introducing children to CS from primary school, including problem solving, critical thinking and communication skills. Kölling et al., in the opening of their paper introducing Stride in BlueJ, describe CS and programming as ‘relevant to all’ to justify its inclusion in school curriculums [9]. Digital fluency or computer literacy are terms that get bandied about a lot when it comes to education, and it seems sensible that schools should be ensuring that students are digitally fluent as they enter an ever-increasing digital world. Resnick, however, argues that while it may have been historically adequate to define digital fluency as the ability to browse and interact with digital media, the definition must now also include the ability to design and create new digital media [10].

In an opinion piece by Gaby Hinsliff in the Guardian from 2015, shortly after the UK introduced CS at A-level, she discusses at length the many benefits of coding education for children [11]. The author does cite the British technology industry as a motivation for coding education, as well as the recruitment crisis they’re experiencing across the Irish sea (similar to the one we wrestle with here). However, she also explores other encouraging factors for teaching children to code. Hinsliff writes that it is becoming
increasingly common to consider mastering code and the skill of computational thinking, as essential to a child’s education as numeracy and literacy. She notes that as years of English lessons do not make a best-selling novelist, hours of coding education may not necessarily make every child the next Larry Page. Nevertheless, it is much more preferable that every child should learn basic coding and computational skills, so that they may at least understand the world and increasingly ubiquitous technology that surrounds them.

Learning to code later in life is not only difficult, but expensive if one chooses the many coding bootcamps for adults that have popped up in recent years. Many of these boot camps as well as being exclusive in their enrolment policies, also cost a pretty penny - one camp cited by Hinsliff running at £8000 for three months. Mandatory CS education, like the UK have, means the new generations of Britons coming up shouldn’t need these bootcamps, and ensure all children regardless of race, creed or gender will receive some level of CS education. This is perhaps especially important when you consider students from less advantaged backgrounds, where there may not be the means for coding education at home, i.e., no laptops, tablets or smartphones. Children from such backgrounds are now exposed to these technologies in their school through their coding lessons. The hope being that mandatory CS in UK schools will help to make the industry more inclusive for children from all backgrounds. The east end of London, historically a working-class area, is home to ‘Tech city’, a thriving neighbourhood of tech start-ups and multinational companies. Mandatory CS education, Hinsliff argues, could potentially enable native EastEnders to engage in the industry in their backyard, by increasing equal opportunities and encouraging children to pursue STEM careers in higher education, regardless of their financial background.

Hinsliff also includes the infamous Steve Jobs quote in her article:

“Everyone should learn how to program a computer because it teaches you how to think.”

Jobs uttered these famous words back in 1995. Learning to code offers the student much more than coding skills. Critical and computational thinking, as well as problem solving, are skills that transfer to virtually any context or industry, and their value cannot be overstated. Whether Jobs heeded his own advice or not, is a topic of debate in many corners of the internet, but his words remain relevant today.

2.1.3 Why Use Specific Tools?
The above sections establish the motivation for including coding and computational thinking in school curriculums and coding education as a wider concept, but how can it be achieved? This subsection discusses the need for specific educational tools and supports in learning to code.

When teaching someone to read, it would be absurd to hand them a copy of some Tolstoy and leave them to it. Similarly, we can’t expect a beginner programmer to be shown a Java program, and understand even a single line of it. Learners must be eased into it. In the beginning, certain complexities should be hidden from a user. Difficult concepts should be simplified where possible, without misrepresenting how things work, or instilling poor developer habits. The best educational languages should support the new programmer by hiding and simplifying where necessary, without being detrimental to the novice’s programming habits as they progress past the initial learning tools. This non-trivial problem of achieving balance, is part of the central research aims in this project and will be returned to throughout this paper.

One approach for beginner programmers, is to begin with block-based languages, with intuitive drag-and-drop interfaces, visual cues, and significant hidden complexities. These languages, like Scratch, offer a ‘low floor’ [10] to support beginners and will be discussed in further detail in the following section. They do, however, have a limit on flexibility and authorship, and when a user reaches these limits, they usually transition to text-based languages. The issues and difficulties with these transitions
are also discussed in further detail in a later section. Many approaches including frame-based editing and hybrid environments have been developed to ease this transition. Another emerging approach is the concept of gradual languages [12], where the language’s syntax becomes increasingly complex as the programmer learns, an idea based on the way natural languages are taught.

As coding education is becoming increasingly common as part of school curriculums, across the globe, more and more research is being completed in the area and more approaches are being developed [9]. But whatever tool is used, there are a variety of reasons to use them. In a study aiming to justify the inclusion of CS in middle school in the United States by Armoni et al., the authors put forward a number of benefits of starting new programmers off on Scratch [13]. The study found the group with Scratch experience performed significantly better in understanding programming concepts like loops and decision making, and the students reported higher levels of motivation and self-efficacy. In addition to the positive impact on students, teachers involved in the study reported increased efficiency in the classroom and a higher engagement with CS in high school, amongst the students who were introduced to Scratch first.

These benefits serve as motivation for beginning your programming education using introductory tools, specifically block-based languages. The following section delves further into the world of block-based programming.
2.2 Block-Based Programming

This section explores in more depth the world of block-based programming, the languages available and their benefits. The first subsection discusses why new programmers should begin specifically using block-based languages, continuing on from the discussion in the previous section on more general education tools. The benefits of block-based languages in teaching programming concepts will then be reviewed, followed by a review of a number of major block-based languages available to use.

2.2.1 Why Start Off with Blocks?

Block-based languages are generally viewed as a gateway to text-based languages like Java or Python [14]. In the same manner as the colourful Lego blocks that allow a child to build their own physical creations, block-based programming languages are designed to engage children to build their own digital creations. Colourful blocks and a drag-and-drop interface, mean they’re intuitive to use, and less intimidating than more traditional text-based languages.

In their study asking students to compare Snap! (a block-based language very similar to Scratch) to Java, Weintrop and Wilensky found 58% of students surveyed found Snap easier to use [15]. When asked for the differences they perceived between Snap! and Java, the students provided a variety of answers.

- **Easier to read**: Students surveyed found the blocks were easier to read than the scroll of text in Java. The descriptor labels on the blocks in natural language, also aided their understanding and the readability of the code.
- **Visual Clues and Cues**: Students cited the shape and layout of the blocks, as well as the colours, and shading as another major difference that made Snap! easier to use. Block shapes allowed them to guess at which blocks were compatible, while visual cues in colour and shading allowed them to group blocks into similar behavioural categories.
- **Drag-and-Drop Interface**: The drag-and-drop interface, compared to Java’s text editor interface, resulted in fewer syntax errors and reduced the burden of slow typing.
- **Blocks Catalogue**: Students found the catalogue of blocks served as a valuable memory aid of what was possible in the Snap! language, and how to go about completing certain tasks. By being able to scroll through the available blocks, recognition was promoted over recall - a common Human Computer Interaction principle to ease cognitive load.

Studies on both Droplet [16] and Alice [17], back up Weintrop and Wilensky’s claim that less typing via a drag-and-drop interface, results in less syntax errors, which in turn increases user confidence. Bau et al. also highlight that blocks allow for rapid code building, and how this is an incentive for novice programmers, particularly children with short attention spans [18]. The paper also acknowledges, like Weintrop and Wilensky, that the block shapes aid a user’s understanding of what is legal and illegal within the language. It is Hansen et al., however, who summarise the benefits for block-based languages for beginner programmers [19]:

“The popularity of block-based programming environments is no surprise; these interfaces reduce the cognitive load required of novice student programmers.”

2.2.2 Benefits of Block-Based Programming in Teaching Programming Concepts

Aside from the aspects that seek to entice younger programmers, block-based languages are powerful tools in teaching programming concepts, regardless of the programmers’ age. Matsuzawa et al. define the goal of introductory programming as developing problem solving, or computational thinking skills [20], rather than understanding the grammar/syntax of a specific programming language. They argue,
in concurrence with countless other authors, that the visual aspect of block-based programming allows new programmers to develop these skills, as they focus on solving the problem or task at hand, rather than the exact syntax required.

Studies deploying block-based programming amongst third level CS students, also demonstrate the benefits of block programming in improving students’ performance and understanding of high-level concepts even amongst older students [14]. The simplicity and powerful nature of block-based environments engage students, piquing their interest, but also allow them to focus on programming concepts and logic over syntax. Students found they could master programming concepts with increased ease, rather than when using Java, proving the benefits of block-based languages like Scratch amongst adults as well as children. Malan and Leitner’s study in higher education also goes further to argue that the large number of keywords and syntax in Java make it unsuitable as an initial teaching tool in third level CS classes, as it only serves to make students ‘masters of syntax’ rather than masters of programming concepts.

Given how many university courses successfully teach programming through object-oriented text languages like Python and Java, including the Computer Science course offered in Trinity College, this argument seems a little far-fetched. With proper pedagogical tools in university lectures that accompany the practical teachings in a programming language, this author believes it is reasonable to assume third level students can handle the burden of keywords in languages like Java and Python, while also gaining an understanding of programming concepts. Certainly, the author’s personal experience as an undergraduate student supports this belief.

Malan and Leitner’s claim does however seem more reasonable for a secondary school context where students are earlier in their overall education. These students will not have the same level of mathematical and logical skills as third level students, so using a block-based approach as the initial approach in secondary schools makes more sense.

It’s clear from the literature in this area that block-based programming languages have valuable traits that serve to entice new programmers, and ease their experience of learning to program, while also offering ways to impart programming concepts on novices without the heavy burden of syntax and keywords. Naturally, there are also downsides to block-based languages relating to lack of authenticity and power, and these will be discussed in the next section. The below subsection will first discuss some of the major block-based languages widely available.

2.2.3 Evaluation of Block-Based Programming Languages

There are a huge number of Block-based programming languages available, and that number is ever increasing as demand continues to grow for introductory programming tools. Scratch is easily the most popular tool, but there are a number of others worth mentioning. This subsection will give a brief description of each and why they are interesting.

**Scratch**

Scratch’s first prototype was developed in 2003 by MIT Media Lab [21] with the intention of teaching children to code via creative exploration. The environment was named after DJs who ‘scratch’ records on turntables to remix media and create new music. Scratch boasts over 93 million users, and over 80 million monthly site visits as of July 2022 [22], making it one of the most popular coding education tools in existence. The environment is divided into a staging area, block catalogue and coding area, and users have the ability to remix existing projects and create their own blocks, making it a very powerful, yet usable tool. A number of extensions further Scratch’s power including pen, music, Lego, Makey Makey and more.

**Alice**

Alice is a block-based programming environment developed in 1994 at the University of Virginia by Randy Pausch, now based at Carnegie Mellon University. It is similar to Scratch in that it seeks to ‘motivate learning through creative exploration’ [23]. On the other hand, it differs from Scratch in that
it allows programmers to create 3D animations and games. Alice aims to prepare students for object-oriented languages like C++ and Java, by imparting computational thinking skills and programming concepts [17].

**Snap!**
Snap! is a block-based programming language heavily based on Scratch, released in 2011, that provides more advanced features than Scratch [24]. When it was released, it had the benefit over Scratch of running entirely in the browser without any installation required. At the time Scratch didn’t have that perk. In addition to this, the Berkeley language has first-order lists and sprites, anonymous functions, and nestable sprites, amongst other perks. Interestingly, one of the creators, Jens Mönig, was previously a member of the Scratch development team.

**Blockly**
With MIT, Carnegie Mellon and Berkeley already in the game, Google entered the playing field of block-based programming in 2012 with Blockly [25]. Blockly differs from the other players in that it is a library for JavaScript, capable of producing code in virtually any language to develop block-based languages and environments. Blockly is now used in Scratch and MIT’s AppInventor for Android development [26], as well as other applications such as RoboBlockly [27] and Code.org [28].

**MakeBlock and mBots**
MakeBlock differs once more from the block-based languages like Scratch and co., in that it is concerned with the field of robotics. MakeBlock produces software based on Scratch for use on their educational robotics equipment. ‘mBots’ allow new programmers to gain programming experience and computational thinking skills, through building, adapting and operating simple Arduino based robots [29].

These are just some of the many block-based options available. These options all target slightly different approaches to teaching coding, but how do they compare to traditional text-based programming languages?
2.3 Block-Based Programming to Text-Based Programming
This section seeks to explore the transition from block-based programming to text-based programming. Not without its difficulties, this transition can be handled in a number of ways, but in any case, there is a strong motivation for novice programmers to make the jump in order to continue learning and developing their coding skills.

2.3.1 Motivation for Transitioning from Block-Based to Text-Based Programming
The primary motivation for transitioning to text-based languages is that novice programmers often hit a wall - if you’ll pardon the pun. There comes a time where users will have reached the limit on what’s possible in their chosen block-based environment, and will have learned everything there is to know. In this author’s personal experience from volunteering in a CoderDojo, there are often students who reach a point within Scratch where they could build programs with their eyes closed. Resnick et al., remind us that educational programming tools should have a ‘high ceiling’ [10], but the reality is that this ceiling, high as it may be, does exist and when a student reaches it, they should move on.

Block-based languages, while powerful in the early stages of learning, lose this power as the student becomes more familiar with the environment and programming concepts. Weintrop and Wilensky, in their survey of high school students, found students found block-based programming to be less powerful, and not suitable for writing complex programs [15]. They also found students felt that block-based programming starts to feel inauthentic and childish, which serves as a deterrent for continuing with their CS education. Authoring programs is slower than in text-based programming, and making small edits is a hassle of moving large blocks of code out of the way to access the relevant section of their code. Once students reach a certain level of computer literacy, block-based programming is simply no longer preferable.

As well as limits on flexibility and authenticity, blocks take up large amounts of screen real estate [30]. This screen real estate problem is one I will return to in the design section, when outlining the criteria for the design which is the basis of this project. These large blocks allow for quick programs authoring, which is advantageous in the early stages of learning where an environment must be enticing for a new programmer. But it also means, programmers fill up the screen quickly, often with badly written, or ‘clumsy’ code [18].

Perhaps, more worrying than students feeling restricted by a block-based language, or embarrassed by its childish nature, is the potential damage done to a student’s understanding of program behaviour and concepts. Earlier in section 1.1.3, the problem of balancing hidden complexities while not misleading the user was discussed. Block-based environments often fall foul of this balance, and the result is new programmers often begin their transition to text-based programming with an incorrect conceptual understanding of programming. Moors et al. highlight a significant weakness of block-based programming in that regard, in their 2018 paper [31]. They suggest such environments can have detrimental effects on new programmers by teaching them incorrect ideas of program behaviour. In their paper ‘Habits of Programming in Scratch’, Meerbaum-Salant et al., offer further poor habits they discovered amongst Scratch users [32]. Contrary to what is considered ‘good practice’ in programming, they argue Scratch promotes ‘bottom up’ development processes and ‘fine grained’ programming, leading to poorly structured programs, and a lack of awareness of concurrency. Powers et al. found the object model in Alice led to misconceptions down the road for novice programmers [17], and Parsons and Haden found novice programmers found it difficult to see a tangible connection between Alice and ‘real programming’ [33]. All of these findings beg the question: are these block-based languages over simplistic and misleading? Are they perhaps hiding too much, meaning the transition to text-based languages that much more difficult?

Recall, once again, the balance that any learning tool must strike between simplicity and accessibility for the sake of easing a learner into something, and building solid foundations for the future as the learner progresses. As Parsons and Haden put it:
“We must be careful that in our quest to make programming easier to teach, we do not wind up not teaching programming at all.”

This is of course an important argument, but is it a little too idealistic? After reviewing many of these block-based programming languages, it seems weighing them on the side of accessibility to get new programmers interested is more important than imparting a perfectly, accurate ideology of what programming is. This author believes it acceptable to put that burden on the transition from block-based to text-based programming, as long as adequate work is carried out to ensure as much support as possible is provided to novice programmers as they make that leap. As discussed in the coming sections, research in the area of supporting a user in transition is not only plentiful and varied, but still growing.

2.3.2 Transitional Difficulties
The difficulties in making the transition from block-based programming to text-based programming are extremely well documented in research literature including but not limited to, Kölling’s work on Frame based editing [36][9], Strong’s work on Pytch [34], Moors’ exploration of the transition from blocks to text [31], and Dorling’s paper on Scratch [37]. These papers have been selected for this review as between them they cover an almost exhaustive array of the types of transitional challenges experienced by novice programmers. But it’s worth noting there are many more papers available covering these challenges, and that research covering these challenges and proposed approaches is still growing.

Mechanics of Typing
While it may be something taken for granted when designing a professional programming environment, when designing an educational coding environment, it cannot be assumed that the user will be proficient at using a keyboard or even particularly computer literate. Thus, one of the first issues cited by Kölling when transitioning to a text-based environment from a drag-and-drop interface is typing [36]. Young children, and anyone new to coding, may struggle with the mechanics of typing, finding it difficult and time consuming to produce the text required to write full programs.

Increased Syntax Errors
In a text-based language, the programmer is now exposed to all the previously hidden complexities of the language’s syntax, and semantics. This means it’s almost certain there will be a sharp increase in syntax errors for the programmer as they grapple with syntax for the first time [36]. Additionally, novice programmers may struggle to understand the importance of syntax, e.g., a space after a bracket, or a semicolon at the end of a line, and will repeat the same mistakes over and over, without some pedagogical input teaching them about the importance of syntax.

Understanding Layout and Spacing
Novice programmers may struggle with managing the layout of their programs, as noted by Kölling [9] and Strong [34]. Where Scratch previously hid the importance of spacing and indentation from the user, they will now have to grapple with it in order to get their programs to compile, depending on the language. Programmers in text-based languages also have to manage program layout in order to know where to alter their program to further functionality or fix any bugs. In Scratch, this was straightforward as code is divided up into sprite coding areas or stage areas. In text-based programming, code ceases to be as visibly structured and becomes instead, a difficult-to-interpret ream of text.

Readability
As mentioned above, a user's program is just a large scroll of text in text-based languages which poses readability challenges for users, especially younger children [36].

Loss of Graphics and Visual cues
Text-based languages are inherently less graphical and visual than block-based languages [34]. This poses readability problems as described above, but it also means a lot of visual cues are lost. In block-based languages, block shape and colour offer the user insight into how they might behave or where/when they can be used, or even which other blocks they may be associated with. Plain text offers
no such clue to the user, and means this learning support is gone. Users must then remember what commands do, with only the textual clue of the name of the command, which increases the memory load.

**Memory**
Leading on from this memory of commands, it must be noted the number of commands increases significantly in the transition too. Kölling also points out that programmers in textual languages must now memorise syntax for their chosen language [36]. They must remember which brackets to use, where to indent or put spaces, whether or not to use capital letters or semicolons, and so on. This is a sharp increase in the cognitive load for novice programmers moving from block-based languages where a block catalogue promoted recognition over recall. It is also especially taxing for non-English speakers.

**Keywords and concepts**
Novice programmers transitioning from block-based languages to text-based languages, also have to grapple with the idea of keywords for the first time, learn how to use them, and remember them [34]. Words like public, class, private, static, this, final, boolean etc are all unknown to programmers transitioning from block-based, and act as major barriers to understanding for users.

**Error Messages**
The introduction of error messages in text-based languages poses another challenge, according to Kölling [36]. Programmers struggle with interpreting error messages correctly to make the necessary changes to their program when errors arise. Given the increase in syntax errors, this poses a significant challenge in the transition.

**Types**
Text-based languages employ the concept of types, which means programmers must learn to understand and use these during the transition [36].

**Function Prototype vs Definition**
In text-based programming, programmers must also understand the difference between prototype vs definition of a function [36].

**Matching Identifiers**
Programmers new to text-based programming struggle with matching up their identifiers [36], such as variable and function names. In block languages with drag-and-drop interfaces, users don’t have to name their functions and their variables are accessible in the block catalogue so they do not have to remember their variable names or type them correctly. In text-based languages they do, and this can cause a litany of issues.

**Writing Expressions**
In Scratch and other block-based programming environments, users are aided in the writing of expressions using expression blocks with visual cues and minimal typing. Writing expressions from Scratch proves to be difficult for novice programmers without the support of these blocks [36].

**Grouping**
In text-based programming languages, where functions are defined within explicit brackets, novice programmers struggle to correctly define functions [36] and grapple with syntax errors resulting from this.

**Loss of Confidence**
All of the above challenges in transitioning from blocks to text, result in more errors, more compilation failures and a loss of confidence for the programmer, as noted by Strong [34] and Moors [31]. This loss
of confidence must be mitigated with support for successful programming, so the programmer is not discouraged from continuing with their coding education.

Varying Needs
With any tool, language or environment varying needs can act as a challenge, and the transition to text-based language is no different. This is noted by Dorling et al. in their paper exploring Scratch [37]. As mentioned above, non-English speakers may also struggle more with the syntax of a language. This is not a challenge specifically targeted by this body of work, but it should always be kept in mind when designing any new tool.

With so many challenges to consider for programmers transitioning from blocks for the first time, how can we support them? What approaches exist for transitioning programmers to text? And what tactics do these approaches employ, that Pytch Junior may emulate in some feature, to scaffold transitioning programmers?

2.3.3 Approaches to The Transition
When teaching a child to read, it’s pretty typical to start off with a baby book, a book with one or two words on each page and plenty of pictures to serve as scaffolding. Then you move on to a children’s book, perhaps with a sentence or two per page and some pictures. Then maybe a book with no pictures, then a young adult book or two before you graduate to adult level fiction. Then, fully literate, the reader may venture into different genres, perhaps some non-fiction, maybe some newspapers. Some readers will stop there, some will only read magazines, some won’t read at all, but some will continue on to the next challenge. Some people will read the classics, and complicated poetry, or convoluted philosophical theories.

Teaching a child to code is no different. You start off small with blocks, and colour and visual cues and you build them up to text-based languages with no visual support at all. Some coders may stop there, some may go into robotics, some may build incredibly complex autonomous systems or artificial intelligence. No matter where the programmer ends up, their first transition is from their baby-book block language to their first text-based language. To get them on the road to their programming ‘Ulysses’, there are a variety of approaches.

Frame Based Editing
A hybrid environment is defined as a ‘blend’ of features from block-based and text-based environments [38], and they generally perform better than both block and text-based to support a programmer in transition. Frame based editing is a type of hybrid environment which maintains some graphical elements to maintain readability and minimise errors, but also introduces text editing which boosts flexibility and authenticity. Kölling and Brown introduce Stride in their paper [36], a Java-like language that is integrated in the Greenfoot development environment, aiming to be error prone and approachable. Novice programmers make use of ‘frames’ which are graphical boxes used to represent scope. In these frames, parts of statements are pre-written in Stride, and there is space for the programmer to type additional code to complete the statement. This eases programmers into the idea of typing out function headers, if-statements, etc., without overwhelming them initially. Stride also has a cheat sheet which mimics the Scratch blocks catalogue to help ease the burden of remembering all the new commands, key words, and identifiers. Stride can also be integrated into blueJ, an introductory environment for teaching beginner Java [9].

When evaluating Stride’s performance amongst students compared to ordinary Java, it was observed that there was no difference in the groups perceived frustration or satisfaction [40]. However, the authors did notice students using Stride progressed at a faster rate than the group using Java, and had fewer issues getting their code to compile, spending less time on syntax errors. This evaluation satisfied frame-based editing's aim of reducing syntax errors and frustration relating to dealing with them.
Strype is another frame-based editing tool for a more specific use case - programming the micro:bit [39]. The micro:bit is a low-cost platform designed for education, and acts as a visual shell around Python.

**Patch**

In the 2016 paper ‘From Scratch to Patch’, Robinson presents a modified version of Scratch for Python called ‘Patch’ [41]. Patch is aimed at supporting students in transition, by providing enhanced teaching support for computational thinking skills. These supports include user-defined functions with Python-like syntax, new iterator blocks, and editable pseudo-blocks to assist with algorithm development and promote good programming practice. Robinson argues that Python and Scratch have strong similarities, and Patch attempts to extend Scratch to Python leveraging the programmer’s existing knowledge of Scratch. He makes the case for Python as a pedagogical tool, citing the language’s simple syntax, widespread use and high ceiling of flexibility. Patch also boasts an integrated tracing tool, to aid novice programmers to understand how code executes, and it supports ‘assignment before use’ for its variables. These features therefore help to build strong foundations of Python principles, for the programmer’s future transition to normal Python.

**PencilCode**

Introduced by Bau in 2015, PencilCode is a block-based tool to help programmers starting off in text-based languages, specifically web languages like Html and JavaScript [42]. PencilCode has a toggle feature that allows a user to switch between block-based and text-based representations of their code, and utilises pseudo-indentation to resemble traditional programming languages. Bau argues that this toggle environment minimises frustration and enables beginners to access professional languages from day one, while not damaging their confidence.

**Droplet**

Droplet is a block-based editor for text code developed by Bau [16]. It works as a drag-and-drop editor for PencilCode, allowing a programmer to edit their code in a text programming language. Droplet loads the existing text program into editable block code, and then converts it back to text code once again. This is a valuable tool for programmers in transition to explore how block code translates in a text world.

**Karel Universe Drag-and-Drop**

This editor was created by Joe Bergin to be used with the Karel J Robot system, and offers students the chance to create programs using a drag-and-drop interface, to piece together blocks [48]. But what distinguishes it from block-based editing, is the blocks in this instance are fragments of correct code in Java. The code pieces are in correct syntax, with the option to simplify it slightly, meaning students don’t need to type at all, but are still exposed to a professional programming language’s syntax nonetheless.

**Jigsaw Puzzle Environments**

‘Jigsaw Puzzle’ programming environments are environments where a user can toggle between a text view and a ‘tiled’ view [43]. These environments are usually based on one special purpose language. Homer and Noble offer ‘Tiled Grace’ as an example of one such environment. In a study described in the paper, students noted features such as colour coding, the method tool box, and in-scope variable lists as being useful in the tiled (similar to blocks) view.

**Pedagogical Approach**

Dorling and White present an alternative approach to a new tool - a pedagogical approach [37]. In the specific context of transitioning from Logo to Python, they put forward the idea that this transition can be managed purely using pedagogical techniques including graphical language in conjunction with text.
They argue that research in this area of the transition is sparse, and more work is needed to embed good pedagogy in the transition process.

While an interesting idea, the authors here are only really considering situations where coding education is happening in a classroom of some description, with an instructor to deliver the pedagogical approach. However, so many novice programmers independently progress their coding education in their own time, and not considering this is a key limitation of Dorling and White’s paper. An environment that can support a learner with or without the outside influence of a third party, seems a more reasonable approach. This logic is presumably why there is such sparseness in purely pedagogical approaches to the transition, as cited by Dorling and White. Most researchers, like this author, don’t see it as a feasible option.

**Gradual Language**

One interesting approach to easing the transition from blocks to text, is the idea of a gradual language. Borrowed from the way natural language is taught, and based on Herman’s insertion that syntax errors are the most common errors when transitioning to text-based languages, Hedy is a gradual language that reveals more and more of its complexity and syntactic structure as a learner progresses [12]. Python based, Hedy, minimises syntax errors and the associated user frustration, as more and more of the learning scaffolding is removed over time. The idea being that the novice programmer is not overwhelmed at first, and eased into the programming language.

**GP**

GP, for general purpose, is another block language. It differs from the other approaches discussed in this section in that it doesn’t really attempt the transition at all [30]. While the creators did experiment with toggling from text to blocks, they ultimately decided it wasn’t necessary for the ‘casual programmer’ who they’re aiming for. They define this ‘casual programmer’ as teen to adult, and for the purpose of this project it can be considered as someone not interested in making the transition from blocks to text at all, and therefore not our target audience. GP is easy to use and debuggable, with more versatility and power than Scratch, eliminating the need for a transition. It achieves this with more blocks, keyboard-based block editing, and blocks that appear like text for a streamlined, more authentic UI. It also removes the blocks catalogue, saying experienced users find this method of block retrieval tedious. Mönig et al., are confident their environment is all the casual programmer would ever need.

This is at odds with Weintrop and Wilensky’s work outlined in section 2.3.1. They report student feedback that suggests that block-based languages are simply too childish, once a certain level of computer literacy is reached, regardless of desired program complexity. This author agrees that there is a point, even for a ‘casual programmer’, where blocks become too juvenile. The proposed changes in the paper by Mönig et al., are not sufficient to transform it into a block-based language that adults would feel is authentic, and not a tool for children.

Regardless of whether blocks can ever truly feel authentic, the creators of GP present their solution for the ‘casual programmer’, and this project is concerned with a more serious one who does wish to transition to text-based languages. The ideas and techniques from the above approaches have been reviewed, their successes and failures analysed to inform the design decisions taken in the course of this project. The design will be touched on in section 2.5, and will be described in further detail in section 3.
2.4 The Wonderful World of Text Editors

Before we continue on with the literature review of Pytch, the basis for this project, we must first take a pit stop into the world of text editors. One of the main aims of this project is to do something to reduce the large scroll of code text in Pytch in its current state, and reintroduce some visual structure to the code, to reduce the user’s cognitive load. To do this, it was important to research the ways in which existing text editors, outside the domain of computer science and programming environments, hide sections of irrelevant text at certain times to allow a user to focus on the relevant text.

First, let us establish a brief history of text editors providing something other than large scrolls of text that threaten to cause bouts of procrastination, in even the most dedicated users. Following that, a number of different text editor formats will be outlined, and the benefits and transferability of their techniques to this project's aim of structuring a user’s code editing experience.

2.4.1 A Brief History of Text Editing

Let’s begin with a definition of a text editor - it is simply a computer application that allows users to edit plain text. The earliest computers from Babbage’s difference engine in the 1820s to the ENIAC in 1946 were a bit busy carrying out calculations to do much else, so it comes as no surprise that text editor applications came much later, but perhaps not as late as one might think. The first text editor called ‘Text Editor and Corrector’ or TECO for short, was released in 1962 [44]. In what is now known as the ‘mother of all demos’ at the ACM/IEEE conference in 1968, Douglas Engelbart presented groundbreaking modern computing elements such as windows, internet video conferencing, and most famously the computer mouse - but for our interests, he also demonstrated an application capable of collaborative text editing. Following this in the 1970’s, computers started to become much more prevalent, especially in the business world - and so text editors did too.

2.4.2 What's Out There Now?

Text editor applications are a very simple premise, yet they can be so very divisive. Ask any developer their opinion on Vim or Nano, and they’re sure to have a strong opinion. Text editors vary in a number of factors including complexity, flexibility and support structures. Depending on what a user uses their editor for they want different things, and the range of uses for a text editor is enormous - so the types available are plentiful too. Characteristics of an editor like navigation, text organisation techniques, and view customisation are pivotal to a user's experience, so it is important to get them right. This project is particularly interested in how text can be organised within text editors, to reduce the scroll of text visible to a user at any given time. By reducing the amount of text visible to a user at a given time, the hope is they will be able to focus more easily on the task at hand, as well as having an understanding of their document’s structure - the document being code in this project’s case.

Folding Editors

These are editors with the capability to fold or minimise subsections of a large scroll of text, in order to focus on the section of the text scroll currently being worked on or read - a relatively straightforward concept. These can be plain text editors or code editors, and are often integrated into programming IDEs. Examples include Microsoft Word and IntelliJ.

Tabbing

Another approach is having an editor that is capable of having multiple editor tabs open at once, that can be toggled between, to select the piece of text currently of interest. This is suitable for text that naturally belongs in separate categories or documents, or in Object-oriented coding. Notepad++ is an example of an editor that uses tabs to keep track of several documents at once, that can be either plaintext or some programming language.
Outliner
Outliers are editors for creating tree structured text documents [46]. An outliner has the ability to collapse and expand the different layers of the tree, to display the desired text. In that regard it is similar to a more structured folding editor. Apple’s ‘Outline’ available on Mac OS is an example of outliner software.

Filtered Views
Not strictly in a text editing context, but email inboxes often have filtering capabilities to only display mail of interest. A similar technique could be applied in a text editor context to only display code or plaintext of interest. Most major email service providers, like Gmail, Outlook and iCloud, all have filtering capabilities in their inboxes.

Integrated Development Environments
These are editors specifically for editing code, as they contain features wholly unnecessary for text editing. IDEs like VS Code, Eclipse, IntelliJ etc, utilise a combination of various text editor techniques, like folding, tabbing, highlighting, filtering, and outlining. Pytch Junior falls into this category, so it too should combine a variety of text editing techniques.

Distraction Free Editors
Distraction free editors or full screen editors are editors that seek to solve the screen real estate problem and aid user focus by reducing other distracting elements on the screen besides text. By optionally hiding the toolbar, scroll bar or other icons, the screen can be totally filled by the document being edited. Examples include Vim and gedit. While an interesting concept, this is not really suitable for a learning environment, as the tools and icons around the text are important in supporting the novice programmer.

Screenplay Writing Tools
Writing screenplays couldn’t be further from coding in an educational programming environment, but the tools developed to aid screenwriters to produce complex scripts could lend innovative ideas to code editors. Countless screenplay writing softwares are on the market [46] and many offer automatic formatting, tagging and metadata extraction to automatically produce prop, setting and character lists, timelines, mind maps and more. In a way Pytch Junior will copy this behaviour by extracting code from the editor provided by the user, and producing Python code to feed to the Skulpt engine. More details of this will follow in section 4.

Aspects of all of these editor types and techniques provide potential uses in some domains, but for the design changes in Pytch to reduce the scroll of text, a combination of filtering and tabbing techniques seem the most suitable. Details of how these fit into the new version of Pytch will be discussed in section 3.
2.5 Pytch

While it’s important to gather background information about the various other research projects, and voices in the space of transitioning from blocks to text, this project revolves around one tool in particular, and how we may adapt it in its existing form to produce a new intermediatory step for novice programmers. The literature review conducted into these other avenues and ideas, as well as the motivations behind them have informed the design for these changes and provide justifications for the decisions and compromises made along the way, which will be discussed in the design section.

This section outlines the initial background information collected on the basis for this project - Pytch. The first subsection provides insight into Pytch and the idea behind it, as a transitional tool, followed by some analysis of its successes and where it has scope for increased learning support. This section will conclude with a brief introduction to the main event - Pytch Junior.

2.5.1 What is Pytch?

Pytch is in short, another Python-based approach to easing the transition from blocks to text. More specifically it seeks to embody the idea of ‘sprite-oriented programming’ in Python. Introduced by Strong and North [34][35], it differs from other ‘transitionary’ environments by focusing on the programming paradigm instead of the use of a text editor, and is based on the idea of using innovative text editing organisation to support the transition to a pure text editing environment.

Pytch utilises an Ace text editor and provides a graphical output, similar to Scratch, to provide users with immediate satisfaction without introducing complicated libraries. A Skulpt transpiler also means live coding is possible, as in Scratch, so users don’t get bogged down with the process of exporting Python code and running it externally. Pytch projects can however be exported in a special file format, just as in Scratch, so that students may share them for marking purposes or just for fun.

2.5.2 How Does Pytch Ease the Transition from Blocks to Text?

Relationship Between Environment and Language

By focusing on the programming paradigm, Pytch enables novice programmers to continue writing their programs in the same style as in Scratch, and this allows them to focus on building a program rather than the nuances of text editing and the mechanical act of typing. Many other similar environments, such as Stride or Alice discussed in section 2.3, require programs to be structured very differently than in Scratch. This increases the burden of transitioning on novice programmers, and discourages them from continuing their coding education. Pytch, however, takes note of the close relationship between environment and language that exists within Scratch, and adopts a similar approach - building an environment, closely tied to the Python language in order to scaffold the transition to the language.

Familiarity

The second way in which Pytch eases the transition is with familiarity - specifically, it’s very reminiscent of Scratch. The layout is very similar firstly, with the coding space on the left, and graphical tab and control buttons on the top right. Even something this simple, can ease the transition for new users, as they at least know where to look for the basics of running a ‘Hello, World!’ program.

Secondly, Pytch copies Scratch in terms of its ‘microworld’ of sprites, costumes and stages. Programmers follow similar processes to create their sprites, and select their appearance and sounds, so the user can more readily focus on writing the unfamiliar text code to control their sprites behaviour.

Ease of Use and Autonomy

Another simple thing Pytch does to ease the transition, is to be accessible by having a low ‘floor’. As discussed in a previous section, having a ‘low floor’ is a criterion for a coding education tool. Pytch is web based, and requires no installation or sign up. Tutorials allow users to easily build and run their first program in Pytch, and the graphical output provides an immediate sense of satisfaction for the user
and engages them. The learning materials mean novice programmers can operate autonomously to progress with their own projects and learning, without needing a teacher or tutor of some kind.

### 2.5.3 Where Can We Add More Support?

Moving to Pytch from Scratch has a number of gains, not least of which are increased authenticity and flexibility. Pytch is more powerful than Scratch. Programmers can do more with Pytch to push the boat out and increase their programming knowledge. Naturally, with these gains some learning supports are lost too: readability, visual cues, memory aids, minimised syntax errors, to mention but a few.

The idea for this project is to produce a new version of Pytch to lie between Scratch and Pytch, that mitigates some of these losses. Where Pytch focused on the programming paradigm in supporting the transition, Pytch Junior builds on that to add editing-based-support. There are four main areas where increased support may mitigate these losses.

**Code Organisation and Structure:**

At present in Pytch the code is presented as a large scroll of text, which does not preserve the readability of the easy-to-read blocks of Scratch with descriptor labels in clear, natural language, organised by sprite. To counteract this, code organisation by sprite could be re-introduced to encourage readability. Furthermore, breaking each sprite’s code into an array of event handler functions, takes away the challenge of mastering function grouping.

**Reduce Typing Requirements**

Typing requirements are the cause of many transitional challenges when moving from blocks to text. If typing requirements are reduced, we can also expect syntax errors, the time it takes to build programs, and expression writing mistakes to reduce. By changing the way, a user writes their code from a typing mechanism to some other format, for some parts of the program, we can also hide some of the confusing complexity and key words from our programmers without misleading them too much.

**Visual Cues**

Visual cues are rife in Scratch. Block shapes, colours, and shading all provide the user with information about the behaviour and type of each block. This is lost when these blocks are replaced with black and white text in Pytch. To reintroduce this, Pytch Junior could leverage colour, padding and other visual elements, to distinguish code belonging to different sprites, event handler heads from bodies, and variables from code text or key words. Spacing could also be used to indicate to the user, the different functionality groupings of buttons and controls within the environment.

**Memory Aids**

Despite the tool box available in Pytch, there is still scope to introduce more memory aids. One option could be a more intelligent autocomplete, to only provide suggestions within scope of the current cursor position. Visual cues, like the ones suggested in the previous paragraph, can also be used as memory aids by creating association between certain parts of the user’s program and the corresponding parts in Scratch. For example, as variable blocks are a dark orange shade in Scratch, perhaps some use of the same colour in this new version of Pytch may help the user remember that a certain piece of text is a variable.

### 2.5.4 Introducing Pytch Junior

The motivation for further work on Pytch lies with the general growing relevance of coding education outlined in this section, as well as the specific introduction of Computer Science to the Leaving Certificate curriculum. Computer Science at Leaving Certificate level doesn’t include object-oriented programming, but Pytch does expose classes to the novice programmer. Feedback from students and
teachers involved in Pytch user trials, cited this and other challenges relating to code management and understanding, as the issues they had with the environment. Thus, the motivation became clear for a project to develop a design for a more supportive version of Pytch.

Identifying the areas with scope for additional support to address these issues, is what led to the conception of Pytch Junior and the eventual design outlined in this paper. Recall the criteria outlined by Papert for a programming language, that they should have ‘low floors’ and ‘high ceilings’ [47]. This design wants to lower Pytch’s floor even more, while retaining its high ceilings.

Pytch Junior involves a major UI restructuring to reintroduce code organisation by sprite to boost readability and manageability, reduced typing requirements, visual cues and supports, and memory aids. It involves taking the research outlined in this section, and carefully applying design concepts and principles, as well as innovation in both block and transitional approaches to programming, to the initial idea to produce a thorough, informed design. Additionally, this paper provides considerations for how this can be achieved technically, and a means of evaluating the success of the design and this project's body of work.
3 Design

The goal of the research undertaken in section 2, was to establish the relevance of coding education tools by exploring the increasing prevalence of coding education in school curriculums, a state of the art in block-based programming and the approach to transitioning to text-based programming. The review also explored innovations in text editors, and the Pytch editor in its current state, its successes and failures in supporting a novice programmer in transition - setting up the motivation for a new solution based on Pytch that increases support in key areas to ease the burden of transition.

The goal in this section is to show how this research is rigorously applied to produce a thorough, informed design for a programming environment. An environment that is a novel solution to the transition challenges unearthed by the research chapter, not resolved by the original Pytch editor. It is neither a block-based, text-based or even frame-based environment, but borrows elements from various research from section 2 as well as retaining some of the existing Pytch features to find a novel design.

To meticulously outline both the process of constructing this design, and the justifications and research basis for every design decision, this section is split into three chronological sections corresponding to the 3 design iterations in phase 2 of the DBR approach. The first section (3.1) describes the initial stages for the design, the original idea from the Pytch team and the identification of the main goals of a new environment. The second section (3.2) documents the second stage of the design process - producing a general UI skeleton to satisfy the restructuring of code management. Stage three began with the UI skeleton and worked through all the finer features, hammering out the details required for a thorough design of the user interface and user processes. Thus, section 3.3 presents the full design with justifications and alternative considerations for every chosen feature.

As outlined in the opening to the literature review section, each stage of design iteration also corresponds with an intertwined iteration of literature review, satisfying the design-based research collective’s criteria that development and research take place through continuous cycles of design, enactment, analysis, and redesign [53]. The literature review first began by establishing the relevance of coding education, and motivation for continuing the research on Pytch in the form of feedback collected during user trials. The second iteration after establishing the research question in collaboration with researchers and practitioners, reviewed literature to identify key areas in Pytch that have scope for added support based on the feedback from the user trials. Stage one of the design, described in the following section, follows this iteration of the literature review.

3.1 Stage 1: Conception of Pytch Junior and Identifying Solid Design Goals

The Pytch team have been considering the idea of a ‘junior’ version of their full editor for some time. In testing sessions with students, they noted the struggles students had with code management and readability, due to the large scroll of text (which corresponds with issues cited in Kölling’s work [36]). Students often struggle with knowing where in their program to make changes when the need arises, and lose track of syntax requirements like closing brackets, semicolons and indentation. These issues with grouping and the increased typing requirements caused increased syntax errors, and difficulties writing expressions. Students also reported confusion over the introduction of key words and classes, particularly understanding the structure of functions within classes. These are noted by Strong and North [34] and discussed in section 2.3.2. Recall all of these issues are also well documented in Kölling’s work on transitional challenges, and are discussed in the same section. Strong and North also cite loss of graphical elements and visual cues as transitional challenges in their paper. Weintrop and Wilensky back up this claim as they cite visual shape and colour of blocks as major benefits of Scratch in their study of high school students [15], discussed in section 2.2.1. While Pytch maintains a graphical output as in Scratch and other block-based languages, in its current state it offers little in the way of the visual cues. Therefore, there was also a motivated interest in exploring the reintroduction of visual cues in a new design.
Of course, Pytch offers many benefits in perceived authenticity and flexibility, as discussed in section 2.5.2, which should and will be preserved where possible. However, this feedback from both teachers and students provides the motivation for another version of Pytch, to be placed in between block-based programming and the current version of Pytch in a novice programmers educational timeline.

### 3.1.1 Conception and the First sketch

When this author began working on this project, Pytch Junior existed only as a piece of concept art in the form of a gif, as seen in figure 2 and 3 below.

![Figure 2: A Pytch class named Banana with some basic functions.](image)

![Figure 3: The same Banana class organised into a new sprite-organised structure.](image)

Figure 2 shows code from the current version of Pytch, specifically a class named ‘Banana’. The banana class consists of a number of functions, all written in a very Python-like syntax. This figure shows the issue of the scroll of text for novice programmers. To experienced programmers, it looks very straightforward. The indentation and spacing clearly indicate function headers from their bodies, and separate functions from each other. Colons and further indentation make iterative loops and if statements readable, and key words like ‘self’, ‘class’, and ‘while’ are all familiar. However, Pytch is not for experienced text programmers, it’s aimed at students moving on from Scratch or other block-based languages. These novice programmers are used to blocks that are divided up by sprite instead of class. They haven’t used indentation or any real text syntax before, or worked with key words. Their iterative loops and decision making came in the form of a special block shape they selected from a catalogue, so indentation and colons mean nothing to them. All they see is a large scroll of text that is largely unfamiliar to them. Putting yourself in the shoes of a novice programmer faced with the contents of figure 2 for the first time, it’s easy to see how they would fail to see the connection between block-based and text-based languages, as pointed out by Parsons and Haden [33] in section 2.3.1.

The first idea for Pytch Junior is seen in figure 3. This figure shows the same piece of code, the banana class, reimagined in a new structure. Here the concept of a class is hidden from the novice programmer by organising code by sprite, as it is done in Scratch. By hiding the concept of classes and instead preserving an aspect of a familiar language, it lowers the barrier to entry in Pytch. It also addresses some of the feedback provided from the testing sessions. The issue of large text scrolls is partially addressed by this division into sprite tabs, but also further tackled by splitting the code for a sprite into a number of boxes - each holding an event handler function. This reintroduction of more rigid code structure mimics the rigidity of blocks in Scratch, with the hope that familiarity fosters confidence.
This gif however, is just that - a gif. More work was needed to research and expand the idea, to establish whether it would address the issues students were reporting, but also to identify ways in which other projects support students in transition to adopt in this project too.

3.1.2 Research and Identifying Priority Features

The aim of this project was to take this preliminary idea and transform it into a comprehensive, informed design, and to do this a research review must be carried out. This review identified four main areas within Pytch with scope for additional support in section 2.5.3. These areas now serve as priority features for Pytch Junior. They are described briefly here, but will be fleshed out further over the course of this section.

**Code Organisation by Sprite**

This is the primary feature of Pytch Junior, as shown in the preliminary UI gif in figure 3. It tackles the reported issues of readability, program management and grouping, and has the added benefit of concealing the concept of classes from novice programmers which lowers the floor for getting started in Pytch Junior. Additionally, as discussed in section 2.1.1, the Leaving Certificate curriculum doesn’t include Object-oriented programming or the concept of classes. Therefore, to allow Pytch to be used in the classroom as part of computer science Leaving Certificate classes, it makes sense to have a version of Pytch that conceals classes from learners.

**Reducing Typing Requirements in Function Declarations**

Another major feature of Pytch Junior is to reduce the amount of typing required by programmers. The mechanics of typing are listed as a common transitional challenge [36], discussed in section 2.3.2, and also as a reported challenge for Pytch users. Expression writing and syntax issues are amongst the problems, Pytch Junior seeks to minimise by changing the way functions are declared and defined.

**Visual Cues**

While Pytch retains the benefits of the graphical elements of block-based languages, with its graphical output, it doesn’t boast any of Scratch’s visual cues. Pytch Junior seeks to use colour, highlighting, and other techniques to prompt the user on programming concepts and behaviour.

**Memory Aid**

Increasingly taxing a learner’s memory capacity is cited by Kölling [36] in section 2.3.2 as one of the largest transitional challenges in blocks to text. Users lose the block catalogue of Scratch, which Weintrop and Wilensky [15] praised in their work, mentioned in section 2.2.1, and struggle to remember what commands are available and how to use them, as well as the burden of remembering syntax. To attempt to tackle this, an autofill feature is identified that only offers suggestions relevant to the current sprite.

These features are all important, but the largest change to Pytch is the major overhaul of the UI to change how code is managed. Therefore, the next stage of the design process was devoted to exploring various UI layouts in a number of hand drawn sketches and producing the final UI skeleton, into which the finer features will sit.

Following the first stage of the design process, another literature review iteration was conducted as part of DBR phase 2, to identify design principles and technological innovations from block-based languages, transitional environments and text editors that could inform the basic layout and interface of Pytch Junior. The focus in this literature review iteration, and in stage two of the design process was on code organisation by sprite, how this is achieved in other environments, its benefits, and how it may be achieved in the design of Pytch Junior.
3.2 Stage 2: Preliminary Environment Sketches

The way the Pytch Junior’s interface is laid out, dictates the way code is managed in the environment. The first feature of Pytch Junior is reintroducing ‘sprite-organised’ code, and thus the interface structure is of the utmost importance. The layout must reflect the underlying structure of the user’s program as a group of sprites and a stage, each with a number of event handler functions, each with a head and body. If the interface is rigidly structured to reflect this program composition, then the hope is it will aid novice programmers in understanding not only program structure and concepts, but also where and how to make changes to their code.

In the second iteration of the design process, time was spent exploring different options for this layout, weighing up the pros and cons of each one in how it supports the motivation behind organising the code by sprite - namely easing readability and the process of code management. A number of hand drawn sketches were produced during this stage. For the sake of readability, these sketches have been reproduced in Windows whiteboard from their original paper-and-pen format.

Figure 4 shows the first sketch created during this process, displaying the three main areas in Pytch Junior - coding and assets, graphical output and textual output including errors. In this layout the graphical output is to the right and the coding area to the left for the sake of familiarity. The buttons for running the program, as well as saving and exiting are on top of the graphics, once again, in Pytch as in Scratch. This familiar layout is listed as one of the benefits to Pytch in section 2.5.2, so it makes sense to retain it in Pytch Junior also.
The text output runs underneath the other two areas along the bottom of the screen. This is how it is displayed in Pytch in its current form. When creating this sketch, it was considered placing the textual output and error output elsewhere, to allow the coding area more screen space to ease the text scrolling issue. However, the vast majority of text programming editors and environments have the text output in the same bottom position. As the goal for both Pytch and Pytch Junior is to progress the programmer to such editors, it was deemed sensible to leave the text output in its original place.

Figure 5: First sketch on how to structure the coding and assets area of the Pytch Junior interface.

With the three main zones designated, the second sketch in figure 5 begins to experiment with how best to layout the coding and assets area of the interface to reflect code organisation by sprite. This first idea involves placing the sprite tabs (as seen in the preliminary idea gif in figures 2 and 3), along the top border of the area with the project's assets in the space to the left of the actual code text. By placing the sprite tabs along the top of the screen, instead of down the side as in figure 3, the idea is to mimic tabs in web browsers which the novice programmer may be familiar with, so that they may intuitively understand how to access each sprite.

The aim with this sketch was to maximise the space for the actual code. By keeping the assets off to the left in a smaller space, it aims to get the programmer to focus on writing the code itself, which we know to be the most challenging part for them, and not get distracted by the costumes or sounds available in the project. This sketch also includes a bolder outline to offer a visual prompt to the programmer of which of their sprites they’re currently editing.

However, with this sketch, there was a concern that having the project assets visible on the side, even in a smaller pane, offered too much distraction for the programmer, so another sketch was created to explore how to hide the assets while still having them easily accessible.
This sketch in figure 6 replicates the Pytch layout of having project assets, in the same area as textual output and errors, effectively separating it from the code entirely. Instead of being the first tab, clearly visible (and potentially distracting) to the user, in this sketch it is the third tab. This sketch also demonstrates the added benefit of increasing the screen space for the coding area.

However, at this point, more concerns arose. If the project assets are separated from the coding area and placed alongside text output, is that visual clue - via proximity - giving the wrong impression to the novice programmer on what an asset is and its relation to code?

Moreover, the concept of an assets tab that houses those belonging to the project as a whole, regardless of what sprite uses them, may undermine the rigid organisation-by-sprite Pytch Junior is seeking to convey. Granted, many professional programming frameworks do have a global assets folder, which any class can use, for Pytch Junior, this author believes it’s more suitable to keep both code and assets organised by sprite. This eases the process of managing the program for a user, and also mimics the setup in Scratch where costumes and sounds are sprite specific, thus allowing the programmer to focus on the task of writing the code itself.

In order to organise assets by sprite, the next sketch moves them back into the coding area.
In this sketch in figure 7 the assets are back in the coding area, but this time split into tabs alongside code as costumes and sounds, mimicking the layout in Scratch that users will be familiar with. This sketch is a lot closer to what Pytch Junior seeks to do, with code, costumes and sounds clearly organised by sprite where a user can see which sprite is active, and whether they are looking at code for that sprite or its assets. It also prioritises code over costumes and sounds, to focus the user on the primary task.

One last tweak to the coding area layout, its orientation to be specific, was decided upon at this stage. The elements that make up a sprite are fixed in number- code, costumes and sounds - but the number of sprites in a user’s program is unlimited. This means it’s very likely that as a program grows, the user will need some sort of mechanism to navigate through their sprites. A scroll bar is the most sensible mechanism for this, but in the current sketch this means scrolling horizontally through the sprites along the top axis. This seems acceptable enough, but novice programmers are more used to a vertical scroll bar, as seen in the Scratch block catalogue and also on most web pages and applications. It is only a small change, but swapping the orientation of the coding area tabs, and thus returning to the layout in the original figure 3 in terms of the sprite’s location, seems the more sensible thing to do.
Having explored other options for how to structure the coding area, this sketch in figure 8 arrives upon the final layout for the sprite and asset tabs. The next sketch in figure 9 explores how to structure the internal section of a sprite’s code tab.
Figure 9 shows the internals of a sprites code as a series of boxes, each containing an event handler function. Each event handler is composed of a head and a body. The body will be the only typing required for the user within Pytch Junior. Both the body and header will be discussed in further detail in section 3.3, but one further sketch is required for the coding window.

This sketch is concerned with variables for a sprite. While the variables could be declared, initialised and used within the code text of the event handler functions, the author believed more could be done to support the programmer in transitioning to Python syntax. In Scratch a programmer can create a variable without initialising it. The same cannot be done in Python, and the same follows true in Pytch. But for programmers moving from Scratch into Pytch, this will be new to them. To support them, Pytch Junior borrows an idea discussed in section 2.3.3 from Robinson in his development of Patch [41]. Pytch Junior should force programmers to declare and initialise their variables in one step to instil the practice in them and to avoid programmers encountering issues creating variables in text programming within function bodies. To do this a new sketch (figure 10) was produced with a simple box at the top of the code window to house and create new variables.
This sketch uses some basic colour cues to distinguish the variables box from the function boxes, to ensure the programmer understands the different nature of the two types of boxes.

With the internal coding window layout now described in figure 10, figure 11 describes the full layout to support code organisation by sprite.
This is the final sketch from stage two, and describes the full layout of the interface design. It omits the finer details of buttons, scroll mechanisms, user flow and processes which will be described and defined in stage three.

Following stage two, a further literature review iteration was conducted with specific focus on visual cues employed by both block-based and transitional technological innovations, as well as innovations to reduce typing requirements and support memory load.
3.3 Stage 3: A Comprehensive Environment Design Justified by Research

With the layout determined and a UI skeleton produced in stage two, stage three’s undertaking was to produce the final design, complete with fine grained details like button placings, colour and design choices, and user flow and processes for adding sprites, variables, functions and assets. The full user interface mock-up can be seen in figure 12 below:

![Figure 12: The full interface mock-up for Pytch Junior demonstrating a typical programmers view for a 'Hello, World!' program.](image)

This figure shows what Pytch Junior may look like running a ‘Hello World’ type program with a few sprites and a stage. As described in section 3.2, the graphics output sits on the right-hand side, with the coding section on the left and the textual output along the bottom.

To break down this design let us return to the areas mentioned in section 2.5.3, where we could add increased learner support for transitional challenges. These areas inform the features of the design outlined at the beginning of this section, and we can use these to dissect the design in figure 3.11. For each area, this section will discuss the aspects of the interface that support that area, in addition to research-based justifications behind design decisions.

3.3.1 Code Organisation by Sprite

*Why Organise by Sprite?*

Perhaps the most evident element of the full user interface design as shown in figure 3.11, is the element of code organisation by sprite. To explain the introduction of this organisational structure, first recall Kölling’s argument in two separate works [9][36], that managing manual layout is a big problem in the transition from blocks to text. It involves understanding classes, extensive amounts of syntax, function grouping and indentation. These are skills that simply cannot be assumed or expected of novice programmers leaving blocks behind them for the first time. Kölling offers one way to address this with his proposal for frame-based editing, as described in section 2.3.3, as do many other authors and researchers. But as Pytch Junior is intended to fall between Scratch and Pytch on a young programmer’s
educational timeline, perhaps we can look back to Scratch for inspiration on how to handle the code management issue?

Once more recall the research outlined in section 2, and specifically Hansen's work on ‘Interactive Design by Children’ [19]. Section 2.2.1 notes Hansen's argument that block-based environments are successful in reducing cognitive loads, but here it’s worth noting that Hansen goes one step further and says this is because of organisation by sprite. This argument forms part of the reasoning behind employing code organisation by sprite in Pytch Junior. The other reasoning for organisation by sprite for structuring the UI, is simply familiarity. Preserving code organisation by sprite in Pytch Junior, means at least parts of the programming process are familiar to users, and this in turn fosters confidence which we know from section 2.3.2 is one of the major transitional challenges in blocks to text.

The UI as seen in figure 12 also addresses the transitional challenge of readability raised by Kölling in section 2.3.2, by separating the old scroll of code text into boxes representing functions, which in turn are housed inside tabs separating code by sprite. By moving from a largely visually unstructured reel of text, to a highly structured representation of a program, it enhances code readability and manageability.

Code organisation by sprite also has the added benefit of concealing some of the complexity of a text-based language like Python from the novice programmer. Return momentarily to Malan’s claim as discussed in section 2.2.2, that Java’s volume of keywords makes it unsuitable for introductory courses at undergraduate level. While Python only has 33, compared to Java’s 50 reserved terms, that’s still a lot of words to learn for the novice programmer. Especially when you consider they didn’t have any to memorise in Scratch. If Malan can argue that 50 is too many for college level programmers, though this author may disagree with that claim, it seems reasonable to say 33 might be too many for secondary school students. Adopting Malan’s philosophy of reducing the number of keywords exposed to novice programmers, ‘organisation by sprite’ offers the additional benefit in hiding classes (which recall from section 2.1.1 aren’t on the Leaving Certificate curriculum) from the novice programmer. Instead of grappling with classes, they can focus on learning to build functions using text for the first time.

**How To Visually Represent Code Organisation by Sprite?**

The decision to separate the sprites codes and assets by utilising tabs, was informed in part by the research into text editors outlined in section 2.4, in particular tabbed text editors. This seemed the most natural choice for clean division between sprites, while ensuring users can see how they access their other sprites. Another factor was considering how tabs within IDE’s are often used when developing more elaborate systems with numerous classes and interfaces working in tandem. In Pytch Junior, while the programs being designed and built are simpler, it’s still a number of class-like objects interacting within an overall larger project, so it is a suitable mechanism for structuring the interface. These sprite tabs essentially allow a user to filter their code by sprite, a concept also borrowed from section 2.4.

Figure 13 and 14 shows the sprite tabs in Pytch Junior, compared to a number of tabs holding different classes in Visual Studio code. These figures demonstrate how, when a programmer moves to writing purely python code in an IDE in the future, they may see the link between sprite-based coding and the concept of classes. By drawing this link, Pytch Junior’s design aims at building solid foundations for the programmer’s future in a text language.
Adding More Sprites
The final design also includes the user process of how a user goes about adding a new sprite to their project. Figure 15 below shows this process:
The ‘Add Sprite’ button is positioned just below the graphical output box for two simple reasons. It’s visible there and it’s in a similar position to the button for adding a new sprite in Scratch. Once again, this fosters familiarity within the user, which in turn makes navigating the interface easier for them.

When the user hits the button, they’re faced with a simple pop up that only asks them one thing: to name the sprite. The name they choose is more important than in Scratch, where a drop-down menu within blocks reminds users of what sprites they have when they want to reference one sprite, from code in another. In Pytch Junior, they must reference other sprites using text. To help programmers remember their sprite’s names, the names are displayed below the image of their sprite in the tabs to the left, and also on the variables box within each sprites code tab. Furthermore, as sensible naming conventions are considered good programming practice, a small reminder in the ‘Add Sprite’ pop up box, helps to reinforce this practice in Pytch Junior users. In addition to its being good practice, allowing users to rename their sprites over time would require complicated code refactoring due to its textual nature. This refactoring has been ruled out of scope of this project as it’s not strictly relevant to the realm of coding education, so the name chosen by the programmer when they create the sprite is the one, they’re stuck with. This is a limitation of the environment, as a programmer can rename a function in Python any time they like, and use the ‘find and replace’ function in their chosen editor to do the refactoring necessary. But by disallowing sprite renaming, it forces the programmer to think about what they want their sprite to represent at the time of creation, and discourages them from creating a great many sprites, just for the hell of it.

In this design shown in figure 15, users are not asked to upload a costume upon creation, and instead a default costume is initially used to represent that sprite. This decision arises from the author's personal experience volunteering in CoderDojo. The author appreciates that when asking a novice programmer to select an image for a new sprite, especially when there isn’t a catalogue to choose from as in Scratch, you run the risk of distracting the programmer and sending them down the rabbit hole of browsing images on their local machine and the internet. It’s reasonable to expect Pytch Junior to be relatively challenging for new users, and this author has experienced Dojo students who either purposely or not, become paralysed by making the choice of selecting a costume, and procrastinating the actual coding activity. To avoid this, Pytch Junior uses a default costume initially, so the user isn’t distracted.

As more and more sprites are added to the project, the sprites tab list becomes vertically scrollable, via simple, intuitive arrow buttons.
The Assets Tabs
When a user does want to add a costume or sound, they access the relevant tab at the top of the coding window. The mock-up of what these tabs look like is shown in figure 16 below:

![UI mock-up showing the costumes page for the 'elephant' sprite.](image)

In this figure 16, you can see the default sprite costume on the far right, as well as the other costumes this particular user has uploaded for the ‘elephant’ sprite. This list, like the sprite list, is vertically scrollable and the button for adding a new costume is in the bottom right of the window. A simple plus sign was used, as a universal sign for adding a new item. The first costume uploaded by the user relaces the default image in the tabs to the left. Subsequent costume uploads do not relace this icon again, for the sake of consistency in the environment.

The mock-up of what happens when a user clicks the button to add a new costume is shown in figure 17 below:
Figure 17: UI mock-up of the process to upload a new costume.

Like the process for adding a new sprite, the user is once again presented with a simple pop up. In this instance the user is asked to select a file from their local machine to upload. The sounds tab and process for adding a new sound for a sprite is identical, except the user uploads an mp3 file instead of a PNG file, and there is no default sound file.

The Stage
While sprites are the primary feature of Pytch Junior, the stage is also important. The stage doesn’t exist within Pytch, but is included in Pytch Junior as a way of providing pseudo global variables and functionality. As Pytch Junior will be used to make things like games, it’s highly unlikely that users won’t at least require global variables for things like ‘scores’ or ‘lives’. To be able to provide this functionality, a number of approaches were considered including a separate global scope tab somewhere in the UI or some tab specifically for global variables. The idea of a global tab of some kind also raised the question of the viability of allowing Pytch Junior users to create non-event handling functions in this global space. This would increase the creative opportunities in Pytch Junior, but it was ultimately decided these approaches were too convoluted, and would overcomplicate the structure of the interface. While there is value in allowing non-event-based functionality, it is reasonable to expect that once a programmer is ready to create such functionality, they are ready to move on to the current version of Pytch. The value of non-event-based functionality does mean its absence is a limitation of this design, and this is addressed in later sections in this paper.

As previously mentioned, Pytch Junior is closer to Scratch than Python, which justifies its retaining of the idea of the stage as a workaround for pseudo-globality. In addition to pseudo-globality, the stage offers users a way to change the background of the graphics window. The stage in Pytch Junior can be seen in figure 18 below:
The stage is almost identical in layout to the sprite tabs in terms of the processes of adding variables, function, costumes and sounds. Its distinguishing colour will be discussed in section 3.3.3. The stage also differs in that it cannot be duplicated, and it is always in the environment. This means it is visible when the project is first created, though it will not contain any code, costumes or sprites. It can be found at the bottom of the sprites list, as it is the secondary feature below sprites. This means once there are three or more sprites in the project, it is hidden from the user’s view. This is not ideal but as it is visible when the project is first started, it seems adequate for the sake of not crowding the interface with a separate window for it.

3.3.2 Reducing Typing Requirements
As discussed in section 2.3.3, Herman’s paper on gradual language cited syntax errors as the most common errors experienced by programmers transitioning from blocks to text. These errors arise from the increased typing requirements, so the simplest way to reduce them is to find a way to reduce the typing requirements. In addition to addressing increased syntax errors, section 2.3.2 also cites Kölling’s inclusion of the slower nature of mechanical typing for beginner programmers in a list of transitional issues. These two papers therefore inspire Pytch Junior’s goal of reducing the amount of typing required to build programs. This is achieved in a number of ways.

Event Handler
First, recall figure 12 and the internal structure of a sprites coding tab. The sprites code is divided into two types of boxes: the variables box, and function boxes. As only event-based functionality is permitted in Pytch Junior, these functions can also be called event handlers. PencilCode [42], discussed as one of the approaches to the transition from blocks to text in section 2.3.3, utilises a technique of having the environment's code visually resemble the structure of real code. A similar technique is adopted here with the event handler boxes intended to convey the idea that a sprite comprises a number of separate functions and some variables or attributes. This is somewhat similar to the notion of a class, which they’ll encounter when they progress to Pytch, so it’s a helpful foundation to give them.
Another part of the reasoning behind putting event handlers into boxes is based on the challenge of function grouping, raised by Kölling [36] and discussed in section 2.3.2. Novice programmers often struggle with the brackets or indentation wrapping functions, depending on the text language they’re attempting to master. By placing functions neatly in boxes, it alleviates the programmers struggle with indentation, and allows them to focus on the content of their functions instead of getting lost in the semantics.

**Adding a new Event Handler**

Figure 19 below shows the process of adding a new function to a sprite.

![Figure 19: Process of adding a new event handler function, using a drop-down menu to select the event.](image)

Figure 19 shows a drop-down menu that is used to select from a list of events, which event to base the functions execution on. The addition of a drop-down menu in figure 19 from figure 3, means the user doesn’t have to type the entire function head which borrows the idea from Kölling’s frame-based editing approach [36] discussed in section 2.3.3, of having to partially fill in a statement. Recall from the discussion on Kölling’s frame-based editor, it aims to be error resistant and approachable, which corresponds to Pytch Junior’s goals as an editor too. In the review of frame-based editing [40], also discussed in section 2.3.3, it’s shown Kölling’s editor is successful in reducing syntax errors and thus it makes sense to adopt a similar approach in Pytch Junior.

The drop-down menu method of writing function heads, also has the benefit of hiding more complexity from the novice programmer. Once again Malan’s argument against keywords for introductory courses acts as justification for the drop-down menu as it conceals the complexity of a Python function definition including keywords like ‘class’ and ‘def’. It doesn’t eliminate all keywords, which would be potentially dangerously over simplistic in any case, but it does simplify the process of creating functions.
In Pytch and indeed in Python, functions require names, so it was considered whether programmers in Pytch Junior should have to do the same. For the sake of simplicity, it was decided the function names could be auto generated sensibly in the background when feeding the users program to the Skulpt compiler (more details to follow in the technology section). In the same way that asking a user to upload a sprite costume upon creation may cause them to be sent down a rabbit hole, asking a user to name a function when they may not have thought about what their function will do is an unnecessary distraction. Especially when functions cannot be renamed, so it would likely result in programmers having poorly named functions, violating good programming practice.

Writing the Event Handler Body

With the event handler header taken care of, let’s turn our attention to the body. In Pytch Junior following on again from Malan’s advice for reducing keywords, the ‘import’ statement required in Pytch is hidden. This is added retroactively by the system when feeding the code to the Skulpt engine. However, as I mentioned in the above subsection, hiding all the keywords and complexity, is potentially dangerously over simplistic. It misleads novice programmers about the language they are seeking to master in the long term (Python in the case of Pytch Junior). Even if calling it dangerously over simplistic is an overstatement, at best it means the student will have to grapple with the same complexity at a later date when they progress to Pytch. Given the step-up they’ll already be expected to take when moving onto Pytch to understand classes and pure text code, it’s unreasonable to also postpone the challenge of understanding at least some keywords in Pytch Junior. With that in mind, it’s reasonably justified that the novice programmer is expected to grapple with keywords in the body of the function such as ‘self’, ‘while’, ‘for’, etc. These keywords, in addition to the fact that the function bodies are entirely text-based, ensure an element of authenticity remains in Pytch Junior which is part of the aims outlined in the introductory section to this paper.

Variables

In addition to altering the way functions are created and defined in Pytch Junior to reduce the amount of typing required, the way variables are declared and initialised has also been specially designed. This process is shown in figure 20 below:
In Pytch, as in Python, creating and initialising a new variable is simply a case of typing something like:

\[
myVariable = 0
\]

In Pytch Junior however, this process has been extracted from function bodies and designed to be an interaction with the GUI rather than a typing process. This is done on purpose to help alleviate the struggles with matching identifiers, as described in section 2.3.2. By having the variables separate from the code, and the GUI interaction for adding a new one, it reminds the programmer of the names they have assigned to their variables when they go to use them in their code. This should reduce the time they spend scrolling through code trying to remember the exact name and value they assigned to their variables. It also instils the Python requirement of declaring and initialising in one step, adding another element to the Pytch Junior design that prepares the foundation for a Python future. This ‘assignment before use’ idea is also used in the Patch editor [41], described in section 2.3.3. An assignment before use policy would be difficult to enforce in Pytch Junior without this variable box - especially as programmers coming from Scratch are unused to assigning a value to their variable when they create it. The policy is enforced by the GUI interaction for adding a new variable to the box, which requires both a name and initial value. Thus, the need for some technical mechanism prohibiting the function body from referencing an uninitialized variable is not required.

Initially it was considered that these variable declarations and assignments could slot into a ‘when green flag clicked’ event handler that is automatically generated upon sprite creation. However, the issues of matching identifiers as well as enforcing the ‘assignment before use’ policy of Python serve as enough justification for keeping the variables entirely separate.

Variables are all instance-level within sprites in Pytch Junior. The absence of class-level variables is perhaps a limitation of the design, but class-level (or ‘sprite-level’ variables to be more precise) were considered and deemed too complicated for novice programmers using the environment to understand given the concept of classes have been concealed from them by the code organisation by sprite.
3.3.3 Visual Cues

Scratch, and other block-based environments, are rich with visual cues. Inversely, text-based languages like Python have zero visual cues, regardless of what editor a programmer chooses. While experienced and professional programmers may find visual cues unnecessary at best, and irritating at worst, novice programmers are proven to find them beneficial. As outlined in section 2.2.1 in the discussion of Weintrop and Wilensky’s study of high school students [15], visual cues, derived from block shape, colour, size and proximity to other blocks in the catalogue, are identified as being helpful to the coding process by the students. Participants in a study reported by Homer and Noble [43], from section 2.3.3, also cited colour as being helpful in understanding how to use their ‘jigsaw’ environment. Both of these studies provide ample reason to think introducing visual cues to Pytch Junior may also offer benefits in supporting the novice programmer.

To review the visual cues included in the Pytch Junior design, let’s first remind ourselves of the UI design in figure 21 and 22:
Figure 21: A reminder of the Pytch Junior UI design.

Figure 22: A reminder of the Pytch Junior UI design - including the stage.
**Colour**

Just as Scratch utilises colour to create association of type and behaviour, so does Pytch Junior. All buttons and controls, with the exception of the green flag and stop button, are the same shade of blue so the programmer can infer they’re all ways to control the environment.

The green flag’s colour is self-explanatory, as is the red stop button, mimicking Scratch for a sense of familiarity for the programmer. Additionally, the green and red stand out amongst the blue, making it easy for the programmer to infer where to start their program.

The variables box is capped by a dark orange banner, reminiscent of the same dark orange shade variable blocks are coloured in Scratch. Event handlers are capped with a yellow banner, to match Scratch’s yellow event blocks. This familiarity aims to create an association in the programmer’s mind, so they may discern what these boxes may correspond to in their familiar environment.

In order to convey the different nature of the stage in comparison to the sprites, pink is used in place of the blue outline of the other sprite boxes. When the stage tab is selected the features within including the buttons to add variables, functions, costumes and sounds, and the scroll bar, as well as the weighted window outline are also shaded in pink rather than blue to further convey this notion.

Finally grey is used to shade the tabs that are not in use, to further highlight which is the active tab to ensure the programmer knows what they’re making changes to.

**Line weight**

In order to highlight which combination of tabs are open, i.e., which sprite and which aspect of that sprite is being edited, line weighting is leveraged. The outline of the open tabs is thicker and encompassing of both active tabs, to create a clean active window.

**Spacing**

Spacing is used in the interface to infer to the user association by grouping. The stage is placed at the bottom of the sprite list, not in between any of them, to ensure it’s not confused with being another sprite simply named ‘stage’. The program controls, i.e., the green flag and the stop button, are also grouped together on the upper left side of the graphics window with padding between them and the buttons for saving or exiting the project, to represent their belonging to different groups of functionalities.

3.3.4 Memory Aids

Weintrop and Wilensky’s study from section 2.2.1 highlights the block catalogue of Scratch as one of the reported successes of the environment due to its prioritisation of recognition over recall. It’s this study, combined with Kölling’s citation in section 2.3.2 of memory of commands as one of the transitional challenges, that motivates the need for including some form of memory aid in Pytch Junior.

The drop-down menu when creating functions for selecting the event of interest, is one way in which this is achieved. It takes the burden of memorising the various types of events off the programmer. This drop-down menu was introduced to the design to reduce the amount of typing, and to relieve the programmer from some of the burden of complexity, but it has the additional benefit of acting as a memory aid.

As documented in 3.3.2 this design aims to keep the bodies of functions as close to pure Python as possible, but that doesn’t mean we can’t add support for producing this code text. To do this, an autocomplete feature is included that can offer suggestions for commands, but also for in scope variables. The idea for this came from Homer and Noble’s work on a ‘jigsaw environment’, as described in section 2.3.3, and their reported findings that students found the feature of variables in scope listed as helpful. The autocomplete offering in-scope variables mimics this.
3.3.5 Other Ideas Considered
Sections 3.3.1 to 3.3.4 all describe features included in the final design, but there are a couple of other features considered that are worth mentioning briefly. While they were ruled out of scope for now, the merits of these features make them limitations of this design that could potentially inform some future work on Pytch Junior.

The first feature considered is the idea of introducing elements of the gradual learning technique Herman describes in the paper covered in section 2.1.3 and 2.3.3. Gradual learning could be achieved by removing the function drop-down menu over time, or reducing how much of the header is pre-populated. More syntax could be revealed over time, or allowing non-event handler functionality after a ‘training wheels’ period. This idea of progressively removing the ‘learning scaffolding’ to bring Pytch Junior to Pytch is certainly an interesting one, and warrants further research in the future to gauge how much benefit it can bring, and how much work would be required to implement it.

Another idea of note that wasn’t incorporated was a catalogue of available commands similar to the one available in Scratch, in Kölling’s frame-based editor Stride from section 2.3.3, and in Pytch in its current state. Homer and Noble also reported positive feedback from their ‘toolbox’ of available methods in their ‘jigsaw environment’, which gives this idea merit to be considered in the future. It was omitted from this design for two reasons. The first being, it was judged unnecessary given the reduced amount of typing and other memory aids included. The second reason is based on something Mönig outlined in the paper discussing GP [30], and previously touched on in 2.3.3. This is the notion that experienced users find it time consuming to use a catalogue to find blocks, and something like an autocomplete feature would be more suitable. While Pytch Junior users are not experienced in the sense that they are masters of programming, it’s reasonable to assume they are experienced Scratch users and may benefit from a speedier method of finding available commands, such as the drop-down menu provided in function headers.

3.3.6 Final Thoughts on the Proposed Design
This section presents the design in full, in a series of user interface mock-ups and written descriptions that explain the thought process behind the decisions as well as the justifications for them based on the literature review of section 2. At the beginning of this section, four main design aspects were outlined: code organisation by sprite, reduced typing requirements, visual cues, and memory aids. The design created to meet these criteria, means additional support is available in Pytch Junior when compared to Pytch in its current state, to address more of the challenges of transitioning. These include the mechanics of typing and syntax errors, improving the experience of code management and readability, reintroducing visual cues, providing memory aids, concealing keywords and some aspects of complexity, providing support to ease the burden of grouping, matching identifiers and writing expressions, and general measures to support programmer confidence. All of these supports are introduced to Pytch Junior, while maintaining a level of authenticity and flexibility by preserving the text nature of, and some keywords in the function bodies.

The description of these design iterations, together with the descriptions of the focus of the corresponding literature reviews iterations conclude phase two of the DBR approach. The next section of this report will turn to the technical approach and provide a theoretical estimation of what’s needed to implement the design described in this section to start the third DBR phase.
4 Implementation

As documented in section 1.3, the design-based research approach falls into four phases, and due to the limited timeframe of this dissertation, this project focuses on phase 1 and 2. However, this section and the following section 5, provide theoretical frameworks for implementation and evaluation to inform the third and fourth phases of the Design-Based research approach. This section proposes a theoretical outline of what’s required to implement the comprehensive design laid out in the previous section. It will provide suggestions for implementing the different design features, as well as the challenges to be expected.

The work for implementing Pytch Junior will be based on the existing Pytch codebase [50], and in particular the Web app component of the repository. Pytch is developed using EasyPeasy React, so Pytch Junior will naturally do the same. The software design for Pytch Junior is structured by a priority list of features, so this section will follow suit, providing an outline for implementing each one, in order.

4.1 The Editing Experience in Pytch

Recall Hansen’s argument from section 3.1.2 and 3.3.1, that Scratch is successful because of its organisation by sprite concept. Bringing this concept into Pytch Junior, eases a myriad of transitional challenges, and therefore makes it the most important part of design. The entire Pytch Junior editing experience is based on this concept, so subsequently is the highest priority in an implementation strategy.

4.1.1 Components

As mentioned in section 2.5.1, editing in Pytch is done using Ace Editors [34][35]. Ace editors are embeddable editors written in JavaScript, that offer a variety of features including code folding, automatic indenting, and syntax highlighting [51]. While these features might not be currently relevant for Pytch Junior, using Ace editors now means allowing for flexibility and power in future work.

Pytch’s interface essentially consists of a number of React components that render the application in a browser. The ‘CodeEditor’ component within the IDE editor, in its current state, defines an Ace text editor which obtains the code from the user. This is what needs to be modified to overhaul the UI into multiple text editors for multiple sprites. This will most likely involve creating an entirely new component that can handle multiple text editors. The challenges of getting multiple ace editors working within one page, is noteworthy. It is possible, but requires some careful configuration. Additionally, some work is needed to alter the application state, so it knows which sprite is active and therefore which editor needs to be rendered, which will be discussed later in this subsection.

The CodeEditor component will require configuration to display the variables and event handler boxes rather than just as plain text. A component to handle the process will be required adding a new variable, as well as the drop-down component for selecting a new event handler.

One challenge for the CodeEditor component, is introducing the autofill feature to show programmers the list of in-scope variables. Ace Editor has an autocomplete feature that could be leveraged to implement this. Another challenge is adding some mechanism to stop users from referencing one sprite’s asset from another sprite.

In addition to the ‘CodeEditor’ component, alterations must be made to the parent ‘IDE’ component to introduce the tabs for both sprites and assets. It seems most logical to create an additional new component to define and render these tabular features, that slot in with the ace editor component.

Additional components will also be required to render the costume, and sound tabs, in addition to components to render the UI that handles the user process of adding a sprite asset from their local machine. This component will also link to the application state to make the necessary updates to reflect the user’s new additions.
Small alterations must also be made to the ‘IDE’ component to remove the assets tab from the text output box at the bottom of the environment.

### 4.1.2 Application State
Pytch uses Easy Peasy, an abstraction of redux, to simplify state management, so Pytch Junior should do the same, as it’s already in use in the code base Pytch Junior will contribute to. Connecting the components to the application state and backend is a vital part of building the editing experience.

The application state needs to reflect the new more structured interface, so something must be added to indicate which sprite tab, and asset tab is active. It also must have an attribute that keeps track of the sprites created, which will be updated when a new sprite is added, and when a sprite gets a new variable, function, costume or sound.

To represent which sprite is active, a simple integer could be used to keep track of which sprite in a list of sprites is the one that’s active - similar to an index. A secondary integer could be used to keep track of which tab within a sprite is active. These integers provided to the ‘IDE’ component from the application state, allow the component to use a selection mechanism to display the correct tabs. More detail on this list of sprites used to represent the user’s program will follow.

Getting the application state to connect to the components is a key challenge for implementation. Ensuring the components correctly pass information about the program state and editor state are crucial to ensuring the user’s code is correctly organised, stored and displayed.

### 4.1.3 Visual Cues
To create the visual cues included in the design for Pytch Junior, alterations will need to be made to the styling sheets in CSS. Firstly, the new components will need to be added to these sheets and then the necessary styling to support line colour, and weight, as well as the coloured banners at the top of the variables and event handler boxes. Stylings for all the new buttons, and UI components will also need to be included to style the components as per the UI mock-ups provided in section 3.
4.2 Running the User’s Code
Section 2.3.2’s citing loss of graphics as one of the challenges for users in transition, conveys the importance of graphical elements to novice programmers. Therefore, it’s important to get the user’s code running, once the editing experience has been implemented, to provide the graphical output that engages and encourages learners.

4.2.1 Representation of the User’s Program
The first thing to consider when trying to get a user’s code running, is what the code actually looks like. The new code organisation means code is a lot more structured than it is in the existing Pytch setup.

A user's code, instead of a scroll of Python code in text form, now consists of a number of sprites each composed of a list of variables, a number of event handlers, and a list of costumes and sprites. These event handlers are in turn composed of a header and a body. To reflect this new organised structure, the application state needs something richer. The ‘codetext’ variable in the project.ts file reflects the code provided by the user. In Pytch this is just a string. For Pytch Junior, this needs to be much more than that so the front-end components can accurately present the user's structured code to them. Figure 23 below shows a suggested approach:

As stated above, a program in Pytch Junior is a collection of sprites and a stage. In the application state this could be represented as an array, starting with the stage and followed by the sprites. It makes sense to start with the stage as this will exist when the program is first created where the other sprites will not. When a user uses the interface to add a new sprite, the components should connect to the application state and ‘push’ a new sprite to the end of this array.

Each sprite array element is itself another array. The first element in this array is an indicator of whether this item is the stage or one of the sprites. This is initialised to 0 for the stage when the project is created, and when a user creates a sprite, this is 1. This simple mechanism allows the components to know whether they are rendering the stage or the sprites, and adjust the visual cues accordingly. It also tells the converter function whether it’s a stage or sprite, as these are handled differently to convert them to Python. This will be discussed further in the next subsection.
The next element in a sprite array is the variables object. This is a collection of key-value pairs where the key is the name of a sprite’s variable, and the value is that variable’s initial value.

The event handler object is next where the key corresponds to the function head, and the value is the function body.

The costumes and sounds are arrays of file names corresponding to the actual files uploaded by the user in the ‘assets’ state attribute. This allows the costumes and sounds to be mapped to sprites, even though they are all stored in the same location. The first costume pushed to this array is used by the tab’s component as the icon for that sprite.

The last element in the sprite array is the name of the sprite, needed to form the class names by the converter function as described below.

4.2.2 Compiling the Code

Skulpt is used to compile the user code to JavaScript to be used by Turtle Graphics to produce the graphical output. Skulpt, however, compiles Python code, so a method is needed to take our heavily structured code and produce a string of compilable Python code. To do this, a ‘converter’ method should be implemented.

This converter method takes the structured representation outlined in the previous subsection, and produces Python code. The converter takes an item from the program array and uses Python syntax to create a class for this sprite/stage using its name that the user provided. Recall that the ‘import Pytch’ statement required to get Pytch to work, is not required in Pytch Junior so the converter would add this also. The converter uses the ‘stage or sprite’ integer to check whether the object is the stage or the sprite. An additional import statement to import the ‘stage’ class is required for sprites to implement the ‘pseudo-globality’ of the stage variables and methods.

Recall from section 3.3.2, it was decided users would not assign names to their functions and that they would be auto generated behind the scenes. This could be done by the converter method. As the function names will be required later for error reporting, these must be generated sensibly in a manner that allows the error reporting system to reverse engineer the generated name to produce the original function header. The converter should use the function header stored in the sprite’s event handler object, to generate these function names and declarations in Python, before using the function body to complete the function, adding the correct indentation as required. In addition to the functions, the converter would take the sprite’s variables object, and add these to the top of the class using Python syntax.

Any sprite, as explained above, technically has access to all costumes and sprites in the project. Because of this technicality, a mechanism will be required to stop users from referring to another sprite’s asset in the code editor. This means the code that arrives at the converter function shouldn’t have any illegal asset references, so no further work is needed in the converter function to stop these illegal references.

Once the converter function has produced the syntactically correct Python code, it can be fed to the Skulpt engine and in theory, the rest is taken care of by the Pytch VM, and doesn’t need to be altered by this implementation. In order for this theory to hold, the program representation must be properly implemented and the converter method needs to be carefully designed and built to ensure the generated Python code is syntactically correct but also corresponds functionally to the behaviour of the user’s program in its structured format.
4.3 Allowing Projects to Persist in the Database
As Pytch Junior is an educational environment, it is reasonable to assume that users may want to leave and return to their projects. Therefore, the third priority on the implementation strategy is allowing projects to persist, and indeed un-persist, from the database.

Pytch utilises the browser's local storage to store users’ projects and Pytch Junior should do the same. In Pytch the code can be considered merely as a very large string, and is stored as such under the ‘codetext’ variable as a JSON string mentioned earlier in this section. Pytch Junior’s code is much richer than a string and storing this structured format could prove challenging and complicated, with no real need to store it as such.

By altering the converter method described in 4.4.2, so that it can work in reverse, it could be possible to use this function for storage purposes too. When saving a project, the environment runs the converter function and saves the Python text version of the project in the local browser storage as a JSON string-as in Pytch. When the project is reopened, the converter function is run in reverse. Taking the stored Python version, stripping it of its added syntax, import statements and placing it back into its structured array format. This may cause a slight lagging in opening the project, but may be the more sensible option than altering the way in which Pytch persists the program. Certainly, for the first implementation in the third phase of the DBR approach, it’s more efficient to alter the converter function than altering the way in which Pytch stores the program, and allows for the quick progression to user testing to get feedback on the more important parts of the design.

It’s worth noting that it may not be so simple to reverse the converter function, and may pose a significant challenge depending on how the algorithm is designed in the first place. Therefore, it would be worth implementing the compiling and storing features in tandem to ensure the system is cohesive, rather than designing the converting algorithm and then having to overhaul that completely in the next stage in the implementation approach.

4.4 Error Reporting
Section 2.3.2 cites both increased syntax errors and understanding error messages, as challenges for novice programmers transitioning to text-based programming. Therefore, it’s paramount to get sensible, and instructive error reporting in Pytch Junior. The challenge of this lies in mapping the line errors reported by Skulpt based on the Python text version to the structured version of the user’s code. For example, if Skulpt returns an error saying there’s a ‘syntax error on line 14’, the user has no way to know what sprite that belongs to, let alone which function.

This mapping method will need to be able to return something more like ‘syntax error in Sprite X in function Y’ at the very least. Ideally, it would also be able to provide a line number within the sprite’s function but for the first implementation this could be omitted until user feedback could determine whether this is necessary or not. Mapping is a non-trivial task. The mapping function will need to find the line with a reported error, and find the nearest ‘def’ keyword in the lines above to discover what function it belongs to by reverse engineering the generated function name to find the original header. It will then also need to find the nearest ‘class’ keyword to discover which sprite it belongs to. It should then take both of these items and formulate an error message something like below:

"Syntax error in 'Elephant' in 'When green flag clicked"

Achieving error reporting similar to the above message would be adequate for the first implementation for user testing. The feedback from this iteration of testing could gauge if this is sufficient for users or whether the mapping function should go one step further, to also provide a line number within an event handler to provide the user.
4.5 Final Thoughts on The Proposed Implementation Framework
This section provides a theoretical strategy for implementing the full design for Pytch Junior, including suggested approaches, considerations and probable challenges. The practical implementation of this design is considered future work, and will be included in the concluding section of this paper under that heading. It also belongs to the third phase of the Design-Based Research methodology that informs this project, to be used in conjunctive iteration with the evaluation framework provided in the following section to refine the design for Pytch Junior.
5 Evaluation

A key part of the DBR approach is the third phase, composed of iterative cycles of testing and refinement of solutions in practice. As discussed, this project focuses on following phase 1 and 2, of the DBR approach, so this section in conjunction with the previous, offers a theoretical framework for following phase 3.

The evaluation of this paper is structured in two parts. The first is a desk-based evaluation of how (i.e., what features) the design addresses the identified areas in Pytch where increased transitional supports could be added. The second part provides an evaluation framework for a practical implementation to assess if the design, by including more support in key areas, aids the programmer in mastering programming concepts and behaviours while maintaining the gains Pytch achieves over block-based languages namely authenticity, flexibility and power.

5.1 How the Final Design Addresses the Original Criteria

The first aim of this project was to identify areas in Pytch where increased learning support could be introduced. These areas were identified in section 2 as code organisation, reduced typing requirements, visual cues and memory aids. These areas provide scope for targeting a number of transitional challenges, and serve as the original design criteria for Pytch Junior, along with maintaining the authenticity and flexibility that a more text-based language entails.

Code Organisation

By addressing the area of code organisation in Pytch Junior, the design stands to reduce challenges such as understanding layout and spacing, readability, keywords and concepts, grouping, and confidence loss. To address this, the design for Pytch Junior introduces code organisation by sprite by using tabs to separate code according to which sprite it belongs to. Additionally, variables and event handler functions are separated into boxes within each sprite’s code tab, resulting in a highly visually structured representation of code.

Reduce Typing Requirements

Lessening the burden of typing for Pytch Junior programmers, also lessens the burden of challenges like the mechanics of typing, syntax errors, keywords and concepts, matching identifiers, grouping, and writing expressions. The drop-down menu feature for defining event handlers, and the GUI interaction for adding a variable both reduce the typing requirements for the user. In fact, the only part of their program they have to type is the body of the event handler functions.

Visual Cues

The loss of visual cues is one of the challenges cited in section 2.3.2, along with readability and confidence loss. Visual cues in Pytch Junior including colour, line weight and spacing, help foster familiarity with Scratch, support inference of variable and event handler behaviour, as well as information about the environment, i.e., which tabs are active, what functionality certain buttons control.

Memory aids

Adding memory aids to the design for Pytch Junior, offers the opportunity to mitigate the transitional issues of remembering keywords and syntax, as well as aiding with the typing requirements and syntax errors. To accommodate this, Pytch Junior includes an autocomplete feature in function bodies to remind users of the in-scope variables. The drop-down menu introduced to ease the process of creating functions, also has the benefit of reminding the user of the available event handlers so they don’t have to remember them also.

This design demonstrates that there are features that can be introduced in Pytch Junior in a novel way, that address the areas identified for more support to meet a variety of transitional challenges. It could
be said that some of the authenticity of Pytch is lost in Pytch Junior, with the UI overhaul that reintroduces code organisation by sprite. The addition of the colour cues, and sprites tab visually also make it seem closer to Scratch than Python, or another text language. But as Pytch Junior is meant to be used in between Scratch and Pytch on the young programmer’s education timeline, this seems reasonable. In addition, the function bodies remain fully in Python-like text, which provides a significant amount of visual authenticity to satisfy the user, and a lot of the flexibility that Pytch boasts over Scratch.

In identifying these features, this evaluation shows the research question identified in section one has been answered by the provision of a research-based design and technical considerations for Pytch Junior. To gauge whether or not the design allows users to master programming concepts and behaviour, we need an evaluation framework to apply to a practical implementation of the design for Pytch Junior from section 4. This framework is provided in this next section.
5.2 An Evaluation Framework for A Practical Implementation of Pytch Junior

This framework involves user trials to gather information about the practical implementation for Pytch Junior, whether it meets the design criteria first outlined in section 2.5.3, and whether meeting these needs has a positive impact on conceptual understanding of how programs are built and how they work. To carry out user trials, research ethics approval is required. This author has successfully obtained that approval from the SCSS ethics committee, and the below section is informed by the framework provided in that application. The full ethics application can be found in appendix A.

Format of the Trials

The trials involve two groups of users aged approximately 15-16. These users can be recruited from specific coding education institutions like CoderDojo, or from schools. It’s worth noting that the person carrying out the trials, must be garda vetted as the participants are under 18.

In this framework, the students are split into two groups. One will use the current version of Pytch, acting as the control, and the experiment group will use the new Pytch Junior. Both groups will be given a set of equivalent tasks to complete, with an equivalent introduction and walk through of a sample task. The aim is for both groups to complete as many of the tasks as possible.

To assess how the groups fared, surveys have been designed for circulation before and after the session. A focus group should also take place at the end of the session, after the surveys have been completed.

Gathering Data

Pre-Session Survey

The first survey obtains information about the participants’ coding experience in terms of history and topics. This is important to compare with the same participants’ final survey to dispel any outliers in the data arising from a participant with unusually high coding experience. It can also help to understand the participants’ answers in the second questionnaire and focus group contributions, to speculate on why they may have found certain things easy or why certain things proved difficult for them. For example, if a participant reported in the second survey that they found everything in Pytch Junior very easy, but they reported in the first survey that they’ve been coding in Python for five years, this data doesn’t provide any value in assessing how Pytch Junior supports a user’s conceptual understanding of programming. This pre-session survey is included in full at appendix B.

Post-Session Survey

The second questionnaire is delivered to the participants after the session. Recall the four steps in the strategy for implementation outlined in section 4: the editing experience, running the user’s code, persisting a project, and error reporting. The two aspects of this implementation actually visible to the user are the editing experience and the error reporting system. Therefore, this survey was designed to gauge the participants' feelings on these two aspects, whether they felt more supported in Pytch Junior and if they reported a better conceptual understanding of programming. The questions designed for gathering feedback on these aspects, were divided into four groups. The full post-session survey is included in this paper at appendix C.

- The Pytch Environment: these questions seek to understand if the participants enjoyed the environment and found it usable, and whether they would be encouraged to continue using it to further their coding education. This doesn’t directly relate to either the editing experience or error reporting, but is important to establish whether the participants had a positive experience using the tool. Discovering from these question responses that the participants, despite finding it perhaps easy to use, hated using it, is an important discovery.
• *Working with functions*: this category enquires about two things: how easy participants found the correct place to add functions, and how easy they found it to write their functions. This is aiming to get feedback on whether the new design supports code management and typing, and to gain some insight into how participants found the editing experience.

• *Working with Bugs*: these questions also seek to evaluate how participants found code management and typing, as well as the error reporting system in Pytch Junior compared to Pytch - by asking how easily they found the correct place to fix a bug as well as writing the actual code to fix the bug. These questions are designed specifically to get feedback on the error reporting system, whether it accurately reports error locations to allow users to find the bugs in their code effectively, and whether they could interpret the error message format.

• *Understanding my Code*: This category asks a series of questions about the participants' understanding of their code including how sprites communicate, how sprites react, how they manage code as the program grows, and how they handle the amount of text in their program. These questions were designed to ascertain if the design of the editing experience is successful in helping with readability and reducing the text scroll. It also seeks to determine if the inclusion of memory aids and visual cues have offered benefits in improving conceptual understanding of program structure, concepts and behaviours. In a nutshell, it seeks to learn whether the supports included in the new editing experience are sufficiently alleviating the lingering transitional challenges in Pytch, to aid users' conceptual understanding.

**Focus Group**

In addition to the surveys, this evaluation framework includes a list of suggested questions for a focus group session. This session should be recorded to allow the researcher to focus on leading the session rather than taking notes.

This session is designed to capture any information that the survey missed, about the participants' thoughts and experiences of both Pytch and Pytch Junior. The group that used Pytch Junior may offer insight into how the design changes fared, but the Pytch group may also be helpful in informing the Pytch Junior design by offering further insight into the benefits and challenges of Pytch. The suggested questions for this session are included below. These questions are just intended to be used to guide the session; they’re not written in stone. In the focus group it’s important to also let the participants guide with their experience of the environment. As such, the questions are designed to be quite open, and not very specific, to allow participants to respond with a wide range of answers. By leaving them quite open, the questions can kick start conversations about aspects of the environment or experience not even considered by the researcher, and ensure the focus group session can gather different, potentially more valuable information than in the surveys.

• Did you enjoy using the Pytch environment?
• How did the programming compare to your previous experience of coding?
• What parts of the session did you find easiest?
• What parts of the session did you find hardest?
• What was the biggest challenge you had in the session?
• How did you find managing your code? Was there too much text?
• Was it easy to locate the part of your code you wanted to edit?
- Was it easy to write the code to make the changes you wanted to, to your program?
- Did you find the code organisation by sprite helped you to visualise the overall structure of your program?
- Would you like to continue using the Pytch environment?

The focus group also offers the researcher an opportunity to get the participants' insight into how authentic the environment felt to them, and whether it offered more power and flexibility than Scratch.

**Analysing the Data**

The data collected from both the surveys and focus groups can be compiled, and the comparison between the control group and experimental group can inform further design revisions, as per phase 3 and 4 of the DBR process.

The first step in analysing the data is linking the pre and post surveys by participant and anonymising all the data. The post surveys should be analysed in each group, the Pytch and Pytch Junior group, and compared by each question group. This will provide insight into how each group felt about the overall environment, how they found editing functions, how they found the error reporting system, and how they felt it aided their conceptual understanding of programming. As outlined above, the pre surveys can be used to dispel any outliers from the data.

Knowing if Pytch Junior users reported either better or worse experiences than Pytch users in any of these categories, can dictate if future design iterations require more or less support in those areas. For example, if testing the first design iteration using the implementation described in section 4, and it’s discovered when comparing the two groups, that the Pytch Junior tested badly on working with bugs, and reported it was hard to interpret error messages, then it can be concluded more work is needed on the error reporting system.

The focus group can tease out more information on why they found certain things difficult or easy, and this can be used to refine the design and implementation of Pytch Junior as well. Carrying on from the previous example of poor feedback on error reporting, when analysing the transcribed session audio, the researcher can look for information provided in the session on what exactly the participants struggled with. Whether they didn’t know where to look in their function, or whether they didn’t understand the error message text in the first place. Additionally, the researcher can analyse the transcript of the Pytch focus group, to find information about why they found working with bugs easier, and how well they interpreted the reported errors to learn what elements of the Pytch reporting system could also be adopted by Pytch Junior. Where the comparison of questionnaire data may inform the researcher that a change is needed in some area in Pytch Junior, data gathered in the focus group session can provide detail on why they found it easy or difficult to complete the tasks, to inform what kind of change is required in the next iteration to address this.

All of this data should then inform the next implementation of the design, followed by further testing and refinement. This process should be repeated until the researcher is satisfied that the editing experience and error reporting of Pytch Junior provides adequate support increases in mitigating transitional challenges to bolster conceptual understanding when compared to Pytch, while maintaining an acceptable level of authenticity and flexibility.
6 Conclusion

In the introduction section of this paper, the aims for this project were discussed. The project aimed to identify areas in Pytch Junior that provide scope for addressing challenges in the transition from blocks to text. The addition of support to these areas serves as design criteria and the project’s goal is to find features that can address these criteria.

The literature review in section 2 provides a thorough overview of the research space of coding education, to identify the areas of code organisation, typing requirements, visual clues and memory aids. Section 3 describes the features of the design to encapsulate these areas and section 4 provides a theoretical implementation of this novel design. Section 5 provides an evaluation that confirms that the design meets the criteria in adding more support in these areas of Pytch, to mitigate transitional issues lingering in Pytch, while maintaining elements of authenticity and flexibility. It also provides a framework to apply to a practical implementation, to ascertain if the design does indeed meet the other design criteria first outlined in section 1, to allow users to have an easier time mastering program concepts and behaviour.

With this work complete, it’s worth returning to the original research question - what features, if any, can be employed in a novel way to add additional support to Pytch in key areas to alleviate the burden of transitioning while maintaining the authenticity, and flexibility benefits of transitioning, and how these features manifest in a new intermediary environment called Pytch Junior.

Section 3 shows there is a novel way of using design features such as tabbing, GUI interactions, drop-down menus, autocomplete, and visual cues, to address these needs and section 4 outlines that these features can be feasibly implemented to produce a new coding environment.

The research question also involves the challenge of maintaining a delicate balance between supporting a user by concealing complexity, and not misleading the user, to ease a learner into a new programming language, while building solid learning foundations for the future. This balancing act is not a trivial problem, nor is it straightforward to decide whether the design is successful in maintaining it or not. The design proposed in this paper for Pytch Junior, succeeds in providing more transitional support for users, and in doing so hides some complexity from them. Therefore, it could be said the environment is weighted in favour of support more than faithfully representing programming concepts and behaviours. But this concealment is justified for the user group Pytch Junior aims to cater to, and elements of authenticity in the fully text function bodies and key words required in those bodies, mean users are still exposed to full Python syntax and the complexities that entails - just on a smaller scale than in normal Python. In that sense, this project has managed to maintain the balance, and future work described below could serve to improve this balance even further.

6.1 Limitations of Design

By following the time intensive methodology of design-based research, this project focused on the first two phases of the approach - to perform iterative literature reviews in tandem with iterative design stages, to produce a design rooted in principles and innovations from the coding education space. A theoretical approach based on the requirements of this design is provided, but the lack of a practical implementation is the key limitation of this work. Thus, the first step in further research on this body of work is obviously to begin phase three of DBR with the first practical implementation and first round of user testing. While theoretical approaches to both the implementation and evaluation are provided to kickstart phase 3, practical implementations, iterative testing and design refinement, will complete the DBR process to produce a fully formed, user tested implementation of Pytch Junior.

While the design that is produced is successful in outlining features in adding support in key areas of Pytch to ease the burden of transitioning, there are some limitations of the design that should be noted.
The first way Pytch Junior’s design is limited is in the support it offers for memory aids. Pytch includes a blocks catalogue of sorts that outlines the translation between common Scratch blocks and text code in Pytch. This is not included in the design for Pytch Junior.

Pytch Junior’s design is also limited by not providing tutorials as exist in the current version of Pytch. These tutorials were ruled out of scope of this project, as the project’s focus was on identifying design features to address key areas that lack support within the practical programming experience of the Pytch Junior environment. Tutorials fall outside of that focus, but offer benefits in helping a user to begin using Pytch Junior for the first time. Recall in section 2.5.4, that Pytch Junior wants to ‘lower the floor’ even more than Pytch does. Tutorials could offer further help to achieve that, and thus are included in future work.

The other limitation involves the question of balance. Recall the main challenge of this project outlined in section 1.2 - maintaining the balance between supporting the programmer, and not misleading the programmer’s conceptual understanding by concealing too much complexity, is paramount to the success of Pytch Junior as an educational tool. While the design outlined for Pytch Junior does maintain some elements of authenticity and flexibility that reveal complexity to the programmer, the addition of code organisation by sprite, and the drop-down menu for creating event handler functions conceal a significant amount of complexity. In addition to this, non-event handler functionality is not possible in Pytch Junior which also limits the power and flexibility of the environment, and omits information about how programs can be built from the programmer. Suggestions for addressing this are provided in the future work discussion below.

6.2 Future Work
The future work for this paper is informed in part by the other design ideas mentioned in section 3.3.5, and in part by the limitations of the design introduced in the previous section.

The first item for future work is to base a practical implementation of Pytch Junior, on the theoretical approach outlined in section 4. This should be followed up by applying the evaluation framework from section 5 to complete the third and fourth phases of Design-based Research methodology. Completing these phases will allow for the analysis of the user's experience of the environment and whether they feel more supported in understanding underlying programming concepts. The data gathered from this evaluation process can, and should then inform revisions to the design outlined for Pytch Junior in section 3.

To tackle the limitations in support described in the previous section, tutorials and a command catalogue should be explored in future work. The tutorials could use the existing Pytch functionality as a basis for adaptation to produce a number of tutorials for use in Pytch Junior. Similarly, the existing command catalogue in Pytch could be used as a basis for introducing it in Pytch Junior. With minor adjustments in the user interface, and minor editing in the commands included to remove the irrelevant ones concerning class, or function headers, a catalogue could be introduced relatively easily.

To tackle the problem of improving the balance of support versus flexibility, and authenticity, also outlined in the previous section, one umbrella approach could broach the problem of non-event-based functionality and concealed complexity. Gradual learning, from Herman as first introduced in section 2.3.3 as a concept and in 3.3.5 as an idea for Pytch Junior, could offer a way to offset the balance from support to authenticity over time, by gradually revealing more complexity to the programmer. For example, function creation could migrate from a drop-down menu interface to a pure text one over time, the class concept could be introduced after a time, and indentation and syntax could be revealed as the user progresses. In addition to revealing complexity, the gradual learning approach could also be used to introduce non-event-based functionality once a programmer has reached some coding milestone within Pytch Junior. Gradual learning would involve significant work to implement, but as it has the potential to address a variety of design limitations, it warrants exploration in the future.
The final piece of future work is a small one - establishing the best name for the environment. Pytch Junior was named as such as a temporary measure, to have a name to refer to during development of the idea. In the future the inclusion of the word ‘junior’ may seem childish and inauthentic to the target audience of teenagers, so may warrant changing. One way to get some feedback on how the target user feels about the name, is to ask them in the focus group sessions described as part of the evaluation framework provided in section 5.

6.3 Final Remarks

This paper follows the first two phases of the Design-Based Research methodology to introduce a novel design and a new approach to bridging the transition of blocks to text, that addresses the identified areas for increased support in Pytch. The intertwined iterations of literature review and design stages, and the implementation considerations constitute the technical content of this project, and result in a thorough design for the first implementation of Pytch Junior. The features of this design successfully mitigate a variety of transitional challenges, with the hope of supporting users in learning programming concepts and behaviour, all the while maintaining authenticity, flexibility and power in the Pytch Junior environment.

At the beginning of this paper, the author cited Papert as saying programming languages should have ‘low floors and high ceilings’ [47]. Pytch Junior started out as the desire to lower Pytch’s floor, and over the course of this project has developed into a rigorous design rooted in literature to not only lower the floor, but ensure novice programmers get to enjoy their high ceilings too.
References


Appendices
Appendix A: Ethics Proposal
Ethics Proposal

Section 1 Project Description

Title of study
Evaluating the effectiveness of the reintroduction of code organisation by Sprite to the Pytch editing environment on young learners.

Dates and Duration of Study
February 2022

Purpose of the project including academic rationale.
The purpose of this study is to introduce novice programmers aged approximately 15/16 to a programming environment with added editor support for helping learners organise and understand their high level program structure, by chunking their code in a way that is familiar to them from their scratch experience, and then evaluating the effect of these changes to the environment. Pytch is a programming environment that serves as an additional step between block based programming and text based programming, aimed at improving the transition experience from the former to the latter for novice programmers. The project’s focus is on enhancing Pytch with improved editor support for program structure and investigates the effect of this support on the users ability to manage the program text.

The changes have been introduced to mitigate the loss of clean, code structure and visual cues in the jump from a purely block based environment like Scratch to a transitional environment like Pytch, while maintaining the increased authenticity and flexibility gained in Pytch. The aim of the study is to allow novice programmers to test out these changes, and then ask them to complete a questionnaire on their experience, to understand if the changes have been successful. This work is a part of a MCS dissertation project being carried out by the researcher.

Research Question
Can the introduction of code organisation by sprite to Pytch (as seen in Scratch) improve learning performance, and reduce frustration for students making the transition from block based programming to text based programming.
Hypothesis

The difficulties of the transition from block based programming to text based programming are well researched and documented[7], and Pytch addresses many of these challenges in its current state. Our hypothesis is that we can reintroduce code organisation by sprite as seen in Scratch, into Pytch to regain some of the intense structure that boosts learning in block based programming. We believe that this code organization will reduce the cognitive load of the learner, and enable them to focus more easily on the parts of the program they are working on, without losing the authenticity and flexibility gained in the transition to Pytch.

The objectives of the project are:

1. Introduce informed changes to Pytch’s programming environment with the aim of retaining the code structure and organisation of block based environments, in order to further ease the transition from block based programming to text based programming.
2. Trial these changes with novice programmers aged approximately 15/16 by conducting an experiment where two groups use the original Pytch environment and the altered one.
3. Gain feedback via questionnaire and focus group on their experience of the environment and the changes introduced.

Procedure of the Study

The proposed study is to run an experiment comparing the current Pytch programming environment against the altered Pytch programming environment. These alterations must first be developed, in order for this experiment to take place.

The preferred mode for this experiment is in an ‘in-person’ setting, but the researcher acknowledges this may change depending on COVID infection rates and regulations, and thus a contingency plan is in place to run this experiment in an online setting.

For the experiment:

1. Split the students into two groups: the first will use the original Pytch environment, the second will use the altered environment.
2. Both groups will be given a set of equivalent tasks to complete.
3. Both groups will be given an equivalent introduction to the tasks and walked through the same sample task
4. Both groups will complete as many of the tasks as possible

The proposed methods for data collection are:

1. Questionnaires
1. Pre-Experiment to determine their prior programming experience.
   b. Post-Experiment to determine their experience of programming in their environment.

2. Focus group
   a. Conducted after the session

3. Observations
   a. Noting any problems faced by children during the session including syntax problem, issue with language, issue with programming environment, other

4. Number of Tasks completed
   a. How many tasks each child completed in the session

The data will be collected and analysed to answer the research question posed, and to test the hypothesis outlined above.

**Expected Results**

The expected results are that the altered Pytch environment with increased code structure and organization will allow learners to focus on the concepts of the language and reduce the cognitive load of a large scroll of code. We expect the group that uses the altered Pytch to find it easier to build up their programs as they will not be overwhelmed by large scrolls of text, in a syntax they are unfamiliar with.

We expect students using the altered environment to complete more tasks, than the students using the original Pytch environment.

**Recruitment Methods**

The original Pytch team have a number of contacts in CoderDojo Ireland, and a number of CoderDojos they have previously worked with, which I will make use of to recruit participants for this study. The participants will be aged 15-16 approximately, and students in 3rd and 4th year of secondary school. These students will be selected using convenience sampling, with all willing participants being chosen, and permission being obtained from their parents/guardians.

In total we hope to recruit 30 students, and hope to run this session once, but can run it more than once to fulfil our articiant quota. The nature of the study will be explained to the participants and an information sheet will be provided to those who express an interest in participating. Interested students will be made aware they are under no obligation to participate, and can withdraw at any time without the provision of a reason. Signed consent forms from the student and their parent/guardian will be obtained before the commencement of the session.

Students who do not wish to participate may still remain in the Dojo session, and another instructor will be present to carry out the normal dojo activities with those students.
Debriefing Arrangements

At the end of the session I will explain further the purpose of the study. I will be available to answer any questions and will make my email address available to them following the session for any further questions.

Section 2 : Ethical Concerns

Ethical Considerations:

1. Working with children, their safety and wellbeing is of the highest importance. To ensure their safety all of the researchers interacting with children will be garda vetted and there will be a CoderDojo tutor in the class at all times during the session.
2. While the participants will be asked to provide their names for the pre and post questionnaires to allow for withdrawal from the study, and in order to link pre and post surveys together for each participant, they will be anonymised for any reports for the findings. This will be done by transferring the data from the questionnaires into an excel spreadsheet (stored on the researchers college OneDrive account) and stripping the names from responses, replacing them with ‘Participant x’. The questionnaires will then be deleted if digital versions are used, or shredded if physical versions are used. The anonymous data will then only be accessible by the researcher and her supervisor, who will also store any data in an excel spreadsheet on his college OneDrive account. In order to allow the researcher to capture meaningful data from the focus group session, the session’s audio will be recorded rather than just employing note taking, which may miss important points raised by the participants. The audio recording from the focus group will be transcribed and any names mentioned will be redacted. The recording will then be deleted.
3. All questions in the questionnaire will be optional.
4. Participants will be informed they may withdraw from the study up to the point of anonymisation without penalty.
5. Participant identities will be kept anonymous.
6. Parents/Guardians will be asked for consent. This will happen prior to the pre-session survey. As the post-session survey is applied directly after the workshop, the first consent/assent will apply for this survey too.

The contact person for this will be Ellen Whelan.
Conflicts of Interest
I must be cognizant of my role as both researcher and instructor during this experiment.

Relevant Legislation

Child Protection Legislation (Ireland) (https://www.education.ie)
In accordance with The Children First Act 2015
- The policies of CoderDojo Ireland will adopted and adhered to

Data Retention Legislation (EU and Ireland)
nsolidation/796.htm

All data collected will be stored in accordance with Trinity College Dublin’s data protection policies in compliance with the General Data Protection Regulation (GDPR) (2018).

As designated ‘data controllers’ as specified by Data Retention Legislation (EU and Ireland), the researcher will perform the following data management duties as specified by TCD in accordance with the management of participant data.

Trinity College Dublin Data Retention Policy
https://www.tcd.ie/about/policies/data_protection.php

In accordance with the principles of the General Data Protection Regulation (GDPR) (2018):

- Obtain and process information fairly - All participants will be issued with ethics information and consent forms, which they will be required to read and agree to prior to data collection

- Keep it only for one or more specified, explicit and lawful purposes - data collected by the project team will only be processed for research under the terms and research aims and objectives of this project.
• Use and disclose it only in ways compatible with these purposes - reported data sets be anonymised and published in accordance with the aims and objectives of the project research publication plan

• Keep it safe and secure - electronic data will be stored on a password protected directory system located on Trinity servers. Any hard copy data will be stored in a locked filing cabinet in the researcher’s supervisor’s office (on Trinity campus).

• Withdraw without penalty. Parents/guardians and their children may withdraw from the research up until the point the data is anonymised without penalty.

Section 3 – Confidentiality And Data Protection

Note that it is up to the researcher (and their supervisor if relevant) to discharge their own professional and legal obligations with respect to the handling of personal data. Please consult the College Data Protection Officer (DPO) if in any doubt regarding your obligations.

3.1 Does this study involve collecting, using, accessing or sharing personal data[1]?

☐ Yes
☐ No

If NO, please go to section 4.

If YES, please list[2] all categories of personal data.

Please see checklist on secure storage available here.

<table>
<thead>
<tr>
<th>Type of Data</th>
<th>Justification: Why do you need the data?</th>
<th>Data Format</th>
<th>Technical and Organisational Controls</th>
<th>Identifiable coded, or anonymised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consent Form</td>
<td>Evidence of Parental Consent</td>
<td>MS Form</td>
<td>Stored in researcher’s college OneDrive account. The supervisor will also have access to these, and will</td>
<td>Identifiable</td>
</tr>
<tr>
<td>Assent Form</td>
<td>Evidence of Participant’s consent</td>
<td>MS Form</td>
<td>Stored in researcher’s college OneDrive account. The supervisor will also have access to these, and will also store them on college OneDrive account.</td>
<td>Identifiable</td>
</tr>
</tbody>
</table>

### 3.2 Does the study involve collecting, using, accessing or sharing sensitive data[3]?

- [ ] Yes
- [ ] No

If NO, please go to question 3.3.

If YES, please list all categories of the sensitive data collected.

Please see checklist on secure storage available [here](#).

<table>
<thead>
<tr>
<th>Type of Data</th>
<th>Justification: Why do you need the data?</th>
<th>Data Format</th>
<th>Technical and Organisational Controls</th>
<th>Identifiable coded, or anonymised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Questionnaire including name, and programming experience. Note</td>
<td>Needed to validate findings.</td>
<td>MS Form</td>
<td>Stored in excel sheet on researcher’s college OneDrive</td>
<td>Anonymised</td>
</tr>
</tbody>
</table>
that all data collected is anonymised as discussed in section 2.

| Audio recording of focus group session. | Needed to provide qualitative data and insights on participant’s experience. | Audio recording and text transcript with names redacted. | Stored on researcher’s college OneDrive account. The supervisor will also have access to these, and will also store them on college OneDrive account. | Anonymised. Recording will be transcribed with any names redacted and recording deleted. |

3.3 Who determines how and why the personal and/or sensitive data is used?

(Data Controller[4] or Joint Data Controllers)

Provide Details: Data Protection Officer: Data Protection Officer, Secretary’s Office, Trinity College Dublin, Dublin 2 - dataprotection@tcd.ie

3.4 Will the personal and/or sensitive personal data be shared with any third parties[5]?

☑ Yes
☒ No
If **YES**, provide details including information on the contractual arrangements in place.

If **NO**, please go to question 3.5

This list should include all Data Processing Agreements with external laboratories, Cloud-based Solutions Agreements etc., and any and Data Sharing Agreements with Collaborators.

Please contact researchDPO@tcd.ie if you need assistance with agreements and/or for any transfer outside EEA (including England, Wales, Scotland or Northern Ireland).

**Provide Details:**

### 3.5 How long will you retain the personal data?

Please see good [research practice guide](#) for guidance on retention of research data. Your school should be able to advice on best practice.

**Provide Details:** The supervisor will retain the personal data for 5 years for research integrity purposes, after which point it will be deleted.

### 3.6 Will the personal data be fully anonymised or deleted after it is no longer necessary?

**See advice on secure data disposal**

**Provide Details:** Data will be deleted when no longer needed
3.7 How will you inform participants of their rights under GDPR[6]:

A data protection section is present on information and consent sheets supplied to all participants

Please note that the DPO’s contact details must be included on any information leaflet or privacy notice if you are using personal data for your research.

Email: dataprotection@tcd.ie

Post: Data Protection Officer, Secretary’s Office, Trinity College Dublin, Dublin 2, Ireland

References

[1] Personal data is information which can identify a person. In particular: a name, address, email, telephone number, an identification number, location data, an online identifier, an IP address, a code key linking back to identifiable data etc. Please note that pseudonymised data is personal data under GDPR. Pseudonymised data means data which cannot be attributed to an individual without the use of additional information which is kept separately. (i.e. a key). It is sometimes referred to as ‘coded data’. Please note that in order to be considered personal data in our hands, Trinity must hold the key.

[2] If using Personal data. Records must be kept pursuant to Article 30, GDPR; https://gdpr-info.eu/art-30-gdpr/

[3] Sensitive personal data means personal data, which poses a higher risk to the individual. It includes personal data revealing racial and ethnic origin, political opinions, religious or ideological convictions, trade union membership, criminal convictions and offences, genetic, biometric (photos, videos, audio etc.) data concerning physical or mental well-being, information relating to education, professional training employment and career history, questionnaires, Information relating to the family of the individual and the individual’s lifestyle and social circumstances.
[4] Employees and students of TCD are not data controllers. TCD is the data controller for the institution. However, if other institutes jointly decide how and why the data will be used, they should also be noted as controllers here.

[5] Third parties could be collaborators (institutes/industry) or service providers (transcribers, cloud storage etc.)

[6] Under GDPR, these include:

- right of access;
- right to rectification;
- right to erasure;
- right to object to processing based on legitimate or public interest;
- right to data portability;
- right to object to profiling or making decisions about individuals by automated means?

Appendix B: Pre-Session Survey
Pre Session Survey

This survey will take approximately 3-5 minutes to complete. Please be assured that while we ask for your name, all of your replies will be anonymized. Participating in the survey is voluntary and you may change your mind at any time. You may exit the survey at any point without submitting by closing your browser window. You may also choose not to answer a question at any time and skip it without penalty.

* Required

About this Survey

Information Sheet for Children

Title: Evaluating the effectiveness of the reintroduction of code organisation by Sprite to the Pyttch editing environment on young learners

Background Context
I am a Masters student studying Computer Science in Trinity College Dublin and a mentor in Trinity CoderDojo for the past four years. As part of my Masters I am completing a research project. I would like your help to test out new changes to the Pyttch programming environment.

Procedures
In this study you will be

1. Asked to complete two short surveys, one before the session and one after the session.
2. Shown a programming environment and taught how to use it
3. Asked to complete a series of tasks
4. Asked to participate in a focus group
5. Asked to provide assent for the researcher to record your voice contribution made within the context of a group discussion, conducted at the end of the session.

Conflicts of Interest
I must remember that I am both a researcher and a teacher during this experiment.

Voluntary Participation
Taking part in this study is completely voluntary and you may stop being a part of this study at any time.
Expected Duration
The expected duration of this session is 1 hour.

Risks
The CoderDojo tutors will always be present in the class and therefore there are no risks to you beyond those encountered by you in a normal class.

Debriefing
I will be available after the session to answer any questions and you can email any further questions you may have to whelane5@tcd.ie

Anonymity
We ask you for your name in this survey so that we can connect the pre- and post-surveys in this workshop. Once that has been done we will anonymise the data by deleting the the original form data and keeping only the anonymised forms.

Keeping you and your data safe
Everyone involved is Garda vetted which means the Guards have checked our details to make sure we are approved to work with families and children. It is important that you know that any notes that we take will not be linked to you or your family or your location. We will store our notes on the Trinity College computer system where nobody can look at them without a password. I promise to look after your data very carefully. You may request for your data to be removed up until the point where I anonymize your data. Only the research team will have access to your data.

Confidentiality, Publishing your data
There may be media reporting, lectures, college projects, including MCS theses, conference presentations and journal articles written as a result of this research. However, none of this will include any information that would identify you.

Research Team
Researcher: Ellen Whelan
email: whelane5@tcd.ie

Supervisor: Glenn Strong
email: Glenn.Strong@tcd.ie
Assent

Trinity College Dublin Informed Assent Form

Lead Researcher: Ellen Whelan

Background of Research:
The research is being conducted to find the best version of a programming environment designed to introduce students familiar with block based programming, to text based languages such as Python.

Procedures of this study:
1. Asked to complete two short surveys, one before the session and one after the session.
2. Shown a programming environment and taught how to use it
3. Asked to complete a series of tasks
4. Asked to participate in a focus group.
5. Asked to provide assent for the researcher to record your voice contribution made within the context of a group discussion, conducted at the end of the session.

The session will last approximately one hour.

Publication:
This study is being completed as part of my Master’s Thesis and will be presented to the School of Computer Science and Statistics in Trinity College Dublin.

Individual results may be aggregated anonymously and research reported on aggregate results.

DECLARATION:
- I am under 18 years and have a signed consent form from a parent/guardian.
- I have read, or had read to me, a document providing information about this research and this consent form. I have had the opportunity to ask questions and all my questions have been answered to my satisfaction and I understand the description of the research that is being provided to me.
- I agree that my data is used for scientific purposes and I have no objection that my data is published in scientific publications in a way that does not reveal my identity.
- I understand that if I make illicit activities known, these will be reported to appropriate authorities.
- I understand that I may stop electronic recordings at any time, and that I until the point of anonymization can have such recordings destroyed (except in situations such as above).
- I understand that, subject to the constraints above, no recordings will be replayed in any public forum or made available to any audience other than the current researchers/research team.
- I freely and voluntarily agree to be part of this research study, though without prejudice to my legal and ethical rights.
- I understand that I may refuse to answer any question and that I may withdraw at any time without penalty.
- I understand that my participation is not fully anonymous and that my name will be used for matching pre- and post-surveys.
- I understand that if I or anyone in my family has a history of epilepsy then I am proceeding at my own risk.
- I have received a copy of this agreement.

**Assent**

I have read, or had read to me, information telling me what will happen to any information about me and how I or my parent can contact the researcher. I assent to take part in this research. By selecting 'assent' I assent to continue with this survey this document, I assent to participate in

1. Do you assent to the above? *

   ○ I assent

   ○ I do not assent, I do not wish to participate
Pre Session Survey

A few questions before we get started!

2. Name

3. How long have you been programming for?
   - [ ] Less than 3 months
   - [ ] 3 months - 1 year
   - [ ] 1 year - 2 years
   - [ ] More than 2 years
   - [ ] No programming experience

4. What programming languages have you used before?
   - [ ] Scratch
   - [ ] Processing
   - [ ] Microbit
   - [ ] HTML/CSS
   - [ ] Python
   - [ ] Turtle Graphics
   - [ ] Other
This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.

Microsoft Forms
Appendix C: Post-Session Survey
Post Session Survey

The survey will take approximately 3-5 minutes to complete. Please be assured that while we ask for you name, all of your replies will be anonymized. Participating in the survey is voluntary and you may change your mind and stop at any time. You may exit the survey at any point without submitting by closing your browser window. You may also choose not to answer a question at any time and skip it without penalty.

* Required

About the Survey

A few questions to help us understand your experience with Pytch.

1. Do you assent to participating in this survey? *
   - [ ] I assent.
   - [ ] I do not assent, I don't want to participate.
Post Session Survey

2. Name

[Input field]
Pythch Environment

Please rate how strongly you agree or disagree with the following statements.

3. I enjoyed using the Pythch Environment.
   - [ ] Strongly Agree
   - [ ] Agree
   - [ ] Neither Agree or Disagree
   - [ ] Disagree
   - [ ] Strongly Disagree

4. I found the programming tasks easy to complete.
   - [ ] Strongly Agree
   - [ ] Agree
   - [ ] Neither Agree or Disagree
   - [ ] Disagree
   - [ ] Strongly Disagree
5. I would use Pytch again.

- [ ] Strongly Agree
- [ ] Agree
- [ ] Neither Agree or Disagree
- [ ] Disagree
- [ ] Strongly Disagree
Working with Functions

A couple of questions about how you got on writing and using functions.

6. It was easy to find the correct place in my program to add a new function
   - [ ] Strongly agree
   - [ ] Agree
   - [ ] Neither Agree or Disagree
   - [ ] Disagree
   - [ ] Strongly disagree

7. It was easy to write the code for my new function
   - [ ] Strongly agree
   - [ ] Agree
   - [ ] Neither Agree or Disagree
   - [ ] Disagree
   - [ ] Strongly disagree
Working with Bugs

A couple of questions about how you found dealing with bugs in your program.

8. It was easy to find the correct place in my code to fix a bug.

   ○ Strongly agree
   ○ Agree
   ○ Neither Agree or Disagree
   ○ Disagree
   ○ Strongly disagree

9. It was easy to write the code to fix my bugs.

   ○ Strongly agree
   ○ Agree
   ○ Neither Agree or Disagree
   ○ Disagree
   ○ Strongly disagree
Understanding my Code

Some questions about your understanding of how the overall program works.

10. I found it easy to understand how the sprite communicate with each other.
   ○ Strongly agree
   ○ Agree
   ○ Neither Agree or Disagree
   ○ Disagree
   ○ Strongly disagree

11. I found it easy to understand how the sprites react to events.
   ○ Strongly agree
   ○ Agree
   ○ Neither Agree or Disagree
   ○ Disagree
   ○ Strongly disagree
12. I found it easy to manage my code as the program got bigger

- [ ] Strongly agree
- [ ] Agree
- [ ] Neither Agree or Disagree
- [ ] Disagree
- [ ] Strongly disagree

13. I found the amount of text in my program confusing and overwhelming.

- [ ] Strongly agree
- [ ] Agree
- [ ] Neither Agree or Disagree
- [ ] Disagree
- [ ] Strongly disagree

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.

[Microsoft Forms]