2018 IEEE Teaching, Assessment and Learning for Engineering (TALE), 4-7 December 2018, Wollongong, Australia, (c) 2018 IEEE

SplashKit: A Development Framework for Motivating and Engaging Students in Introductory Programming

Jake Renzella
School of Information Technology
Deakin University, Geelong
Melbourne, Australia
jake.renzella@deakin.edu.au

Alex Cummaudo
Applied Artificial Intelligence Institute
Deakin University, Geelong
Melbourne, Australia
ca@deakin.edu.au

Andrew Cain
School of Information Technology
Deakin University, Geelong
Melbourne, Australia
andrew.cain@deakin.edu.au

John Grundy
Faculty of Information Technology
Monash University, Clayton
Melbourne, Australia
john.grundy@monash.edu

Jonathon Meyers
School of Information Technology
Deakin University, Geelong
Melbourne, Australia
jon.meyers@deakin.edu.au

***Abstract*—Learning to program is known to be challenging for many students. Upon entry, students often have poor perceptions of their capabilities with some anxiety around the challenges they expect to face in learning to code. Lowering the barriers to entry will help ease students into programming and enable a broader range of student to continue programming. SplashKit is an educationally focused development framework designed to aid the teaching of programming by empowering students to create interesting and dynamic programs from their first programming tasks. This paper explores how SplashKit can be used in tertiary education to underpin a range of introductory programming approaches.**

Keywords—programming framework, programming education, introductory programming, motivating programming students

I. INTRODUCTION

Since the introduction of computer programming coursework at the undergraduate level students' understanding of code has always posed a challenge to educators [11], [16], [17]. Accordingly, a range of approaches aimed at helping to ease this process have been developed. These approaches range from innovations associated with tool support, curricula, pedagogy, and language choice [20]. In relation to tool support, there are challenges with many programming tools being developed and focused on professional software engineers. In this context, the array of options provided to professionals will overwhelm students for whom even the basics are challenging [2].

Within the education tool space, there are a range of graphical programming environments that aim to simplify the construction of valid programs by switching from a text-based language to a building block-based approach [12]. These graphical languages abstract away part of the software development process, thereby focusing the student's attention on algorithmic thinking, rather than syntax and language-specific features. While these approaches have great merit, they are not likely to be appropriate for computer science and software engineering students who need to start building their proficiency and familiarity with the tools that are primarily used within their industry. In this context, a different approach is required.

The approach we propose is based around two main goals: to enable students to create interesting, fun and engaging programs, while also ensuring that students use industry standard languages, and tools, and are put in control of the programs they create.

The benefits of these goals have been shown in innovations like Georgia Tech's Media Computation course [1], which suggests that allowing students to develop programs they engage with is likely to increase the time students spend on the task and has seen improved outcomes. Integrating professional tools is more challenging, the goal here is to focus on selecting options that are not likely to increase the cognitive load of tasks students perform [2]. By getting started using only basic tools we aimed to help students better understand the processes behind more complex environments, while still enabling them to create programs that are likely to engage their interest.

As the focus of the introductory programming units is around developing algorithmic awareness, and understanding of the notional machine [3], it is important that these elements are the focus of student attention. In this regard, the second goal requires such tools to empower students while requiring them to implement the controlling logic. The application programming interfaces (APIs) students first use when learning to program need to be simple and flexible, with a focus on ensuring that students can rationalise about the programs they create.

Teaching programming has a varied range of methodologies, both for future software engineers and practitioners who will need to program to fulfil domain-specific duties. Moreover, programming is rapidly moving from an undergraduate to K-12 curriculum. To make programming fun and engaging from day-one, and not to overwhelm students with complex programming tools such as IDEs, we developed SplashKit, a cross-platform, multi-language, open-source library and set of tools with accompanying learning resources. Using SplashKit, students can build a wide range of programs — such as video games, programs to consume and publish resources via web services, programs to perform data manipulation and persistence in databases, and programs that interact over a network. The design of the SplashKit API enables these complex features to be implemented with less code and in a way that helps students develop appropriate conception of computation.

In this paper, we first detail SplashKit and its features in Section II, explore how SplashKit supports student learning in Section III, and discuss possible benefits and implementations of SplashKit in curriculum in Section IV.

II. Related Work

Since the development of the BASIC programming language in 1963 [8], computer science educators have been designing or simplifying programming environments in order to reduce the barrier of entry for students new to programming. There are two large categories of student motivation for learning programming [9]:

a. Students or practitioners seeking to solve domain problems with computer programs; e.g. biology students who write algorithms for DNA sequencing.

b. Students learning to program for their own sake; i.e. computer science students.

The BASIC programming language is an example of category a. Kemeny and Kurtz [8] attempted to reduce the complexity for non-science students when learning to program. Existing languages — such as Fortran or Algol — required a level of specificity, that they were concerned may deter students. More recent programming languages such as Blue (1996) and Grace (2010) were designed for category b, addressing the problems in teaching programming in an introductory programming unit [10], [12]. Grace's development panel describe its primary goal as an aid to cover first-year programming principles, such as program design, data structures and algorithms [11].

Furthering these developments, the recent push to deliver programming curriculum from the undergraduate level to the K-12 level has resulted in an increasing number of teachers making use of teaching-first languages and environments [18], [19]. An example of this is the Scratch programming environment [12], a live, visual programming environment aimed for students between the ages eight and sixteen. Scratch has largely been very successful: as of May 2018, there have been over twenty-eight million users registered in the Scratch programming framework website [14]. Even with Scratch targeting younger learners, it has also been used in early introductory programming undergraduate courses with a transition to Java mid-semester [15]. Table 1 shows further examples of common educational resources and their associated teaching approaches.

TABLE 1. EXAMPLES OF LANGUAGE APPROACHES

Approach	Examples
Primarily graphical "building-blocks".	Scratch, Alice, Google Blockly.
Educational-first designed programming languages.	Grace, Pascal
Industry languages commonly used in education (with or without frameworks).	Python, Java, C#

These past and current efforts highlight the ongoing pursuit of and demand for high quality educational resources designed to ease new students into programming. While tools like Scratch and Alice are very popular resources, their purpose and approach comprise mainly of graphical "building blocks". This approach differs widely to that of SplashKit which we explore in later sections.

III. Introducing SplashKit

A. Overview

SplashKit consists of a core API, dependent libraries, associated tools and educational resources. These components work together to enable SplashKit to be used in support of a range of different teaching methods and approaches (see Fig. 1).

Central to SplashKit is the *SplashKit Core API*, which exposes features for students to use. This API is written on dependent libraries including: libCURL, SQLite, SDL. The API is read by the *SplashKit Translator* to generate website documentation and multiple language adapters (currently Python, Pascal, C# and C/C++). Students can then write programs in any of these supported programming languages, with the aid of the accompanying open education resources available online. To compile these programs, students make use of the *SplashKit Manager (SKM)*, which works as an interface between the source code and language's compiler, giving the programmer access to a broader range of features from the SplashKit API, as well as optimisation tools to more easily initialise a new project.

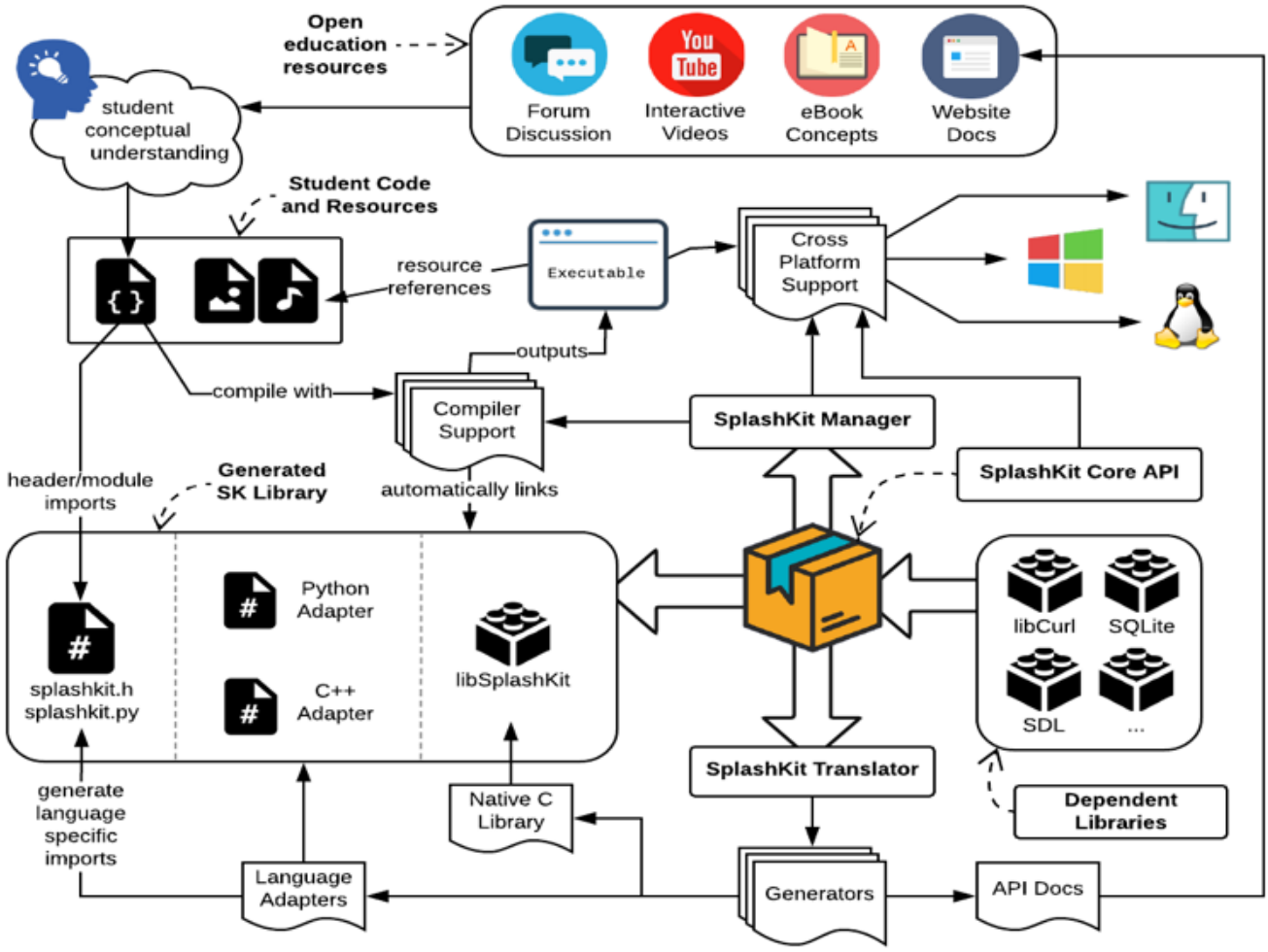

Fig. 1: Overview of SplashKit.

B. *SplashKit Features*

Having a context in which to apply the use of computing has been shown to make programming more relevant, and to improve student success [4]. This can be challenging when delivering a general introductory programming unit, as a single context may not engage all students. For many years, we used a games programming context with an earlier version of SplashKit, and while this has been successful for many students, we have found that this one context does not appeal to all, for example business students may not be motivated to create games. To help address this, we redeveloped the framework used for games programming to incorporate a wider range of potential contexts. When used in conjunction with a flexible assessment approach — such as the approach used by Cain et al. in portfolio assessment [5], [6] — core work can be targeted in a general context, with students free to pursue their own interests across a range of tasks. We anticipate that the broader set of capabilities is likely to appeal to a greater number of students, and to help engage students from other disciplines.

SplashKit encapsulates several third-party libraries to provide a simple, consistent interface for students to program against. External libraries provide means of developing multimedia applications, interacting with embedded databases, accessing web resources, creating web servers, parsing and creating JSON structures, interacting with social media, and other capabilities. The SplashKit API packages these and exposes them in a consistent manner, so that they are accessible to beginner programmers. This provides students access to advanced capabilities while removing the need to address issues such as compatibility, installation and configuration.

For these capabilities, we spent time designing the interface of the SplashKit Core API so that it is used in a **procedural style**, with the beginner programmer always in control of the program sequence. This typically meant introducing some inefficiencies to prioritise simplicity and to enable a clear programming style to be adopted by the students.

An example of this is the way students create and engage with the embedded web server. The underlying library uses an asynchronous, call-back-based architecture, allowing professional programmers to register call-backs that will execute when the web server receives different requests. This saves the programmer from having to write what would be considered

boiler plate code, but requires that the programmer understand the idea of a call-back, and the idea that the library code has some processing that will call the registered functions when given criteria are met. Similarly, standard graphical applications typically abstract away the idea of the event loop. Instead developers would register for events, and receive call-backs when these events occur.

In these cases, the call-back structure is advantageous to professional developers, but hides away important details for beginners. These architectures require a greater level of understanding to comprehend how they function, and therefore tend to hinder beginner students as they attempt to understand how their programs are behaving within the computer.

The approach we took with SplashKit, was to hide these more advanced concepts and require students to actively control the program's behaviour themselves. An example can be seen in Fig. 2, which shows the listing of a simple web server that hosts a RESTful web service and includes a graphical user interface. This program creates a web server to accept requests and opens a window to provide some feedback on the current state of the program. The student would then need to implement an event loop that runs until the user quits the program. Within the loop the program is then responsible for updating the window and handling any incoming requests. In terms of programming constructs, this uses basic programming structures, making this more accessible for beginners.

The program in Fig. 2 may easily be extended by students as they are introduced to more concepts. The string data could be expanded to a custom data type using a structure. Multiple values could be handled with an internal array or with data saved into an embedded database. The graphical interface could be enhanced to track the number of times values are changes, added, or deleted, and so on. The capabilities of SplashKit allow students and staff to explore a wide range of interest areas.

SplashKit supports both object-first and object-later approaches of programming curriculum. For instance, the code in Fig 2. shows an example of a Web Server, using the *has_incoming_requests(message_service)* function (line 33). When used in C# (not shown), the codebase makes use of an equivalent object-oriented call, by mapping such functionality to a *getter* property: *message_service.HasIncomingRequests*. Similarly, the function *is_get_request_for(request, "/message")* is mapped to the instance variable: *request* as a method: *request.is_get_request_for("/message")*. This is discussed further in Section IV.

```
1.  void process_request(web_server message_service, string &me
    ssage) {
2.    http_request request = next_web_request(message_service);
3.
4.    if ( is_get_request_for(request, "/message") )
5.      send_response(request, message);
6.    else if ( is_put_request_for(request, "/message") ) {
7.      message = request_body(request);
8.      send_response(request);
9.    }
10.   else if ( is_get_request_for(request, "/index.html") )
11.     send_html_file_response(request, "message_index.html");
12.   else
13.     send_response(request, HTTP_STATUS_BAD_REQUEST);
14. }
15.
16. int main() {
17.   string message = "Get started with SplashKit";
18.
19.   web_server message_service = start_web_server();
20.
21.   open_window("Quote of the Moment", 300, 100);
22.
23.   while ( not quit_requested() ) {
24.     process_events();
25.     clear_screen();
26.
27.     draw_text(message, COLOR_BLACK, 10, 10);
28.     draw_text("Activity:", COLOR_BLACK, 10, 30);
29.     draw_text("Close to quit server", COLOR_BLACK, 10, 60);
30.
31.     refresh_screen();
32.
33.     if ( has_incoming_requests(message_service) ) {
34.       fill_rectangle(COLOR_GREEN, 80, 30, 10, 10);
35.       refresh_screen();
36.       process_request(message_service, message);
37.       delay(150);
38.     }
39.   }
40.
41.   return 0;
42. }
```

Fig. 2 Example Web Service written in SplashKit in C++ in 42 lines of code.

C. *Compiling SplashKit Projects*

SplashKit incorporates a number of underlying libraries, and needs to be linked or otherwise made accessible to the executables students create. To enable this, we needed a way to easily allow students to compile their programming projects. We did not want to involve complex setups of IDEs, or long convoluted compiler calls from the command line.

To address this, we created the *SplashKit Manager (SKM),* a tool designed to act as the student interface to SplashKit on the command line. To get started with SplashKit, students first need to install SKM. The SKM installation is designed to be simple, but not oversimplify development practices such that it also teaches *transferrable* development skills. Students install SKM using a single command line instruction. This instruction downloads and installs the latest release of the SplashKit Core API, language adapters, and SKM.

One challenge to this process is the different command line environments and tools available across the different operating systems. While the process is relatively straightforward on Linux and macOS, the same approach requires an extra step on Windows as Windows users need to install a POSIX compatibility layer, such as MSYS2 [23], to access the command line tools. To ease this process, an installer is provided which is preconfigured for SplashKit/SKM. It is believed that as the Windows Bash Subsystem matures, this could replace the need to install a tool such as MSYS2 [7].

Once the installation download is complete, SKM will be added to the student's PATH environment variable so that they can build and run SplashKit code for any of the supported languages without any further configuration. An example usage of SKM to build a SplashKit program is seen in Fig. 3.

We designed SKM to wrap valid calls to the compiler. This can be demonstrated again in Fig. 3, by removing the prepended *skm* from the compilation command, the user is left with a completely valid call to the *clang++* compiler. This is done to ensure students are learning transferable skills, and will develop an appropriate understanding of how to use the compiler themselves across the duration of introductory programming units. Ideally, students would start compiling with SKM, and as they learn more about the compilation process, they could eventually transition to providing these compilation flags themselves.

```
$ skm clang++ ./my_program.cpp -o my_program
```

```
$ clang++ ./my_program.cpp -o my_program \
    -L/usr/local/lib \
    -lSplashKit -lSplashKitCpp \
    -I/usr/local/include \
    -rpath @loader_path \
    -rpath /usr/local/lib
```

Fig. 3 Example SplashKit compilation commands using SKM and without SKM. Prefixing a *clang++* call with *skm* simply adds all required compiler flags that would, otherwise, confuse first-time programmers.

Rather than have students interact with the command-line, using an Integrated Development Environment (IDE) with SplashKit and SKM is also a possibility. There is a concern that the introduction of fully-featured IDE in introductory programming may overwhelm the students, and increase the initial cognitive load of learning to program. In an introductory Java course, Z. Chen and D. Marx [23] proposed having students use the JDK directly (command-line) for tasks in the first few weeks, before switching over to a fully featured Eclipse IDE. Another approach is to use simplified IDEs such as Visual Studio Code [24], which provides the benefits of traditional IDEs without overloading the students with complex features that are not relevant in learning the basics of programming.

In addition to compilation, SKM also includes features for keeping the SplashKit Core API and language adapters up to date. This simplifies the process of updating SplashKit, enabling students to get the latest version by running *skm update*.

IV. Supporting Student Learning With SplashKit

To accompany the SplashKit software we have also developed a range of online resources aimed to help students develop their understanding of computation and to make effective use of the SplashKit API as they learn to program.

Website documentation provides a key reference resource for students, enabling them to explore SplashKit and get details on its various features and how these can be integrated into their code. While the main goal of the website is to facilitate access to the API, this process also helps students develop important skills associated with reading API documentation, an essential software engineering skill.

Accessible from the SplashKit YouTube channel are a range of video resources developed to help students understand the concepts that underlie their code. These videos portray programming concepts using a conversational format, with the presenters typically taking on student and teacher roles to help discuss and address common issues and misconceptions. Programming concepts are inherently abstract, making them difficult for students to engage with. To help address this, the videos use visualisations and analogies to provide students with an effective means of approaching these concepts. Programming concepts are also highly interrelated, and the videos make use of hyperlinks within the videos to connect to associated concepts and to provide an interactive experience for students online.

The open and extensible nature of these resources provide educators with a platform for teaching introductory programming. One of the great benefits of a free and open-source platform is that others are welcome, and encouraged, to provide contributions to extend SplashKit where desired features are not present in any of the resources or APIs.

A. Decoupling SplashKit from Languages and Paradigms

Another key goal of SplashKit was to ensure that the API would work across a range of imperative programming languages and across procedural and object-oriented paradigms. To this end, the SplashKit Core API and translator were designed to enable the generation of language specific adapters. This is shown in Fig. 1.

The SplashKit Core API is read by the SplashKit Translator which outputs adapters that include the code necessary to connect to SplashKit from different programming languages. The translator reads marked comments in the Core API code, to provide appropriate mappings between the different languages, ensuring that each language adapter mirrors its language's style and practices.

Using language adapters enables SplashKit to support a wide range of different programming languages, and the translator is designed in such a way that new languages can be added relatively easily by specifying the new language's coding conventions and means of integrating with external libraries.

Students develop their programs in a supported language, and the adapter is responsible for ensuring that data is correctly marshalled between the adapter's programming language and the native Core API library. In addition to generating language adapters, the translator is also used to generate API documentation published on the SplashKit website (https://splashkit.io). The documentation produced by the translator is primarily language neutral, while allowing students to select their programming language to see the relevant signatures. In this way, the one set of API documentation can be used to support multiple programming languages.

To support both an objects-first and objects-later approach to introductory programming, HeaderDoc comments in the Core API code include additional details which are used to generate both procedural and object-oriented APIs. Internally, the Core API is implemented using the procedural paradigm with resources passed to functions that operate on them. SplashKit resources are implemented as pointers, which allow them to be

passed in or encapsulated as an object. When generating an adapter, the Translator exports functions for a procedural adapter, or wraps the resources as objects and converts the functions to methods on these objects.

Fig. 4 shows an example of the marked up SplashKit code for some *Sound Effect* related functions. The *class* attribute is used to indicate the class these functions will be attached to in an object-oriented adapter. This translation also requires a change to the method name, which is facilitated through the *method* attribute. For example, the *Play Sound Effect* function in a procedural language is change to the *Play* method on the *Sound Effect* class in an object-oriented language. In this way, the translator is able to create both object-oriented and procedural APIs.

```
1.   * Plays a sound effect once at full volume.
2.   * @attribute class    sound_effect
3.   * @attribute method   play
4.   * @attribute self     effect
5.   * @attribute group    audio
6.   * @attribute static   audio
7.   */
8.   void play_sound_effect(sound_effect effect);
9.
10. /**
11.  * …
12.  * @attribute class    sound_effect
13.  * @attribute method   play
14.  * @attribute self     effect
15.  * @attribute suffix   with_times
16.  * @attribute group    audio
17.  * @attribute static   audio
18.  */
19.  void play_sound_effect(sound_effect effect, int times);
```

Fig. 4 Sample documentation in the SplashKit Core API.

```
1.  void play_sound_effect(const string &name)
2.
3.  public static void Audio.PlaySoundEffect(string name);
4.  public static void SplashKit.PlaySoundEffect(string name);
5.
6.  procedure PlaySoundEffect(const name: String)
7.
8.  def play_sound_effect_named(name):
```

Fig. 5 How the *play_sound_effect* function appears in several languages on the SplashKit API documentation website. *From top to bottom*: C++, C#, Pascal and Python.

Function and method overloading is another issue that needed to be tackled by the translator. Examine the *Play Sound Effect* function in Fig. 4. The function is overloaded to allow the caller to provide either the sound effect or its name, as well as combinations that allow changes in the volume and number of times the effect is to be played. To address this, the document attributes also include a *suffix* that can be applied in cases where the adapter language does not support overloading. In each of these cases, the suffix attribute is added to the function or method name in the adapted language. Fig. 5 shows examples of how the sound effect functions from Fig. 4 would appear in C++, C#, Pascal and Python adapters, as shown in the SplashKit API documentation website. Note how the C# listing shows two ways of calling this method: either through the *Audio* class (as bound via the *static* attribute) or through the generic *SplashKit* class.

The multi-lingual nature of SplashKit provides additional advantages in helping ease student's transitions to new programming languages. Learning a new language involves learning both the syntax of the language, as well as becoming familiar with the language's standard libraries and utilities to perform basic actions. By using SplashKit, we can help reduce this complexity by having students program against the SplashKit API in the new language. This enables students to focus initially on learning the language, while making use of the familiar SplashKit API.

B. *Sample Curriculum*

We have used SplashKit to teach introductory programming for several years at both undergraduate and postgraduate level. In our case, we have adopted an objects-later approach, with the first unit of study exploring structured procedural programming, and the subsequent unit introducing object-oriented design.

In the introductory programming unit, SplashKit is used to guide students through the core programming concepts. Topics were organised so that students can understand and rationalise about all the code presented with as little "magic" (unexplained code) as possible. The intent here is to guide students through the software development process and understand core concepts without abstracting too much of this process away. Content within each topic is organised in three categories:

- **Artefacts** are programming abstractions students create in their code, with accompanying material encouraging them to visualise these as entities that exist in their code.
- **Actions** relate to the instructions for the computer to perform, with these statements interacting with the software artefacts to achieve the program's objectives.
- **Terminology** encompass all other concepts and include things students need to memorise and understand to operate effectively in the areas of computer science and software engineering.

The curriculum starts by introducing students to *program, library,* and *procedure* artefacts, the *procedure call* action, and the key concept of *sequence*. Programs are the overall coding artefact, with the logic being divided into many procedures. Procedures are then an artefact that contains a sequence of instructions designed to perform a single task. At this stage, each instruction is itself a call to a procedure, where the procedure may exist in the program's code, or in an external library such as the language's standard library or the SplashKit library. Within Topic 1 we have students use SplashKit to create a scripted sequence, such a drawing a picture using basic shapes, or creating a short animation with images and sounds. We have found that this task helps students familiarise themselves with programming processes, while exploring the basic programming concepts in a creative fashion. It is always interesting to see how creative students can be even with such limited programming abstractions.

Data, parameters, and functions follow in Topic 2. These build on the ideas from Topic 1 and help overcome one of the limitations where similar tasks would need to be duplicated even if they differ only in data values used. This topic introduces students to the *variable* and *function* artefacts, the *assignment statement* action, as well as the terminology associated with variables (i.e. parameters, arguments, local, and global variables etc.). SplashKit functions provide a convenient means of helping students explore this topic. Students can create custom drawing functions that accept parameters, use local variables to calculate

coordinates, and then perform the required drawing. An example may be to draw a framed image, where the frame and image may be of different sizes, requiring appropriate scaling for the frame to correctly surround the picture.

This leads nicely on Topic 3, *control flow*, which helps address the obvious limitation that is: up to this point programs have been just a sequence of actions. This introduces the ideas of *selection* and *repetition*, which can be motivated by exploring limitations students will have faced. For example, implementing an event loop is a nice motivating example for this topic. A small program can be written to show a picture, with a short delay to keep it on the screen. When it disappears, this can be used to discuss the current limitations and hopefully spark a desire to overcome this through additional programming understanding. This exploration can then lead onto the idea of using a loop to repeat instructions until the window is closed, and then even further to using if statements to selectively run code *if* a certain key is held down, or when a request is received over the network. With just these few concepts, students can now start to explore the API to make exciting small programs including games, database applications, and web servers.

Following on from this topic, we have students learn about organising data with custom data types (structures) and then on to working with arrays and objects. In each case, SplashKit empowers students to build fun and engaging programs, while keeping the focus on programming concepts.

C. Student Programs

In our experience teaching with SplashKit, students have been able to use these tools to create a wide range of fun and engaging games, as well as small business-like applications. Fig. 6 shows three small games which have been developed. *IsoCity* is a City Simulator in which the user can build and demolish buildings in a simulated city. *Dart Dodger* has the player controlling a balloon as it aims to rise high into the sky. As the balloon ascends the difficulty increases with wind, and the number of obstacles increasing, ultimately leading to a high score saved to disk. Finally, *Shape Platformer* has the player navigate a green square through several levels that include platforms, spikes, and jumping pads and provides a level editor in which students can create and save levels to a file and load later. These programs were all designed and implemented by first-year students in their first programming unit.

SplashKit is also often used by students less motivated by the idea of programming games, and these students have created basic programs making use of databases and web servers.

Fig. 6 Sample student projects developed with an earlier version of SplashKit.

V. Conclusion

Since the early days of tertiary introductory programming courses, there have been a number of programming languages, approaches, and tools developed to help lower the barrier to entry for new students. These tools are often designed for professionals — with interfaces and features overwhelming students new to programming.

SplashKit is a free, open-source, language-agnostic, and cross-platform development framework designed for tertiary education. SplashKit is flexible, supporting a wide range of approaches to programming education including: objects-first or objects-later curriculum design, programming language choices, and a range of engaging and relevant programming tools such as a fully-featured game engine, databases, and web-servers — all with an easy to approach API design.

The Core API can be developed to add new capabilities and libraries such as Machine Learning or Data Analytics libraries, which when complete will automatically be available across the different programming languages and platforms with no extra effort. New languages can be added to the translator, thereby bringing SplashKit to educators with existing materials and curriculum which make it difficult and costly to switch programming languages at their institution.

We are motivated by the amazing applications and games which first-time students have been able to build using SplashKit, and the engagement we have seen so far. Where others adopt SplashKit for their teaching, we also hope they will contribute to the education resources so that we may grow as a community.

VI. Acknowledgements

The authors acknowledge the many helpful comments from students and tutors who have used the SplashKit development platform in improving this work.

References

[1] M. Guzdial, "*Exploring Hypotheses About Media Computation",* Proc. 9th Ann. Conf. International Computing Education Research, 2013, pp. 19-26.

[2] R. Mason, G. Cooper, "Distractions in Programming Environments", Proc. 15th Australasian Computing Education Conference, 2013.

[3] B. DuBoulay, "Some difficulties of learning to program", J.Educational Computing Research, 2(1), 1986, pp. 57-73

[4] L. Porter, M. Guzdial, C. McDowell, and B. Simon, "Success in introductory programming: what works?" Commun. ACM, 56(8), 2013, pp. 34-6.

[5] A. Cain, C. Woodward, "Toward Constructive Alignment with Portfolio Assessment for Introductory Programming", Proc. 1st IEEE Int. Conf. Teaching Assessment and Learning for Engineering, 2012, pp. 345-350.

[6] A. Cain, "Factors influencing student learning in portfolio assessed introductory programming.", Proc. 3rd IEEE Int. Conf. Teaching Assessment and Learning for Engineering, 2014, pp. 55-62

[7] J. Hammons, et.al. "Bash on Ubuntu on Windows", Microsoft Developer Network at https://msdn.microsoft.com/en-us/commandline/wsl/about

[8] KURTZ, T. 1981. BASIC. In Wexelblat, R., Ed. History of Programming Languages. Academic Press, New York, 515–537.

[9] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," ACM Computing Surveys (CSUR), vol. 37, no. 2, pp. 83–137, Jun. 2005.

[10] M. Kölling, J. Rosenberg, M. Kölling, and J. Rosenberg, Blue—a language for teaching object-oriented programming, vol. 28, no. 1. ACM, 1996, pp. 190–194.

[11] A. Gomes and A. J. Mendes, An environment to improve programming education. New York, New York, USA: ACM, 2007.A. Black, K. B. Bruce, and J. Noble, "Designing the Next Educational Programming Language," 2010.

[12] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch Programming Language and Environment," ACM Transactions on Computing Education (TOCE), vol. 10, no. 4, pp. 16–15, Nov. 2010.

[13] "Scratch - Imagine, Program, Share", Scratch.mit.edu, 2018. [Online]. Available: https://scratch.mit.edu/statistics/. [Accessed: 28- May- 2018]

[14] D. J. Malan, H. H. Leitner, D. J. Malan, and H. H. Leitner, Scratch for budding computer scientists, vol. 39, no. 1. ACM, 2007, pp. 223–227.

[15] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari, "From Scratch to 'Real' Programming," ACM Transactions on Computing Education (TOCE), vol. 14, no. 4, pp. 25–15, Feb. 2015.

[16] T. Jenkins, "On the difficulty of learning to program," Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences, vol. 4, Feb. 2002.

[17] P.-H. Tan, C.-Y. Ting, and S.-W. Ling, "Learning Difficulties in Programming Courses: Undergraduates' Perspective and Perception," presented at the 2009 International Conference on Computer Technology and Development, pp. 42–46.

[18] S. Grover and R. Pea, "Computational Thinking in K–12: A Review of the State of the Field," Educational Researcher, vol. 42, no. 1, pp. 38–43, Jan. 2013.

[19] Y. B. Kafai and Q. Burke, "Computer Programming Goes Back to School:," *Phi Delta Kappan*, vol. 95, no. 1, pp. 61–65, Sep. 2013.

[20] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, J. Paterson, A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson, "A survey of literature on the teaching of introductory programming," arXiv, vol. 39, no. 4, pp. 204–223, Dec. 2007.

[21] "SplashKit", *Splashkit.io*, 2018. [Online]. Available: http://www.splashkit.io/. [Accessed: 06- Jun- 2018].

[22] "MSYS2 homepage", Msys2.org, 2018. [Online]. Available: https://www.msys2.org/. [Accessed: 06- Jun- 2018].

[23] Z. Chen and D. Marx, "Experiences with Eclipse IDE in programming courses," J. Comput. Sci. Coll., vol. 21, no. 2, pp. 104–112, Dec. 2005.

[24] "Visual Studio Code - Code Editing. Redefined", Code.visualstudio.com, 2018. [Online]. Available: https://code.visualstudio.com/. [Accessed: 07- Jun- 2018].